Report Number: WUCSE-2002-30

2002-08-29

# Economy of Interaction in Program Visualization: Designing Effective Visualization Tools for Reducing User's Cognitive Effort - Doctoral Dissertation, August 2002

Mihail-Eduard Tudoreanu

Program visualization has the potential to be an important tool for people who seek to observe and understand the behavior of a running computation. This thesis focuses on alleviating barriers to the realization of this potential that pertain to the design of a visualization system and to insufficient knowledge about how people take advantage of program visualizations. Our major contribution is the design of a visualization approach capable of improving user's performance through the use of economy of information and tasks. We present evidence from our empirical studies that this type of economy promotes animations capable of significantly improving... Read complete abstract on page 2.

# Economy of Interaction in Program Visualization: Designing Effective Visualization Tools for Reducing User's Cognitive Effort - Doctoral Dissertation, August 2002

Mihail-Eduard Tudoreanu

Complete Abstract:

Program visualization has the potential to be an important tool for people who seek to observe and understand the behavior of a running computation. This thesis focuses on alleviating barriers to the realization of this potential that pertain to the design of a visualization system and to insufficient knowledge about how people take advantage of program visualizations. Our major contribution is the design of a visualization approach capable of improving user's performance through the use of economy of information and tasks. We present evidence from our empirical studies that this type of economy promotes animations capable of significantly improving people's understanding of the computation. We apply this knowledge to develop a system for creating application-specific visualizations solely through interactions with program visualizations and textual views of the computation, thus promoting economy of interaction. The system is built around the principle that animation viewers are also the creators of animations and systematically refine the visualizations to suit their momentary goal.

2002-30

# Economy of Interaction in Program Visualization: Designing Effective Visualization Tools for Reducing User's Cognitive Effort - Doctoral Dissertation, August 2002

Authors: Tudoreanu, Mihail Eduard

August 29, 2002

Abstract: Program visualization has the potential to be an important tool for people who seek to observe and understand the behavior of a running computation. This thesis focuses on alleviating barriers to the realization of this potential that pertain to the design of a visualization system and to insufficient knowledge about how people take advantage of program visualizations. Our major contribution is the design of a visualization approach capable of improving user's performance through the use of economy of information and tasks. We present evidence from our empirical studies that this type of economy promotes animations capable of significantly improving people's understanding of the computation. We apply this knowledge to develop a system for creating application-specific visualizations solely through interactions with program visualizations and textual views of the computation, thus promoting economy of interaction. The system is built around the principle that animation viewers are also the creators of animations and systematically refine the

SEVER INSTITUTE OF TECHNOLOGY

DOCTOR OF SCIENCE DEGREE

DISSERTATION ACCEPTANCE

(To be the first page of each copy of the dissertation)

DATE: July 19, 2002

STUDENT'S NAME: Mihail-Eduard Tudoreanu

This student's dissertation, entitled Economy of Interaction in Program Visualization: Designing Effective Visualization Tools for Reducing User's Cognitive Effort has been examined by the undersigned committee of six faculty members and has received full approval for acceptance in partial fulfillment of the requirements for the degree Doctor of Science.

APPROVAL: ______________________ Chairman

______________________

______________________

______________________

______________________

______________________

Short Title: Interaction in Program Animation Tudoreanu, D.Sc. 2002

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

---

# ECONOMY OF INTERACTION IN PROGRAM VISUALIZATION: DESIGNING EFFECTIVE VISUALIZATION TOOLS FOR REDUCING USER'S COGNITIVE EFFORT

by

Mihail-Eduard Tudoreanu

Prepared under the direction of Profs. Gruia-Catalin Roman and Eileen Kraemer

---

A dissertation presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Doctor of Science

August, 2002

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

ABSTRACT

# ECONOMY OF INTERACTION IN PROGRAM VISUALIZATION: DESIGNING EFFECTIVE VISUALIZATION TOOLS FOR REDUCING USER'S COGNITIVE EFFORT

by Mihail-Eduard Tudoreanu

ADVISOR: Profs. Gruia-Catalin Roman and Eileen Kraemer

August, 2002
Saint Louis, Missouri

Program visualization has the potential to be an important tool for people who seek to observe and understand the behavior of a running computation. This thesis focuses on alleviating barriers to the realization of this potential that pertain to the design of a visualization system and to insufficient knowledge about how people take advantage of program visualizations. Our major contribution is the design of a visualization approach capable of improving user's performance through the use of economy of information and tasks. We present evidence from our empirical studies that this type of economy promotes animations capable of significantly improving people's understanding of the computation. We apply this knowledge to develop a system for creating application-specific visualizations solely through interactions with program visualizations and textual views of the computation, thus promoting economy of interaction. The system is built around the principle that animation viewers are also the creators of animations and systematically refine the visualizations to suit their momentary goal.

to my family

# Contents

# List of Tables

# List of Figures

# Acknowledgments

This thesis is based upon work supported by the National Science Foundation under Grant No. NSF-CCR 9619831.

My research would have not been possible without the support of my advisors Eileen Kraemer and Catalin Roman, and the help from Del Hart. Many thanks to Maggie and Raluca for their patience, especially during late night hours.

Mihail-Eduard Tudoreanu

*Washington University in Saint Louis*
*August 2002*

# Preface

The work presented in this thesis is part of a larger endeavour that began at Washington University and later extended to the University of Georgia and University of Alabama Huntsville. The project was funded by NSF and Boeing. The purpose of the endeavour was to develop an approach for monitoring distributed computations that permits selective and interactive exploration of the computation's space. This entails collecting the relevant portion of the local states of the processes in the computation, and assembling a global state that satisfies certain constraints. The stream of global states produced by the monitoring system is intended to be made available for end-users to analyze. One method of analyzing the computation is through the observation of animated graphics. This thesis elaborates on the techniques through which an end-user may visually explore the state of the computation by creating and refining custom animated views of the program.

# Chapter 1

# Introduction

Program visualization holds great potential for conveying information about the state and behavior of a running program. However, barriers exist to the realization of this potential. These barriers include elements of the design of a visualization system that reduce user's performance, insufficient knowledge about how people take advantage of program visualizations, and difficult or unreliable extraction of data from a live computation. The focus of this thesis is on the elements of the design, but we also rely on and present our experimental studies which add to the knowledge about how people use visualizations. Our major contribution is the design of a visualization approach capable of improving user's performance through the use of economy of information and tasks. This type of economy promotes animations that significantly improve people's understanding of the computation. We apply this knowledge to develop a system for creating application-specific visualizations solely through interactions with program visualizations and textual views of the computation, thus promoting economy of interaction. The system is built around the principle that animation viewers are also the creators of animations and systematically refine the visualizations to suit their momentary goal.

## 1.1 Program Visualization

Program visualization, also termed algorithm animation[1], is a form of presenting the execution of a running computation through the use of animated graphical displays.

[1]In some publications, program visualization is considered to denote a research area that is either different from or inclusive of algorithm animation. For the purpose of this dissertation, the two terms are regarded as synonyms.

Such animations visually encode and present how data is processed inside a running program. For example, an animation of a sorting algorithm might show step by step how the elements of an array are swapped pairwise until the array is finally ordered. Program visualization is believed to be conducive to the understanding of algorithms and programs [84], and consequently, has been applied to solving problems that require a solid insight into the complexities and abstractions of computations [71]. For example, graphical displays have been used for monitoring and debugging of programs [60], for learning about algorithms[87, 4], for communication within a group of programmers[46], and for the optimization of code [41].

Current trends in software engineering and visualization suggest that a demand for program understanding techniques exists and that graphics have the potential to supply significant help. Thus, algorithm animation may become increasingly important for all categories of users, ranging from novices to seasoned programmers. Current and future software packages demand new and improved techniques, beyond traditional software engineering and formal analysis approaches, to cope with the sheer complexity and size of code while maintaining an acceptable level of reliability (see [59] for a mainstream media report on the challenge of developing reliable software). The behavior of software systems is very often unpredictable and strays from what developers intended or assumed. Hence, there is a need to make software analysts more aware and knowledgeable of both the programs they write and the pre-existing, unfamiliar components included in their product. Program visualization can play a role in bridging the gap between human reasoning and computational processes from the early stages of training when graphics can be used to give students extra insight, to production settings when visual aids can be used for programmers and administrators who seek to manage large software, and even to end-users who must become more and more aware of the high-level mechanics of the software they employ.

On the supply side, there are encouraging signs that visualization is an effective conveyor of information. Scientific visualization is gaining ground and popularity among researchers, which can be seen for example in the use of visualization for analyzing the results of detailed simulations of various natural phenomena on supercomputers [5]. Information visualization, which focuses on the presentation of more abstract data than does scientific visualization, exhibits an upward trend in significance, providing help in domains such as marketing and stock market analysis [3], and being envisioned as the future of data analysis in the intelligence community (see call for proposal [47]).

The benefits of graphics as a presentation medium are two-fold, as Bertin notes [15] in his book “Semiology of Graphics”, as a tool for conveying large amounts of information and as an environment for solving logical problems. Scaife and Rogers [77] view graphical representations as a form of external cognition, in which mental internal representations are offloaded onto an external medium to relieve the cognitive burden and speed up processing. A stylized representation of the external cognition principle is given in Figure 1.1, in which the visualization extends the cognitive resources of the user. Note that these considerations are applicable to graphics in general, and it is unclear whether users of program visualization take advantage of animation in this manner.

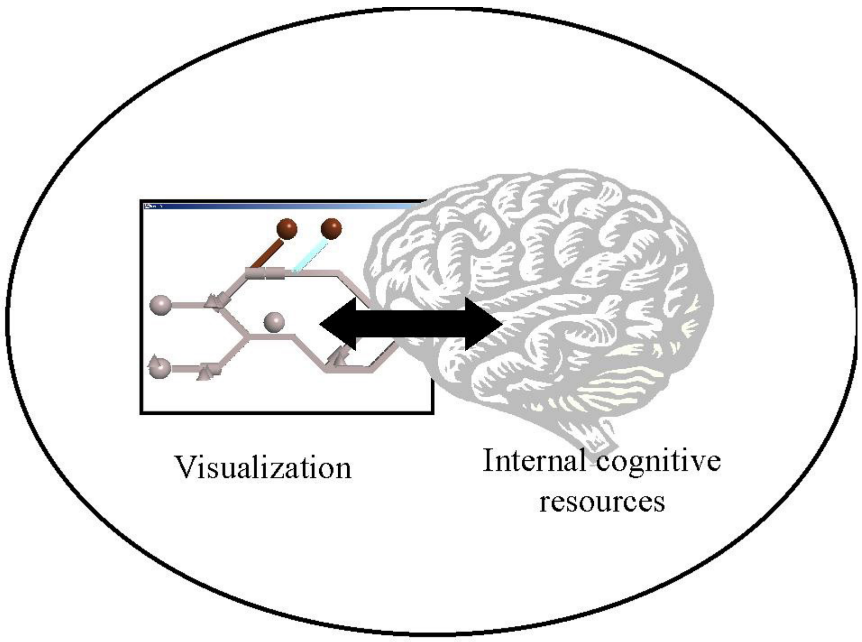


Figure 1.1: Visualizations are considered to extend cognitive resources because they are a form of external cognition. More resources improve human performance in performing a task.

## 1.2 Barriers to Exploiting the Potential of Algorithm Animation

Despite program visualization's expected potential and the need for richer tools to connect people with computations, animation is largely underutilized [51]. Since the introduction of program visualization tools, program visualization technology has undergone significant advances, though not enough to have shown consistent advantage to visualization users over non-users. One barrier to effective program visualization is a poor understanding of how animations improve user's performance and, more precisely, of some of the factors that may lead to effective algorithm views. This might explain the mixed results of empirical studies into the practical importance of algorithm animation [19, 39, 50, 54, 83]. Consequently, another barrier is that program visualization tools include only a few elements that enhance the effectiveness of animation.

This dissertation concentrates on barriers to increased performance for visualization users that relate to cognitive load. More precisely,

**A.** on the adverse effect of increased cognitive load on the benefits provided by animations, and

**B.** on the barriers to promoting reduced cognitive load in program visualization tools. Animation tools may increase the cognitive load in the following instances:

   **1.** when the animations are created and refined;

   **2.** when the user needs support for a number of unrelated or unforeseen tasks; or

   **3.** when the user is becoming familiar with and applying the visualization.

**A.** Algorithm animations and high cognitive load may have opposite effects on the mental resources. On one hand, animations have the potential to free internal cognitive resources. On the other hand, increased cognitive load uses more of the same resources. Thus, program visualization and increased cognitive load may offset each other's effects from the point of view of internal resource usage. In other words, a situation in which both graphics and high cognitive load exist resembles a situation when neither is present. Hence, a visualization environment that requires users to handle additional information and tasks, which increases cognitive load, offers similar performance advantages to that of a user who has no visualization at all. The problem is aggravated when the additional information and tasks are not related to the observed computation, leaving fewer resources for analyzing the program.

**B.1.** The creation and refinement of an application-specific visualization often relies on indirect structures, such as specialized languages with their compilers, or dedicated graphics packages and editors [1, 16, 74, 88]. These indirect structures create a barrier by consuming the cognitive resources that the user devotes to understanding and handling the structures and tools. A schematic of such an interactive visualization process is shown in Figure 1.2. A modification of the animation is often performed by switching through multiple environments. It often happens that changes involve some guesswork, as they might not be apparent immediately due to the delay in updating the visualization or to the placement of the program view out of sight, obscured by the extra tools. The user must store the associations between tools in the working memory and must internally process the structure when guessing whether a change results in the desired view.

**B.2.** Pre-defined animations are limited in the range of tasks with which they can efficiently assist the user. Although such views are easy to obtain and quite helpful for their intended tasks, they can quickly become a burden as the task shifts. The user's task is likely to shift due to changes in the user's goals or understanding of the computation. An attempt to apply a pre-defined visualization to a different task, even if derived from the original task, might force the user to devote substantial cognitive effort to associating the information in the visualization to the information required by the task. This might have the undesirable effect of increasing the data that has to be mentally stored and might result in an increase in the time and cognitive burden to perform the task. In such a situation, the task of understanding the program is shifted to understanding and applying the relations between the available visualization and the problem to be solved. The mere existence of an animation does not lead to reduced cognitive load.

**B.3.** Algorithm animations might employ numerous concepts and representations to present the abstract data and mechanics of a computation. To accomplish any task, the user must expend additional effort for understanding what the animation is presenting during its execution. The user must learn and recall the visualization syntax, the mapping between changing entities of the computation and the dynamic graphical elements. If the syntax is unknown, the visualization is perceived by the

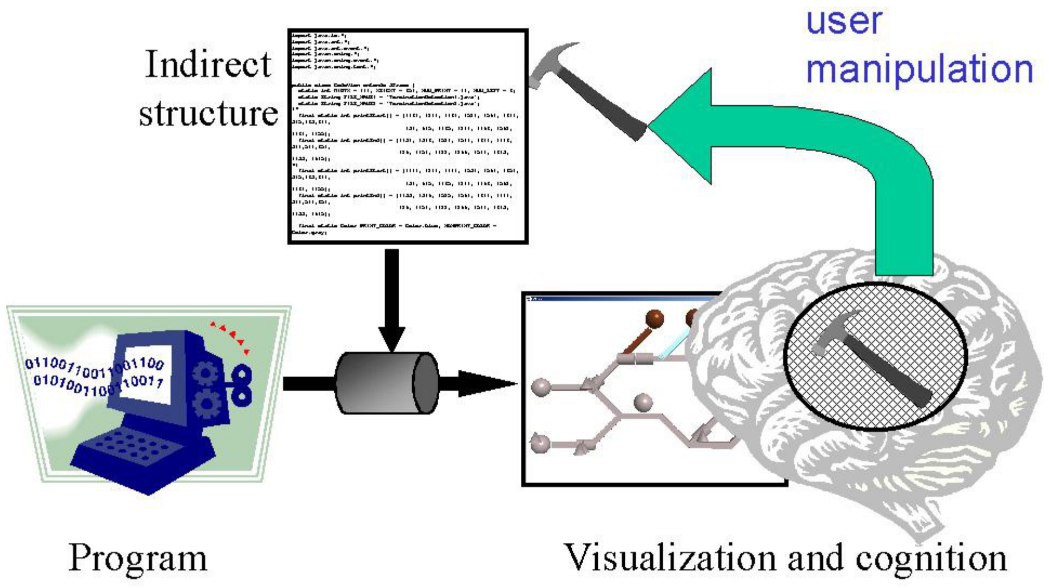


Figure 1.2: The running program can be observed through a visualization, a form of external cognition. Existing algorithm visualization tools often require the user to create and refine indirect structures, which in turn modify the final animation. To accomplish this, the user must allocate cognitive resources to the indirect structures (hashed area), reducing the resources available for processing the program and visualization. More than one type of indirection may exist between the program and visualization.

user as a set of randomly changing shapes and colors. Unfortunately, in classical program animation, no provisions are made for reducing the time and cognitive resources diverted toward learning the syntax.

## 1.3 Contributions: Removing and Reducing Barriers

Our research is aimed at creating an improved program visualization tool that is rooted in theoretical and empirical results. The most important design goal of our tool is toward increasing the performance of animation users as compared to those to whom visualization has not been made available. The secondary goal is to allow users to build a wide range of animations for any arbitrary computation. The target audience includes users who observe the manner in which an already-written program manipulates its variables and data structures. Our tool is intended for monitoring and detecting errors at programming language and algorithmic levels, and for communicating the behavior of software components among teams of programmers.

The unifying theme of our approach is the notion of *cognitive economy*, the application of the concept of external cognition to program visualization. *Cognitive economy* seeks to reduce both the amount of "extraneous" information that the user must manipulate and store in the working memory, and the number and complexity of tasks related solely to the animation. "Extraneous" information includes any information not directly pertaining to the observed computation such as the code that defines the animation. The goal of cognitive economy is to allow mental resources to be allocated to understanding the computation rather than to "extraneous" information and tasks. On the positive side, the balance of allocation is improved (because of external cognition) by graphical representations that can be customized to various computations and tasks. On the negative side, the balance can become unfavorable to the understanding of computations as "extraneous" tools, including animations, are added into the user's environment, and as the user must allocate more resources to managing the extra tools.

Cognitive economy is supported by an empirical investigation in which we found that, under conditions of improved cognitive economy, animations promoted significantly more accurate problem-solving by the user. Poor cognitive economy, on the other hand, is among the factors that may cancel the benefits of the visualization, leading to the situation in which users and non-users of graphics register similar performance.

We developed an algorithm animation tool that is designed for cognitive economy and eliminates the need for indirect representations to create visualizations. Cognitive economy is accomplished by simply eliminating information and tasks related to the indirect structures (compare the new approach in Figure 1.3 with the traditional one in Figure 1.2). Since cognitive economy is a factor that promotes effective program visualization, our tool has the potential to significantly help its users. Our tool is designed to be used with a stream of program states that are collected from a running computation with a system like PathFinder [40]. Users can create and refine animations solely via interaction with graphical and textual representations of the computation.

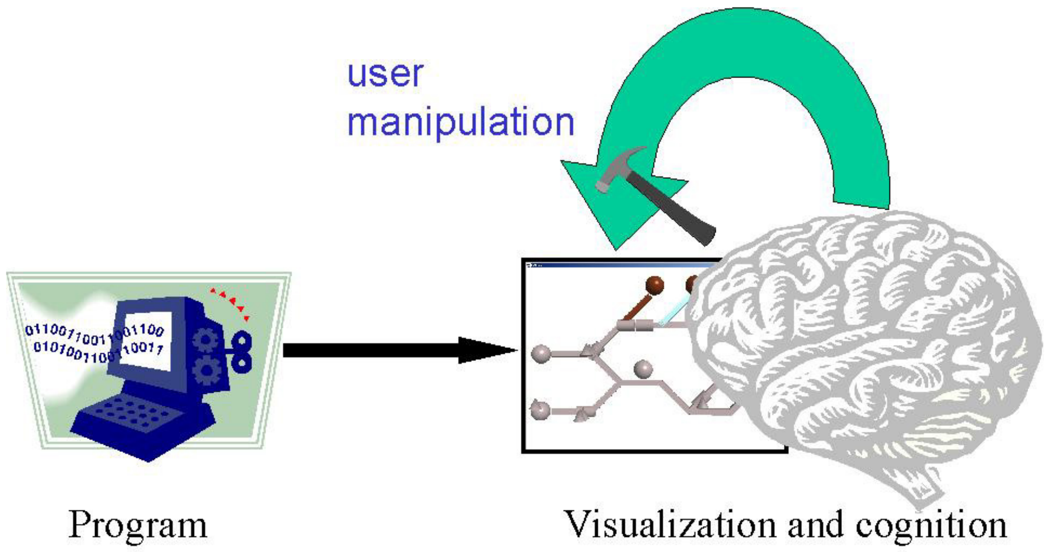


Figure 1.3: The running program can be observed through a visualization. The visualization can be customized directly via user interactions.

Both practical and theoretical technical contributions make the customization of visualizations possible in the absence of indirect structures and with cognitive economy. The technical contributions listed below address the interactions with visualizations on a spectrum from a high, conceptual level, to the creation of dedicated widgets for continuous modification of visualizations, and to very basic interaction with physical and software input devices. The conceptual-level interaction is supported by an algorithm that involves the computer in the automatic handling of lower-level graphical details. These contributions include:

- **A model of interaction with algorithm animations.** The model is based on the content of the visualization, and it is independent both of the particular graphical details employed in the animation and of the features of the graphical user interface. The focus is on discrete operations, on how the animation of a computation can be changed in discrete steps. Cognitive economy is further supported by the content-oriented model because the user is not required to specify the graphical details of the animation, which can be quite difficult and tedious. Additionally, the operations, once learned, can be applied uniformly across animations and computations.

- **Automatic presentation of running programs.** Algorithms for automation presentation can create custom graphical representations for the information content specified and manipulated by the user. Automatic presentation is employed in conjunction with content-based manipulation of the animations, but can also be useful for starting a visualization session and for novices with little experience with computer graphics. The user can still operate at the graphical level if desired, and the technique is flexible enough to include the graphical preferences of the user in the final animation. The main advantage is that some of the user's tasks are transfered to and handled by the computer.

- **Interactive legends.** These legends, an extension of those found on a road map, allow continuous adjustment of an animation. They are an integral part of the visualization serving both display and input functions. As a display, these legends explicitly convey the visualization syntax, the mapping between program entities and graphical features. The placement of visualization syntax in plain sight reduces or eliminates the need to train the user for the visualization. As an input, legends permit the adjustment of the manner in which the program is encoded as graphics, and provide an intuitive mechanism for refining the data to graphics mapping. In serving these functions, legends are important for cognitive economy, for removing barriers in understanding the animation itself, and for complementing the interaction capabilities of discrete operations.

- **Spatial transformation of input.** We conducted an empirical study to analyze the accuracy and speed of interaction when user's gestures performed on an input device appear "distorted" on the screen. The results of the study enabled us to develop a prediction model that can be employed to evaluate the speed of a user interface that relies on continuous input devices. The model is a theoretical contribution, which can be used in practice to choose an interface with visualizations that reduces the difficulty of basic continuous interaction, such as moving elements in the visualization. Accurate and fast basic interaction has the potential to reduce the difficulty of the tasks performed on the screen. Spatial transformation of user's gestures is quite typical and occurs because the user cannot access objects directly on the screen (except through touchscreens and electronic pens), but only via input devices such as physical mice or software scrollbars. The experiment investigated affine transformations, a combination of rotation, translation and scale.

## 1.4 Dissertation Overview

The next chapter is an overview of the current state of the program visualization field. First, an overview of the existing program visualization approaches is given, followed by a brief description of important empirical studies. An expanded list of barriers, some of which are beyond the scope of this thesis, is given in "Why Johnny Won't Visualize," which takes a closer look at possible factors that may be responsible for preventing the wide-spread use of program animations

Chapter 3 presents two controlled experiments, in which we found that algorithm animation under conditions of cognitive economy can significantly improve user performance.

An abstract model for the content of a program visualization is given in Chapter 4. Later in that chapter, we identify general characteristics of the user operations on animated graphics, and present an example of an operation set and visualization session. The abstract model is also used by the automatic presentation algorithm of Chapter 5. User interactions have the power to modify this abstract model of animations, and automatic presentation provides the capability to build visualizations that reflect the changes produced by the user at the high-level of the model.

Legends are presented in Chapter 6. They complete the discrete operations of Chapter 4 and provide a way to explain the decisions of the automatic presentation algorithm of Chapter 5. We detail the visual design of interactive legends and describe the actions available to the user. The chapter also introduces animated legends.

Chapter 7 describes an experiment that assessed what happens when the user input is spatially transformed through a combination of translation, rotation and scale. The procedure and results of the study are detailed, and a statistical model of the interaction speed as a function of spatial transformation is derived. The chapter contains an example of how the prediction model can be applied to choose between two basic interfaces to three-dimensional visualizations.

Finally, in Chapter 8, we summarize the contributions of the thesis and present future work.

# Chapter 2

# Program Visualization Technology

## 2.1 Overview

In this chapter, we provide a picture of the current state of program visualization research. We begin by elaborating on developments and approaches that permit users to obtain a program animation. We explicitly state the differences between these approaches and our approach. Next, we present the results and conclusions of experimental work that concentrates on the interaction between animations and users. Finally, we compile a list of open research problems.

## 2.2 Obtaining Animations

A wide range of tools, each built on different principles and with its own strengths and weaknesses, can be employed by a user who seeks to observe an animation. A number of comprehensive taxonomies of program visualization have been compiled by Myers [61], Price *et al.* [66], and Roman and Cox [72]. However, from the perspective of our research, the primary feature of interest in classifying program visualization systems is their reliance on the user manipulation of indirect structures. The *indirect group* contains systems in which the visualization is defined by writing a piece of code or by constructing a visual program. The *direct group* allows the users to obtain a view of the observed computation by manipulating the animation directly. For completeness, we consider that pre-defined program visualizations are part of the direct group because such animations do not require the construction or manipulation of any structure, and that *animation authoring* belongs to the indirect group. In animation authoring systems, there is no running computation to be monitored, and the user focuses on the animation itself rather than on an existing program. Authored animations may or may not present an exact computational process. The main reason for classifying these approaches under indirect structures is the different paradigm the animation authoring entails. Moreover, it happens that most authoring tools expose users to coding.

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

### 2.2.1 Approaches that Involve Indirect Structures

#### Monitoring Running Programs

A diverse body of research has produced systems and tools that are able to create graphical representations of running programs. We present both systems that are specially designed for program visualization and techniques that were intended for other types of visualization but have been employed for the creation of program visualizations.

The most versatile program visualization approaches are powerful enough to produce virtually any animation. However, they require the user to have programming skills, as the visualizations have to be coded, and to be knowledgeable of the animation packages provided by the tool. Marc Brown's BALSA [16], one of the earliest algorithm visualization tools, is based on the *interesting event* paradigm. The observed program is annotated at certain interesting points with calls to user defined animation functions. The main role of these calls is to change the picture displayed on the screen, with the end result that interesting events in the computation are visually represented. Stasko in Tango [86] introduced the *path transition paradigm* which was later adopted by Stasko and Kraemer in Polka [88], a visualization tool for parallel and distributed computations. The *path transition paradigm* is also based on annotations of the observed program and assumes that multiple events may be called in parallel from the processes of the computation. These calls, which are user-defined, specify the path of various graphical objects on the screen and the transformations they are to undergo, as opposed to actually moving or changing them. The system then animates these objects along the path specified by the user and synchronizes their movement/change. Samba [81] allows the use of a simpler, scripting language to access the power of Polka. Samba was designed as a simpler method of generating animations. One Samba application was to support students in building their own program visualizations [87], which is similar to our goal of empowering viewers to become creators of animations.

A *declarative approach* to program visualization emerged in Pavane [74], designed by Roman and Cox. Unlike the other approaches in which the user defines how the animation is changed when the execution reaches an interesting event, the declarative paradigm requires the user to specify a mapping from the program state to a graphical representation. When the program state changes, the mapping is re-applied by the computer to update the picture. Pavane supports smooth animations of the transitions from one frame to the next. The mapping is specified by the user as a program, which is written in a programming language that allows the definition of arbitrarily complex functions [23]. Demetrescu and Finocchi developed Leonardo [25], which is also based on the declarative paradigm, but uses a logical language to create and modify visualizations. One advantage of the logic-based approach is that the visualization can be modified without recompiling the logical program that defines it. A virtual CPU is integrated into the visualization system, which allows the underlying program to be executed in reverse. Leonardo also includes a declarative method [28] for specifying the path of timing of the transitions from one frame to the next.

Visual programming can replace the traditional textual code that defines the visualization. AVS [1], a commercial product developed by Advanced Visual Systems Inc., has been used to construct program visualizations [27]. It is interesting that AVS was designed for scientific visualization and was adapted to the building of program animations. With AVS' visual programming environment, the picture is produced by specifying a data flow graph. The main drawback of such an adapted system is the inability of the user to specify a wide range of *animation behaviors* solely through direct interaction. The behavior of the animation may need to be purposely modified from the behavior of the observed program in order to maintain information from previous states of the computation and combine it with the data from the current state. This approach is helpful for tasks that require the user to understand temporal causal relationships in the computation. A further drawback of AVS is that the user must understand and learn the functions of various modules, which are typically coded textual programs, and must connect and patch these modules together by making use of visual programming.

Automatic presentation tools are perhaps the easiest to use, as they can typically produce a graphical representation of the information in which the user is interested. Their main advantage is that, unlike previously presented systems, the user has only limited exposure to the graphical details and can operate at the information level. The two automatic presentation tools discussed here, Boz [20] and VisAD [43] still require the user to create simple pieces of code (a more comprehensive list of automatic presentation techniques and their relation to our work is presented in Section 2.2.2). Casner in Boz [20] is able to generate a picture of a database that is also customized for the user's task. The specification of the task, however, is provided as a piece of code. VisAD [43] was developed by Hibbard *et al.* to visualize the data types that appear in an algorithm. VisAD requires users to specify, in a textual manner, the correspondence between scalar data types and graphical attributes. Based on this mapping, the algorithm can present composite data types.

#### Animation Authoring

Animation authoring, as defined in this thesis, refers to approaches for the creation of animations that focus on data structures used in algorithms and may illustrate computational steps, but that are not used to monitor an existing program. They are employed in situations that entail the explanation of an algorithm and its variations. The dominant application is the instruction of students in algorithm and data structures courses, where both normal operations and problems are demonstrated by an instructor.

The goal of animation authoring is radically different from the goal of the rest of program visualization research. For the former, the animation can be regarded as the final product, while for the latter, a computation is the final product and the animation just a tool in shaping the product. "Traditional" program animation can supply the animations needed in authoring tasks, but entails a high overhead because the creator must implement the operations in a programming language, then collect the data from the running program, and then build the visualization. Animation authoring systems presented below attempt to combine and, when possible, to skip some of these steps.

Roessling developed a direct interaction system named Animal [69] which supports the drawing of each frame of an animation and contains a number of transition effects. Animal can be used to build general animations, not necessarily related to algorithms. Special attention, however, is given to graphical representations of pointers and linked data structures, as well as to the display of code. Animal is not coupled to any computation, which provides flexibility to convey erroneous algorithm steps, but makes it more challenging to create an exact rendering of an algorithm. The visualization can be played both forward and backwards. AnimalScript [70], which uses the same engine as the direct manipulation tool, is a simple and extensible language with which animations can be defined via textual scripts and some limited Java coding.

Hundhausen and Douglas developed a language SALSA and an environment ALVIS [44] for animation authoring. Like our approach, their work is rooted in empirical results (see [46, 53]) rather than solely on the authors' intuition. The system provides reversible execution and low fidelity animations. This kind of animations is based on cut-outs sketched by the user. Spatial logic is employed to assemble cut-outs, which seems to support our position that computer graphics (based on Cartesian coordinates) is too complicated to be specified directly by the user. Although ALVIS provides a friendly interface for building and editing SALSA programs, the user typically must handle SALSA code. No other code is necessary for the definition of an underlying algorithm.

SKA [38] is another authoring system designed by Taylor. It supports cloning of data structures and parallel tracing of the execution of the same algorithm on the cloned structures. Data structures can be modified at any time by the user. SKA actually uses an underlying Java program that encodes the steps of the algorithm to be animated.

Animations can also be authored with common presentation tools such as HyperCard$^{TM}$ from Apple or PowerPoint$^{TM}$ from Microsoft. Each frame of the animation and transitions can be defined in a card or a slide respectively.

### 2.2.2 Approaches Without Indirect Structures

This section is organized in three subsection in the order of the capabilities of these systems. The order is subjective and takes into account the number of tasks for which the user can customize the visualization. The tasks that can be covered by a tool is closely related to the general flexibility of the tool in supporting a wide range of animations and computations.

### Direct Interaction with Visualizations

Direct manipulation and visual definition of visualizations was explored as a means of eliminating the coding requirement. Mukherjea and Stasko integrated into a debugger a program visualization tool named Lens [60]. The user can click on statements of the observed program and add visualization requests mainly by filling out various forms. A graphics editor is integrated in Lens to produce the graphical objects of the visualization. The values of graphical attributes can be smoothly animated by choosing among pre-defined transitions. Lens is not designed to construct a large range of animations, but instead is designed to permit easy creation of most used types of animations. Lens is further restricted by its dependency on the control structures of the program such as `if` or `while` blocks. Steve Reiss' BEE/HIVE [68] is another system for creating program visualizations in a direct manipulation environment. It consists of a visual query interface that can be used to process the data collected from a running program, and a number of visualization templates to convey the processed information. Although the query mechanism can help explore relations in the program state, the final appearance of the animation must rely on a pre-existing template.

### Automatic presentation

Automatic presentation techniques appeared in the context of structured, static information such as database systems. An early approach, APT [58] by Mackinlay, was able to create a picture to present relations from a database. The algorithm ensures that all and only the data chosen by the user is depicted, and that the relations between values are shown by an appropriate visual feature such as color, shape or position. SAGE [76], created by Roth *et al.*, relies on a richer characterization of information than APT and is capable of creating visualizations that can present data more clearly.

The University of Washington illustrating compiler (UWPI) [42] can automatically create visualizations for simple programs. UWPI recognizes a set of abstract data structures and has a built-in representation for these data structures. The visualization for each state is generated through a simple composition of these pre-defined graphical representations. UWPI is limited in the programs it can correctly analyze (it only recognizes a subset of Pascal and it may fail to recognize the type of data for programs with bugs), and in the appearance of the final picture.

Automatic presentation techniques, although they require little effort from the user, are subject to a number of limitations which include:

**Continuity of animations** Some of these systems (APT [58], SAGE [76], and Boz [20]) were designed for static information, and for dynamic information may produce animations that are hard to understand. Specifically, the problem is that there is no guarantee that an automatic presentation tool will maintain the same visual encoding for similar information. As program entities appear and disappear, the algorithm might decide on a different “best representation” of the same piece of data in each frame of the animation. This may happen in an attempt to accommodate additional program entities or to simplify the picture when entities disappear.

**Animation behavior** The tools that handle dynamic data (VisAD [43], and UWPI [42]) support only a limited range of animation behaviors. Namely, only the information from the current state of the computation is displayed.

**Types of program entities** No automatic presentation technique, except UWPI, distinguishes between the type of program entities and properties (relations) of those entities. The state of the program, however, is comprised of various types of entities, which represent very different concepts such as queues, trees or scalar integers. Nonetheless, entities of different types might have properties in common. A good visualization, in our opinion, must convey the information in a manner that allows the viewer to easily recognize classes of entities, even when properties that span across several classes are also shown. A strong visual distinction should be made between types of program entities.

### Pre-defined views

Pre-designed visualizations have the main advantage that they are simple to use and quick to employ. Additionally, such visualizations typically benefit from the insight of an expert into the problem for which the animation was created. Pre-defined visualizations tend to be created for common tasks that occur in a certain programming language, type of computation, or class of problems. Such views might increase cognitive load when applied to solve more specific problems or tasks outside the intended scope of the visualization.

Tools for enhancing the understanding of programs written in a specific language have been developed for Java $^{\text{TM}}$ and Prolog as well as other languages. Jinsight [49] and VisiVue [2] are commercial tools that present the execution of Java programs. Jinsight, built by IBM, offers a number of perspectives such as program slices, execution of threads, and memory allocation. Jinsight has interactive features that allow the user to organize the wealth of data collected from a program. VisiVue shows a diagrammatic view of the objects and the references between objects. Eisenstadt and Brayshaw developed the Transparent Prolog Machine (TPM) [30] with the purpose of providing insight into the inner workings of a Prolog program. TPM shows the backtracking process of a Prolog execution through animated trees and can provide both coarse-grained and fine-grained level of detail.

A specific visualization for distributed computations that employ the PVM [91] library is provided by PVaniM, developed by Topol *et al.* [93]. The system provides a number of visualization that show message passing patterns and statistics. Some of the views were inspired by Heath and Etheridge's ParaGraph [41], a pre-defined system dedicated to the analysis of the performance of parallel programs.

An example of a system created for a particular class of problems is COMIND [62]. It was designed for understanding and guiding the search algorithms in a constraint satisfaction problem [33]. A number of views present how the search explores the space of possible solutions and how constraints are considered. It is assumed that, based on these animations, a user can customize the search algorithm to be more efficient for a given set of constraints.

## 2.3 Evaluations of Program Visualization

Empirical studies of program animations have concentrated on two main directions. In the first direction, researchers answered the question of whether program visualization provides benefits to users. In the second direction, studies analyzed the manner in which participants in the experiment create or make use of program visualizations. The studies we conducted are along the first direction and differ from previous work in the user's task (problem-solving as opposed to learning) and theoretical background (cognitive economy). A good survey of program visualization studies can be found in Hundhausen *et al.* [45].

### 2.3.1 Measures of the Effectiveness of Animations

Although a reasonable number of techniques exist for obtaining program animations as described above, little is know about the practical effectiveness of such animations. Previous user studies of program visualizations have had mixed results. An early study by Stasko *et al.* [83] showed a non-significant advantage on a post-test for participants using an XTango visualization and text description of a pairing heap algorithm compared to a group which had only the text.

A more comprehensive study by Lawrence *et al.* [54] looked at students who attended a lecture, then constructed data sets individually while doing a related algorithm visualization lab exercise. These students performed significantly better on a post-test than those who just attended the lecture. Students who were given a data set did marginally, though not significantly, better than those who only attended the lecture. However, among students who only attended a lecture, those who viewed algorithm visualizations performed marginally worse than those who viewed slides containing static depictions of the algorithm's execution.

Hansen *et al.* [39] found that a group of students who used a hypermedia environment that included algorithm visualizations as well as pseudocode, text, and illustrations performed significantly better on a post-test than another group that used a textbook. However, it was not clear which aspects of the environment contributed to that result.

Byrne *et al.* [19] conducted a series of experiments to measure the effect of making predictions about the next state of an algorithm, with and without visualization. A significant result was obtained for the use of visualization on the conceptual portion of a post-test for a basic graph search algorithm, but no difference was found for a more complex binomial heap algorithm.

### 2.3.2 Observational Studies

To investigate the use of algorithm visualization and other media as a problem-solving resource, Kehoe *et al.* [50] conducted an in-depth observational study of students solving a set of binomial-heap questions, given a textbook section, pseudocode, and either algorithm visualizations or a series of static figures captured from the algorithm visualizations. No time limits were set. They found that students using the visualizations spent more time comparing and moving between the media types, especially the pseudocode and the visualization, and that the visualization group performed significantly better on the questions.

Hundhausen conducted a pair of ethnographic field studies [46] to observe how students in an algorithm course construct visualizations. Various field techniques were employed. One study required students to use Samba [87] to build visualizations of the algorithm learned in class. The animations were then presented to other students. The students spent substantial time both individually and in discussions on low level coding of graphics. The other study let students use office and art supplies to build low fidelity visualizations. These students spent less time on creating visualizations and concentrated more on the underlying algorithm. The students also actively marked up their paper visualizations and often backtracked the presentation to a previous point. Note that this field technique is different than a controlled experiment.

## 2.4 Why Johnny Won't Visualize

This section is named in the same style as a series of famous articles, such as "Why Johnny Won't Read", that also addressed possible problems. Many potential users of program visualization have a strong intuitive belief that visualization is a valuable tool for communicating information about the state and behavior of programs. Yet, in practice, the use of visualization is less pervasive than the notion that it is useful. Several factors may contribute to this apparent disconnect between belief in the usefulness of visualization and the extent to which visualization is actually employed. The factors are related either to the user or the visualization technology.

### 2.4.1 User-Related Factors

Program visualization may be underutilized because of user misconceptions, lack of trust, or perceptual limitations. Furthermore, the benefits provided by animations are not very obvious, and there are few clear guidelines to obtain them.

The manner in which users regard visualizations often prevent users from employing them. One problem is a lack of trust in the stability of the visualization system, which might make users suspect that an application error is instead caused by the visualization. Another problem is the preconception/misconception among potential users that they must understand the program before beginning to visualize it. Finally, users may have doubts about the relative benefits of visualization given the perceived effort.

The actual benefits of algorithm animation, independent of users' opinion, are not consistently apparent. The studies presented in the previous section found only infrequently an improvement for visualization users. Further, the exact aspect of the user performance that is enhanced by program visualization is still not obvious.

Limitations of human perception, especially those particular to algorithm animation, have the potential to make visualizations difficult or impossible to observe and follow by users. Little is known about a number of perceptual issues including:

- the number of concurrently moving objects that can be tracked and related to the underlying program by a typical viewer;
- the instances in which the user can map the movement of an object or objects (an animated action) to the operation the visualization is trying to portray;
- the instances in which the user can assign the correct meaning to a sequence of actions;
- the effect of the complexity of an algorithm on the ability to track and interpret individual or group actions;
- the ability of visual and/or audio cues (e.g., arrows, sounds) to signal events we want to highlight (e.g., exchange completed, end of pass, etc.);
- the manner in which the visibility of representations affect the viewer's ability to reason about the algorithm.

A lack of proven guidelines for good animations makes the creation and refinement of program views an art more than a science. This produces a mismatch with the target audience, primarily engineers and scientists. The guidelines should include perceptual knowledge, hints of the general format of a visualization based on the current type of task to be performed, and suggestions for organizing the information of the program in order to solve a type of problems. Steps toward creating guidelines, such as Roman's proposal to visualize formal properties (invariant, progress), have been few and not thoroughly tested in practice.

### 2.4.2 Technology-Related Factors

The wide-spread use of program animation may be hindered by problems in connecting a visualization to a running program, for creating (authoring) animations, and for navigating through the computation via modifications of the program views.

Linking a visualization system to a live computation is sometimes challenging. The requirements may include annotation of the original code of the computation, and data collection/assembly systems that are flexible enough to allow varying levels of granularity. Sometimes, it is only possible to monitor a restricted class of programs or executables written in certain programming languages. Further complicating the process is the use of off-the-shelf and third-party modules that either do not support monitoring or support other monitoring paradigms. A similar problem arises in the use of distributed systems that do not fall entirely under the administrative domain of the visualization user.

The creation of an animation may be considered difficult and time-consuming by the user. As already pointed out, the user may be required to handle indirect structures and multiple unrelated tools, that increase mental effort and time, and provide poor feedback to user changes.

In addition to creation, users and designers of visualizations must perform the tasks of navigation and refinement, often as a result of the insight gained from previous graphical representations. These otherwise natural tasks are often difficult and tedious, or may be limited by poor interaction mechanisms in the visualization tool. In addition, often the factors impeding easy creation also negatively affect the refinement of visualizations.

## 2.5 Concluding Remarks

Program visualization technology and, to a smaller degree, knowledge about the use of such animations is becoming increasingly advanced. Experience in both subfields can be used to develop more efficient visualization systems, yet does not seem to be enough to make animations commonly utilized. Current program visualization approaches either require indirect structures or have limited capabilities. At the same time, little is known about the factors that may result in an increased performance for animation users. Our work focuses on cognitive economy as such a factor, and on the development of a program visualization tool that supports cognitive economy.

# Chapter 3

# Empirical Studies of Program Visualization

## 3.1 Overview

The knowledge of factors that promote beneficial program visualizations is still in its infancy. Indeed, the question of whether program visualization is beneficial in helping users to perform certain task has received mixed results. In this chapter, we present two studies designed to answer the question "Does visualization promote understanding of distributed computations?" One study failed to show any significant benefit when visualizations were used to answer questions about distributed computations. The other study, in which both the animation and the testing environment were designed in light of the concept of cognitive economy, showed a clear benefit for visualization users. Performance of the users who did not have access to visualizations was comparable across the studies, providing us with the opportunity to compare these studies and to postulate possible causes of improved performance of visualization users.

## 3.2 Introduction

Despite the high expectations of the visualization community and a warm reception from users, program visualization has previously failed to demonstrate its usefulness in most empirical studies. Overall, the results of the studies have been mixed, with slightly better results recorded in situations in which algorithm animation was employed, and with only a few experiments that reported a significant difference in performance between visualization and non-visualization settings [19, 39, 45, 50, 54, 83]. The variation in the results shows, at the least, that evaluation of algorithm animation is complex and that not enough is yet known about the factors that affect the effectiveness of program visualization. Such factors, when properly identified and understood, may guide the design of more beneficial animations and animation tools.

In the process of conducting and comparing two controlled experiments, we identified several factors that can critically influence the effectiveness of program visualization. One of the studies found that visualization significantly improves user's performance, while the other did not show any improvement for animation users. The majority of the factors that were modified from one study to the other fall under the concept of cognitive economy. Cognitive economy seeks to reduce both the amount of information handled by the user and to eliminate or reduce the user tasks that do not pertain to the observed computation.

The two studies are similar in that they employed the same computation, user profile and type of tasks to be performed by the participants. Moreover, about half of the tasks, which consisted of answering questions, were identical in the two studies.

The methodology of both experiments allowed users to examine the visualization and any other learning material while completing their task (answering questions). Our studies captured the situation in which users have a problem to solve, may employ any available materials, and are not focused on remembering the computation afterward. We measured the importance of animations as a problem-solving tool. This differs from most previous experiments, which focused on learning, and in which the participants underwent a training session to examine the learning materials (including visualization), followed by a test session to answer the questions. Typically, the test session consisted of a number of problems to be solved without the help of the materials from the training session. In those experiments, user's attempts to recall aspects of the visualization may have increased rather than decreased the cognitive load. We recognize that this is a simplified view, as some learners might quickly form better internal representations (Craik introduced the notion of "mental models" [24]) of the algorithm by observing the visualization. In any case, a study that allows the use of visualization while solving a problem will capture both external cognition and the creation of internal representations.

Distributed computations were employed in both experiments, to the best of our knowledge, for the first time in algorithm animation evaluations. Such computations can be less intuitive than sequential code because of the interactions among multiple concurrent processes. One of the studies, henceforth to be referred to as study $\mathcal{A}$, included two computations. Only one of the computations was used in the other study, which is named $\mathcal{B}$.

Experiment $\mathcal{B}$ was designed specifically to minimize the amount of information about the animation and testing environment to be memorized by the participants. Navigation tasks in the environment and visualization were simplified or eliminated in an effort to reduce the user's tasks not related to the computation.

The next section describes the distributed algorithm used in both evaluations. The two experiments are presented in Section 3.4 and Section 3.5. An analysis of the results across the studies is the subject of Section 3.6. The list of factors that differed between the studies is compiled in Section 3.7, followed by a discussion in Section 3.8.

## 3.3 Termination Detection Algorithm

The algorithm selected for our empirical study is a classical termination detection algorithm for distributed systems by Dijkstra and Scholten [29]. The algorithm assumes that a computation is performed on multiple nodes of a distributed network. The purpose of the algorithm is to allow the node that initiated the computation to determine when the computation has finished, which means that no node in the network is still working on any part of the computation. The initiator process, named root, starts the computation by optionally performing some processing and then distributing the computation among some of the neighboring nodes. Each of the nodes that receives a request to perform a sub-task might individually fulfill that sub-task or decide to divide the request further among a subset of its neighbors. As the request wave continues to propagate in the network, possibly to processes with which the initiator has no direct contact, it may become challenging for the initiator to determine when the entire computation has finished.

To better explain the Dijkstra-Scholten algorithm, we make the assumption that at most one computation takes place in the network at any given time. Thus, a node is either processing a job for the computation or it is idle. Three variables are kept by the each node in the computation: `idle`, `count` and `parent`. A boolean variable `idle` is sufficient to describe whether the node is working or not. The algorithm has each node keep track of the number of requests (messages that are part of the computation) it sends, and requires a node to send an acknowledgment (new message type, specific to the termination detection) back for each request. A request is generally acknowledged when the request is fulfilled. At each node, a variable named `count` contains the number of requests sent by that node that are not yet acknowledged. `count` is incremented every time a request is sent and decremented when an acknowledgment is received.

Each node maintains a record of its parent, the neighbor that introduced that node into the computation. The initiator is the first node involved in the computation, and becomes the root of a tree that, at any time, spans all and only the nodes involved in the distributed computation, which includes nodes that are actively computing sub-tasks or waiting in the termination detection. The tree is defined by the parent variable of each node, and may grow or shrink as nodes become part of the computation or finish their part in the computation. A node is considered to be involved in the computation if it is not idle (is still performing a sub-task) or if it still has children. Each participating node either has a parent or is the root. When a node finishes the computation (it is idle and it has no offspring), it sends an acknowledgment to its parent and sets its parent variable to null, effectively canceling its status as a child. Because of the acknowledgment, a node, A, knows that it has no offspring when `count=0`, which means that all its requests have been acknowledged, and consequently any neighbor that might have considered this node a parent has acknowledged. To show that this is true, consider one request sent from A to N. Either N already has a parent, or it does not, in which case A becomes the parent of N. When N exits the computation, N sets its parent to null and sends an acknowledgment to A. Thus, N is no longer a child of A. In the case where N already had a parent, A was never considered the parent of N. In such a case, N acknowledges immediately after receiving the request. The parent of N will handle N's exit out of the computation.

In this algorithm, when the root is idle and its `count` variable is zero, the computation has finished because the root does not have any offspring and all nodes are out of the computation. This condition can be checked locally at the root without the need to communicate to other nodes.

## 3.4 Experiment $\mathcal{A}$

### 3.4.1 Goal

The study examined two distributed computations to obtain a more complete picture of how algorithm animation can help users to gain insight into such computations. In addition to Dijkstra and Scholten's termination detection algorithm [29], the distance vector routing algorithm was used in the test. Routing algorithms are designed to facilitate communication between nodes of a distributed network that are not directly connected to each other. In distance vector routing, nodes directly connected to each other exchange an array of known reachable nodes and their distances according to a predefined protocol. A description of the routing protocol can be found in most networking textbooks such as [92]. For the comparison between our two experiments, the specifics of distance vector routing are unimportant.

To determine the variations in the results that were due to the use of animations, participants answered questions both in the presence and in the absence of a visualization. The individual skills and knowledge of each participant could be factored out via analytical tools. Note that, for any single participant, it is difficult to make a comparison between visualization and text-only situations for the same algorithm because the knowledge about the algorithm that is gained in the first-encountered situation will improve the results of the second. Switching to another algorithm when the visualization is added or removed makes both situations begin with similar user knowledge.

### 3.4.2 Materials

The experiment took place on four personal computers. Two of them were equiped with 17" monitors and served as hosts for the text-only environment. The computers used by the visualization group had one 19" monitor each.

The algorithm was rendered in Java 3D$^{TM}$ [90] and the testing environment was written in Java$^{TM}$. Parallel simulations of the distributed computations were also implemented in Java$^{TM}$.

### 3.4.3 Subjects

Thirty-nine students enrolled in the "Human Computer Interaction" class (a senior/graduate level course of the Computer Science Department at the University of Georgia) participated in the study to earn class credit and compete for cash prizes. The same class credit could be obtained by completing an alternative assignment, so the participants may be regarded as volunteers. The top three performers (tied performers considered as in the same place) earned cash prizes. Performance was again measured by the number of correctly answered questions.

### 3.4.4 Procedure

The experiment took place in a computer science laboratory. Participants were handed and lead through a print-out describing the testing environment and test procedure. They were also presented with a hard copy description of one of the algorithms. After they had completed answering questions about that algorithm, subjects moved to another computer and received the hard copy description of the other algorithm. The participants were asked to answer nine multiple-choice questions for each algorithm. The participants were allowed to navigate back and forth through each of the two groups of nine questions and to revise their answers as desired. Once the subjects moved to the next algorithm (and group of questions), they could not modify their answers for the first algorithm.

Effectively, the participants completed two consecutive sessions, in each learning and answering questions about one algorithm. In one of the sessions they had access to a visualization for the corresponding algorithm. The order in which the algorithms were presented to each participant varied, as did the order in which the visualization was made available. Table 3.1 summarizes the number of participants in each of four possible orderings.

Table 3.1: The number of participants in each of the four situations:termination detection first and visualization first, termination detection first and text-only first, routing first and visualization first, and routing first and text-only first.

| | **Visualization first** | **Text-only first** |
|---|---|---|
| **Term. detection first** | 11 | 10 |
| **Routing first** | 10 | 8 |

Each participant was required to spend between fifty to seventy-five minutes in the experiment. Fifty minutes was the minimum to ensure that the participants took enough time to utilize all the supporting materials. Seventy-five minutes was the maximum participation time to avoid participant fatigue and to comply with the class schedule.

While answering questions, participants could refer to the:

**text:** the textual description of the algorithm;

**code view:** a static, textual presentation of the algorithm's pseudocode that could be randomly "executed";

**output view:** textual output of a random execution of the algorithm; participants could run the algorithm again with another set of random inputs and events;

**visualization:** each subject could observe one of the algorithms through an animated display;

**visualization description:** a one page print-out describing how the animation encodes the algorithm.

The pseudocode was a stripped-down version of an actual Java implementation of the algorithm. The code view was not interactive and was formatted in a single column that could be explored via a scrollbar. The text for the termination detection was the same in both studies.

The output view could not be filtered and presented the output of a live program. The input and events of that program were randomly generated, and as such the users could see a virtually unlimited number of sample executions of any of the two algorithms.

The program animation was based on a three-dimensional world as shown in Figure 3.1. The meaning of the graphical elements employed in the visualization was explained on a piece of paper handed to the participants at the beginning of the visualization session. One characteristic of the visualization was that the point of view in the three-dimensional world could be modified by the user via mouse controls built into the Java 3D distribution. Changing the point of view helped users in determining the parent-child relationship, which was encoded on the z-axis; the position of the parent was below that of its children. A snapshot of the termination detection visualization is presented in Figure 3.1. A different visualization was used for distance vector routing.

The visualization could be run continuously or step-by-step. In the continuous mode, the visualization could show multiple steps being performed in parallel to convey the flavor of the distributed computation. Note that, in study $\mathcal{B}$, the continuous

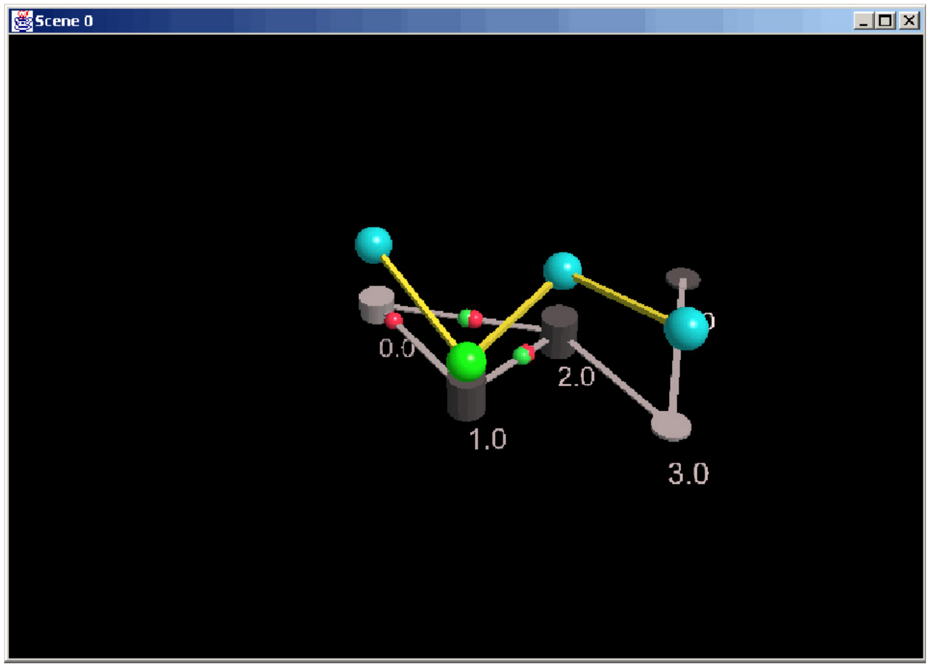


Figure 3.1: A snapshot of the termination detection visualization from experiment $\mathcal{A}$. The tree that spans the distributed network is presented above the network topology. The offspring of a tree node are always "higher" than the tree node.

mode presented only one step at a time. There was no difference between the two experiments in the step-by-step playback feature.

The arrangement of the windows in the testing environment (questions, code, output, and optionally the visualization) was left to the discretion of the user and operating system.

### 3.4.5 Design

The experiment had two independent variables, VISUALIZATION and ALGORITHM, and measured the rate of correctly answered questions. VISUALIZATION has two values $\{vis, nonvis\}$, and ALGORITHM also has two $\{termination, routing\}$. Because not all possible combinations of visualization and algorithm could be presented to every participant, the participants were randomly assigned into four groups, as shown in Table 3.1. The same eighteen questions of Appendix A were given to all subjects.

One hypothesis was tested:

**H1** The use of visualization can significantly improve the rate of correct answers.

### 3.4.6 Results

One participant ran out of time before starting the second algorithm (termination detection). Only the answers from the participant's first session were considered. Thus, 39 participants are included in the analysis of distance vector routing and only 38 in that of termination detection.

Table 3.2: Statistics on the number of correctly answered questions for termination detection.

| **Group** | **Num. of Participants** | **Mean** | **Std. Deviation** |
|---|---|---|---|
| *vis* | 19 | 3.26 | 1.95 |
| *nonvis* | 19 | 3.92 | 1.75 |

Our preliminary analysis revealed that the experiment failed to prove **H1**. Tables 3.2 and 3.3 present statistical values for the number of correct answers individually for each algorithm. It is apparent in the tables that animations did not help their users under the conditions of the experiment. The performance of the participants when using visualizations is slightly worse than when only text is employed. No significant difference was found between the two situations ($t = 1.111, df = 36, p > 0.05$ for termination detection; $t = 0.358, df = 37, p > 0.05$ for routing).

Table 3.3: Statistics on the number of correctly answered questions for distance vector routing.

| **Group** | **Num. of Participants** | **Mean** | **Std. Deviation** |
|---|---|---|---|
| *vis* | 20 | 3.26 | 1.95 |
| *nonvis* | 19 | 3.92 | 1.75 |

## 3.5 Experiment $\mathcal{B}$

### 3.5.1 Goal

To accurately measure the contribution of algorithm animation in performing a task, our study concentrated on reducing the cognitive noise that the visualization design and testing environment might impose on the user. Further, we sought to reduce variability in the participants' experience with the testing environment and with program visualization in general.

The design of the visualization and of the testing environment can add significant cognitive noise to the measurements. Remember that our goal is to test the effectiveness of visualization, not that of the testing environment. Toward this end, we designed visualizations to contain legends, which relieve the user from remembering the encoding of algorithm properties into graphical features or the relation between a measure on the screen and a measure in the program ("is this rectangle supposed to be that long for only two allocated objects?"). Interestingly enough, classical algorithm animations tend not to include a legend. Rather, the user is typically trained for the animation. In our study, the visualization was simply given to the user without any explanation, except of what the algorithm does.

Problems in using a testing environment can also introduce noise into an experiment. One approach to simplifying the user's task involved the addition of interactive print statements (see Figure 3.2) that saved, compared with real life, the time and cognitive effort required for re-compilation and syntax correction. This also provided the users with a filtering capability to enhance searching through the textual program output. To further reduce the effects of the testing environment on the user's performance, we also positioned and sized various windows for the user. Note that these simplifying features applied equally to both the visualization and non-visualization groups.

Since previous studies raised concerns about variations among individuals we took several steps to ensure greater uniformity in the participant's experience. Instructions for the test were recorded and then replayed for each participant, to be sure that every participant saw and heard exactly the same information. Part of the instructions familiarized participants with the testing environment by guiding them through a sample test that used a very simple problem and algorithm animation, but that employed each of the navigational and informational features of the environment. Finally, we prepared in advance the executions of the algorithm that were presented to all participants. This can be a very important factor since random executions might go through different parts of the code and expose different users to quite different aspects of the program. This is especially important when using distributed algorithms, which are intrinsically non-deterministic.

### 3.5.2 Materials

The experiment took place on two Intel® Pentium® personal computers, one of which had the visualization of the algorithm installed. The computer with the visualization was equipped with two monitors. For the non-visualization case, it was possible to display the various windows of our testing environment on a single screen. For the visualization case, the second screen was used only for the visualization, while the first screen displayed the same windows as for the non-visualization case.

The visualization of the algorithm made use of Java 3D$^{TM}$ [90] and the rest of the testing environment was written in Java$^{TM}$. A PowerPoint® presentation, which included speech recordings, was created to deliver instructions about the experiment to the users. The goal of using this type of presentation was to explain the study in the same manner in each user session.

### 3.5.3 Subjects

Twenty college students volunteered to participate in the study. Five were female and fifteen male. To motivate the participants, a cash prize was offered to the person that showed the "best understanding" of the algorithm. Understanding was measured by the number of correctly answered questions. In the end, two cash prizes were awarded, one for the visualization group and one for the non-visualization group.

### 3.5.4 Procedure

Participants were presented with the PowerPoint® instructions about the study environment and with a textual, hard copy description of the termination detection algorithm. During the instructions they were presented with a demo algorithm (finding the maximum number in an array), which provided the opportunity to explain the features of the testing environment and how the algorithm can be examined. The visualization of the demo algorithm was a simple two-dimensional view. In the actual study, the participants were asked to answer ten multiple-choice questions about the algorithm. The questions were presented on the screen and the participants were allowed to navigate back and forth through the questions and to revise their answers as desired. Unlike in study $\mathcal{A}$, no time constraints were imposed.

While answering questions, participants could refer to the:

**text:** the textual description of the algorithm;

**code view:** an interactive presentation of the algorithm's pseudocode that could be interactively "executed," and where the progress of the participating processes could be observed;

**output view:** textual output of the program produced by a series of print statements in the program; output of individual print statements could be toggled on and off by the participant;

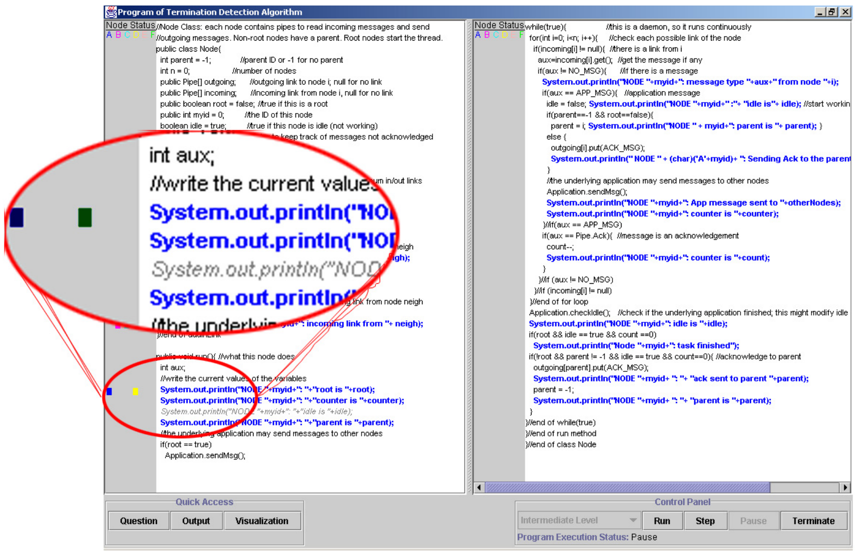
**visualization:** the visualization group was also able to view an animated display of the algorithm.

The pseudocode was a stripped-down version of an actual Java implementation of the algorithm as in $\mathcal{A}$. The actual implementation had been previously run with various inputs, and a trace file for each of these executions had been saved. When the pseudocode was "executed," one of the trace files was played back. As such, the user could select and observe simpler or more complex executions. The trace file could be played continuously or in steps. Moreover, every participant in the study had the opportunity to see exactly the same algorithm runs, which made the comparison between subjects more accurate than with random executions of the program.

The presentation of the pseudocode was interactive in the sense that all print statements, which determine the textual output of the program, could be turned on and off by simply clicking on them. Turning a statement off had the effect of filtering out all the print-outs that statement made during the program execution. In the real world, this would be similar to commenting out the print statement in the program, re-compiling, and then re-executing the program to produce the filtered output. This interactive feature allowed the participants to quickly add/remove information in the output based on whether they considered that information significant for their current task (answering a particular question).

Another feature of the pseudocode view was the presentation of the last statement executed by each process (more like a pseudo program counter). Remember that this is a distributed computation, where multiple processes execute the same sequential program. The code view marked the last statement executed by each process in the computation with a colored rectangle at the left of the statement (Figure 3.2). Each process was assigned a separate color.

The program execution could be observed through a textual output or an animation. Every participant was presented with the textual output produced as the result of print statements. Only half of the participants had access to the visualization.



```
//Node Class: each node contains pipes to read incoming messages and send
//outgoing messages. Non-root nodes have a parent. Root nodes start the thread.
public class Node{
  int parent = -1;            //parent ID or -1 for no parent
  int n = 0;                  //number of nodes
  public Pipe[] outgoing;     //outgoing link to node i; null for no link
  public Pipe[] incoming;     //incoming link from node i, null for no link
  public boolean root = false; //true if this is a root
  public int myid = 0;        //the ID of this node
  boolean idle = true;        //true if this node is idle (not working)
                              to keep track of messages not acknowledged
                                                          um in/out links
                                                          neigh
                                                          gh);
                                                          link from node neigh
                                  yid+": incoming link from "+ neigh);
}//end of addLink

public void run(){ //what this node does
  int aux;
  //write the current values of the variables
  System.out.println("NODE "+myid+": "+"root is "+root);
  System.out.println("NODE "+myid+": "+"counter is "+counter);
  System.out.println("NODE "+myid+": "+"idle is "+idle);
  System.out.println("NODE "+myid+": "+"parent is "+parent);
  //the underlying application may send messages to other nodes
  if(root == true)
    Application.sendMsg();
```

```
while(true){                //this is a daemon, so it runs continuously
 for(int i=0; i<n; i++){    //check each possible link of the node
  if(incoming[i] != null){  //there is a link from i
   aux=incoming[i].get();   //get the message if any
   if(aux != NO_MSG){       //if there is a message
    System.out.println("NODE "+myid+": message type "+aux+" from node "+i);
    if(aux == APP_MSG){   //application message
     idle = false; System.out.println("NODE "+myid+" :"+ "idle is"+ idle); //start workin
     if(parent==-1 && root==false){
      parent = i; System.out.println("NODE " + myid+": parent is "+ parent); }
     else {
      outgoing[i].put(ACK_MSG);
      System.out.println(" NODE " + (char)('A'+myid)+ ": Sending Ack to the parent
     }
     //the underlying application may send messages to other nodes
     Application.sendMsg();
     System.out.println("NODE "+myid+": App message sent to "+otherNodes);
     System.out.println("NODE "+myid+": counter is "+counter);
    }//if(aux == APP_MSG)
    if(aux == Pipe.Ack){  //message is an acknowledgement
     count--;
     System.out.println("NODE "+myid+": counter is "+count);
    }
   }//if (aux != NO_MSG)
  }//if (incoming[i] != null)
 }//end of for loop
 Application.checkIdle();   //check if the underlying application finished; this might modify idle
 System.out.println("NODE "+myid+": idle is "+idle);
 if(root && idle == true && count ==0)
  System.out.println("Node "+myid+": task finished");
 if(!root && parent != -1 && idle == true && count==0){ //acknowledge to parent
  outgoing[parent].put(ACK_MSG);
  System.out.println("NODE "+myid+ ": "+ "ack sent to parent "+parent);
  parent = -1;
  System.out.println("NODE "+myid+ ": "+ "parent is "+parent);
 }
}//end of while(true)
}//end of run method
}//end of class Node
```

Figure 3.2: The view of the algorithm pseudocode. In the inset, one of the print statements is turned off. Also, to the left, rectangles mark the last executed print statement for each of the nodes. Two rectangles are visible in the inset (their original color was changed to increase the contrast on black and white media.

Note that the code view, output view, and visualization were all based on the same trace file, and thus presented the same data. Further, these views were synchronized to present the same steps at the same time.

The program animation was based on a three-dimensional world as shown in Figure 3.3. The main difference between this view and traditional program visualization is the addition of a legend. Instead of talking to users or having them read a description of what the graphical attributes encode, we added the legend to the visualization. We hoped that the legend would reduce the amount of information the user must remember. One difference from experiment $\mathcal{A}$ was that the point of view in the three-dimensional world was fixed and the users could not change it, which reduced the number of tasks to be performed by the user.

The windows in the testing environment (questions, code, output, and optionally the visualization) were pre-arranged on the screen in such a way that none of them was completely obscured.

### 3.5.5 Design

The experiment had both between-subject and within-subject variables. The participants were randomly divided into two groups. One of the groups was presented

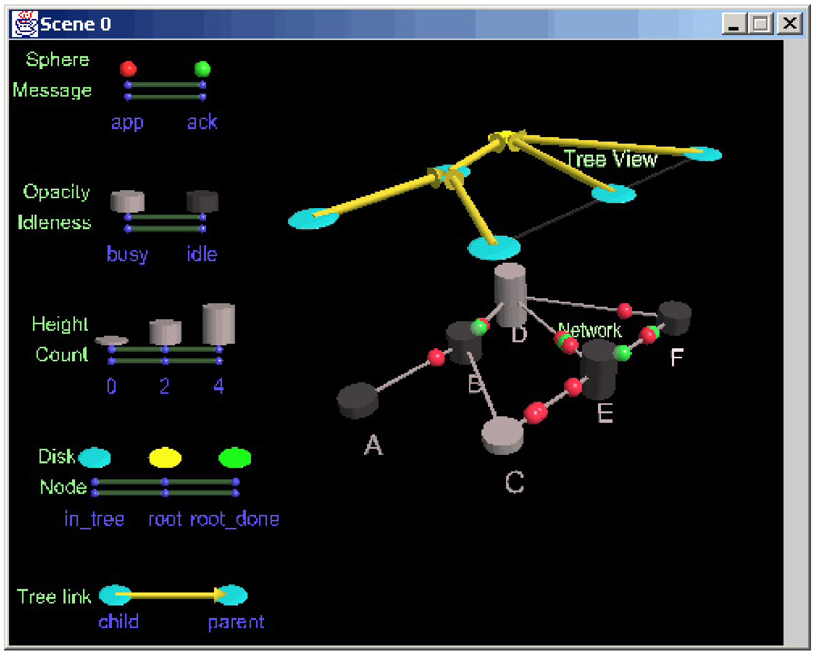


Figure 3.3: A snapshot of the termination detection visualization. The tree that spans the distributed network is presented above the network topology. A legend is displayed on the left.

with a visualization of the program and the other was not. The between-subject variable is named VISUALIZATION and has two values $\{vis, nonvis\}$. Subjects in both groups were presented with exactly the same questions (see Appendix B for sample questions), but the questions themselves were of two types QUESTION $= \{execution, general\}$. Four out of ten questions inquired about a specific *execution* of the program, and it is likely that the user had to run and observe that execution to gather all the information referred to in the question. The other questions were more *general*, in the sense that they were referring to properties common to more than one execution of the program, and could have been answered merely by reading the question. The dependent variable was the rate of correctly answered questions.

Two hypotheses were to be tested:

**H2** The addition of a visualization to the other support materials significantly improves the rate of correct answers.

**H3** Questions of type *execute* have a higher rate of correct answers than more general questions.

### 3.5.6 Results

The average time required to answer the questions was about the same for both groups: about 45 minutes for visualization and about 46 for non-visualization.

Table 3.4: Statistics on the number of correctly answered questions for each group.

| **Group** | **Num. of Participants** | **Mean** | **Std. Deviation** |
|---|---|---|---|
| *vis* | 10 | 7.2 | 1.93 |
| *nonvis* | 10 | 3.7 | 2.26 |

An analysis of variance (ANOVA) on the correctness of the answers reveals that the *vis* group answered significantly more accurately than the *nonvis* group ($F_{1,18} = 14.54, p = 0.0013$). In effect, hypothesis **H2** holds. Table 3.4 presents numerical values for the average number of correctly answered questions.

As illustrated in Figure 3.4, *execute*-type questions had a higher rate of correct answers. However, the study did not find a significant difference between *execute* and *general* questions ($F_{1,18} = 0.19, p = 0.6676$). Thus, hypothesis **H3** was rejected. The analysis also failed to find any interaction between the question type and the use of visualization ($F_{1,18} = 0.10, p = 0.7587$). That means that the rate of correct answers for the two types of questions is nearly constant within each group.

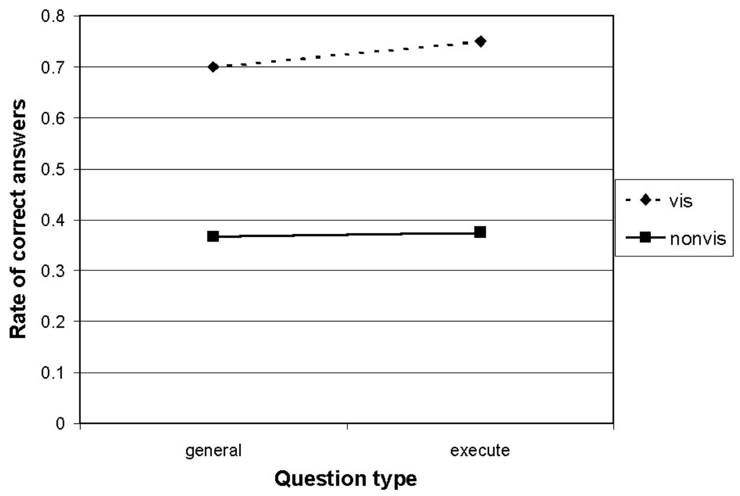


Figure 3.4: The rate of correctly answered questions for each group and question type.

Random assignment of people to groups resulted in four of the five female participants being placed into the non-visualization group. Concern about the effects of this on the validity of our results prompted us to compare the scores of the females within the non-visualization group to those of the non-visualization group as a whole. We found that the mean of the females' scores was above the mean for the overall group. Thus, gender differences can be excluded as the cause of the difference in performance between the visualization and non-visualization groups.

## 3.6 Analysis of the Common Questions in the Two Studies

Although one of the experiments failed to find an improvement for the situations in which visualization was employed, the collected data can be analyzed to determine whether the effect of visualizations was different between the two studies. If such a difference exists, there is a good probability that the factors that were not the same in both studies, or a subset of these factors, critically affects the performance of algorithm animation users. Furthermore, since all changes influenced visualization sessions while only a few affected text-only sessions, we might find the results from text-only situations to be comparable.

The data to be analyzed were restricted to the questions that were common to both studies. Five questions about the termination detection algorithm were exactly the same in both $\mathcal{A}$ and $\mathcal{B}$. The rate of correct answers was computed for each such question in one experiment and then in the other. This rate was further used in an analysis of variation.

We found that there is a significant, though not very significant, increase in performance for the visualization situations in experiment $\mathcal{B}$ ($F_{1,4} = 7.87, p = 0.0485$). This suggests that the factors listed in the next section are important for animation users. For the non-vis groups, no statistical difference was registered between the two studies ($F_{1,4} = 0.62, p = 0.4745$), although in study $\mathcal{A}$, the performance of the text-only situations was slightly higher than in study $\mathcal{B}$. Table 3.5 summarizes the rate of correct answers for each experiment and situation type.

Table 3.5: The rate of correct answers for visualization and text-only cases in each experiment. Only the five common questions are considered.

| | **Study $\mathcal{A}$** | **Study $\mathcal{B}$** |
|---|---|---|
| **vis** | 0.3888 | 0.64 |
| **non-vis** | 0.4108 | 0.3 |

## 3.7 Differences Between the Experiments

The factors that are different in study $\mathcal{B}$ from study $\mathcal{A}$ are listed below. They are then assigned to either cognitive economy, perceptual, uniformity in the experiment design, or probably irrelevant type.

1. Legends were used in $\mathcal{B}$, while a hard copy description of the visualization was used in $\mathcal{A}$.

2. Navigation in $\mathcal{B}$ was improved by pre-arranging windows on the screen, using two monitors for the visualization group, and adding navigation buttons that allowed jumping to the desired window.

3. In $\mathcal{B}$, arrows presented the parent-child relationship as opposed to increased height on the z-axis and capability to change the 3D point of view in $\mathcal{A}$.

4. *execute*-type questions were only presented in study $\mathcal{B}$.

5. In study $\mathcal{B}$, an interactive code view allowed limited filtering of the output.

6. The continuous play of the animation depicted only one step at a time in $\mathcal{B}$ as opposed to multiple steps being shown at the same time to convey the parallelism in the computation in $\mathcal{A}$.

7. In $\mathcal{B}$, the same traces of the algorithm were viewed by all participants, while $\mathcal{A}$ exposed subjects to random executions.

8. Recorded PowerPoint instructions and a simple computation were used in $\mathcal{B}$ as opposed to oral explanations for each participant in $\mathcal{A}$.

9. No time limit in $\mathcal{B}$ as opposed to a minimum and maximum duration for the experiment in $\mathcal{A}$.

10. Only one algorithm was used in $\mathcal{B}$ compared to two in $\mathcal{A}$.
11. The nodes in the distributed network were labeled with letters in $\mathcal{B}$, not with numbers as in $\mathcal{A}$.
12. Physical location was an office in $\mathcal{B}$ as opposed to a laboratory in $\mathcal{A}$.

### Factors Related to Cognitive Economy

Factors one through five can be regarded as promoting cognitive economy. Legends free the users from learning and remembering the mapping between abstract concept in the program and the graphical features of the animation. Navigation, although typically a trivial task, becomes distracting and consumes cognitive resources because it is performed frequently (every time the user is interested in another perspective on the computation or problem). Eliminating the need, and the possibility, to modify the point of view eliminates a task unrelated to the problem at hand to be solved by the user.

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

### Perceptual Factors

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

### Factors that Reduce Measurement Noise

Factors seven through ten did not seem to have contributed to the difference in the results of the two experiments. All these factors targeted the design of the experiment, not of the visualization or visualization environment. The slightly better performance of the text-only group in the second experiment suggests that the factors did not have an effect on the user performance, though they might have helped reduce the noise in the evaluation.

### Factors that May Have Little Significance

We believe that the last two factors, labeling system and physical location, have an insignificant influence on the results. It might take an extreme variation in the physical location or the type of labels to offset the results.

## 3.8 Discussion

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

This chapter also points out important factors for the methodology of controlled program visualization experiments. One group of factors fall under cognitive economy and the other under improving the uniformity of the participants' exposure to the features of the testing environment. These two goals are somewhat conflicting because the information to be memorized during training may lead to increased cognitive load for the participants. An empirical study should concentrate on reducing even trivial tasks not related to the algorithm and on making the information accessible at a glance as to reduce the data that the user feels must be remembered.

Moreover, the design of a study should ensure that most participants go through the same amount of unrelated tasks.

Study $\mathcal{B}$ found program visualization beneficial for understanding distributed computations under conditions that reduced the user's cognitive load. The usefulness of animation seems to be manifest in the accuracy rather than speed of the people's performance (Table 3.4).

Visualization appears to serve as a more effective cognitive resource than textual displays in this problem solving context, which is consistent with the observations of Kehoe *et al.* [50]. The visualization group must have dedicated part of their time to observing the animation instead of studying traditional textual materials. The results of study $\mathcal{B}$ show that the visualization group, with additional displays to view and interact with, required on average about the same time as the non-visualization group to complete the assigned tasks, and at the same time, the visualization group obtained higher scores.

We speculate that the rejection of hypothesis **H3** is due to a possible relationship between performing *execute*-type tasks and comprehending the general behavior of the algorithm. *General*-type questions, which often require knowledge about properties that hold on all executions of the observed computation, may be dependent on an accurate understanding of several individual executions. Knowledge about the general behavior is derived, from among other sources, from the observations of and knowledge acquired from sample runs of the algorithm. Thus, the quality of a tool presenting a running computation affects in a similar manner both how individual executions are perceived and how the overall behavior is understood.

Planned future work includes performing studies to further isolate the contribution of individual factors to the improvement in user performance, including:

- legends v. no legends
- 2D v. 3D
- simple v. complex algorithms
- effects of different navigation schemes
- attributes of algorithm animations that affect their usefulness

## 3.9 Concluding Remarks

Cognitive economy is a principle to be considered in the development of program animation tools and in the methodology of user studies. Under cognitive economy, a three dimensional algorithm animation was found significantly beneficial for an accurate understanding of distributed computations. An initial study that did not focus on reducing cognitive load and variations in the user's experience failed to find visualizations helpful. Important features of the study include ensuring the visibility of all windows, limiting unnecessary interactions, and introducing an initial sample task. Legends attached to the visualizations made explicit the mapping between graphical elements and aspects of the algorithm.

# Chapter 4

# An Abstract Interaction Model for Reshapeable Visualizations

## 4.1 Overview

Over the course of a visualization session a user may employ a wide range of graphical representations, each appropriate for one of the user's tasks. Adaptable visualization displays are needed to meet these diverse interests and applications. This chapter presents an approach for improving cognitive economy during the creation and refinement of visualizations by eliminating indirect structures. Displays are modified solely via on-screen interactions with existing graphical views of the computation and textual representations of the program state. To interpret these user actions, we develop an abstract model for reasoning about visualizations that focuses on the information that is visually communicated and that is independent of the particular choices of graphical elements and interface. By acting through visualizations on the screen, users can control the aspects of the computation and the properties that are displayed.

These actions by the user modify the visualization in a given program state. However, users must have a means to define how the visualization will react to changes in the program state. We provide a way to specify the effect of actions for future program states. Through this means the user can express the desired animation behavior. To achieve their effect and update the program representation, actions are automatically re-applied, in the order they were performed, for each state of the computation. We define a sample set of operations on the model that allow users to take advantage of the information on the screen and to explore related parts of the computation.

## 4.2 Introduction

Visualizations can be categorized on a continuum from being application-independent to application-specific. Application-independent visualizations are useful because they present the same information about any computation, thus reducing any learning curve associated with understanding the visualization. Application-specific visualizations are useful for seeing properties and behaviors particular to the computation being studied. The drawback of application-specific visualizations is that they need to be created specifically for that application. This overhead can prevent them from being used. Ideally, these visualizations could be created on-the-fly, under cognitive economy.

We have developed a system to support the creation of application-specific visualizations and an underlying model upon which the system is based. Our approach is to allow new custom visualizations to be created from empty views or by refining existing visualizations. This refinement process occurs directly through the manipulation of visualizations. Development of the system requires the definition of a semantic model in which to express the effects of user actions. The model captures the information being conveyed by a visualization without considering the visual features through which the information is communicated. The model is not focused on the specific graphical attributes used in the visualization, but rather on how these attributes fit together to represent the computation.

The model includes a dynamic component that defines the manner in which the animation is updated in response to the evolution of the computation. This is done by keeping a history of user operations to be re-applied upon each change in the computation. By defining how the operations are re-executed, users control the evolution of the visualization based on the evolution of the program. The model allows the behavior of an operation, and thus of the animation, to be dependent on previous states of the computation. Rich animations that explicitly depict temporal patterns extracted from multiple states can be created. The explicit depiction relieves the user from remembering information from previous states, and thus improves cognitive economy.

This chapter builds upon work done in both information and software visualization. Information visualization studies techniques of visually conveying various types of data. Specifically, this work builds upon the area of automatic data visualization ([20, 58, 76]) and employs the underlying concept of separation of what is being communicated from how it is communicated. To create an animation, it suffices for users to specify the information they wish to present. The method of automatically generating the dynamic graphical representations is presented in Chapter 5.

To apply automatic data visualization techniques to program animation two issues need to be addressed. First, the state of the program to be visualized is dynamic and, unlike databases, has no relational structure, making it difficult for the tool to extract data and relations to be visualized. Moreover, the actual information users usually wish to communicate is not exactly the program state, but instead needs to be derived from a subset of the state, or even from multiple states. We address this issue in the interaction model and the flexible temporal behavior of animations. Second, an animation of a computation must maintain continuity across frames. Graphical elements that encoded a program property in one frame cannot be used to encode a different property in another. For example, if idle nodes of a network are depicted red at one time, red cannot be used in a later frame even if no idle nodes exist. This issue is addressed by the automatic presentation technique of Chapter 5.

The next section presents a brief overview of the techniques for interactive manipulation of visualizations. Section 4.4 details the abstract model of interactive visualization. The architecture of the system and an example of a visualization session are presented in Section 4.5 and Section 4.6, respectively. A summary is given in Section 4.7.

## 4.3 Previous Work in Interactions with Visualizations

Interactions with visualizations approaches are presented from the perspective of creating visual representations for running programs and from the point of view of presenting large information spaces.

In the program visualization field, Lens [60] is most similar to our work. It allows the creation of visualizations at run-time via direct manipulation. Users interactively specify how the attributes of graphical objects are derived from the program variables or other attributes. Animation annotations are introduced at different points in the program text. In our approach users work with the state of the program, not with the program code. This reduces the dependency on the control structures (e.g., constructs like `if`, `for`) of the program. During the refinement of the program presentation, we also allow users to take advantage of the underlying information associated with graphical objects because of our data-based abstract model, which is different from the graphics-driven approach in Lens.

In information visualization, interactive modification of the content and appearance of a visualization is used as a solution for analyzing and exploring large or complex collections of data. The user can modify the subset of information that is presented in the visualization. To maintain context, an overview of the entire data space is also depicted. The focus subset is presented within this overview. FISHEYE view [34], Table Lens [67] and Magic Lenses [31] are a few examples of such techniques, termed focus+context. The user can choose what part of the visualization to focus on, and in some of these systems can also specify the type of details the user is interested in and even can perform transformations of the raw data. Our system provides similar capabilities through the use of the same operations that make the creation of the animation possible.

Multiple displays can also be employed to convey complex information by presenting the data from a different, and often incomplete, point of view in each display. Generally the windows are linked to each other, which allows the user to infer how the same information appears in multiple windows via semantic brushing. Semantic brushing is the technique of highlighting in all visualization the information selected in any of the views. Visage [75] is a general-purpose tool that support manipulation of multiple heterogeneous visualizations. A specialized system for optimizing problem solving algorithms, which employs multiple visualizations, is COMIND [62]. Our approach assumes that multiple animations are created and refined by the user and supports brushing between animations. The abstract model of the content of a program visualization provides a data-driven approach to relating graphical elements of multiple views.

## 4.4 Visualization Model

The goal of the model is to simplify the creation and refinement of algorithm visualization by allowing users to define a program view via specifications of the information to be visualized. The model promotes an information-based approach in which animations can be created without human calculations of the values for the graphical attributes such as exact position, width, or color. Our approach is based on a static model that abstracts away from the graphical details and captures the data and relations conveyed in a program visualization, and on a technique for specifying the behavior of the animation over time.

This section is organized as follows: a discussion about the high cognitive effort required to calculate graphical details, a short, high-level example of the type of visualization session we consider, the definition of an abstract model for reasoning about static visualizations that circumvents the need to specify graphics, and techniques for creating the flexible dynamics of the animation.

### 4.4.1 Graphics and Interaction

Graphics in an interactive visualization environment are powerful in their roles of information conveyor and of providing "concrete" objects with which the user can interact in a direct style. Nonetheless, the creation of a graphical world, which entails the calculation of numerous interrelated graphical attributes, may be a complex and tedious task that has the potential to lead users away from visualization. The complexity increases for program visualization because the graphical attributes are not static, but dependent on and are intended to depict the evolving values and objects of a program state. Users often have to produce formulas and programs that take into consideration all possible scenarios that may occur during the execution of the program. Consider the simple example of displaying the elements of an array as rectangles arranged horizontally on the screen (depicting the relation defined by the index of the elements). In a world in which visualizations are created/modified by computing graphical attributes, typically a number of rectangles equal to the size of the array are created. Since the size might change, the x-position of each array needs to be specified as a formula based on the width of each rectangle and on the gap between them, something such as $10index + 2(index - 1)$ for a width of 10 and a gap of 2. The formula may be further complicated by the need to center the graphical objects around the origin of the coordinate axes: $10index + 2(index - 1) - 6size$. In any event, the mere listing of a number of program variables requires graphical calculations irrelevant to the observed computation. In a world centered on the information of the visualization, the listing may be accomplish by just adding the program variables of interest to a view. The computer is typically able to handle the computation of the graphical details, as presented in Chapter 5.

A further drawback of a world that consists solely of graphical attributes is that such a world masks the underlying program data in the values of the attributes (note that this does not imply that the pictorial representation is hard to interpret visually). It becomes tedious for the user to relate graphical attributes back to the values of the observed computation. In our previous example, knowing that some rectangle is positioned at 550 does not directly point to the index of the array element represented by that rectangle.

We envision a visualization model that does not rely on graphical properties and, consequently, allows users to specify and modify the information to be conveyed. The model provides for users that are unable or unwilling to process computer graphics. Nonetheless, graphics provide a "materialization" of the program entities that can be chosen, dragged or otherwise manipulated directly on the screen. We also recognize that, as an information conveyor, the choice of graphical elements of a picture significantly determines how the image is perceived by a viewer as explained by Colin Ware in [96]. The abstract model, although focused on information, still allows users to operate at a graphical level (see Chapter 5). Note that the perceptual importance of graphical elements does not invalidate our choice of abstract visualization model. Perception and creation/modification are different tasks with different requirements.

### 4.4.2 Constructing Visualizations and Exploring the Computation

From the user's perspective, the computation appears as a set of variables whose values change as the program runs. The values of the variables, the state of the computation, are assumed to be collected and fed into the visualization tool by a monitoring system such as PathFinder [40]. The user can observe the state and select the variables of interest in a pre-defined textual view that presents each variable as a tuple.

Consider that the user is monitoring a distributed computation in which four node descriptors exist for nodes A through D. Each node descriptor is initially seen as a tuple, and the user would like to build a graphical view to observe the computation.

The user creates a visualization by instructing the system to depict the first three descriptors; the fourth one may not be of interest. No graphical feature needs to be specified because the system can automatically produce a graphical design.The new visualization shows three objects. Note that no property of the objects nor relation among objects was specified by the user, and consequently, no property or relation appears in the animation.

Next, the user refines the existing visualization by requesting that the CPU utilization, which is a field of the node descriptor, be shown for each of the three nodes. The inclusion of the CPU utilization relation adds a structure to the visualization that enables the user to analyze the patterns among the utilization of CPUs. Again, the system can automatically employ a graphical attribute, such as height, to encode the CPU relation. Note that a second user might be interested in comparing the CPU of node A with the number of messages sent by node B, instead of the CPU of B. Such a user can compute a single relation from the CPU of A and message count of B, or in effect from any available data. The computed relation is then depicted as in the case of the first user.

In general, the visualization pipeline starts with the tuples, from which the user derives the information to appear in the visualization, and is completed by an automatic presentation technique. The user's task is to choose the subset of computation for the objects in the display and the structure (relations and properties) to be visually shown in the animation.

### 4.4.3 Scenes

Our goal is to circumvent the use of graphical attributes in the creation and modification process and to base the interaction on the information content of a program view. We have developed an abstract structure termed a *scene* that captures the data and relations portrayed in a picture, but does not concentrate on graphical features. Any program view can be captured as a scene, and a scene can be transparently transformed into a visualization with the techniques in Chapter 5. As such, the user can create or refine a scene via interactions with a visualization.

We introduce the scenes from the perspective of a system designer rather than a visualization user because the former supports a more precise definition. The role of a scene is to provide an exact description of the information in a static program visualization. Consider Figure 4.1 which presents two images that convey the same information, but with different graphical features. One observation is that both images have a number of objects, such as rectangles, lines and bars. These objects have certain properties, and some of the properties suggest relations between objects. Moreover, each graphical object has a meaning in that it represents some entity or entities in a computation. The same picture can be used, in one instance, to present the utilization of each node in a distributed computation (as in Figure 4.1), or in another instance, to present the utilization of entire sub-networks in a domain of linked networks. The particular meaning of the objects is the distinguishing feature of a program visualization compared to other types of visualizations.

| **Data** | **Structure** |
|---|---|
| node A | Name=A; CPU=40; OS=Linux; Connection=AB,AC. |
| node B | Name=B; CPU=20; OS=Irix; Marked=true; Connection=AB. |
| node C | Name=C; CPU=65; OS=Irix; Connection=AC. |
| link AB | Marked=true; Connection=A,B. |
| link AC | Connection=A,C. |

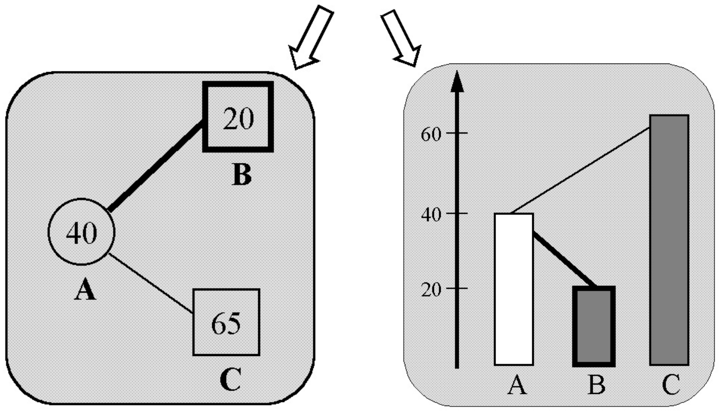


Figure 4.1: The same data, a network and the CPU usage of each computer, is presented in both visualizations. Labels on the left, and height on the right show the usage. Nodes are grouped in two classes by their type of OS, which is depicted via shape or fill intensity. Node B and channel AB are marked and appear bold in both pictures.

Based on the previous observations, a scene is defined as a set of *scene elements.* Each of the elements is a pair of *structural specifications* and *data specifications* because it both has a number of properties and represents some subset of the state of the observed computation. The properties of the elements define a structure among scene elements, and we termed that part *structural specification.* The subset of the state that each scene element depicts is termed *data specification.* Typically, a one-to-one correspondence exists between scene elements and graphical objects, and each structural specification is depicted through a graphical feature such as color, width or adjacency. Table 4.1 is a concise description of the notions mentioned above.

Table 4.1: Overview of the elements of the model

| *Conceptual Element* | Scene | Scene Element | Data Specification | Structural Specification |
|---|---|---|---|---|
| *Captures* | window | graphical object | meaning of an object | property of an object |

**Rendering Specifications**

*Structural specifications* capture the interrelationships between different parts of the visualization. The structural specifications of a scene are described as named relations of the form $[< name >=< value >]$. There exists a *relation* among scene elements that contain a structural specification with the same name. A *logical equivalence* is assumed among elements with the same $[< name >=< value >]$ pair, while a *logical distinction* is being made between elements with the same name and different values. Any existing ordering on the domain of the values is extended on the elements. In Figure 4.1 right, nodes are grouped in two classes by shading. The structural specifications describing this relation is $[OS = Linux]$ for A and $[OS = Irix]$ for B and C.

A scene element can be involved simultaneously in multiple relations (see Figure 4.1), each of them with a corresponding structural specification name. A scene element can be logically equivalent, within the same relation, to an arbitrary number of other elements. The other logically equivalent elements can be distinct, as AB and AC, which are both related to A, appear in Figure 4.1.

Graphically, a relation between scene elements is shown through at least one graphical attribute. Within a relation, the objects corresponding to scene elements that are logically equivalent have identical values for the graphical attribute(s) of the relation. As such, the graphical objects have some visual characteristics in common. Graphical objects for logically distinct scene elements have different values for the attribute(s) depicting that relation.

### Data Specifications

The state is regarded as a set of tuples (tuple space). This space continuously changes through atomic steps as the computation runs. *Data specifications* describe what part of the state of the computation is represented by a scene element. Note that structural specifications do not directly express the subset of the program state that is being visualized because their main role is to express relations; structural specifications may be derived from the values of the data specifications.

Users may rely on data specifications to identify the program objects represented by a scene element (graphical object). The same type of object, such as a circle, may represent one computer node in one visualization or an entire network in another. Having the explicit portion of the state associated with each scene element helps in finding the components of the visualization based on properties of the state (e.g., find all messages sent by this graphical object). It is also useful in the exploration of the computation because it can add to the scene parts of the program state that are related to existing elements (e.g., show all nodes that are members of the same group as the target node). Finally, it is useful for changing the level of detail of a graphical representation (e.g., moving from the representation of Figure 4.1 to a single circle for the entire network and back).

A tuple has the form $< ID, TYPE, field_3, ..., field_n >$, and typically corresponds to a program variable. The number of fields depends on the tuple type. The ID and the TYPE are fixed for a tuple. Each $field_i$ has a value and a name. Tuples of the same type differ only in the ID and field values. A tuple space is content-based accessed, i.e. the tuples are matched by the values of their fields, which are likely to change over time. The unique ID for each tuple helps to provide continuity in monitoring the same portion of the state while the program runs.

### Using Scenes

Visualizations are created by selecting the subset of the computation state to be represented, grouping the subset into smaller units based on the desired granularity of the presentation, and structuring the units by deciding on the individual properties and relations to be observed. These tasks can be performed naturally in the scene framework as explained below. Refinement of the visualization entails the modification of the existing scene.

In the scene framework, the subset of the computation state that the user is interested in is wrapped as one or more scene elements via user operations. The data specification of an element might contain the tuple for a single process, while another scene element might contain the tuples for an entire network. In effect, the grouping is performed when the data specification of each scene element is defined. Each scene element has a graphical entity representing it in the visualization.

The graphical entities (and the scene elements they represent) are structured by the user's definition of the structural specifications. Properties of an individual scene element can be specified simply by adding a structural specification (e.g., mark node A as busy by adding structural specification $[busy = true]$) based on the user's preference or by computing structural specifications from the data specifications of that element. A global property, which implies a relation among scene elements, can also be described by structural specifications. The CPU utilization of the nodes can be added by $[CPU =< x >]$, where $< x >$ is copied from the data specification of each scene element (Figure 4.1).

### 4.4.4 Temporal Behavior of Animations

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

A scene is just a static description of an animation frame that corresponds to one state of the computation. In an animation, there exists a scene for each state. The model does not require the user to specify each scene; an updated scene can be created automatically by starting with the initial view and re-applying the operations the user performed to obtain the previous scene. Thus, a *history* is maintained to record all operations. The history is a sequence of operations. Note that the history records all operations, including the ones that refined an old visualization because we support a scenario in which an animation is created and observed for a period of time, then it is refined, and the observation resumes.

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

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

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

For example, suppose that the user wants to highlight idle nodes in a computation that has three processes, and they become idle in the order A, B, C with only one idle at a time. The first time the operation is executed, the marked node is A in all modes (see Figure 4.2, first column, which corresponds to the static behavior). After the program state has changed, one possible behavior of interest to the user might be to keep marking process A as a reference point that A is the one that was idle. The animation must display the same process that was initially idle, A, even if it is not idle anymore (first row of the figure). Another possibility would be to ignore the past and show the process that is idle now, node B (second row). There is yet another option: to present both processes that were idle and the ones that are idle. More precisely, processes that were idle in any state between the initial and the current one. This mode allow easy tracing of momentary properties. Of course, the user can input the same operation twice in different modes, such as absolute and relative in order to observe both the node that was initially idle and the current idle ones.

A selection operation in *absolute* mode has the same effect regardless of the current state. The same items are selected as when the operation was first applied.

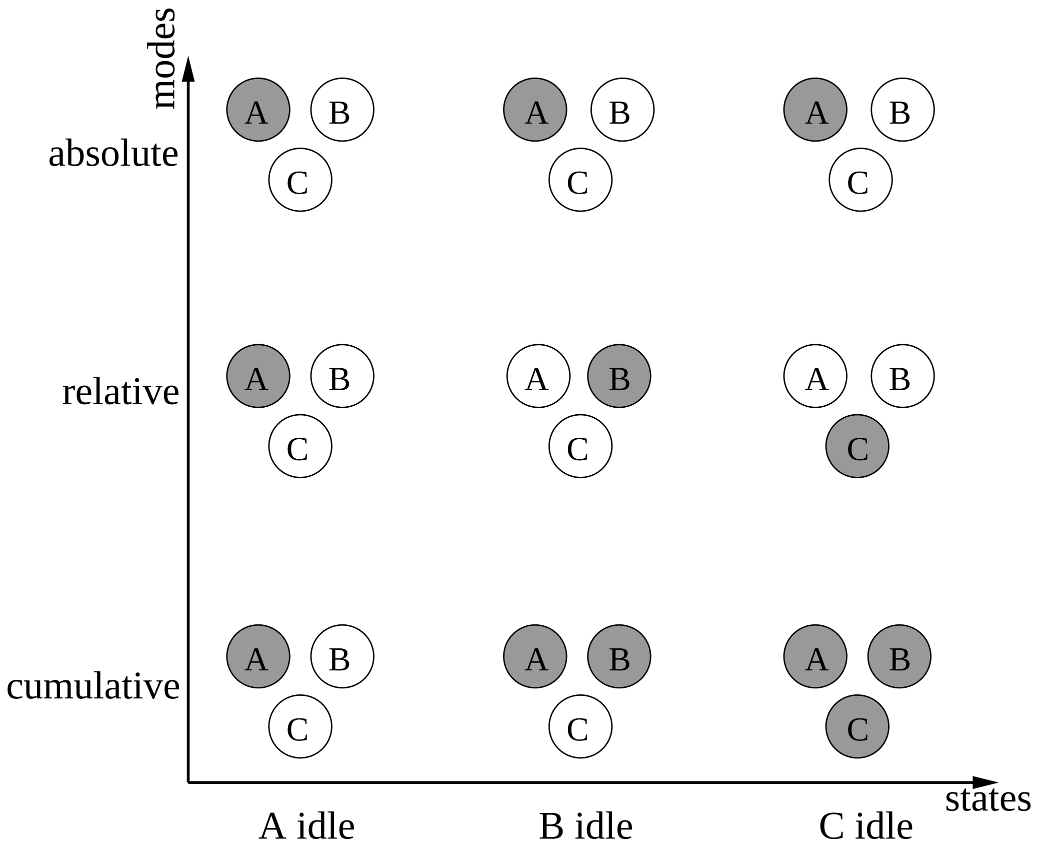


Figure 4.2: A row presents a visualization during three states of a program. Initially, only A is idle, then only B and then only C. The shading of the idle nodes is applied in a different mode in each row.

Such an operation is not directly recorded in the history. An operation that specifies exactly the instances of selected items is recorded instead. In the example above, the operation being recorded would be: find the scene element for node A (because idle nodes = $\{A\}$) and add the structural specifications $[marked = true]$ to it. When the history is re-applied in later states, the recorded operation keeps marking node A.

A *relative* operation does not depend on the previous program states, only on the current one. Such an operation behaves as if the user executes it for the first time every time the state is modified. It is added to the history in its original form. Non-selection operations, the ones that modify scenes, are always in the relative mode.

Finally, a *cumulative* operation depends on the sequence of states that has occurred since its first execution. It is similar to having that operation performed in absolute mode for every state in the sequence. Two operations are added to the history. One is the operation as if in absolute mode and the other is the operation itself, in cumulative mode. When the history is re-applied, the second operation causes another operation in absolute mode to be recorded between the first and the second. The history will contain one absolute operation for each state since the operation was performed. In the example, before node C becomes idle, the history will contain an operation 'mark A idle' (from the first time the operation was applied in cumulative mode), 'mark B idle' and 'mark nodes with CPU=0 idle' (from its application for the second state of the program).

New operations can be defined by combining a sequence of operations into a single, atomic one. A sequence that contains selection in cumulative mode is recorded once for each state of the computation, with the selection operation(s) in absolute mode for the corresponding state. This ensures that the atomicity of the operation is preserved. In the example, first idle nodes are selected (operation $Sel(idle)$), a new structural specification is added to them (operation $Mark$). The sequence would be recorded as $< Sel(A), Mark >$ in absolute mode, as $< Sel(idle), Mark >$ in relative mode, and as $< Sel(A), Mark >< Sel(B), Mark >< Sel(C), Mark >$ in cumulative mode.

To reduce the length of the history, only simulated operations that were not previously recorded are included in the history. It is superfluous to record 'mark A idle' twice in a row since the second operation does not produce any modification to the scene.

## 4.5 The Reshapeable Visualization System

The abstract model was implemented in a prototype system. The Reshapeable Visualization system implements the framework described in the previous section. The system provides an environment in which users can build graphical representations of a computation through direct manipulation of scenes. The information and structure of a visualization is defined by user actions on scenes. Visualizations corresponding to those scenes are automatically drawn by the system. Users can intervene and determine entirely or partially the graphical elements to be used.

The architecture, depicted in Figure 4.3, consists of two modules, implementing a two-step mapping of program state to graphics. The first module generates scenes based on the user actions and the state of the computation. The second is a rendering system capable of realizing each scene as an actual image.

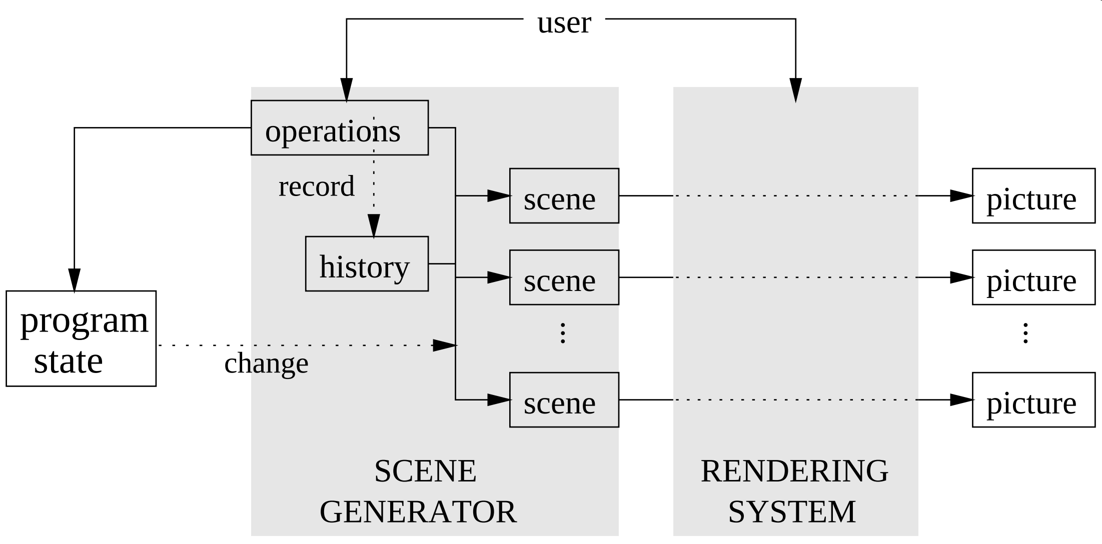


Figure 4.3: The architecture of the visualization system.

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

The scene generator processes user operations and records them into a history as defined in the previous section. The state of the computation is provided by the monitoring system PathFinder [40]. PathFinder notifies the scene generator when the computation changes and provides the data describing the state of the computation. Upon notification, the scene generator re-executes the history.

The rendering system chooses a graphical object for each scene element in a scene. The system also assigns graphical attributes and values to the rendering specification such that the final view presents the structure of the scene. Users can specify the exact graphical elements to be assigned to a scene element or structural specification, or they can override the decisions of the rendering system. In the former case, the names and values that appear in the structural specifications can influence the system decisions. The system interprets a structural specification of the form $[SHAPE =< ObjName >]$ as a user request to select the graphical object named $< ObjName >$ to depict that scene element. The system also tries to match the name or value of a structural specification with the name of an attribute. As such, a specification [$width = 3$] is assigned to the width attribute if possible (there exists an unassigned attribute named `width`).

The prototype currently has a very basic user interface, based mostly on dialogues and limited point-and-click features. It is possible to replace some of the dialogues with more "natural" drag-and-drop techniques, but our main goal is to test the high-level interaction model. The rendering system uses Java 3D [90] to draw the graphical objects on a 3D canvas. Navigation capabilities, zooming, translation and rotation of the point of view in the 3D space, are available to users. The transition from one picture (state) to another is performed smoothly by interpolating linearly between old and new values of attributes.

## 4.6 Sample Visualization Session

An example of constructing and refining an animation provides the opportunity to describe the use of the abstract interaction model in practice and a set of operations that take advantage of the model. The set of operations presented here is simple and was developed for the minimal user interface of our prototype system, which allows point-and-click and dialog-based interactions. It is conceivable to develop more complex operations targeted at graphical interfaces that allow richer gestures such as dragging. We consider the set of operations an implementation issue, rather than a contribution of our model.

First, the computation chosen in this example is presented, followed by a discussion about the selection framework, and then by the step-by-step visualization session.

### 4.6.1 Computation

A hypothetical user investigates how a railroad system functions. The railroad components (trains, signals, and switches) are "smart" devices that communicate with each other to assure the safe passage of trains to their destinations. The railway is divided into segments, each guarded by a signal. To enter a segment, a train sends a request to the corresponding signal. If the request is approved, the train moves onto that segment. Otherwise, it either waits or tries a different route. The computation to be monitored is a simulation of this railroad system.

The types of tuples in the program state and their corresponding fields are listed below.

- `track(x, y, orientation, status)` represents a piece of track starting at position (x, y) with the given orientation and status. Orientation expresses a compass direction such as North, North-West, West or South-West.
- `train(x, y, orientation, number)` describes a train at position (x, y) with a given orientation. The last field is a unique identifier for the train.
- `light(x, y, locked, status)` defines a light at the given position. The light status is either 'go' or 'stop'. The train may move to the next track if the light is 'go'. The locked field is used by trains to reserve the track guarded by this light and is employed in deadlock prevention.
- `switch(x, y, status)` represents a switch at position (x, y). The status field indicates the direction that the switch is thrown.

### 4.6.2 Selection Framework

The paradigm on which the operation set was developed relies on selection to determine the information used by most operations that modify the animation. The user can select the elements to be changed and any additional information from the visualization or computation. The selection can be performed in multiple steps, and may be quite complex.

The system maintains three selection sets, one for structural specifications, one for tuples and one for scene elements. The graphical objects of the selected scene elements are highlighted in the visualization, in any scene in which they appear. Similarly, selected tuples and structural specifications are highlighted in the textual displays in which they appear. A tuple field that is the same as a selected structural specification is also highlighted. Note that tuple fields and structural specifications have the same format $< name >=< value >$, and as such, one can be used instead of another.

The user has the capability to add and remove elements to any of the selected sets via a number of selection operations. The selected information is included in the input for some operations.

### 4.6.3 Visualization Session

The visualization session consists of the following high-level steps:

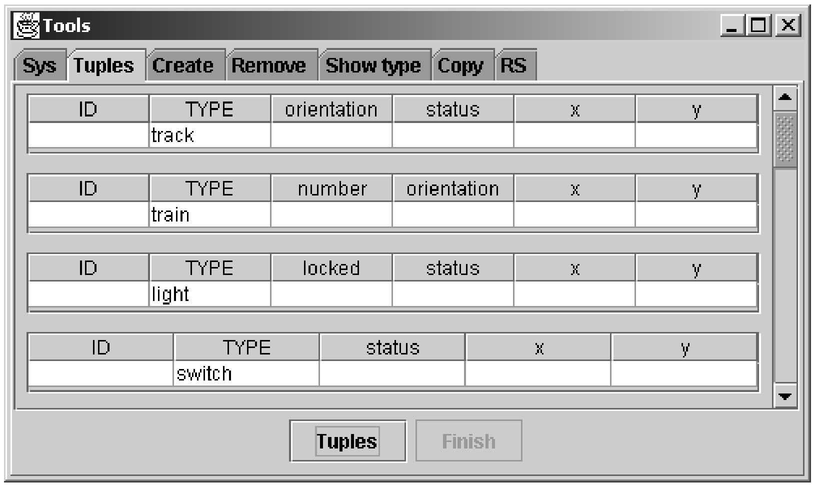
1. the tracks are first presented in the visualization, followed by the signal and trains;
2. the trail of each train is included in the animation to permit observation of the route taken by the trains, as well as their current position;
3. the amount of time spent by a train on each piece of track is encoded visually.

Initially, no visualization exists: only a tools window and a pre-defined view of the computation state, named *State View*. The tools window allows the user to choose operations and to control the system. The State View presents in a textual form the tuples of the current state of the program. The user has the option of viewing all individual tuples or only the types of tuples, as captured in Figure 4.4. When no operation is in the process of being input, this view permits basic selection to be performed by clicking on a tuple or a tuple field. In the former instance, the tuple is added to or removed from the selected tuples. In the latter instance, the tuples with the same field are (de)selected if the name of the field is clicked, or the field is (de)selected as a structural specification if the value is clicked. Similar views are used to display the data specifications of scene elements, which are also sets of tuples.

#### Creating Elements

The user begins by creating a view of the tracks. The tracks are first selected from the State View. The Create Scene Element operation is then selected in the tools window, and a panel (dialogue) as seen in Figure 4.5 appears in the tools window. The dialogue not only creates scene elements (and consequently graphical objects), but also provides for adding structural specifications to the newly created elements. The operation uses the selected tuples and, for each tuple, creates a scene element. Then, for each created element, the operation adds the structural specifications given in the dialogue.

When a structural specification is added to a scene element, the value introduced by the user is evaluated. An identifier prefixed by the special symbol '$' is considered a reference. To evaluate a reference, a name-value pair whose name is the



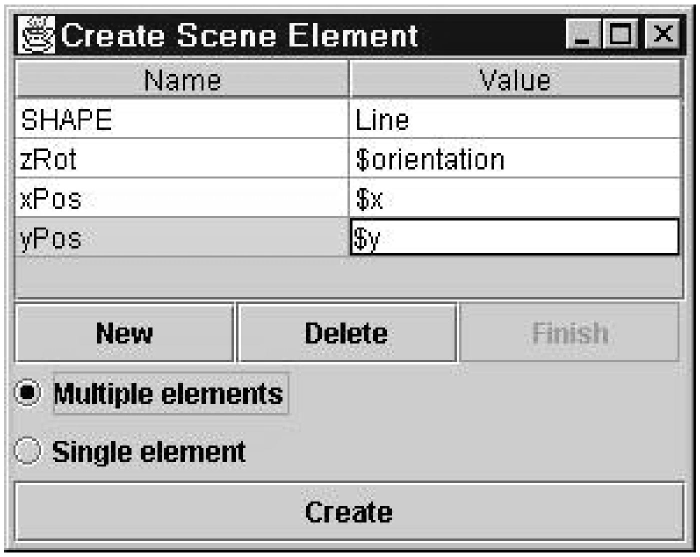
| ID | TYPE | orientation | status | x | y |
|---|---|---|---|---|---|
| | track | | | | |

| ID | TYPE | number | orientation | x | y |
|---|---|---|---|---|---|
| | train | | | | |

| ID | TYPE | locked | status | x | y |
|---|---|---|---|---|---|
| | light | | | | |

| ID | TYPE | status | x | y |
|---|---|---|---|---|
| | switch | | | |

Figure 4.4: The textual view of the computation state showing only the types of tuples.



| Name | Value |
|---|---|
| SHAPE | Line |
| zRot | $orientation |
| xPos | $x |
| yPos | $y |

● Multiple elements

○ Single element

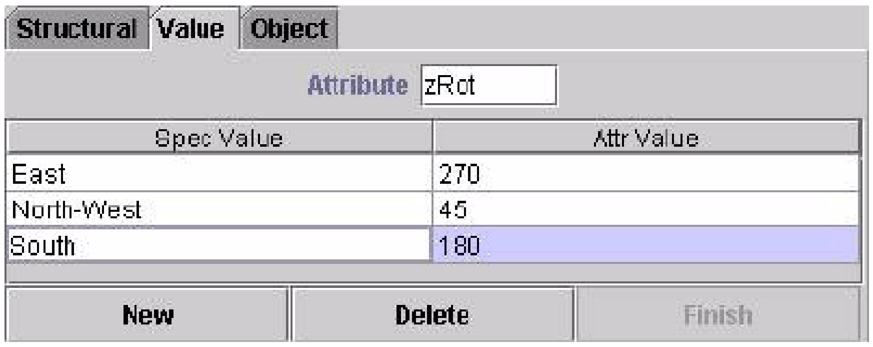
Figure 4.5: The dialog for creating the track scene elements and setting their structural specifications.

identifier must be located. The prefixed identifier is then replaced by the value of the found pair. The user can direct the Create Scene Element operation, and any other operation that recognizes references, to search for a pair in the scene element to which the structural specification is added, either in the selected specification or in the selected tuples.

In Figure 4.5, the value `Line` remains the same for all new elements. The reference `$x` is replaced by the actual value of the `x` field of the `track` tuple. This is the tuple that was wrapped inside the scene element when it was created. Thus, scene elements have the specification `xPos` equal to the x-position of the track they represent.

The hypothetical user has a clear idea of how the tracks should visually appear and requested a line as a graphical object. Moreover, the user instructs the automatic presentation system to use the x-position to encode the field `x` of the tuple. A specification named `xPos` is always assigned to the x-position. Similarly, `yPos` is assigned to y-position and `zRot` to z-rotation.

Two interesting issues arise in the drawing of the tracks: the system has no knowledge of the semantics of constants like North or South, and neither the system nor the user know the length of a track. The user solves the first problem by "teaching" the system the meaning of direction constants. Either the registry editor from the tools window or a legend like the ones in Chapter 6 have the capability to adjust the image. In the registry editor, the user modifies the values assigned by the system so that the rotation angles correspond to the right directions. This is shown in Figure 4.6 (angles are in degrees). The second problem can be solved by the user by tweaking the length value with a legend or trying different length values for the track segment.



Attribute: zRot

| Spec Value | Attr Value |
|---|---|
| East | 270 |
| North-West | 45 |
| South | 180 |

Figure 4.6: Assigning correct rotation values to symbolic compass directions.

Figure 4.7 shows the visualization containing the tracks. The user adds the signals and trains to the visualization in a similar manner, via selection and scene element creation. To ensure that the status of signals and position of trains change with the computation, the operations are executed in relative mode. Figure 4.8 is a frame of the current animation.

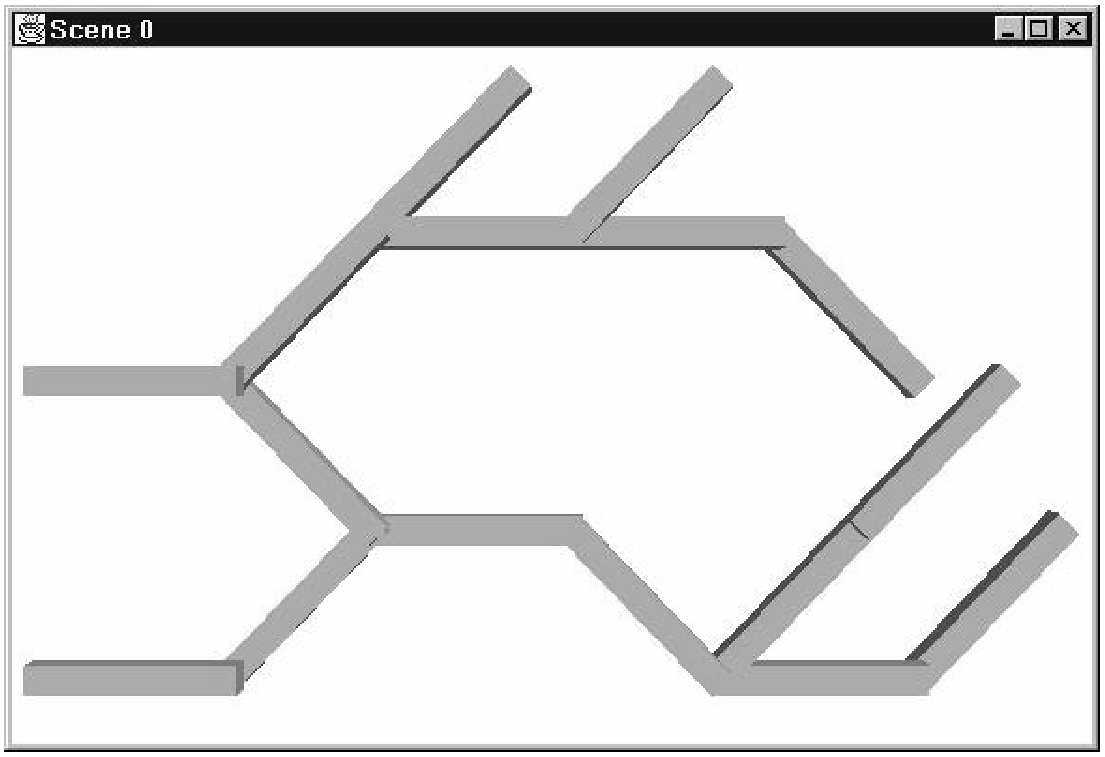


Figure 4.7: A snapshot of the visualization after the tracks have been added.

### Property-Based Selection

Consider that the user in the scenario needs to select all train elements. A possible approach is to click on each train. Clicking on a graphical object when no operation was previously activated results in the selection or de-selection of the scene element that corresponds to that object. This instance-based selection is useful, but it may become tedious for a large number of elements.

An alternative approach is for the user to specify a property of the elements, tuples, or structural specifications of interest. Property-based selection is also a powerful instrument that lets the user take advantage of the information and relation captured by the model.

Property-based selection takes as input a set of name-value pairs and matches the elements, tuples or structural specifications (the user chooses one of the three) that have all those pairs. The name-value pairs may contain references. An operand such as '<' or '>' can be used instead of equality. The matched elements, tuples, or structural specifications are added to or removed from the selected set chosen by the user. For scene elements, the user can restrict the match to data specifications

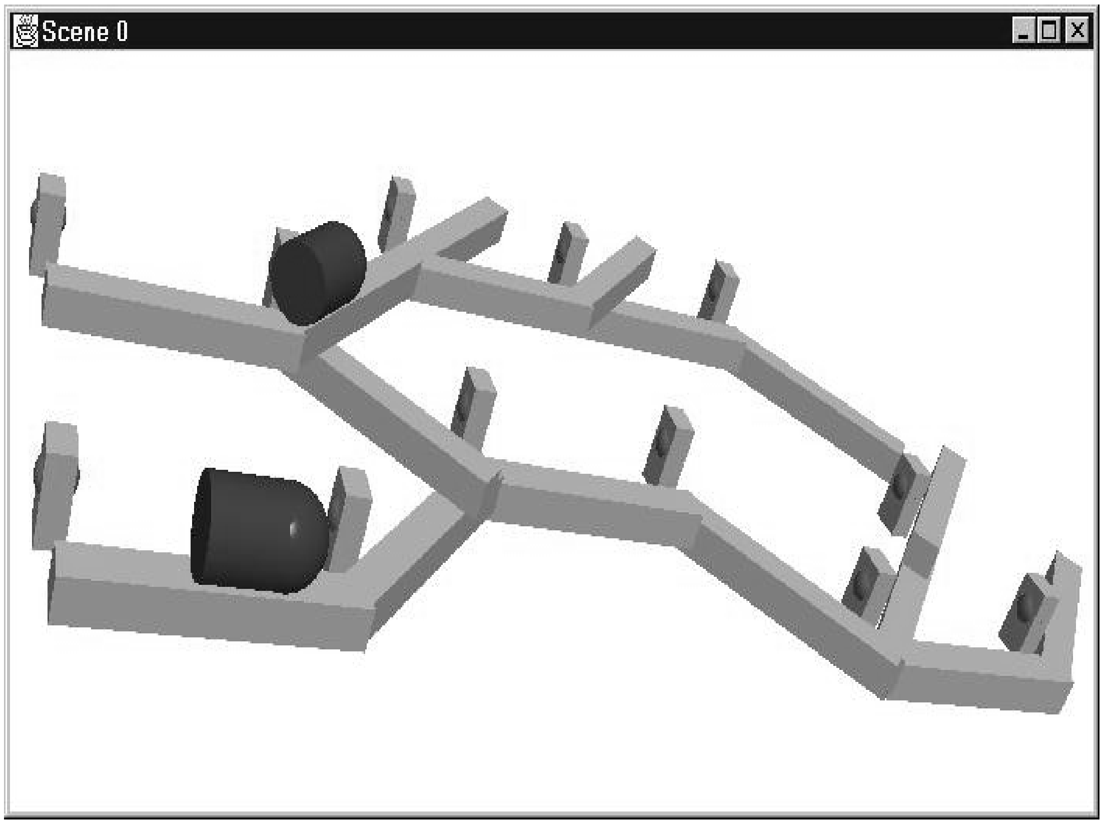
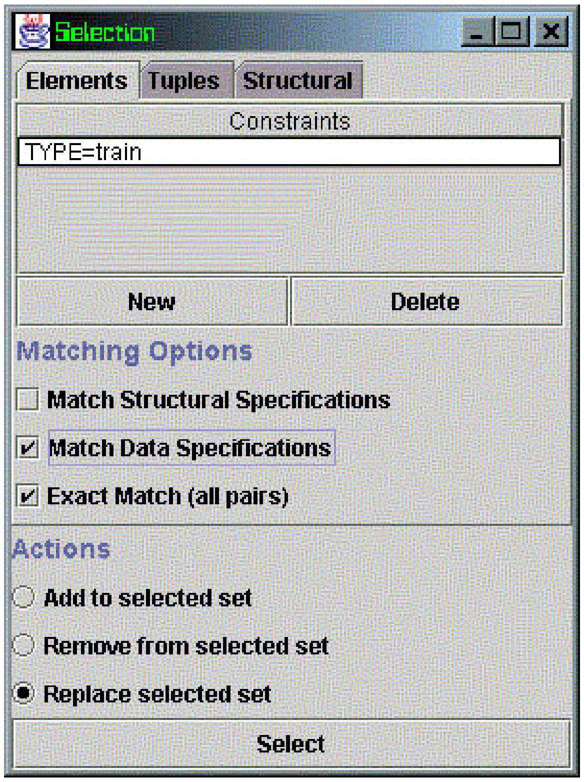


Figure 4.8: The visualization showing the trains and the railroad.

or structural specifications. Figure 4.9 shows how the user replaces the selected set with the scene elements that contain a tuple of type train.

The user can simply click on the `TYPE` field of a track tuple instead of writing the constraint in the dialogue. This feature allows the user to point to one object that looks interesting and select all other object. After choosing the select operation, the user can click on one of the train objects with the third mouse button, which will open a Data Specification View (the second button opens a view of the structural specifications of any object). While select operation is activated, a click on one of the fields in the Data Specification View is interpreted as a request to select all tuples that have the same field and value. Each click automatically updates the selection dialogue. The user just pushes the select button of the dialog after pointing to the fields of interest.

More complex queries are defined via `struct.`, `tuple.`, `selectedStruct.`, and `selectedTuple.` modifiers. They may be specified before the name or value of a pair and direct the matching options and the de-referencing for that pair. For example,



TYPE=train

☐ Match Structural Specifications

✓ Match Data Specifications

✓ Exact Match (all pairs)

○ Add to selected set

○ Remove from selected set

● Replace selected set

Figure 4.9: The property-based selection dialogue.

typing `tuple.TYPE=train` forces the matching of scene elements that have a tuple of with a `TYPE` field equal to the value `train`. The modifier `struct.` refers to the structural specifications of an element, while `selectedStruct.` and `selectedTuple.` instruct the operation to search in the already selected tuples or structural specifications to de-reference a '`$`'-prefixed identifier.

### Refining Data and Structural Specifications

The visualization described so far does not offer information about the route followed by a train. Now we will consider how to modify the visualization to show such information. For each train, it is enough to add a structural specification $[visited =< x >]$ to a track that has been visited by that train, where $x$ is the number of the train. The same structural specification needs to be added to the train object so the route and the train share a graphical characteristic. No specific graphical elements need to be given by the user, they are automatically assigned.

The operation to add structural specifications to selected tuples was already discussed with the Create Scene Elements operation, which adds structural specifications after creating scene elements. Thus, to add $[visited = < x >]$ to all trains, it is enough to enter `visited=$number` in a dialogue similar to that in Figure 4.5 after selecting all train objects. The Add/remove Structural Specifications operation goes through all selected elements and adds/removes the specifications given by the user. The specifications are evaluated in the same manner as for scene element creation.

Adding the specification $[visited = < x >]$ to a track, where $< x >$ is the number of a train, is performed by first adding the train tuple to the track element. The new element presents information from both the track and train program objects and should have the data specification updated accordingly. The second step, just adds a new structural specification `visited=$tuple.number` to the modified tracks, which has the effect of copying the value of the `number` field for each selected track.

An operation named *join* permits aggregating selected elements. The operation needs the pairs (tuple fields or structural specifications) that have to be common across some elements in order to aggregate them. The aggregation entails the union of the data specifications, structural specifications, or both as determined by the user. The operation can also join selected elements and selected tuples, which adds to a scene element the tuples that have the user-specified fields in common with that element.

In the scenario, the user intends to add a train tuple to the track on which that train resides. So, the user selects all trains tuples and tracks elements, then activates the Join operation. Next, the fields `x`, `y` and `orientation` are clicked, which is interpreted as a specification of the common fields; a train is on the track that has the same location and orientation. The Join operation is executed for the tuples only. So, the tracks that have a train, also contain the tuple of that train. Note that this might be quite difficult to specify using only graphical features, but the information-based model makes it simpler to choose.

The Join operation is performed in cumulative mode. Join is regarded as a composite operation that first selects the scene elements and tuples, and then joins them. The cumulative mode ensures that, in future states, the tracks that were previously selected will continue to be selected. The visualization thus far is shown in Figure 4.10.

The visualization, at this point, presents all trains moving on the railroad. As they move, their trail is marked by a different color, each corresponding to a different

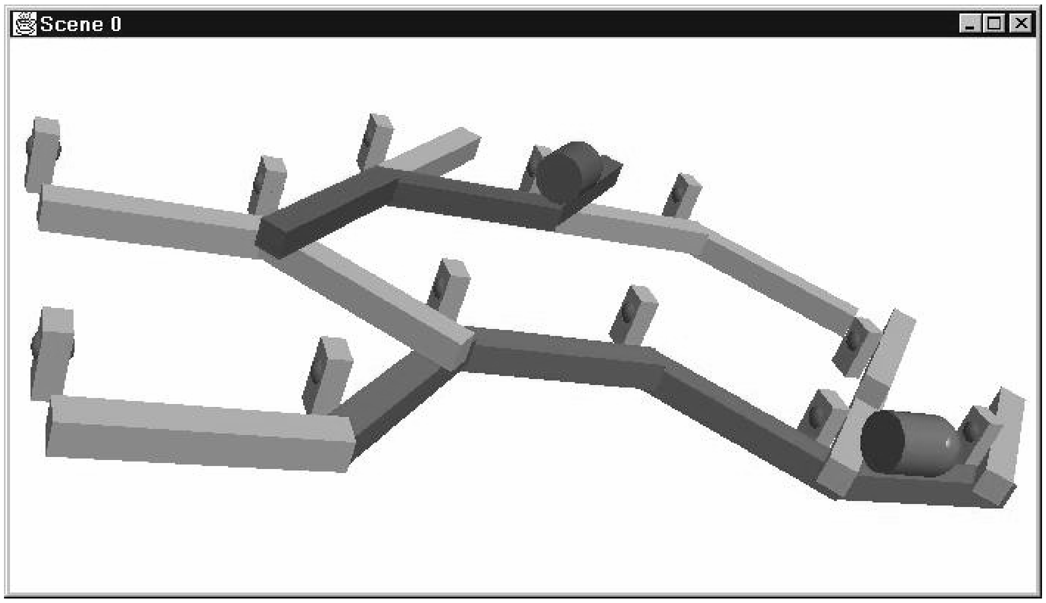


Figure 4.10: A snapshot of the visualization that display the trails of the trains.

train. Notice that the trails do not appear in the state of the computation, but they are computed by the visualization system.

### Advanced Behavior

We explore the use of Add/remove Structural Specifications in different modes and the results that can be obtained through specifying different behavior. The visualization will compute and display the rate of progress for each train. The trails will show the time (number of states) passed since the train was on that track. If the width encodes the time, then the user can compare the relative speed of the trains via the average width of their trails and can also find places where the train waited longer that show up as a larger difference in track widths.

In the scenario, the user is interested in determining the progress of the trains. To accomplish the task, a structural specification named `age` is used. First, the user adds the specification `age=0` in cumulative mode to all tracks currently visited by a train. The cumulative mode ensures that the initialization of the age is performed only once for each track visited by a train. Next the user selects all scene elements that have a specification `age` in cumulative mode and replaces it with the specification `age=$struct.age + 1`. For each element, the operation de-references the value of the age specification and increments it. Due to the cumulative mode, age will be incremented in every new state. After observing the visualization for a while, the user can establish a cutoff value for the age after which the age is not displayed anymore. In relative mode, the user selects all elements that have a structural specification `age=20` and removes the age specification. The selection can be made by clicking with the second mouse button on an elements that has reached the cutoff age and pointing to the age specification.

## 4.7 Concluding Remarks

Supporting the creation of and interaction with application-specific visualizations through direct manipulation enables the use of visualizations tools for many who otherwise may not have the technical expertise to create custom mappings from a program execution to animations. The Reshapeable Visualization system accomplishes this task by using a history list of operations to maintain the user's current view of the computation while it runs. The system is based on a conceptual model that separates the choice of information to be communicated from the graphical representations of that information.

### Advantages of the Data-Driven Interaction Model

The interaction model presented in this chapter is most appropriate for users who explore a computation and are interested in discovering and analyzing various relations among the elements of the computation. In such a scenario, the user might have no particular visualization in mind, and this model allows such a user to delegate to the computer the task of determining a visual representation for the information of interest.

The data-driven interaction allows querying of both the scenes and computation state. The same information space is present in the animations and the computation and is independent of the particular values of the graphical attributes. Users can create visualizations by combining the data already existing in other program views and in the program state. Cognitive economy is improved because the user is not required to convert graphical values to and from data values.

Finally, the interaction model does not require users to have knowledge of computer graphics in order to create and refine animations. The automatic presentation algorithm of the next chapter can automatically manage the graphical design.

### Disadvantages of the Interaction Model

The interaction model can become tedious when a particular graphical appearance must be reproduced by the user. Although the appearance of the picture can be controlled, the need to focus on all graphical details reduces the effectiveness of our approach and makes the creation and refinement of pictures comparable to that of other visualization tools such as [60].

Our approach is not intended for the development of arbitrarily complex visualizations. The interaction model together with the set of operations currently implemented can produce a range of animations that is a subset of those that can be created using code-based visualization approaches that have the computational power of a Turing Machine [25, 74, 88].

# Chapter 5

# Automatic Presentation of Running Programs

## 5.1 Overview

This chapter presents a technique for automatically generating graphical presentations of a program execution. Viewers can customize the presentation and examine particular aspects of the running computation by interactively creating a specification of the program's entities and properties of interest as described in the previous chapter. We identify three goals for visualizations that display consecutive computation states. First, a visualization must present all of the entities and properties desired by viewers and no other information. Second, the graphical representations assigned to various program entities must be sufficiently distinctive to permit viewers to easily recognize entities of different types, despite similarities in graphical characteristics denoting common properties of those entities. Third, to maintain continuity of the animation over time, graphical elements used to present one state of the program must be reserved and subsequently used to represent the same or similar entities or properties in other states. Based on these goals, we have developed an algorithm that assigns graphical objects to each program entity of interest. Attributes of these objects are chosen to present properties of the entity in a consistent manner. The algorithm relies on a characterization of the available graphical objects and attributes to determine the graphical elements that best display the data contained in the entities and their properties. For views that have a greater number of properties than the available number of graphical elements, we have developed heuristics for deciding which properties can be depicted by overloading the same graphical attribute. The automatic presentation is flexible and permits viewers to intervene and determine entirely or partly the graphical design of a visualization.

## 5.2 Introduction

Visualization, as an efficient medium for conveying complex information, is suitable for monitoring the execution of running computations. Visualization has the potential to convey complex program behavior to a viewer. Nonetheless, the use of program visualization is not prevalent in industry and only somewhat more commonly used in academia. One possible reason for this state of affairs is that the process of constructing graphical representations that capture properties particular to the monitored application is complicated and generally requires a user (i.e., visualization builder) to possess knowledge of computer graphics or to be familiar with a particular visualization package. Users typically need to specify both what data and properties they are interested in and how to map them to the particular graphical elements forming the final animation.

We seek to simplify the visualization-building process by automatically producing graphical designs for program visualizations. Users are freed from specifying and understanding low-level computer graphics. Automatic graphical design not only promotes cognitive economy by simplifying the user's tasks and reducing the need to operate with computer graphics, but also provides important assistance to novices and viewers that are unwilling to design the animation.

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

Our assumption is that the information to be depicted is made available for each state of the program by the interactive tool presented in the previous chapter. This information describes both the entities and the relations between them that are to be shown on the screen. A collection of such data is termed a *scene* and does not typically contain graphical information. Instead, in this approach, the content of a scene is automatically translated into a graphical representation. At a high level, the translation can be regarded as a constraint satisfaction problem [33], in which each element of a scene is assigned an element from a domain of graphical representations. From all possible solutions, it is desirable to select one that produces an intuitive and easily understood visual representation of the scene. The technical contributions of this chapter reside in identifying the specific constraints that arise in program visualization and in providing a method of computing corresponding graphical representations.

The constraints that must be satisfied by the automatic presentation algorithm can be described informally as a set of three goals for scene depiction. These goals appear as common features of classical examples of program visualization, such as Stasko's POLKA and TANGO visualizations of sorting and bin-packing algorithms [89, 85, 82], Pavane visualizations of trains, ATM Networks, and elevator controls [74, 73], Brown and Najork's animation of a shortest-path algorithm [17], and Baecker's depiction of sorting algorithms in his "Sorting Out Sorting" video [11]. We note that these program visualizations have a different flavor from the ones created for program optimization. The goals are:

**Expressiveness** Display all the entities and relations of the scene and nothing else. Consider the monitoring of a generic distributed computation. For a given state, a scene describes three computers and one message as entities as well as their x and y position as relations. According to this goal only the four entities are to be drawn and they should be placed according to their position.

**Visual Classification** Distinguish between different kinds of entities. In the previous example, messages and computers would be drawn differently to permit a viewer to easily infer the type of each object. The visual appearance of one data category should be significantly different from the appearance of another's. A graphical attribute like color or length is typically insufficient. Consider the case of a visualization that presents every entity, computers and messages, as spheres with a different color for each category. Such a visualization fails to emphasize the category of each object and makes message passing subject to misinterpretation as computer movement. A better visualization might depict the two categories as two different types of graphical objects, such as small spheres for messages and cubes for computers.

**Continuity** Ensure that consecutive depictions of the same scene present similar data using common graphical features and unrelated data using different features. For example, once a message has been presented as a sphere it will continue to be presented as such at a later time. This also prevents a mobile computer that enters the environment of interest from being depicted as a sphere even in a state when no messages, and consequently no spheres, are displayed.

In the case of large, long running computations the display might run out of graphical attributes available to encode entities and properties. We propose a trade-off that might degrade the clarity of the view, but allows a user to continue to monitor the program through the same visualization. The trade-off consists of techniques for the overloading of attributes and for the re-mapping of program data to graphical values. We propose a novel heuristic to determine when an attribute may be overloaded to encode more that one property. The method analyzes the data up to the current state to establish if two properties are derived from two "unrelated" categories of data. If so, then it is possible for these unrelated categories to share a common attribute without confusing the viewer. For example, both the hard disk drive activity of a computer and the status of a message could be presented using color, with little chance of confusing the viewer as to whether the message status and hard drive activity were related.

A version of the automatic presentation technique was implemented in our program visualization system. With this system, users can interact with the animations to build program visualizations at runtime, refine existing visualizations, and explore various aspects of the computation. Users operate in an environment that permits them to select the program entities of interest and to derive the properties and relations to be observed over time. No graphical elements are required to be specified or calculated by the users because the final picture is automatically generated as described in this chapter. If desired, users can control the automatic presentation system by specifying the graphical design entirely or in part.

In the remainder of this chapter we seek to explain the details of these automatic presentation techniques and to show how they can be used to produce useful visualizations of program executions. In Section 5.3 we provide context by re-visiting related work. In Section 5.4 we define terms and provide detail on the type and form of information that is available about a program that permits automatic presentation. A description of the presentation algorithm follows in Section 5.5, and the heuristic that is used to determine when a graphical attribute can depict multiple relations is explained in Section 5.6. Section 5.7 relates how the user can control the appearance of the visualization. In Section 5.8 we summarize this work, describe its limitations, and discus continuing and future work.

## 5.3 Related Work

Two types of techniques, one designed for information visualization and the other for scientific algorithms, are closely related to the work presented in this chapter. The former includes tools, such as APT [58], Boz [20] and SAGE [76], which focus on the automatic generation of visualizations. APT and SAGE rely on the semantics of the underlying data to generate an appropriate display. In APT these semantics are embodied in the functional dependencies in a relational database. Boz focuses on an analysis of the task which the graphic is intended to support. The **Visual Classification** and **Continuity** constraints distinguish our approach from these traditional automatic presentation tools. Although these tools employ expressiveness and effectiveness criteria [58] to determine the best graphical representation for a given set of data, they are designed mainly for static, relational data. While they can be used successfully to present the relations between data elements (**Expressiveness**), the problem is that no special provisions are made to distinguish the category of information (messages or computers) being shown. Even if the category is considered as an additional dimension of the displayed relations, the distinction between categories is in most cases illustrated by a single graphical attribute (color, length). This might be insufficient for some visualizations. For example, it is possible with these other tools that messages and computers would be depicted as points at the corresponding locations, with messages distinguished from the computers only by virtue of having a different color. The resulting visualization would be less than ideally informative. A further problem with the traditional tools is that if applied to each state of the program in succession, they can lead to a series of scenes in which the representation of a particular piece of information can vary drastically from one scene to another. This drastic change in graphical representation could occur if the range of values associated with a variable appears to change during the course of execution. For example, if all prior message ids had been numerical, one type of attribute would have been assigned to message. However, if a message id appeared that also contained letters, a new mapping from message id to graphical attribute would be selected, perhaps even causing a cascade effect in which other program attributes were also reassigned to new graphical attributes. Such a mid-stream re-mapping from program entities and their relations to new graphical objects and attributes could easily confuse a viewer.

VisAD [43] is an algorithm visualization system that can present arbitrary data types as they appear in scientific algorithms. Scalar data types are mapped by the user to scalar display types. Complex data types, combinations of scalar data, can be assigned a representation by combining the corresponding scalar display types. VisAD maintains **Continuity** in the sense that the representation of the same piece of information does not vary greatly over time, since it is encoded by the same scalar display type(s). But, VisAD does not satisfy **Visual Classification**. For example, if the scalar data that defines the message type and the computer type are similar, such as a tuple $< ID, TIME >$ for message and $< ID >$ for computer, then the appearance of the graphical objects that represent messages and computers might thus be quite similar.

## 5.4 Input Data

As stated in the introduction, we assume the existence of a description of the entities and relations to be depicted. The description necessary for visualization could be automatically derived from program state alone. However, such an automatically derived specification is typically not sufficient to produce meaningful visualizations, as it would include all available entities and relations. More meaningful visualizations can be produced by permitting the user to specify the particular entities and relations of interest. Such a specification was introduced under the name *scene* in the model of the previous chapter. We will elaborate next on the practical importance of scenes by showing how different two views of the same computation can be.

Consider as an example a computation that regulates the transport of components between various assembly lines in a factory. The transport is performed via carts that move independently on a network of tracks. Parts are loaded and unloaded at special stations. The program has a representation of the real world entities (stations, tracks, carts and parts) as software objects and controls these entities by modifying the fields of the objects. For example, the program can have a part moved from a station to the cart by replacing the station name with the cart id. A description of the object (program entity) types is presented in the form of tuple types in Figure 5.1. The state of a station or track (busy or ready) determines whether a cart can enter the station or move over the track. The track is defined by a starting position, the angle at which it is oriented and its length. The angle is also used for carts to define their moving direction.

```
station(name, x, y, state, capacity)
 track(x, y, angle, length, state)
 cart(id, x, y, angle, destination)
 part(x, y, location, destination)
```

Figure 5.1: Types of entities that are manipulated by the transport program.

Users that are interested in understanding the transport computation might need to observe multiple aspects of the program. In general, the same computation can be viewed from multiple perspectives. Two views of the transport program are shown in Figure 5.2 and Figure 5.3. The first view places all entities relative to their positions. The angle of the tracks and carts as well as the length of the tracks are also shown in the view. Color is used to indicate state; the topmost stations and tracks have a darker or lighter color that shows their ready or busy state. Two relations are presented for each entity: the x and y coordinates. Objects with the same coordinate value are shown at the same x or y location. These relations are ordered and impose a distance between its elements. Other relations in the picture are the angle, length and state of some program entities. The second view presents, for each destination, the number of carts and parts that are headed to that destination. Note that although each picture element of Figure 5.2 represents one program entity, in Figure 5.3 an element of the picture may represent multiple program entities, such as all parts headed to station `a23`.

A scene is a description of a program view. It consists of a number of elements, each specifying an item to be displayed. An element is a pair of data specifications and structural specifications. The former represents the subset of the program entities the element stands for, while the latter determines how the element is integrated with other elements to form the scene, more precisely, what relations and properties are to be shown. The entities in the data specifications and program state are expressed as tuples. Structural specifications have the form *name* = *value*, and each name expresses a different logical relation between scene elements. Elements that have a structural specification with the same name are considered logically related either similar or distinct depending on their value. The values may also impose an ordering between distinct elements. For example, two elements that have structural specifications `x=1` and `x=3` respectively can be considered ordered because `1` and `3`

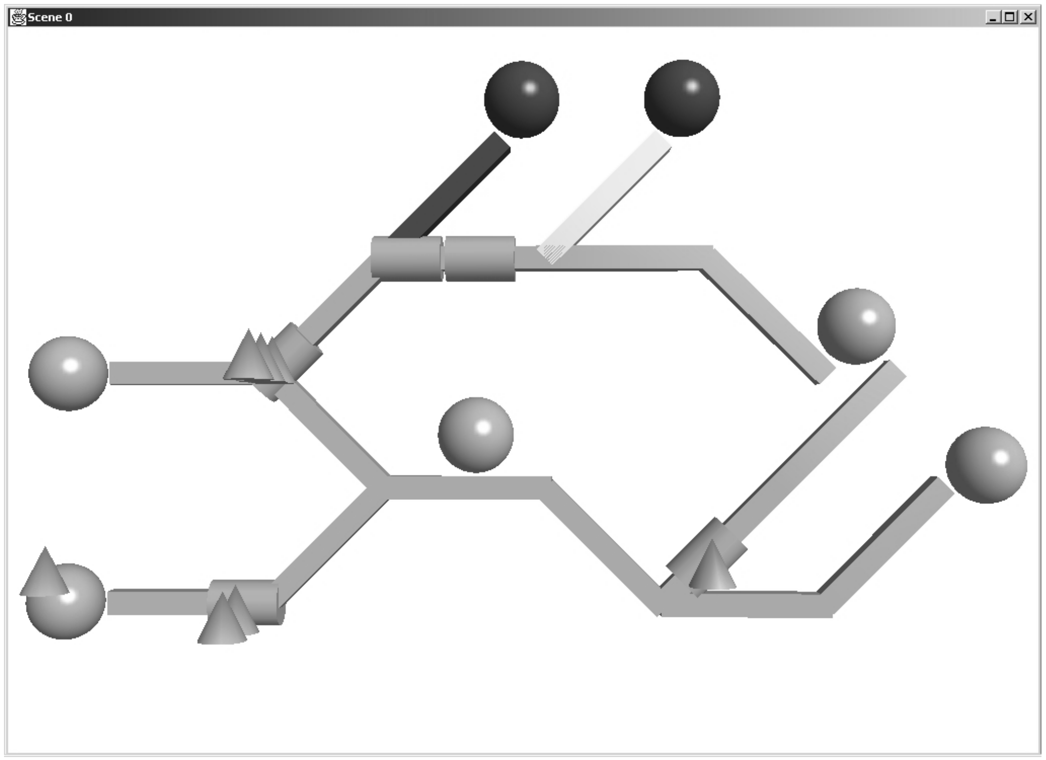


Figure 5.2: A presentation of the transport network. Program objects are shown at their relative physical location. Stations are shown as spheres, carts as cylinders and parts as cones.

are ordered as numbers. This is not true for structural specifications such as `x=ready` and `x=busy`. Table 5.1 and Table 5.2 present some of the scene elements describing the two views of the transport program.

Table 5.1: Three elements of a scene that presents each entity at its position. The first element also displays the state of the station, while the third shows the angle of the cart.

| Element number | Structural specifications | Data specifications |
|---|---|---|
| 1 | `x=3, y=4, state=ready` | `station(a23, 3, 4, ready, 40%)` |
| 2 | `x=7, y=2` | `station(b14, 7, 2, ready, 99%)` |
| 3 | `x=15, y=2, angle=90` | `cart(123, 15, 2, 90, a23)` |
| 4 | `x=15, y=2` | `part(15, 2, b14, a23)` |

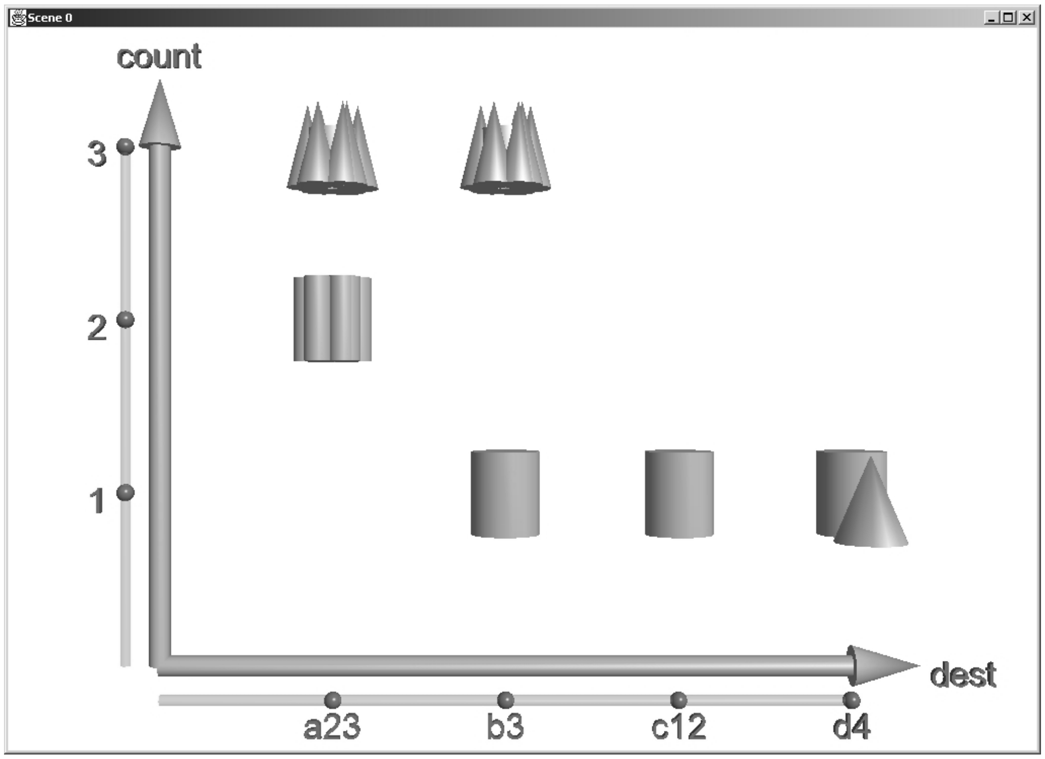


Figure 5.3: A presentation of the transport network. The number of carts and parts for each destination is shown. Carts appear as cylinders, and parts as cones. A graphical object for multiple entities, as shown for `count` greater than one, was constructed from the object representing a single entity, which appears at `count=1`.

Table 5.2: Two elements of a scene that presents a count of carts and parts that are moving to a destination. The program entities of a scene element have the same destination value. `count` is derived from the number of entities that exist in a scene element.

| Element number | Structural specifications | Data specifications |
|---|---|---|
| 1 | `dest=a23, count=2` | `cart(123, 15, 2, 90, a23)`<br>`cart(31, 4, 7, 45, a23)` |
| 2 | `dest=b3, count=3` | `part(2, 2, cart22, b3)`<br>`part(15, 2, cart123, b3)`<br>`part(2, 2.5, cart23, b3)` |

## 5.5 Creation of Graphical Representations

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

Scenes are translated into graphics following a top-down, three-step framework as shown in Figure 5.4. In the first step, graphical objects are chosen to present the elements of the scene. The objects are based on the types that exist in the style gallery. In the second step, graphical attributes of the allocated objects are assigned to the structural specifications of the scene. Across the entire scene, all structural specifications with the same name must be depicted by similar attributes, e.g., color or length. This step is designed to ensure that all relations are shown distinctly, as required by **Expressiveness**, and to choose different types of objects for different types of program entities (**Visual Classification**). In the third step, values are generated for the attributes of the graphical objects. The values of the attributes that present a structural specification must make the relation defined by the specification visible in the appearance of the corresponding graphical objects. In some cases, values of the unused attributes, which present no structural specifications, must also be computed in order to satisfy the presentation goals, specifically to avoid hiding graphical objects (scene elements) and presenting false relations via object intersection (required by **Expressiveness**). A decision made in one step may lead, in a later step, to a graphical design that does not satisfy the presentation goals, in which case the process backtracks and an alternate decision is tried.

The framework can produce graphical representations by analyzing the type of data and relations described by the structural specifications of a scene and by finding the graphical attribute that is most fit to present those relations, as employed in other automatic tools ([58],[20] or [76]). Our approach, however, also makes use of the names present in the scene to build a graphical representation that is more likely to reflect the viewer's intentions. The naming of the program objects and their fields can aid the system in choosing the graphical object for a scene element. The names given to the structural specifications typically show the viewer's mental model for the final visualization. A structural specification that refers to a graphical property, such as `width=4`, is most likely intended to be depicted through that attribute, while a more abstract name, like `idleness`, might not have a clear visual representation in

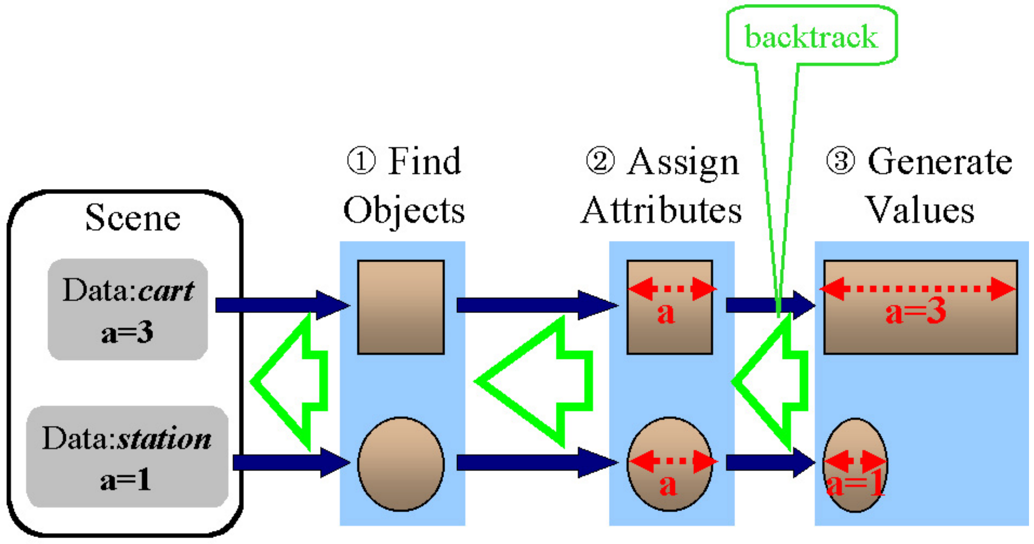


Figure 5.4: An example of a cart and station scene elements as they go through the three rendering steps. They have one structural specification named a and different data specifications.

the viewer's model and consequently no specific choice for the graphical attribute is given.

The graphical representation of a scene can be further enhanced by trying to use as much as possible those objects of the style gallery that can function together to form a coherent view. Such sets of objects, denoted *themes*, were either designed to be used together or were employed in the same past presentation. The sphere and line objects that display an arbitrary graph layout form a theme. Another theme might consists of styles, or glyphs, that are designed to present statistical data such as minimum, maximum, or distribution for a number of variables. Graphical representations should use as few themes as possible because the existence of multiple themes in a picture typically leads to aesthetic and perceptual problems. For example, using statistical styles instead of the spheres of the graph layout raises the question of where a line should be connected to a statistical object such as a "whisker" object (a bar with some external markings for the standard deviation). Trying to connect the line object towards the middle of the statistical object, as designed for the sphere, might interfere with the depiction of the standard deviation.

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

The remainder of this section provides more details for each of the framework steps. First, the steps are described under the assumption that a new scene is displayed. Then, the manner in which the algorithm depicts an updated scene is discussed in a separate subsection. The transport computation is used as an example by focusing on the generation of the first view (Figure 5.2).

### 5.5.1 Style Gallery

The style gallery defines the domain of graphical elements that can appear in a scene presentation. It consists of a set of descriptions of graphical object types. Each description provides a characterization of the data specifications that the style was designed to represent and of the graphical attributes of the object. For the data specifications, typical names of the preferred program entities together with the name and type of their most common fields are stored in the gallery. A graphical attribute has a name and is characterized by one or more of the following.

- The domain of values the attribute can take. This can be an interval as for size, a hypercube for a multi-dimensional domain such as a $(red, green, blue)$ triple for color, a list of values for attributes like texture, or textual for labels.

- The recommended minimum distance between two values of the attribute which are distinguishable by a typical viewer. For textual domains, this value specifies the maximum length of the label. This is not applicable when the domain is a list.

- The default value for this attribute.

- The type(s) of data for which the attribute is intended. As in [58], data can be classified into *nominal* or unordered, *ordinal* or ordered, and *quantitative* or an interval (both order and distance are defined). In addition, the preferred size or the domain itself might be specified.

- The type of attribute: position, size, color, rotation, texture, transparency, label, connection or containment. An object designer can add new types if an attribute does not fall into one of these categories.
- A list of the structural specification names that are preferred to be displayed.

To demonstrate the process of building representations, a style gallery with four object types is assumed to exist. The possible types are sphere, line, cylinder and cone. Some of their characteristics are provided in Figure 5.5. A 3D world is considered because the current implementation of the system is using Java 3D [90].

### 5.5.2 Choosing Styles

An important role in deciding the graphical object fit to represent a scene element is played by the data specifications. The gallery is searched for styles that were designed to display the type of program entities described in the *signature* of the scene element. In the simplest case, when data specifications consist of one tuple, the signature of the element is defined by type of the tuple (also referred to as *class*) and by the name and type of its tuple fields. For example, the signature for an element depicting a track is `track(x:integer, y:integer, angle:real, length:integer, state:string)`. Elements that represent different classes of tuples have as a signature the set of individual signatures of each class.

Scene elements can also represent multiple program entities of the same type. Such objects are depicted by the same style regardless of the actual number of similar tuples. An object that represents two carts in Figure 5.3 should appear the same as an object that represents six carts. The signature of the corresponding scene element is `cart+(x:integer, y:integer, angle:real, destination:string)`. The '+' sign symbolizes the presence of multiple tuples of the same class, in this case `cart`. Notice that this is different from the signature of an element that represents only one cart.

For each type of scene element, the algorithm searches the style gallery to find graphical objects that match the signature of the type. Styles are filtered by the requirements listed below. If they do not pass a filter, they are not considered for the next higher numbered filter. The set of styles that pass the highest-numbered filter are forwarded to the next step in which only one of them is selected to display all

| | | |
|---|---|---|
| **Sphere:** | | Preferred data specifications: `vertex(id:integer), node()`; |
| | | Attributes: `x, y, z, radius, color`; |
| | **x, y, z:** | Domain: $[-100, 100]$; |
| | | Minimum distance: 0.5; |
| | | Default value: 0.0; |
| | | Type of attribute: positional; |
| | | Type of data: domain with more than 2 values. |
| | **radius:** | Domain: $[0.5, 15.0]$; |
| | | Minimum distance: 0.5; |
| | | Default value: 1.0; |
| | | Type of attribute: size. |
| | **color:** | Domain: cube between $(0.0, 0.0, 0.0)$ and $(1.0, 1.0, 1.0)$; |
| | | Minimum distance: 0.5 (Euclidean distance); |
| | | Default value: $(0.2, 0.2, 0.2)$; |
| | | Type of attribute: color; |
| | | Type of data: nominal with less than 10 values. |
| **Line:** | | Preferred data specifications: `edge(), link()`; |
| | | Attributes: `x, y, z, orientation, length, color`; |
| | **orientation:** | ... |
| | | Structural specification names: rotation, angle, slope; |
| | | ... |
| **Cylinder:** | | Preferred data specifications: none; |
| | | Attributes: `x, y, z, orientation, length, color`. |
| **Cone:** | | Preferred data specifications: none; |
| | | Attributes: `x, y, z, color`. |

Figure 5.5: A simple style gallery. The complete description of the sphere is presented. For other styles, only excerpts are shown.

scene elements of that type (with the same signature). If all styles fail the first two filters, the algorithm decides that it is not able to show the scene due to the lack of appropriate graphical objects.

1. Only styles that are not already assigned to other signatures are considered. Condition **Visual Classification** requires that different kinds of graphical objects must represent different types of scene elements.

2. Only styles that can represent all objects with the given signature are chosen. This is determined by the capacity of a style to depict all the relations present in the structural specifications of those objects, i.e., there are enough attributes for all the names in the structural specifications.

In the transport visualization, the scene elements for parts are members of the `x` and `y` relations. The stations contain three structural specification names: `x`, `y` and, for some stations elements, `state`. The carts are involved in `x`, `y` and `angle` relations. The tracks must present `x`, `y`, `state`, `angle` and `length`. In this case, all styles can display any type of scene element. The only exception is `Cone` which has only four attributes and therefore cannot represent some of the tracks.

3. The algorithm prefers styles that match the signature if the '`+`' sign is ignored. An new object type is created by replicating the object of the style as shown in Figure 5.6. The resulting object has the same graphical attributes as the original. A "glue" of the same color is placed between the replicated objects.

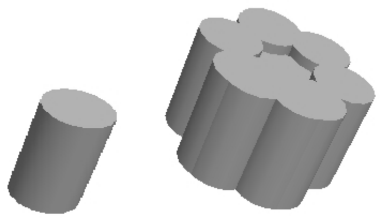


Figure 5.6: To the right, a style for depicting multiple tuples of the same class is constructed from the style on the left, which was designed to display a single tuple of that type.

4. Styles that are designed to represent the given signature are given preference. The signature recorded in the style must be a subset of the signature of the scene element. The same classes must be present in both signatures, although the element might have fields that do not exist in the style. In this case, it is very likely that the style was designed for precisely the kind of program entities that are to be displayed. Notice that additional fields might be present in the element signature. These extra fields are allowed for the case of object-oriented programming when an object of the computation might be a specialized version of a super-class. For example, "movable station" might be a sub-class of "station". A style designed to represent a station may also be used for a movable stations.

   In our example, the signatures of the styles (Figure 5.5) do not match any of the signatures of station, cart, part or track types of scene elements.

### 5.5.3 Selecting Graphical Attributes

This step of the framework assigns a graphical attribute to each structural specification name. The input consists of scene element types and, for each type, of the set of styles that are possible candidates for displaying the type. Once the graphical attribute that will be used to show a relation (a structural specification name) is determined, the algorithm decides on only one style for each type of scene element.

The input in our example is four types of scene elements (cart, part, station and track), each with its own set of styles. The sets for the first three types are the same {`Cone`, `Cylinder`, `Line`, `Sphere`}. The set for track is {`Cylinder`, `Line`, `Sphere`}. Recall that `Cone` cannot represent some tracks because it does not have enough attributes.

One relation (structural specification name) of the scene is displayed through one type of graphical attribute, such as color or width, regardless of the style used for presenting individual scene elements that are part of that relation (have a structural specification with that name). The algorithm examines each possible assignment of attribute types to the set of structural specification names. The attribute types to be considered are the ones that belong to one of the input styles. Out of the *correct* assignments, the algorithm selects one that can depict the relations most *effectively.*

The *correctness* of an assignment is decided by verifying all the factors listed below.

1. Different attribute types must be assigned to different structural specification names. This ensures that a viewer can distinguish two separate relations and therefore that requirement **Expressiveness** is satisfied. For example, the `length` and `state` relations cannot be encoded by a single type of attribute such as size.

2. Attributes already assigned to other relations in the current visualization session are not used (requirement **Continuity**). The algorithm can be simplified by considering only unassigned attribute types.

3. For each type of scene element in the input, there must exist an input style that is capable of depicting all elements of that type. That style must contain every attribute that is assigned to a structural specification name appearing in an element of that type.

Consider in our example a situation in which the length attribute is assigned to the `length` relation, and size to the `state` relation. In this situation, there is no style that can depict some of the tracks, because there is no style that has both length and size types of attributes. Such an assignment is not correct.

4. In order to satisfy the **Visual Classification** requirement, there must exist a *style solution*, a set of distinct styles such that there is one style for each type of scene element. A style of this set is both associated in the input with a type of scene element and capable of presenting all elements of that type, as explained in the previous paragraph. Notice that it is possible to have multiple style solutions for a single assignment.

   A possible assignment for the transport view is z-coordinate to `x` relation, x-coordinate to `y`, y-coordinate to `state`, length to `angle` and color to `length`. A possible solution is `Line`, `Sphere`, `Cone` and `Cylinder` for cart, part, station and track respectively. The other solution is `Cylinder`, `Sphere`, `Cone` and `Line`.

The *effectiveness* of an assignment is evaluated separately for each of its style solutions. This measure estimates how good a solution is by quantifying the fitness of each attribute, for each solution style, to depict their assigned relation. The fitness is derived from Mackinlay [58], where, for each kind of data (nominal, ordinal or quantitative), perceptual tasks (graphical attribute types) are ranked according to their efficiency in presenting that data. The fitness of a graphical attribute for a given relation is its ranking in displaying that relation's data. If the attribute has a description of the preferred data type in the style gallery, that description is considered to have the first rank, followed by a default ordering of attributes as in [58].

In the first example, consider the use of length to present the `angle` relation. The data in this relation is quantitative, and according to Mackinlay[58] length is the second choice. Because there is no description for preferred type in the style, the fitness for this choice is 2. Assuming that there are two styles (one for carts and one for parts) that use length to depict `angle`, this adds 4 to the effectiveness of an assignment. In the second example, let the y-coordinate represent `state`. The values of the state are ready and busy, which suggests nominal data. The default ranking for such data is 1, as positional values are preferred for nominal information. However, the y-coordinate has a preferred type in the style gallery, for which the values of the `state` do not qualify. So, the y-coordinate is actually the second choice for the `state` relation. Again, this adds 4 to the effectiveness of the assignment when two styles use y-coordinate to show the state.

In order to reward the use of attributes that have the same name as the encoded relation, the fitness of such a choice is considered to be zero. Zero fitness is also awarded to attributes that depict their preferred relation name as described in the style gallery. So, if the x-coordinate is used to depict relation `x`, the fitness is zero, and nothing is added to the effectiveness of the assignment. The same holds for orientation as a choice for depicting `angle` because "angle" is listed as a preferred structural specification name (Figure 5.5).

The solution that has the lowest sum of fitness values is the one chosen in this step. Note that the sum is adjusted to discourage the use of multiple themes. In our example, the lowest effectiveness ranking is obtained when the `x`, `y` and `length` relations are depicted through the attributes with the same names. The use of orientation for `angle` is also rewarded, adding zero to the effectiveness. The use of color for `state` adds only two to the effectiveness. This assignment is shown in Figure 5.2.

### 5.5.4 Choosing Values for Attributes

The input for this step consists of the mapping of each scene element to a style as well as a mapping of each relation (structural specification name) to a graphical attribute type. At this point, the algorithm must decide how to display individual scene elements, especially their structural specifications. This is the same as setting the attributes of the graphical objects that form the final visualization. A graphical object corresponds to a scene element and is an instance of a style associated to that element.

Some attributes of a graphical object might display structural specifications that exist in the scene element, while other attributes might be unused, do not have a related structural specification. For example, in Figure 5.2, color is not used for the stations on the left, but it presents the `state` specifications for the ones on the top. Values are set for all attributes, showing either a structural specification or the default value. In the former case, values are generated if the data to be conveyed is nominal or ordinal. For quantitative data, the value of the structural specification is copied directly to the value of the graphical attribute that displays the specification. If, however, the structural specification values are not in the domain of the graphical attribute as described in the object's style, different values for the attribute must be generated.

Most attributes that do not present a structural specification are set to the default value specified in the style of their object. Position, size, length and transparency types of attributes have default values computed based on the overall appearance of the visualization. If possible, the default values of such attributes can be adjusted to ensure that all objects (scene elements) are visible and that no unspecified relations are shown through object intersection as required by **Expressiveness** (unless connection or containment type of attributes are assigned). The situations where the default values are computed are given below.

- Objects with unspecified position must use a layout scheme to be spread out on the screen. We prefer to show each class of elements in their own region on the screen, e.g., all un-positioned carts in the upper region and all un-positioned stations in the lower. Advanced algorithms for the layout, such as those for graph layout, can be plugged in the automatic presentation. Currently, we only implemented a random layout scheme.

- An object that completely encircles another and has the position and size already set can be made more transparent to allow the occluded object to be viewed. This might have been the case if the cylinders (carts) were smaller than the tracks. The tracks could have been made transparent so that the carts looked as if they were moving inside a pipe.

- Objects can be prevented from overlapping by reducing the length or size of unused attributes. This is not possible when objects lie at the same position, or more generally when some of their internal axes intersect as it is the case for carts and tracks. An internal axis is always inside the object regardless of the values of unused attributes.

  In both Figure 5.2 and Figure 5.7, it can be seen that the length of the carts is set to prevent the two uppermost carts from overlapping.

- Objects lying at the same position can be separated by listing them on an unassigned dimension such as the z-coordinate. Although this introduces a false relation among objects, it is still useful and is employed only when multiple objects hide each other. In Figure 5.7, the z coordinate is employed to list the parts carried by one cart because they are at the same (x, y) coordinate.

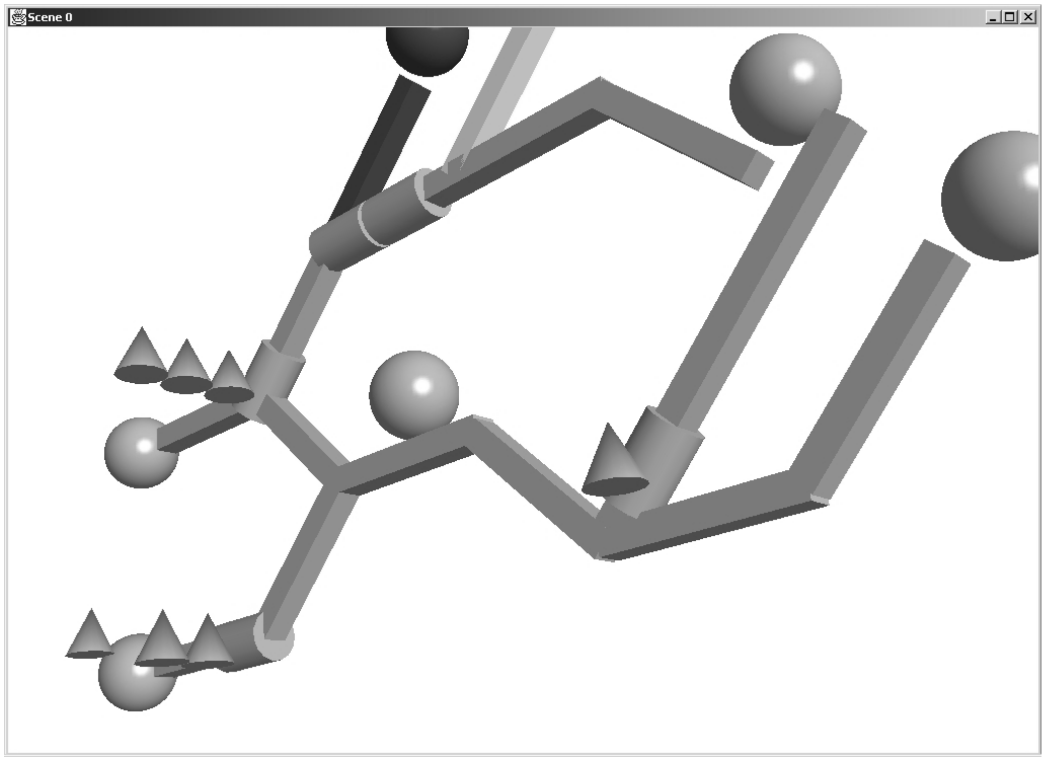


Figure 5.7: The presentation of the transport network as seen from a different viewpoint in the 3D world. The parts that are in the same cart or station are distinguished on the z-coordinate.

The system determines values for positional attributes last, preceded by the value for size attributes, preceded by transparency, preceded by the other types. Default values for non positional attributes are the same for all objects of one style (e.g., all carts have the same length), yet different from any values used for displaying structural specifications. Default values for positional attributes might be particular to each object if one coordinate is used to distinguish among occluded objects. In Figure 5.7, cones do not have the same default z-coordinate.

### 5.5.5 Repainting a Scene

The presentation of the algorithm thus far assumed that the content of the scene was new, more precisely, that the types of scene elements and names of relations were encountered for the first time. Typically, a scene is frequently updated to present new states of the program and to react to changes made by users. The new visualization frame has a large number of graphical features that are similar to the previous ones.

This is because a large portion of the new content of the scene is copied from the previous content, and according to **Continuity** common graphical features should be employed.

A registry is maintained by the system to record the mapping between scene entities and graphical elements. In effect, the decisions that lead to a successful visual representation are recorded. The registry keeps the style associated to each scene element signature, the graphical attributes assigned to a relation, the mapping of structural specification values to graphical attribute values, and old default values of unused attributes. For recorded scene element types, the first step of the framework associates only the style from the registry to that element type. In the second step, known relation names are automatically assigned the graphical attribute previously used.

In the third step, the values for already seen nominal data are kept the same, but it is possible that different graphical attribute values are generated for ordinal and quantitative data, even if that data is present in the registry. Since ordinal data has an implicit ordering between values, the insertion of new values invalidates the previous ordering. This requires the system to re-assign graphical values to display the new ordering, with the elements at equal distance. For example, let the carts be qualified by 'bad', 'fair' and 'good' attributes which are shown using width 2, 4 and 6 respectively. If a new qualification 'so-so' is added between 'fair' and 'good', the system should use width 6 for 'so-so' and width 8 for 'good' to order carts uniformly. Quantitative data does not require the elements to be at equal distance, although values that are out of the attribute's domain may appear. In that case, all data values need to be re-encoded, in effect scaled, through different attribute values to show the relative distance among them. Finally, non-positional default values are kept the same for the new visualization, while default positional values might be adjusted. In Figure 5.7, when the number of parts carried by a cart changes, z-coordinates are updated to allow all parts to be displayed.

## 5.6 Graphical Attribute Overload

Data overload occurs when the amount of information to be presented exceeds the available graphical features. Overload can be detected in any of the three steps, when all styles have been used, when there does not exist a correct assignment of attributes to relations, and when all the values in the domain of an attribute have been exhausted. If data overload occurs, the visualization session may have to stop, or users might be required to manually design the entire graphical presentation. Heuristics that can alleviate the last two problems, albeit by sacrificing the clarity of the visualization, are presented next.

If no additional graphical values can be produced in step three, when values of the structural specifications are encoded, the system can continue displaying the visualization by showing multiple data values through the same attribute value. For example, if stations have a large number of states, the same color might need to encode more than one state. Nominal and ordinal data is evenly distributed across the values of the attribute. That means that each attribute value encodes about the same number of data values. In contrast, quantitative data is scaled to fit in the domain of the attribute. It is possible that one attribute value encodes most of the data, yet viewers are not able to infer this from the visualization. We are studying the use of legends, like the ones that appear on road maps, to display the distribution of the data values. Legends are the subject of the next chapter.

In step two, if no graphical attribute is found for a new structural specification name, the system looks for an attribute that is assigned to a specification name *unrelated* to the new name. Two structural specification names, A and B, are unrelated if both of the following conditions are true. First, at all times during the current program execution, the data specifications of any scene element that contained a structural specification named A had nothing in common with the data specifications of any elements that contained B. That is, the classes of tuples of such two elements were distinct. Second, at all times during the execution, no structural specification named A was derived from the same subset of the program state as a structural specification named B.

The first condition does not allow two properties of one scene element to be presented through the same graphical attribute, since the data specifications of the element are common to both properties. Furthermore, this condition prevents the same style from showing two relations through the same attribute. In our example, it is not possible for color to present the state for some stations (spheres) and the capacity for other stations. It would be impossible for viewers to distinguish when color represents state and when it represents capacity.

The second condition is aimed at preventing two different relations that are changing synchronously from being encoded with the same attribute. Because the two relations are derived from the same program data, a change in the data is likely to modify both relations. If the relations were shown by one graphical attribute, the synchronized modifications would make it difficult for a viewer to perceive that the two are different properties.

No two relations encountered in our example can be encoded through a common attribute. Consider a new relation `capacity` that shows the percentage of used capacity of each station. It is possible in Figure 5.2 for `capacity` and `length` to be shown through the same attribute. Under the assumption that spheres have a length, the resulting visualization depicts stations as ellipsoids whose length varies with the capacity. This visualization might prove to be clear enough for a viewer to distinguish the meaning of the track's length from the meaning of the station's length.

## 5.7 User Control

Users can control the presentation through the choice of structural specifications, by editing the registry and by creating custom styles and themes. The exact name and value for a graphical attribute can be given in the structural specifications. As described in Section 5.5, the algorithm encodes a structural specification by the attribute with the same name whenever possible. Moreover, the value of the specification is copied to the value of the attribute when the former is in the domain of the attribute. Users can also control the style for each scene element. A special structural specifications named `STYLE` is interpreted by the system as a request for that particular style. As such, a scene element that has a specification `STYLE=Sphere` is shown as a sphere.

To override the decisions of the system, the registry can be edited. Beyond a textual editor, legends provide a direct manipulation alternative for editing the registry.

## 5.8 Concluding Remarks

The technique described in this chapter simplifies the construction of visualizations that depict the execution of a program. The viewer specifies, via interaction with animations, the program entities and relations to be observed over time. The presentation technique automatically designs visualizations to depict the sequence of states of the running computation as specified by the user. The result is an animation that interpolates smoothly between representations of each state. We introduced three requirements for such visualizations. First, the pictures must express all and only the entities and relations specified by the viewer. Second, various types of program entities must be easily classified through their visual appearance. Third, similar information displayed in multiple frames of the animation should not have a radically different representation. For large amounts of data, these requirements are relaxed to permit the viewer to keep using the same program view.

The presentation algorithm relies on a style gallery that defines the graphical elements available for a visualization. A style provides the algorithm both with a description of the capabilities of its graphical features and with a characterization of the type of information the style was designed for. The system produces a graphical representation by finding the styles that match the information specified by the viewer and combining them into a picture.

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

# Chapter 6

# Interactive Legends

## 6.1 Overview

The benefits of program animations may be increased by adding to the program view a visual representation of the data-to-graphics encoding employed by the animation. This chapter focuses on interactive legends that provide both an economical format for conveying a mapping and a widget through which the mapping can be adjusted by users. Legend keys promote a continuous style of interaction that allows users to adjust the appearance of the observed computation according to their understanding and interest. We show the flexibility of legend keys by using them to query the information based on the properties of interest and to focus the presentation on the program objects and properties relevant to the current task.

## 6.2 Introduction

The effectiveness of program animations may depend on the existence of an explicit representation of the *visualization syntax*. The syntax refers to the graphical means through which the information of a running program is visually communicated, and it can be regarded as a mapping between the abstract data of the underlying program and the graphical features of the visualization. In the experiments presented in Chapter 3, the use of legends to depict the syntax is among the factors that appeared critical for animations to benefit the user in performing a problem-solving task.

Traditionally, program animations included little explanation of the visualization syntax and did not regard the mapping as part of the program view. The missing syntax representation raises problems related to an increased cognitive effort and to a reduction in the elements that can be "naturally" manipulated via direct interaction. First, the increased cognitive load is probably a consequence of the user storing the data-to-graphics mapping in the working memory. Otherwise, the user risks not understanding the processes conveyed by the animation because of the inability to relate animated graphics to the abstract concepts in the observed program. A poor understanding of the syntax may result in overlooking important events in the program, in identifying false problems, or in overloading the user with questions such as "is that rectangle supposed to be that wide for a list with two elements?". Second, the lack of a syntax representation may restrict the user's ability to adjust the data-to-graphics mapping. This mapping is the core of the visualization process because it enables users to control the appearance of the program view and to query the subset of the computation that is of current interest. The user may be able to modify the syntax through slow, discrete widgets, such as textual input for the exact values of graphical attributes, or through widgets that can specify a single range at a time, such as sliders.

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

In our approach, legends are regarded as an integral part of visualizations. As such, legends have a dual role, as both a presentation mechanism and an interaction element that can be modified, included or removed as a result of user actions and program execution. The main contribution of this chapter is the design of an interactive widget that is based on traditional legend keys. The widget is expressive enough to convey the choices of the automatic presentation technique of the previous chapter. The discussion presented here focuses on continuous adjustment of the visualization syntax. More discrete operations, such as combining the elements of a key or transferring keys between different scenes, are the subject of our early work in [94]. They are not included in this chapter because of their different purpose and appearance.

The versatility of the interactive legends is demonstrated by considering solutions to query and focus+context problems. Legends can be used to specify (query) the elements of interest based on properties of those elements. Focus+context refers to a range of techniques for handling large quantities of information that cannot be properly depicted at once (see Figure 6.10 for an example of focus+context). The visualization shows only a subset of the entire information space in detail. In order to maintain context, the remaining data is also included in the visualization in an abbreviated form that can be accommodated with the graphical attributes available.

The legend key widget is general enough to express relations between arbitrary domains. The legends can present relations concurrently between more than two domains, although we typically employed the widget with two domains: a graphical attribute and a data property.

The next section presents related work, followed by a description of the appearance of our legend keys in Section 6.4. Section 6.5 presents the style of interaction with the key widget. The use of legend keys for querying and focus+context is illustrated in Section 6.6. Conclusions and future work are presented in Section 6.7.

## 6.3 Related Work

Legends are commonly used in visualizations and charts in the fields of scientific and information visualization. Office tools include legends in the charts they produce. However, the interaction with these legends is limited at best to coarse selection tasks, such as clicking on a legend key of a chart and highlighting the entire series depicted in the chart. Our approach emphasizes continuous manipulation of the legends. Similar continuous flavor appears in dynamic queries, zooming interfaces and focus+context techniques, all applied to handle large or complex data spaces.

Dynamic queries [9, 80] are a class of interaction widgets that allow the selection of an interval of values for a data attribute. The user can control the upper and lower limits for any attribute by grabbing and moving those limits directly on the widget. In response to the user actions, the visualization promptly presents those elements whose attributes fall within the specified range. For example, a user can effectively slice the information space, continuously move the slice, and set its size. Because of the natural interaction style, many variations and extensions of dynamic queries were developed, such as the widgets in the Attribute Explorer [95] by Tweedie *et al.*, where an overview of the data distribution is drawn on the widget itself, or the Verizon Laboratories' EZChooser [98], which emphasizes the querying power of dynamic queries although it sacrifices their continuous nature. We integrate dynamic queries into the design of interactive legends, which provide the ability to adjust not only an interval, but to adjust the mapping itself.

Zooming interfaces such as Perlin and Fox's Pad [63], or similar user interfaces like Pad++ [13] and Jazz [14] developed by Bederson *et al.*, are intended to provide smooth zooming into the components of a user interface. The continuous nature of these interfaces is different than the continuous manipulation of legends.

Focus+context techniques include FISHEYE view [34], Table Lens [67], Magic Lenses [31] and SDM [21]. FISHEYE view, through a spatial distortion, adjusts the position and size of objects, but does not handle other graphical attributes. Table Lens, as the name suggests, is designed for tabular data representations. Magic Lenses are capable of emphasizing various information space ranges, but does not allow the specification of more complex constraints such as properties that depend on each other. SDM allows the modification of the parameters of graphical objects via direct manipulation of those objects. The appearance and positioning of the set of interest can be modified to allow an easier analysis of the set. Although SDM provides clues as to the relation between the selected objects and the rest of the picture it does not convey the data to graphics mapping. SDM does not directly support querying of the elements based on their properties as well as the encoding of a data attribute through a segmented function such as the one in Figure 6.2.

## 6.4 Legend Keys as Displays of the Visualization Syntax

### 6.4.1 Basic Legends

Legends consists of a number of *keys*, each presenting a relation between data and graphical domains. Basic legend keys, such as the ones in Figure 6.1 and Figure 6.2, consist of a data thread and a graphics thread. Threads encode a domain of values and are generally depicted as lines. The threads of a key are parallel, commonly either horizontal (Figure 6.1) or vertical. Threads can be labeled as in Figure 6.8. On the thread, a number of discrete values are shown by ticks. A tick is labeled by the textual value of the discrete point or by a graphical representation. A graphical representation is always used on the ticks of a graphics thread. Typically such a thread conveys a graphical attribute. Each tick label visually conveys a value for that graphical attribute. In Figure 6.1, each tick on the graphics thread has a label with a different height.

Threads encode discrete points or continuous domains. For threads encoding discrete points, each value of the domain has a corresponding tick, while for continuous domains, the ticks sample a range in the domain. We concentrate on the continuous case in the discussion that follows, noting that similar issues apply to discrete domains.

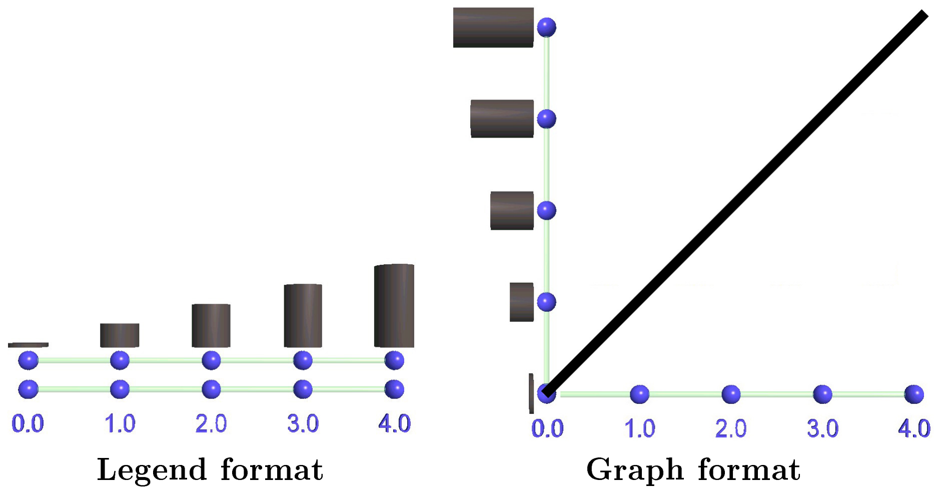


Figure 6.1: A basic legend key is shown together with the graph format that corresponds to the same function. A linear function is displayed.

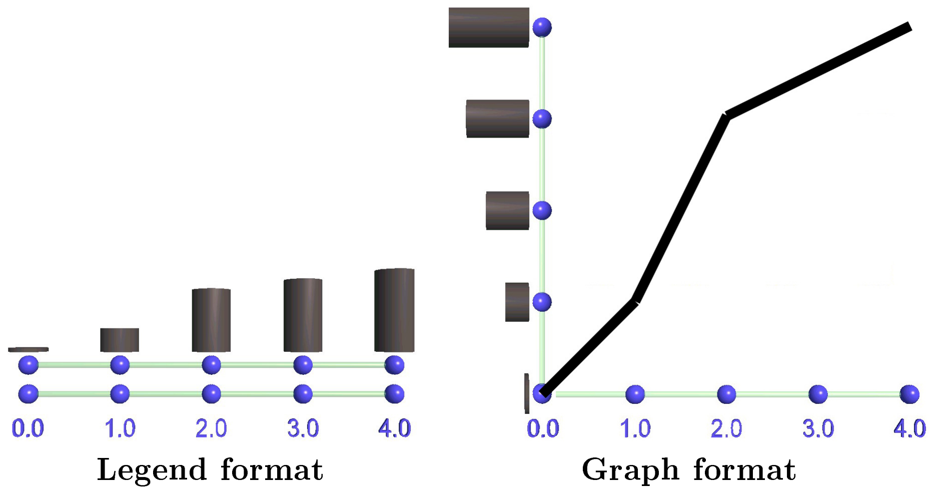


Figure 6.2: A segmented linear function is shown in both legend and graph formats.

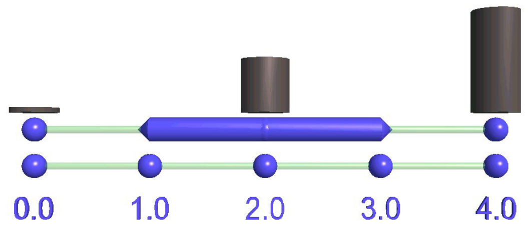
A legend key is a brief representation of a mapping. Mappings are also customarily illustrated with a graph, but the legends may be better because they reduce estimation error and occupy a smaller space than the corresponding graph format, as seen on Figure 6.1 and 6.2. The appearance of the two functions in key and graph formats is illustrated those figures. For a visualization user, it is important to establish how certain data is visually encoded, or conversely to estimate the data value that a graphical object is representing. For this kind of tasks, legends may reduce the estimation error because of the placement of the two domains next to each other. In a graph, the domains are displayed on the coordinate axes, and the estimation is performed by tracking a course parallel to one axis until the function mark is encountered and then, by tracking another course parallel to the other axis.

The type of functions that can be represented without loss of precision by a basic key include functions that appear as a segmented line on a graph. The best depiction is obtained by creating a tick for each point where two segments join. Alternatively, a legend key can approximate more complex functions and relations via interpolation. The coarseness of the interpolations is dependent on the display space available for the key, as more detailed sampling produces keys with more ticks and is consequently longer. The keys presented in Figure 6.2 can be reduced to only three ticks by eliminating the data ticks 1.0 and 3.0 with their corresponding graphics ticks. The representation of the segmented function becomes an approximation of the actual function, but it may be enough for estimating the visualization syntax.

### 6.4.2 Extensions of Basic Legend Keys

Basic keys are extended to accommodate constant functions and inverse of constant functions (appear as horizontal respectively vertical lines in a graph form), to depict relations among more than two domains, and to provide control over coordinate axes. Functions that have a constant segment and the inverse of such functions associate one value to a range of values. The relation is made more apparent in a key by replacing the spherical representation of a tick with a cylinder that spans the desired range (Figure 6.3). A cylinder marks only one value on the thread it lies on, but because of the cylinder's increased width, the wide tick can show a relation with an entire range of values on the other thread.

Multiple threads added to the same key are able to concurrently present mappings among more than two domains. An example of such a key is given in Figure 6.4.



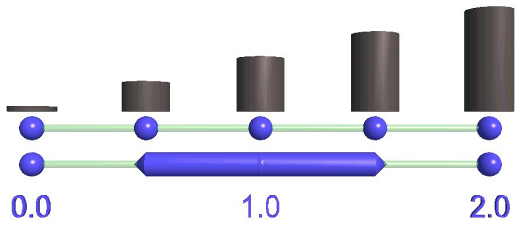


Figure 6.3: Two legends that present a function with a constant segment, in which a range of values are mapped to the same graphical feature (left) and a relation that contains the inverse of a constant function, in which one value can be depicted through a range of graphical features (right).

A viewer can estimate the relations between any of the three domains in that figure. This type of key extension can be employed for graphical domains with multiple dimensions, such as color that has red, green and blue dimensions, and for advanced querying of the information space.

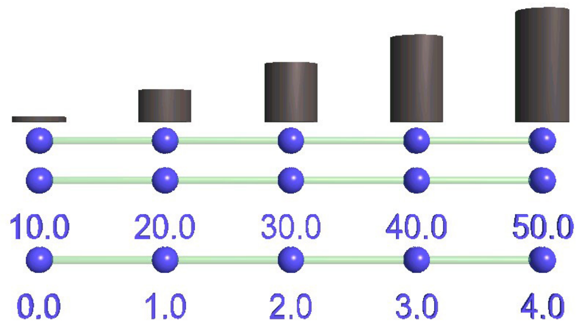


Figure 6.4: A key that presents concurrently the association among three domains.

Coordinate axes can be regarded as a legend key because they depict the relation between a spatial dimension and a data value. The design of a basic key is slightly modified to resemble a coordinate axis, as illustrated in Figure 6.5. The ticks on the graphics thread have no explicit label, yet they mark the location in the visualization that have the same position as the tick on that axis. Note that the ticks of the position thread can only mark a single value, therefore they have only a spherical representation.

### Automatic Presentation of the Visualization Syntax

The automatic presentation algorithm of the previous chapter can present the generated encoding via basic keys and coordinate axes. The algorithm maintains a registry

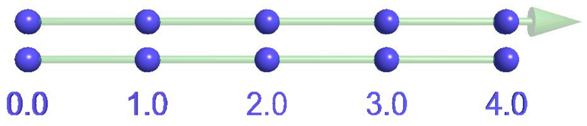


Figure 6.5: A key extension to be used as a coordinate axis. An arrow is added at the right end of the graphics thread. A graphics thread marks a hyperplane with the same position on the axis as the tick.

in which three types of mappings are recorded: tuple types to graphical objects, structural specifications to graphical attributes, and values of the specifications to values of the graphical attributes. The first mapping is discrete and is depicted as a basic key with textual labels on the data thread and with sample graphical objects on the graphic thread. For the second mapping both threads have textual labels. Finally, the third mapping is presented through multiple keys, one for each graphical object–attribute pair.

The focus of our implementation is on the legend keys for the automatic presentation algorithm. Basic keys, which are sufficient to present the registry, are fully implemented, while the extensions of basic keys are provided as an API that can be programmed into a Java 3D application.

## 6.5 Interaction with Legend Keys

Keys can be manipulated through discrete operations, such as adding and removing ticks and threads, or requesting an even spacing for ticks, and through continuous adjustment of the mapping. The continuous adjustment is performed by sliding the ticks and threads and by adjusting the range of cylindrical ticks.

We identify the following three goals for the continuous manipulation of legends.

- to update both the animation and the elements of the keys affected by a user gesture while the gesture is performed;
- to provide cues as to all elements of a mapping that are being changed;
- to maintain the consistency of key, which include preserving the ordering of a continuous domain.

During a user gesture, the ticks that lie in the path of the movement are generally raised above the thread. The user can complete the move without becoming confused with regard to the element being manipulated. In Figure 6.6, which incidentally presents how the linear function in Figure 6.1 can be transformed in the segmented function of Figure 6.2, the graphics tick that is being dragged appears highlighted. The dragging caused the graphics tick of data value 1.0 to be raised. The raising mechanism provides feedback to the user as to what parts of the mapping are changed and consequently is also employed to show a that a dependency between two ticks is changed. In Figure 6.7 the graphics tick for 2.0 is raised and moved in tandem with the data tick. Note that the element grabbed by the user is the data tick.

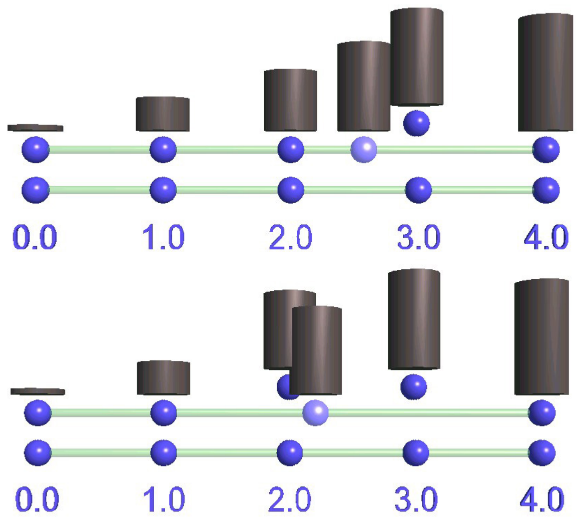


Figure 6.6: Two snapshots of an interaction with the linear legend of Figure 6.1 by moving the graphics tick of 3.0 to 2.0. The element grabbed by the user is highlighted. It can be seen that a new graphics tick was created for 3.0 (top) and that the graphics tick for 2.0 was raised when the user gesture brought the manipulated tick close to the graphics tick for 2.0(bottom).

The ordering in a domain is maintained by a pushing mechanism. When the tick being manipulated "touches" another tick of an ordered domain, the other tick is also moved as if the two ticks were beads on a string. This prevents the two from switching order, as it would happen if the second tick were raised. Raising is replaced by pushing for ordered data threads. In Figure 6.7 it can be seen that the data tick 3.0 was pushed by the leftward movement of tick 2.0. Also note the raised graphics tick which points out the change in the mapping of 2.0 and 3.0 to heights.

The key always has a graphics tick associated to a data tick for ordered graphical domains. This feature permits the user to always observe how data is encoded

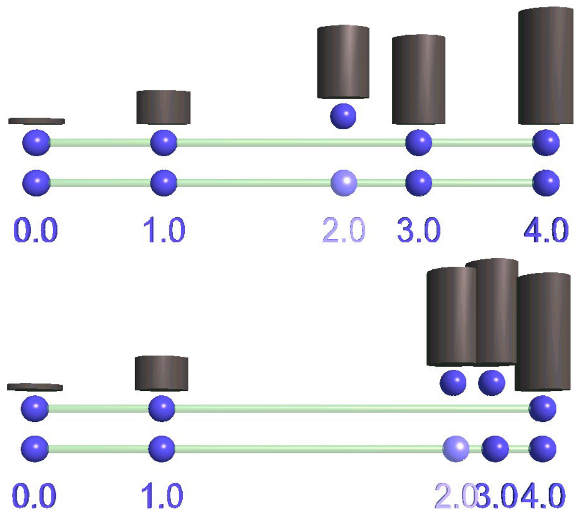


Figure 6.7: Snapshots of an interaction with the linear legend of Figure 6.1 by moving data tick 2.0 to the right. The top figure shows that a graphics tick is always maintained for a data tick. The bottom figure shows both ticks for 3.0 pushed together with 2.0.

visually. A graphics tick can be moved away by the user, as illustrated in Figure 6.6. The widget, however, creates a new graphics tick for the "orphan" data tick, and updates the new tick in a normal fashion. For discrete domains, the tick is not created because its value (or appearance) cannot be inferred by analyzing the neighboring ticks.

## 6.6 Using Legends

The purpose of interactive legends is to allow a mapping to be modified. For program visualization, the most basic modification is the "tweaking" of the appearance of an animation. "Tweaking" can have different flavors as part of higher level problems the user is addressing. Keys are powerful enough to permit users to perform higher level tasks such as querying of the program state and focusing the picture on certain elements while displaying the rest of the information in smaller detail. Querying and focus+context are important especially in the visualization of large amounts of data.

### 6.6.1 Queries

Legend keys can be regarded not only as defining a mapping, but also as specifying a query. The mapping, which associates values of a domain with values of another, can be interpreted as a constraint that requires that a value in the first domain be equal to the associated value in the other domain. The domains can be properties of the elements in the visualization, in which case the key determines that only elements that have the properties satisfying the key constraints be depicted. For example, the legend in Figure 6.1 can be viewed as requesting the system to display, out of all possible graphical objects, those that are cylinders and have the height and the data value in the relation shown in the legend.

The queries expressed by a legend key can be as simple as specifying a range, similar to dynamic queries [9, 80]. A legend whose threads have two ticks is sufficient to select a range. Nonetheless, legends can specify more complex queries in which there exist a certain relation between the properties of the objects. In Figure 6.8, the query selects those elements that have $prop1 = prop2$ for $prop1$ between 0.0 and 4.0. The query also specifies, via the key on the right, that selected elements should satisfy the property $prop1 = prop2 - 1$ for $prop1$ in the range 6.0 to 9.0.

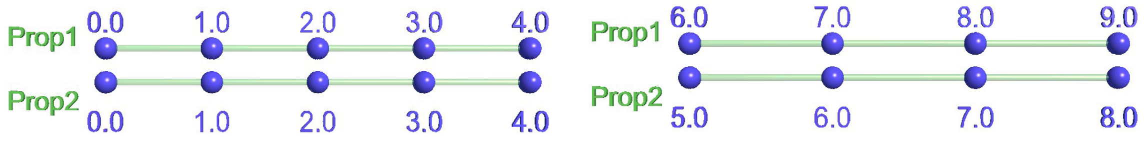


Figure 6.8: A query that selects both objects with $prop1 = prop2$ (left) and objects with $prop1 = prop2 + 1$ (right), depending on the range of the property.

### 6.6.2 Focus+context at Graphical Attribute Level

Focus+context techniques present a subset of the data in more detail than the rest. In other words, more graphical features, such as color, width or x-position, are assigned to the focus subset than to the rest of the information. Legend keys allow users to choose the range of values of a graphical attribute that are assigned to a range of data. The advantage is that multiple focus sets can be select, each presented through the graphical features chosen by the user. The choice of data and graphics can be modified via “natural” and continuous interactions with keys.

Consider a visualization of a distributed system that presents the CPU utilization at each node (Figure 6.9). The user is interested in low utilization nodes, but it is difficult to perceive the individual differences between those nodes. During the animation session, it is also difficult to observe small changes in the CPU. The difficulty is a result of the view assigning only a small number of heights to low CPU values between 0 and 10. The majority of the heights are assigned, in a uniform manner, to the rest of CPU values (range 10 to 100). There is no focus set in the visualization with respect to the height attribute.

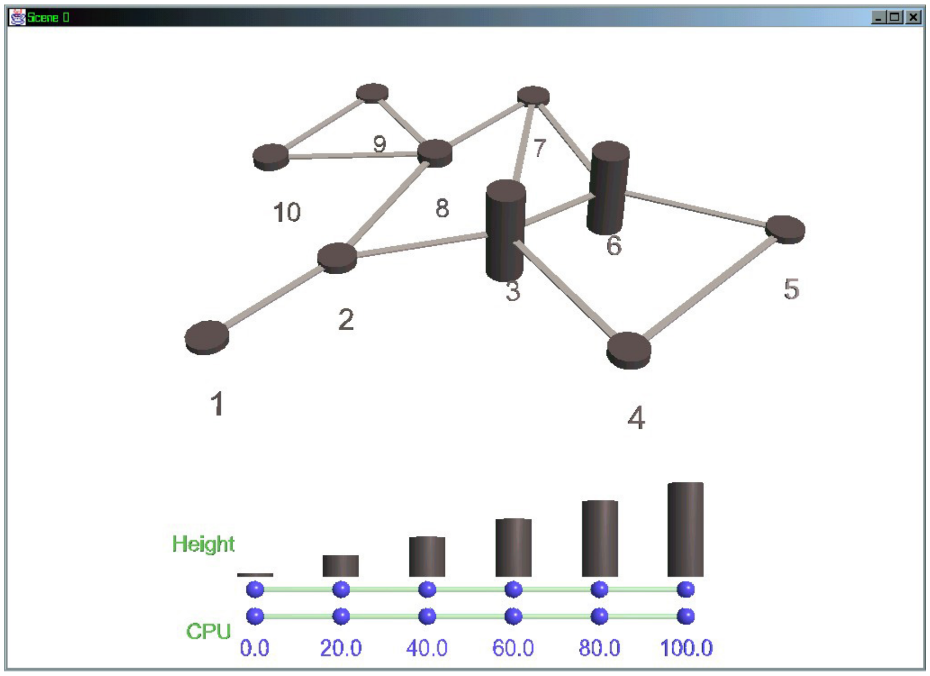


Figure 6.9: The CPU utilization for the computers of a distributed network is encoded by height. The same number of height values encode any interval of CPU values of a given length. Most CPUs are in the range 0–10, while most heights assigned to the CPU range 10–90. Individual values of low CPU nodes cannot be distinguished among.

The user can focus the visualization on the low CPU nodes by assigning more graphical values to the range of interest (CPU between 0 and 10). As such, the key is manipulated, and a large range of heights is dedicated to the CPU in the range 0 to 20 (Figure 6.10). In the new visualization, the relative difference between low CPU nodes is clearer because the height difference is larger. At the same time, the utilization of the other nodes is still visible, which provides the user the overall behavior of the distributed network.

## 6.7 Concluding Remarks

Legends can be added to a program view in order to obtain a quick and concise explanation of the visualization syntax. Interactive legends improve a user's control

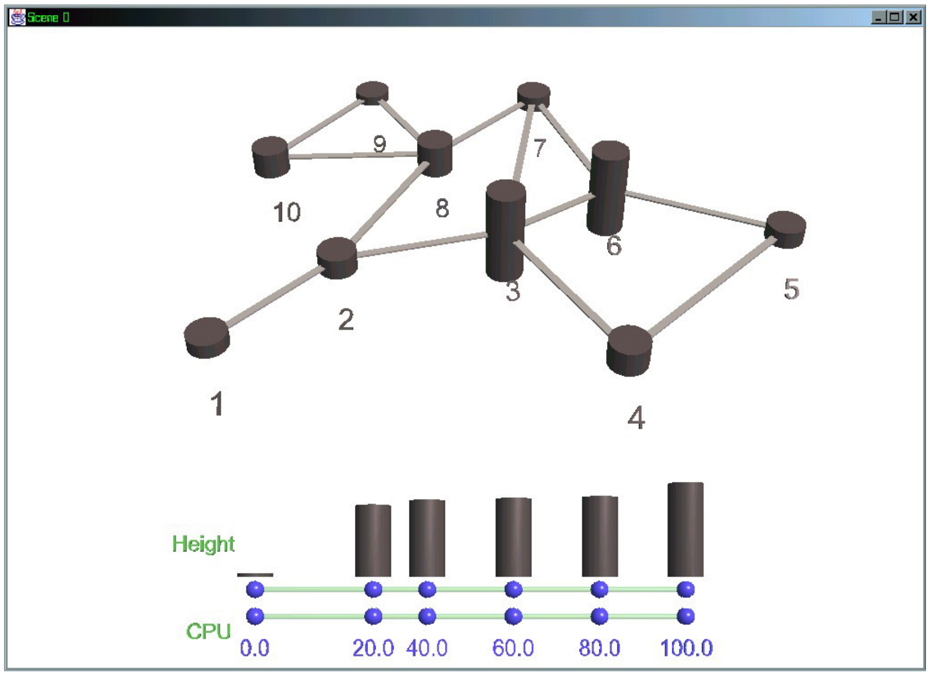


Figure 6.10: The CPU utilization for the computers of a distributed network is encoded by height. The view focuses on low CPU nodes by assigning most heights to presenting CPUs in the range 0–20.

over the visualization process and support visual navigation, via a "natural" manipulation style, in the space of the observed computation and graphical displays. The continuous interaction is appropriate for exploration and discovery tasks, and complements the discrete, more precise operations of the interactive visualization model (Chapter 4). Legends also provide a means of visually conveying the the choices of the system that produced an automatically-generated animation (Chapter 5).

Future work includes empirical studies of the usefulness of legends, a method of organizing and accessing the legend keys, and the development of an animation key capable of describing the transitions of the graphical elements from one frame of the animation to the next.

# Chapter 7

# A Study of the Performance of Steering Tasks under Spatial Transformation of Input

## 7.1 Overview

Indirection exists between the virtual objects that form the computer interface and the input devices through which the user interacts to manipulate these objects. This chapter studies the effect of spatial indirection on the speed and accuracy of user interaction. For continuous input devices, spatial transformation can be decomposed into translation, rotation, and scale. Translation alone simply shifts a movement from the device space to a different position in the virtual space, preserving the direction and size of that motion. Rotation changes the direction, while scale modifies the size. This study found evidence that rotation and scale are significant factors in interaction performance. We propose a model based on these factors that can be employed to predict the time required for a task of tracing and staying inside a non-linear shape. Contrary to our initial hypothesis, moderate translation changes did not register significant variations in the required time. The results of this experiment are applied to the analysis of two competing user interfaces for selection in a three-dimensional environment. The results of this study are also applicable to the placement and ergonomics of physical input devices.

## 7.2 Introduction

Direct manipulation interfaces permit users to control some activity through interaction with virtual controls, graphical representations of objects. For example, a user presented with a visualization of the 3-D structure of a protein might interact with a mouse or pen to fine-tune the structure by dragging a sub-unit into a preferred position. A supervising surgeon in a telemedicine setting might interact with a visualization to mark the location of an incision. A student at a remote location might interact through a touch pen to project annotations on a whiteboard seen on a live camera feed that includes the classroom, whiteboard and professor.

Despite the name, direct interaction actually involves several layers of indirection stemming from the representation of the world in which the user interacts and from the design of input devices. The three-dimensional structure of a molecule is an example of such a world, which is commonly viewed and manipulated through a two-dimensional projection on a screen. A user movement, such as dragging an atom of the molecule straight up (movement only on the vertical axis), is not directly applied from the two-dimensional screen, but instead must be converted and interpreted as a motion in three dimensions. In the molecule representation, the atom might also move along one of the other two coordinates (horizontal or depth), at an angle with respect to the original user motion. Furthermore, since not all molecule structures are physically possible and consequently not all locations are possible for the dragged atom, the straight motion performed by the user may result in a wavy path in which the atom moves along the edge of feasible locations. To generalize, the world in which the user interacts may introduce indirection in direct manipulation because, on one hand, gestures might have to be interpreted into a complex world with a high number of degrees of freedom, and, on the other hand, user actions might have to be adjusted to the constraints that govern that world.

Input devices may add an additional layer of indirection to the user interaction. Although there are devices such as touch screen and pen that permit the user to directly point at the elements of interest, most other input devices operate by design in a different space than the controlled objects. For example, a typical mouse resides at a certain distance from the screen, moves on a horizontal plane, which differs from the vertical orientation of a typical screen, and travels a different distance than the controlled objects. Direct interaction with the molecule of the previous example actually happens by looking in one place, at a vertical screen, and acting in another place, upon a mouse that moves horizontally. Software devices such as graphical user interface (GUI) widgets can also introduce indirection. Software input devices, as the physical ones, act as intermediaries in controlling other GUI elements and often move in a different space than that of the controlled objects. A scrollbar or three-dimensional handle (see Conner *et al.* [22] and Ware and Rose [97]), for example, help in moving other objects whose motion might not be constrained in the same manner as that of the controls.

The ability of the user to interact efficiently may be confounded by these layers of indirection that can produce quite complex transformations of the user's input. However, the overall spatial indirection can be regarded as a combination of only a few basic transformations. Three basic transformations are considered in this chapter: translation, rotation and scale. Translation, defined by an offset, is the distance between the location of a single action and its effect (Figure 7.1b). Translation does not change the size or direction of the motion. Rotation is the angle between the line connecting two actions and the line connecting their effects (Figure 7.1c). For example, a mouse motion is rotated from a horizontal to a vertical plane. Scale, related to control-display ratio, expresses the ratio of the distance between two effects to the distance between their actions (Figure 7.1d). A unit displacement of the thumb of a scrollbar may be scaled so that a document moves a few lines or a few pages depending on the size of the document. Scale changes the size of a motion, while rotation changes its direction.

In this chapter, we attempt to study the impact of these spatial transformations of input on the effectiveness of interaction. The manner in which the user is affected is likely dependent both on the type of input transformation and on the task performed by the user. In the former respect, our study focuses on transformations that involve a combination of translation, rotation and scale (an affine transformation) that remains constant during interaction, both over time and over the entire area of the display and device footprint. The final goal is to quantify the effect of translation offset, rotation angle and scale on the speed of interaction by paying attention to the interactions among these three factors. We artificially create various transformations of the user's input on a personal digital assistant (PDA), which allows us to compare in the same environment both direct (not transformed) interaction and various types of spatial indirection. In the latter respect, we fix the task in the study to be a steering task. Steering is the action of moving the cursor through a tunnel, or in other words, of connecting two points while staying inside a predetermined shape.

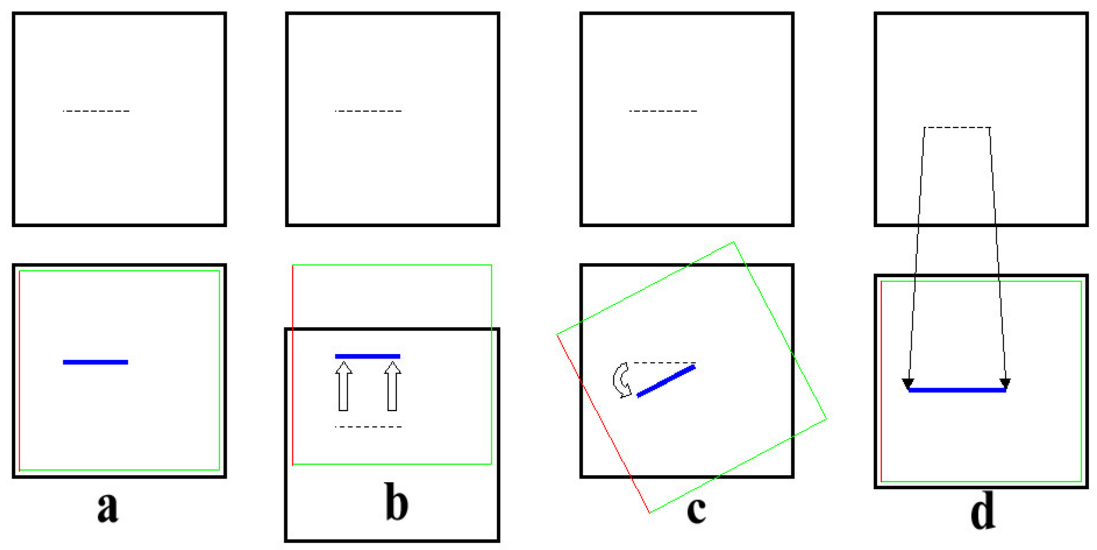


Figure 7.1: The correspondence between the input device and display: (a) no spatial distortion, (b) translation up, (c) counterclockwise rotation around the center, and (d) scaling. The upper row presents the footprint of the input device (thick rectangle) and the movement of the control (dashed line). The lower row superimposes, on the input device, the display area (thin rectangle) and the effect of the movement (thick line).

The result of this chapter can be applied to evaluating the fitness of an input device for a task, and has the potential to add to the ergonomics of the work space and increase awareness of the limitations of human-computer interactions. Moreover, because spatial indirection also occurs at the software level in the graphic interface, understanding the influence of distortion can guide the design of the GUI and the manner in which control elements are used.

The next section presents background work about the evaluation of input devices under various conditions. The design considerations and hypothesis of the experiment are the subjects of Section 7.4. The experiment is described in Section 7.5, followed by a discussion of the results in Section 7.6. The model developed by this chapter is then applied in the context of a (common) steering task in three-dimensional space in Section 7.7. A future extension of the prediction model concludes the chapter in Section 7.8.

## 7.3 Background

### 7.3.1 Steering Law

An extensive body of literature assesses the performance of continuous input devices under diverse tasks and conditions [12, 18, 36, 48, 52, 55, 56, 57]. The research is based on a robust result, Fitts' law [32] for target acquisition. According to the law, the movement time to a target, $MT$, increases as the distance to the target, $A$, increases and as the width of the target, $W$, decreases. More precisely,

$$MT = a + b\log_2(\frac{A}{W} + c), \tag{7.1}$$

where $a$ and $b$ are constants determined empirically for each task and input device, and $c$ is usually selected from one of 0, $\frac{1}{2}$, 1 by different researchers (see MacKenzie [55] for a discussion of $c$). The logarithm term is referred to as the index of difficulty. This measure of performance ($MT$) and the formula above have successfully been used to model a plethora of pointing, dragging and selection tasks. This chapter focuses on the results that are pertinent to steering tasks and spatial input transformation (see MacKenzie [55] for a survey of research related to Fitts' law).

Tunnel steering [6], the task we focus on in this study, is similar to target acquisition tasks in the sense that the pointer is moved from a starting position to a target. The path of the pointer, however, is restricted to remain within the boundaries of a tunnel. Accot and Zhai showed that when the tunnel is a shape with constant width the movement time depends linearly (in contrast with the logarithmic dependence of Fitts' law) on the ratio of the tunnel length and width. The performance measure, derived from Fitts' law, is

$$MT = e + f\frac{A}{W}, \tag{7.2}$$

where $A$ and $W$ are the length and width of the shape, and $e$ and $f$ are constant for particular conditions of the experiment. The index of difficulty for steering ($ID$) becomes the fraction $\frac{A}{W}$. It is this index of difficulty that we refer to in the remainder of the chapter. The steering law, which can model even tunnels with variable width, was obtained through mathematical analysis and verified empirically in [6]. The law was also successful in modeling the movement time in a follow-up comparison of pointing devices conducted by Accot and Zhai [7].

### 7.3.2 Distortion via Physical Input Devices

Although no comprehensive analysis of the effects of the transformations induced by input devices is known to the author of this thesis, research exists on temporal distortion (i.e., lag) [57] and on spatial scaling, largely related to control-display gain [8, 10, 18, 35, 48, 99]. Scattered results pertaining to rotation [26, 37, 78] and translation [37, 56, 65, 78, 79, 97] can be inferred from other work. MacKenzie and Ware [57] presented a variant of Fitts' law that quantifies the effects of lag on the user performance. Their study was conducted for a classical pointing task and is orthogonal to the spatial effects discussed in this chapter.

#### Control-Display Ratio

Control-display (C-D) ratio, together with the related control-display gain and movement scale, was the most extensively studied spatial transformation [8, 10, 18, 35, 48, 99]. The C-D ratio is calculated as the controller movement divided by cursor movement, while the gain is the inverse (for positional control devices). Movement scale considers only the controller motion by maintaining the same visual perceptual feedback as explained by Accot and Zhai [8]. Movement scale is still part of C-D ratio research in that the display size is kept fixed and the ratio is varied only through the controller size. With all the research however, there is little consensus about the value of C-D ratio as a predictor of interaction efficiency. On one hand, some findings indicate that the scale can predict human performance, although different functions are proposed. On the other hand, no correlation seems to exist between C-D measures alone and movement speed, as found by Buck [18] and Arnaut and Greenstein [10]. In the former category, human performance books [99], as well as Accot and Zhai [8] for the steering task, determined a U-shaped function of C-D gain versus movement time. The bottom of the U-form represents an optimal range of C-D gain for which MT is small. Another formula was reported by Gibbs [35], which yields a U-shaped function when lag is present in the system and a linear function otherwise. In contrast, Jellinek and Card [48] concluded that human performance would be constant but for the inadequate resolution of input devices and displays, suggesting that the drop in performance is an effect of quantization.

Our work differs from previous C-D gain research in the manner that scale is considered. We recognize that scale alone does not completely describe the effects of transformation on the user's ability to perform tasks, and we concentrate on understanding the interactions of scale with translation and rotation. It is possible that the combination of these three factors, rather than scale alone, better expresses human performance.

### Rotation

Cunningham [26] approached the issue of rotation between the visual and motor space and reported high error rates for angles ranging from 90° to 135°. At 180°, the error rates were relatively low. That study concentrated on inferring the internal spatial representations of human visual-motor processes, and the impact of adaptation to rotation. The emphasis was on accuracy rather than movement time. In a pilot study, we also found that the drop in accuracy for the steering task is very steep for angles of the second quadrant (90° to 180°) rendering steering almost intractable. Consequently, our study focused on angles in the first quadrant (up to 90°).

### Translation

Ware and Rose [97] approached the problem of translation in a virtual reality (VR) environment. The task of their study was to rotate 3D objects under various conditions. One experiment studied the displacement of the haptic handle from the visual one. The displacement condition resulted in a degradation of speed, while the accuracy remained approximately the same. The main difference between Ware and Rose's study and ours resides in the type of tasks; VR rotation may be different than a 2D steering (translation) from a motor and cognitive point of view. The applicability of the results of our study to the VR task might be further influenced by the interaction between spatial transformation factors.

Translation was also studied for remote pointing devices by MacKenzie and Jusoh [56]. Such devices are handy when the user is at a rather large distance from the display as in conference presentation settings or interactive television systems. MacKenzie and Jusoh's study focused not on translation effects, but on input devices. The study compared the mouse (ball technology), and two types of remote pointing controllers, one an isometric-type joystick, and the other with both a gyroscope that sensed side-to-side and up-down movement and a mouse ball. This last device was evaluated in both close and remote settings, but in each setting using a different technology (gyroscope or ball). The results showed that remote pointing is significantly slower than traditional (at the desktop) pointing. However, a conclusion that translation effects impede performance cannot clearly be made, both because of the radically different interaction styles of the devices studied and because of the lack of direct comparison between close and remote conditions for any one technology.

The effect of all three transformation types, though not investigated explicitly, is evident in studies of devices that permit direct control and in which no angle, scale or offset exists. The touchscreen and lightpen are representative direct control devices. Even early work, using pioneering touchscreen technology, found direct pointing devices faster, although less accurate, than other devices [37, 65]. The problems cited for touchscreen and lightpen range from user fatigue and occlusion of the display to lack of precision and screen smudging [65, 79]. We anticipate that most of these problems will not be evident in the stylus interactions on a PDA, except perhaps a small degree of occlusion. A later study by Sears and Shneiderman [78] of high-precision touchscreens concluded that single pixel selection can be achieved with a stabilized touchscreen whose raw input data is filtered and processed. That study also confirmed that for large target width the touchscreen is faster than the mouse, in agreement with earlier experiments [37], but for smaller targets the performance is about the same as with the mouse.

## 7.4 Design Considerations

The design of the empirical study was strongly influenced by the findings of an informal pilot study and on the potential problems reported by other researchers. Our design goal was to reduce noise caused by uncontrolled factors, to avoid known pitfalls, and to make the experiment manageable. As a result, the angles were restricted to the first quadrant, and the scale range was chosen around and close to one.

A small scale interval centered around one reduced the quantization effects and restricted the limbs (arm, fist, and finger) involved in the interaction. Quantization appears when a continuous movement is mapped into a coarse discrete coordinate system. This effect is believed by Jellinek and Card [48] to be the cause of performance variation caused by C-D gain. Furthermore, the effect was noticed in our pilot study even for scale values of two. It has been noted by the researchers that limb segments and combination of limbs involved in the motor action are among the causes of performance variations with scale. Langolf *et al.* [52] found the finger more effective than the wrist, which was better than the arm, but Balakrishnan and

MacKenzie [12] expressed skepticism about that result. Zhai *et al.* [100] found that fingers significantly helped in a multiple degree of freedom input. In our experiment, the size of the PDA display and input control limits the classes of muscles and limbs required for the experiment. Authors such as Guiard *et al.* [36] may regard this as contrary to the "spirit of Fitts' research", which they maintain was intended to model human performance independent of the body movement. Nonetheless, limiting the limbs involved in the motion may also reduce the noise in the experiment and perhaps produce better results in the controversy regarding C-D ratio. This experiment should provide a good model for the impact of spatial transformation effects on tasks that require mainly finger and some wrist movement - actually mainstream human-computer interaction tasks.

### 7.4.1 Hypotheses

All hypotheses assume a steering task.

1. Translation has an effect on the movement time and error rate of the interaction. Both movement time and error rate increase as the offset gets larger.

2. Rotation has an effect on the movement time and error rate. For higher values of the rotation angle, the movement time and error rate are larger than for smaller angles.

3. Scale has an effect on the movement time and error rates. Movement time decreases with scale, while error rate increases.

4. Interactions exist among translation, rotation and scale that affect the movement time.

## 7.5 Experiment

### 7.5.1 Materials

The experiment made use of a PDA and required subjects to trace through a shape under various spatial transformations. The PDA was selected for its ability to offer zero-translation interaction. The PDAs used were two Palm Pilot m505s with 16-bit color displays. The display had a backlight feature, which was kept on during the testing. The screen resolution was 160 x 160 pixels, with dimension 5.5 x 5.5cm.

Henceforth we will use pixels as a measurement unit. Standard styli shipped with the Palm Pilots were used for drawing directly on the screen.

The software for the experiment was developed under Java$^{TM}$ for Palm OS. Lag was not measured, nor was it apparent during typical operation. For the purposes of the study, lag can be considered very small and constant.

### 7.5.2 Subjects

Twenty-four college students volunteered for the study. Sixteen participants were male, eight were female, and six had previous experience with the stylus. They were offered a soft drink and competed for a cash prize that was to be awarded to the fastest and most accurate participant. Note that a trade-off exists between speed and accuracy. Our goal was to encourage "normal" behavior, in which users are neither overly meticulous and slow nor overly fast and sloppy.

### 7.5.3 Procedure

Thick circles and vertical lines were the two types of shapes used in the steering task. The shapes were red and contained either two or four black control stripes, as shown in Figure 7.2. The screen background was white. The steering task started by painting blue ink at the bottom of the shape to be traced, slightly under the thick line (outside the red shape itself) and in the southern region of the circle (inside the shape). A black crosshair, 3 x 3 pixels, provided the subjects with a hint at the starting location for the stylus. This location, when touched with the stylus, caused ink to be drawn at the bottom of the red shape, as explained above. Note that, in the presence of spatial transformation, the point where the stylus touches the screen is not the same point at which the ink is drawn. During a trial the crosshair follows the stylus on the screen and lies at the tip of the stylus while the stylus is pressed.

Both the line and circle were intentionally displayed off-center of the screen, which is especially important for the circle. If the circle were displayed on-center, the rotation would not have any real effect.

The participants were instructed to trace with blue ink a path inside the red shape so that the following two constraints were satisfied: (1) The blue ink must be drawn inside the red shape except when starting and ending a line trial. Remember that a line trial actually starts outside the red line, on the white background. The participants were also allowed to overshoot the top of the line, and consequently the

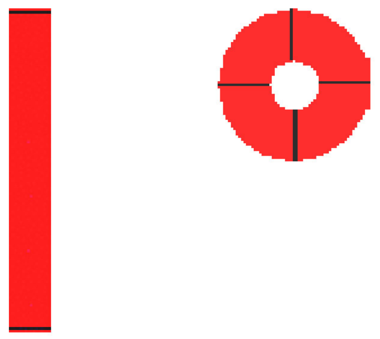


Figure 7.2: The two shapes in the steering task.

line trials might have ended outside the shape. (2) All black control stripes had to be touched to force the subjects to follow the main features of the shape. Of course, this constraint is satisfied automatically by every successful steering task; we made this constraint explicit to assist subjects in performing the task and to simplify the analysis of correct results.

To move to the next trial, subjects pushed one of the physical buttons on the Palm Pilot. No time pressure or constraints were imposed between the trials. Timing of a trial began when the stylus first touched the screen and ended when the stylus was last lifted. Lifting the stylus during a trial did not count as an error nor influence the timing of the trial. This manner of recording the duration of a trial differs slightly from that employed in similar experiments by Accot and Zhai [6, 7, 8]. Those experiments detected when the path of ink exited and entered the tunnel shape and considered those times as the beginning and end of a trial. However, detecting whether the shape was entered/exited for each "stylus moved" event was simply overwhelming for the PDA processor (each event would require costly multiplications and divisions to determine whether two segments intersect), and the lag incurred by such computing was unacceptable for our experiment. We correctly predicted that the intercept (values of $e$ in the steering law) would be higher than in previous studies due to this difference in methodology. Informally, the intercept can be explained as the minimum amount of time required for a trial, which is the time the user takes to start a trial. In our methodology, the intercept included the instant needed to move inside the shape, while in the previous study this time period was ignored.

The trials were recorded on the PDA and analyzed later on a regular computer. Trials that did not satisfy the two correctness conditions mentioned above, namely blue ink fully inside the red shape and all along the shape, were counted as errors and excluded from the analysis of the movement time.

The display of the Palm Pilot conveyed the current transformation of the input in a manner similar to that of Figure 7.1, bottom row. For the PDA the thick rectangle represents the physical edge of the screen, which actually defines the footprint of the stylus input device. The thin rectangle, the drawable area, was translated and rotated by the current offset and angle of the spatial transformation. To provide a cue as to the direction of rotation, the top of the rectangle was colored red, while the other three lines were green. Portions of the thin rectangle that are beyond the edge of the screen (outside the thick rectangle) were not visible. A number representing scale was printed at the bottom of the screen.

### 7.5.4 Design

Conditions included three offsets O $= \{0, 30, 60\}$ pixels to the right of stylus, four angles R $= \{0°, 30°, 60°, 90°\}$, three scales S $= \{0.8, 1.0, 1.5\}$, two shape types Sh $= \{line, circle\}$, two shape lengths A $= \{64, 125\}$ pixels, and three shape widths W $= \{12, 16, 20\}$ pixels. The ratio of length to width produced six IDs (indices of difficulty) ranging from 3.2 to 10.42. The experiment may be viewed as being performed within subjects, but with some empty cells. That is, because of the large number of combinations (3x4x3x2x2x3=432), the need to perform multiple repetitions (5) for each combination, and the need to avoid fatigue by limiting the time per subject to one hour, each subject performed a subset of the 432 possible combinations. Specifically, the subjects were randomly divided into two groups, with the first two scales assigned to the first group and the last two scales to the second group. Within a group, all subjects were tested for all 3x4x2 distortion conditions of offset, angle and scale. The conditions were presented in a random order to each subject. For each distortion condition, a shape was randomly chosen out of the 12 Sh-A-W possible shapes. However, at the end of their sessions all participants encountered each shape twice, although under different O-R-S conditions. Moreover, within a group all distortion conditions were fully crossed with all shapes.

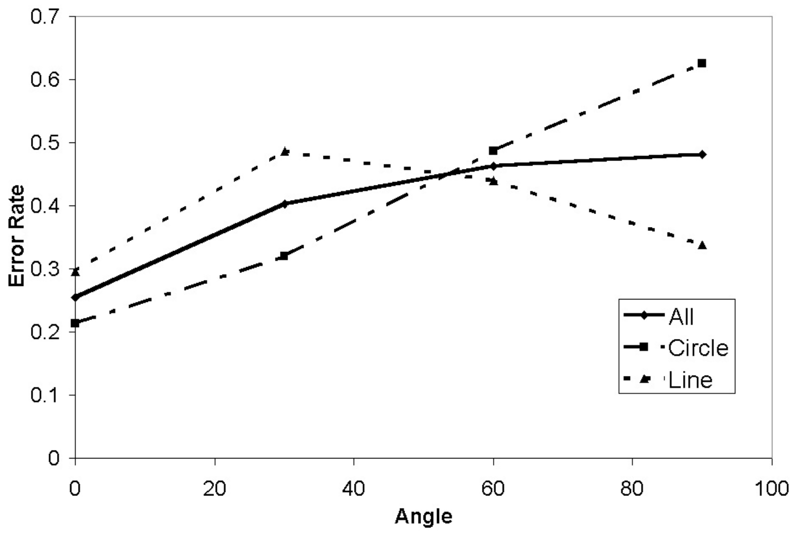
A training pre-session that consisted of 18 different trials was given to all participants. Subjects were allowed to go through the 18 trials multiple times. Subjects spent about an hour in the experiment, including this training session.

### 7.5.5 Results

The first two repeats for each combination were excluded from further analysis because they showed a strong learning trend; the second and third repeats differed significantly in the error rate ($F_{1,23} = 6.37, p = 0.0189$) and movement time ($F_{1,23} = 6.52, p = 0.0177$). Differences in error rate ($F_{2,46} = 0.97, p = 0.3867$) and movement time ($F_{2,46} = 0.31, p = 0.7349$) were not significant among the last three trials.

The overall percentage of incorrect trials is quite high at around 40%, although for no rotation, R=0, the error percentage is at 25.5%. This is just slightly higher than previously reported error rates [7], which were computed under fixed scale and translation conditions. As justified by Accot and Zhai in [7], the steering task offers the user a "chance" to make a mistake anywhere along the path of the tunnel, not only at the of the end of the path as in goal passing tasks. This becomes especially significant when the user input is transformed, as it becomes harder to continually control the path of a cursor that seems bent on moving outside the tunnel. Angle has a large impact on the error rate ($F_{3,69} = 23.49, p < 0.0001$). Figure 7.3 shows that, overall and especially for circles, the number of wrong trials increases almost linearly with the angle. The lines behave differently from circles, and a possible explanation for this is given in the next section. Scale is also a significant factor for the error rate ($F_{2,22} = 37.33, p < 0.0001$). In Figure 7.3 bottom, it can be observed that the error rate increases with the scale. Contrary to our hypothesis, translation does not change the error rate in a significant manner ($F_{2,46} = 1.52, p = 0.2296$). Shape type is also not a significant factor ($F_{1,23} = 0.96, p = 0.3365$), as can also be inferred from Figure 7.3, which shows that the circle outperforms the line for small angles, while the line is better for larger angles. As expected the $ID = \frac{A}{W}$ was found to be significant ($F_{5,115} = 23.73, p < 0.0001$) for errors.

The dependence of the movement time on distortion and shape factors is analyzed only for successful trials. Spatial distortion components, except offset, proved to be significant factors for movement time ($F_{3,68} = 53.37, p < 0.0001$). Again, variations in offset do not entail significant variations in the movement time ($F_{2,46} = 0.97, p = 0.3853$). Figure 7.4 plots the movement time for circles, lines and



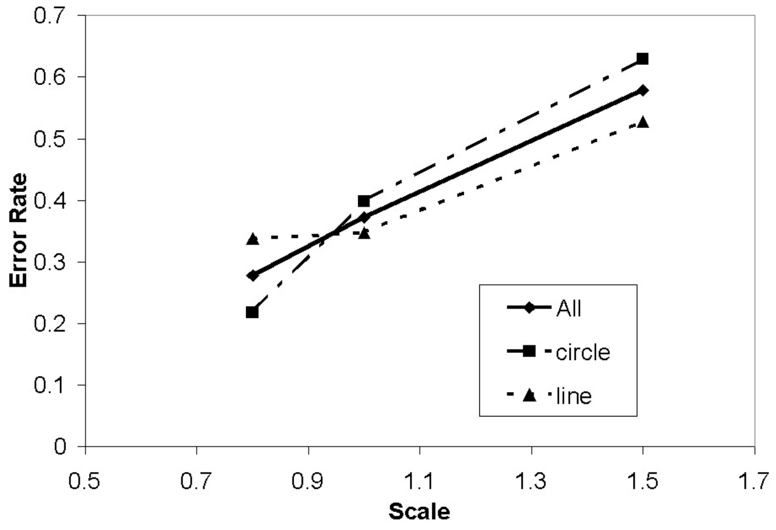
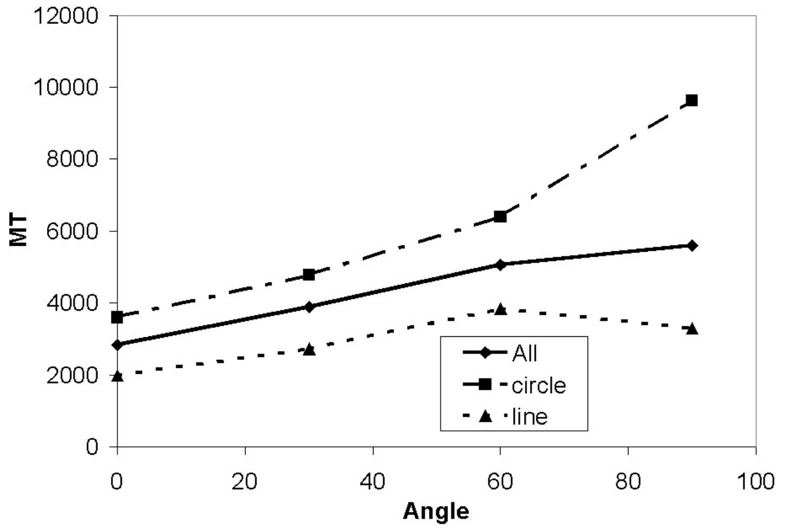


Figure 7.3: Error rate as a function of angle (top) and scale (bottom).

overall as functions of distortion parameters. Notice on the upper plot how at 90° the movement time for the circles is higher than at 60°. The lines behave differently in that the duration of a line trial decreases at R=90°. Figure 7.5 suggests that an interaction between angle and scale may exist, as does the F-test ($F_{6,54} = 3.13, p = 0.0105$).

It is apparent in Figure 7.4 that the average steering time for circles is higher than for lines. Statistically, shape is indeed significant ($F_{1,23} = 270.94, p < 0.0001$).



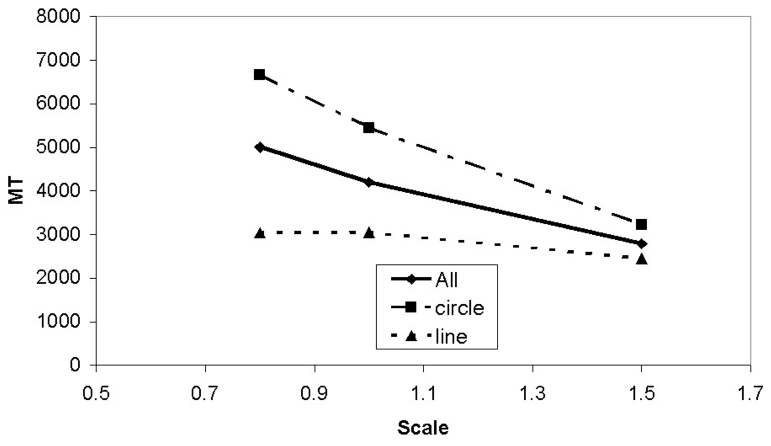


Figure 7.4: Movement time as a function of angle (top) and scale (bottom).

As expected, the index of difficulty, ID, passes the significance test for the steering time ($F_{5,113} = 32.73, p < 0.0001$).

We also developed a model for the effects of spatial transformations based on the steering law. Translation is ignored in the model because the offset component is clearly not a factor in either efficiency or accuracy for the steering task. Based on the interaction between angle and scale, a new regressor termed *spatial index* is

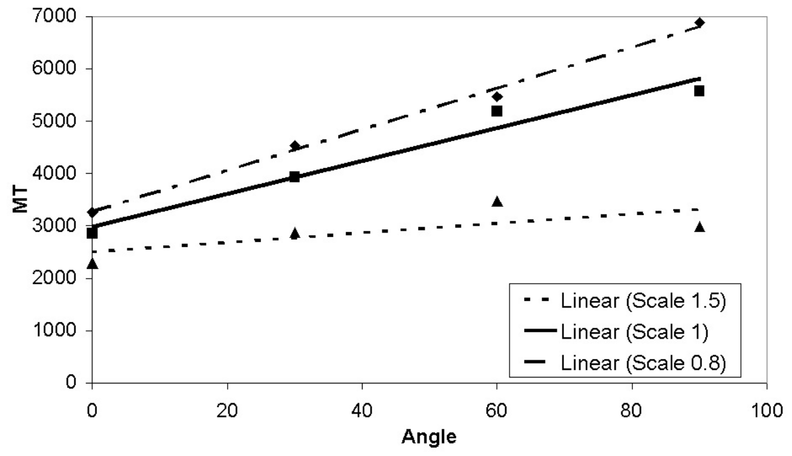


Figure 7.5: Interaction between angle and scale. Trend lines are included for each scale value.

introduced:

$$SI = \frac{R + s_0}{S}. \tag{7.3}$$

The constant $s_0$ depends on the conditions of the experiment, and ensures that scale is taken into account when there is no rotation ($R = 0$). Interaction between $ID$ and $SI$ is marginally significant ($F_{55,116} = 1.54, p = 0.0261$). Figure 7.6 shows the interaction between SI and ID for circles only.

Table 7.1 summarizes the models and their correlation to the data. $ID$ alone cannot explain the variance in the steering time; models based solely on it consistently have a lower correlation than models that take $SI$ into account. In the next section we discuss possible reasons for the poor correlation values for the line shape.

We do not favor the additive model ($ID + SI$) for statistical and semantics reasons. This model has lower correlation (equal for lines) than the multiplicative one. Moreover, an interpretation of the additive model cannot easily be given; the large negative intercept and negative $s_0$ cannot be easily assigned a meaning. It is also unlikely that spatial transformation affects a task independent of the task $ID$. The multiplicative model, however, clearly expresses the intuitive notion that the effects of scale and angle are more pronounced in more difficult tasks than in simpler tasks. Further, the multiplicative model yields $ID$ coefficients comparable to the

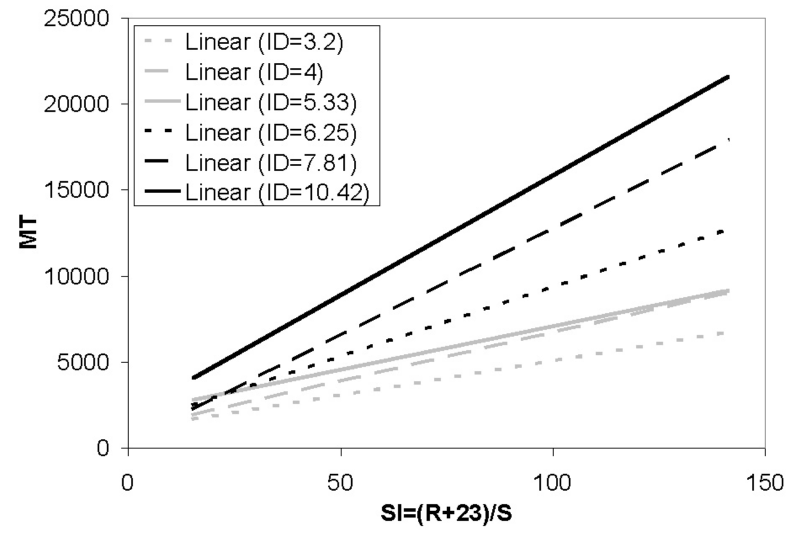


Figure 7.6: Interaction between SI and ID for circles. Only data trends are presented.

Table 7.1: Regression models: $ID$ alone, additive and multiplicative. ($p < 0.0001, n = 72$)

| | | | |
|---|---|---|---|
| $ID$ | line | $MT = 1011 + 334ID$ | $r = 0.481$ |
| | circle | $MT = 139 + 1073ID$ | $r = 0.534$ |
| $ID + SI$ | line | $MT = 458 + 334ID + 19(R - 15)/(S)$ | $r = 0.621$ |
| | circle | $MT = 4013 + 1059ID + 85(R + 6.2)/(S)$ | $r = 0.860$ |
| $ID \cdot SI$ | line | $MT = 1504 + 3.3ID(R + 35)/(S)$ | $r = 0.620$ |
| | circle | $MT = 926 + 14ID(R + 23)/(S)$ | $r = 0.904$ |

results reported in [8] for no rotation and scale factor of 1.0. We note, however, that the intercepts are higher in our experiment as explained in Section 7.5.3.

## 7.6 Discussion

From discussions with the test participants at the end of the sessions, two main strategies for steering emerged. In the first strategy, subjects relied on visual feedback to correct the tendency to move out of the shape, while in the second strategy subjects made a blind guess as to how the stylus should be moved, with little regard for visual feedback. The latter entailed only a swift ballistic motion that likely required less time than the visual steering. The lack of visual feedback has a negative effect on accuracy (Prablanc *et al.* [64]). We speculate that with circles the second strategy would produce too many errors to actually influence the steering time of the correct results. For lines however, this strategy probably worked reasonably well and thus affected the movement time. This might explain the drop in duration for the 90° angle, when tracing the vertical line is the same as drawing a horizontal line from the starting point. Assuming that participants noted that lines can often be guessed and circles cannot, the ballistic strategy might also provide a reason for the relatively high rate of errors for lines, which is almost the same as for the circles and contrary to the findings in [8]. Consequently, the line behavior is different from the circle's, and might be the reason for the poor correlation of the regression model for the line shape. We believe that practical steering tasks, such as navigating through a hierarchical menu, cannot be performed blindly, and thus will exhibit behavior more similar to that of the circles than the lines. Note that another exception is likely to occur at 180°, as observed in Cunningham [26].

The experiment failed to prove hypothesis 1 that translation is significant for efficiency of interaction. This may explain why there was no degradation of interaction speed with touchscreens when the cursor was deliberately translated. In effect, as can be inferred from Sears and Shneiderman [78], the induced translation reduced the occlusion of the cursor by the hand and resulted in improved interaction accuracy. In our case, it may be possible that occlusion occurred at the 30-pixel offset and not at 0 or 60, where the hand is either too far from or close to the stylus to block the view. Our results show a mean time higher for an offset of 30 than for the others. Note that the faster interaction for touchscreens observed in [37, 65, 78] is most likely due to the lack of rotation and scale, rather than lack of translation, in direct input devices.

The relation between the movement time and scale is almost linear, as revealed by Figure 7.4. Thus, scale variations are important for steering time, at least when other spatial distortions also change. The hypothesis that the function is U-shaped is neither supported nor rejected. The observed shape in Figure 7.4 might well be part of the left side of the U shape.

The interaction between angle and scale was shown to be significant (almost at the .01 level) in the experiment. The regression model is also based on the spatial index, which is mainly the ratio between angle and scale. The implication is that the effect of spatial transformation is better explained by a combination of its determinant factors than by any single factor. Individually, these factors offer a poor justification for the performance results.

## 7.7 Applying Spatial Index To Selection Task

The spatial index introduced in the previous sections, besides its theoretical significance as a descriptor of spatial indirection of user input, can also be applied to assess and choose among multiple designs for a user interface. As a case study, we consider the problem of selecting a "slice" of a three-dimensional visualization. Examples of such selection include deciding where to perform an incision on a patient's body for a remote/virtual surgery, or choosing, from a visualization of a mountain range, an area that has certain mineral deposits that are not covered by layers of extremely hard rock. In both cases, the selection takes place based on a two-dimensional surface such as the patient's epidermis or the surface of the Earth at sea level. For simplicity, we consider that the surface is a plane. To perform the selection, the user must be able to draw a shape on the reference plane, perhaps relying on information displayed above or below that plane. Such information might be the organs and blood vessels that lie beneath the patient's skin or the deposits and rocks in the mountain. The selection includes the portion of the visualization that lies directly above and below the reference plane and that is projected on the inside of the user-drawn shape. If the shape were a circle, the selection would be a cylinder that is perpendicular to the reference plane and intersects that plane on the drawn circle.

The point of view of the user cannot be directly above the reference plane because that would make it difficult for the user to perceive what is directly beneath and above that plane. It may happen that one of the rock types might obscure another in the mountain range. Therefore, the point of view of the user would most likely be at an angle from the plane of reference. Figure 7.7 depicts a reference plane rotated about 45° around the horizontal axis from the screen plate. The implication of this angle is that one unit in the reference plane, when projected onto the screen, appears as approximately 0.7 units[1]. The area to be selected may not be in the center of the screen (users' viewpoint) or may need to be moved off the screen center in order to permit the user to observe it sideways in addition to the up-down view. For the mining example, the user might be satisfied if the mineral deposits are reachable from

[1]Depending on the projection technique, the size might actually be smaller that 0.7 units if the measurement is taken far from the user's point of view.

one side as long as the rock above does not cave in. As a consequence, a line that is off center and is vertical in the reference plane does not appear vertical on the screen, but slanted as illustrated by the angle marked a in Figure 7.7.

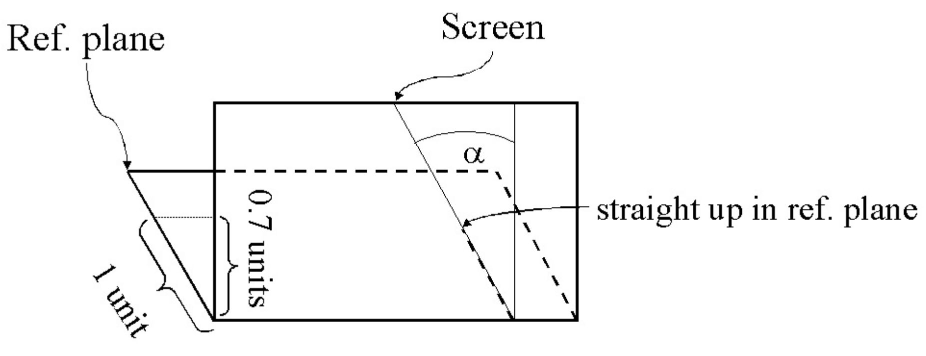


Figure 7.7: The relation between the screen and the reference plane in which the selection is made. Dashed line is what the user would see on the screen from the reference plane. Note that a line that is straight up on the plane appears slanted on the screen.

Consider two designs for a selection user interface that rely on the same widget that moves in the reference plane, but interpret the input from a tablet, or any other input device, differently. An *in-plane design* moves the widget relative to the reference plane, in the same manner regardless of the user's point of view. An upward input motion would move the widget towards the north side of the reference plane (parallel to the vertical axis of the reference plane, not of the screen plate). An *in-screen design* has the user specify where the widget should move as opposed to steering the widget. The motion of the widget is now dependent on the point of view of the user, and an upward motion on the tablet does not necessarily displace the widget toward the north of the reference plane, but toward the top of the screen plate. Figure 7.8 depicts what happens when a square is selected with each of the two designs.

To compare the two designs, it is useful to note that the selection problem is in a fact a steering task. The user must maneuver a widget inside a shape in order to delimit the area of interest of the visualization. For steering tasks, we model movement time as a function of $ID \cdot SI$, the index of difficulty and the spatial index. The two designs have the same $ID$ because they do not involve any alteration of the underlying visualization. The user perceives the same width and length for the tunnel through which it has to maneuver.

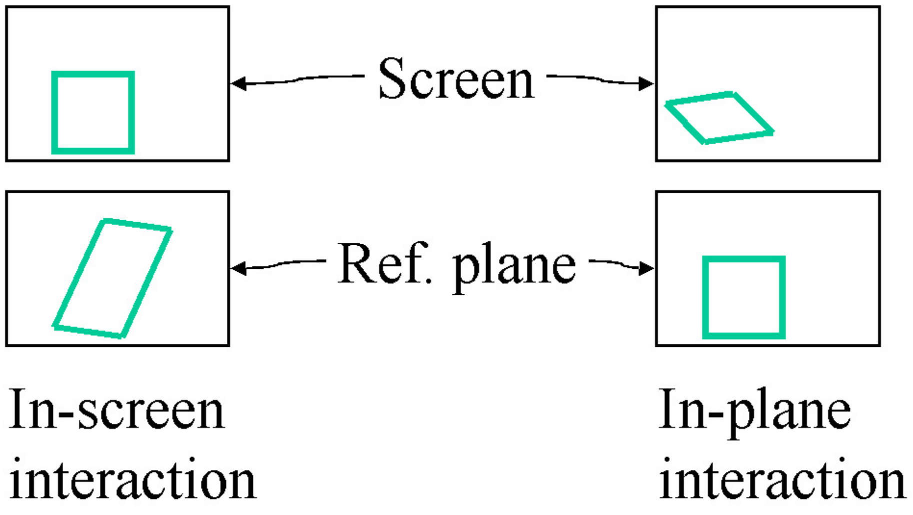


Figure 7.8: A square is selected using the two interface designs. The picture shows how the selection appears on the screen and what is selected in the reference plane.

The spatial index is dependent on the scale and rotation values between input and display. In both interfaces, the same input device is used, which is considered to have scale $s$. In the in-plane design, when the widget moves one unit in the reference plane, the user perceives only 0.7 units on the screen. Therefore, the scale of the input device is further scaled on the screen to a value of $0.7s$. In the in-screen design, the widget appears to move in the screen plate, and the scale remains $s$.

Rotation appears in the in-plane design because a straight up motion on the tablet appears slanted on the screen (Figure 7.7 and Figure 7.8). The angle might be anywhere between 0 and 45 degrees, depending on the placement of the area of interest on the screen and on the method of projecting the three-dimensional world. We approximate the angle at around $20°$, which is probably common for most projections and screen positions (including Figure 7.7). The in-screen design does not incur rotation.

We can now quantify $ID \cdot SI$ for both designs under the assumption that the interaction is regional, that is does not take place across the entire screen and leads to large variations of the rotation angle. In the in-plane design, $ID \cdot SI_{plane} = ID(20+c)/(0.7s)$, while in the in-screen design $ID \cdot SI_{screen} = ID \cdot c/s$. The values of $c$ would depend on the particular conditions of the experiment, but if the shape were a line (such as a surgical cut) and the input a tablet, very similar to a PDA input, then $c \approx 35$. This makes the spatial index for the in-plane design about twice as large as the spatial index for the in-screen. Thus, in order to perform the selection in the same amount of time, the $ID$ for the in-plane design has to be half of the $ID$ for the in-screen design. In other words, the in-screen design appears more effective because it takes about the same time for the user to draw shapes that are twice as long or are within tunnels half wide than with the in-plane design ($ID = A/W$ is dependent on the tunnel's amplitude and width). The downside of the in-screen design might be a higher rate of errors, because there is usually a trade-off between speed and accuracy.

## 7.8 Concluding Remarks

The model presented in this chapter enabled us to compare different user interfaces designed for the same task. The model captures one of the simplest forms of input transformation, a constant affine transformation, and we believe that a simpler model that excluded rotation or scale would be very limited. The model was developed from a rather extensive empirical study. A study for modeling the effect of non-constant transformation, although it may prove useful, may become very complex.

The constant transformation model can be extended in the future to deriving prediction models for more complex effects, namely transformations that vary with the position of the cursor in the display space. The variation is negligible for small enough regions of the display, and the constant model is applicable for these small regions. A spatial index can be calculated for the entire display by decomposing the interaction space and computing the local spatial index (similar to the derivation of the steering law for variable width tunnels [6]). The model obtained through such analytical means can then be verified empirically. Note that the empirical verification of a model often requires fewer test cases than the derivation of the prediction model from empirical observations.

# Chapter 8

# Conclusions

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

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

The effectiveness of this approach is increased for exploratory tasks in which the user is interested in discovering both the properties of a computation and the visual manner in which to better analyze the program. The user is not required to calculate or be knowledgeable of computer graphics and can directly query the computation state and the visualization based on common properties and relations.

The clarity of the notions presented by animations is improved through the use of legends, which provides for an easier application of program views to a problem and for a continuous interaction method of adjusting the appearance of the data.

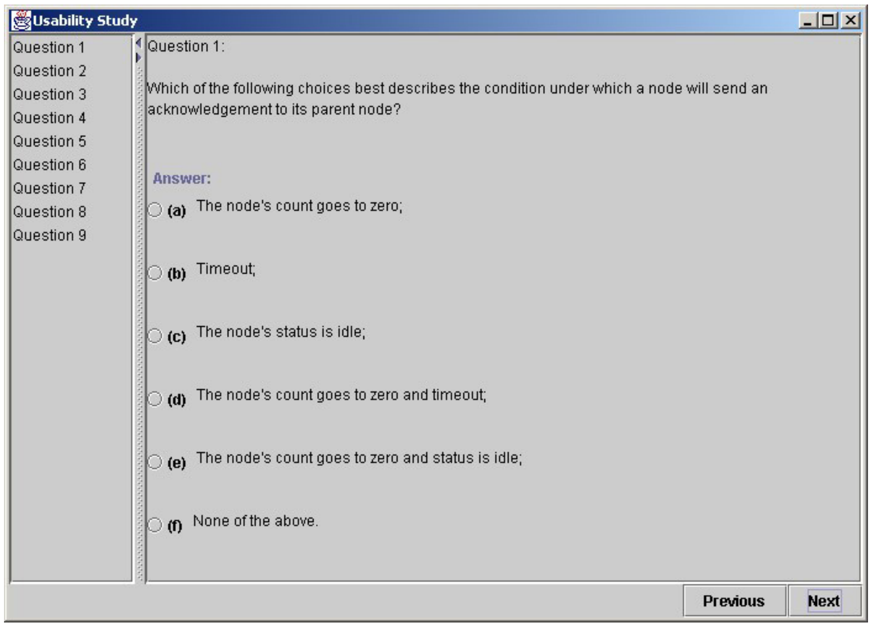
The limitations of this program animation architecture include a tedious task of reproducing known program views and a possible mismatch between the "real-world" semantics of some particular data, such as North and South, and the graphical feature chosen by the automatic presentation system. First, users that already know the visual appearance of a computation can input the exact values for the graphical attributes, but the process may become as difficult as with other visualization architectures. Second, the automatic presentation algorithm does not have the same knowledge as the user about the meaning of the information and about the appropriate layout for objects with unused positional values. The user has to inform the system about the right representations and layouts.

Future work will concentrate on evaluating the usability and usefulness of the visualization system. Particularly, it is important to determine the effectiveness of animations for users that start with no visualization and create a custom view while learning about the computation, as well as for users that have an existing program animation that needs to be refined.

Our plans also include the development of techniques that rely on time multiplexing to allow the user to observe multiple program views. The automatic presentation algorithm can be extended to partition the data streaming from the computation and to produce visualizations that are presented in turn, for a short period of time, to the user. An interesting visualization can be selected by the user and further refined. The system can also detect interesting events, or alarms, in the observed computation and briefly present them to maintain user's awareness and ensure that these events are not overlooked.

# Appendix A

# Questions for Study $\mathcal{A}$-Termination Detection



Question 1:

Which of the following choices best describes the condition under which a node will send an acknowledgement to its parent node?

Answer:

(a) The node's count goes to zero;

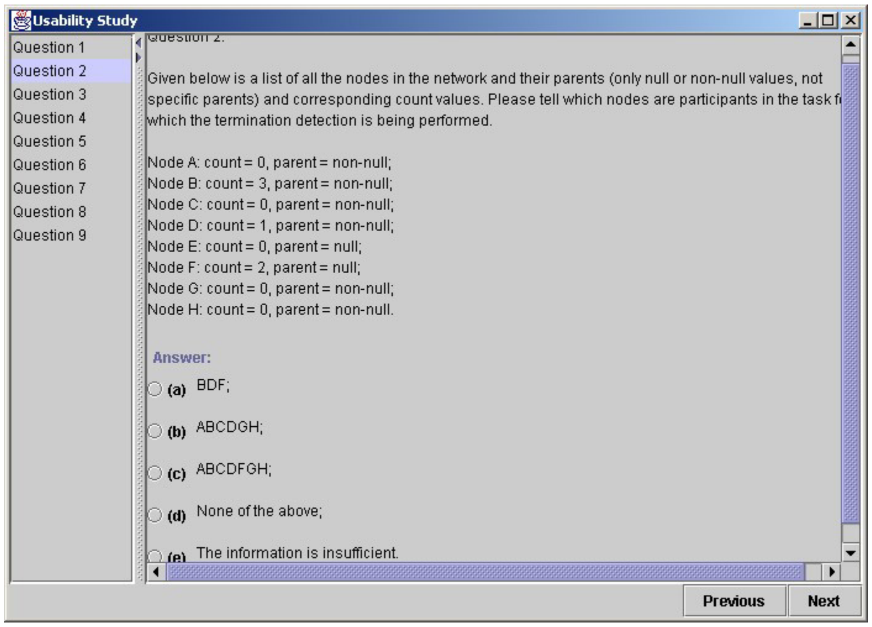
(b) Timeout;

(c) The node's status is idle;

(d) The node's count goes to zero and timeout;

(e) The node's count goes to zero and status is idle;

(f) None of the above.



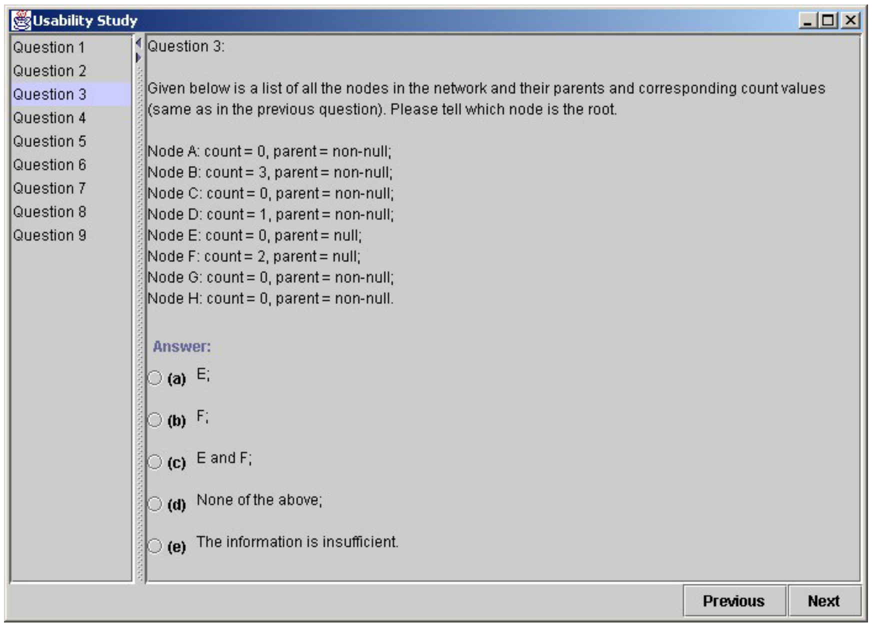
Question 2:

Given below is a list of all the nodes in the network and their parents (only null or non-null values, not specific parents) and corresponding count values. Please tell which nodes are participants in the task f
which the termination detection is being performed.

Node A: count = 0, parent = non-null;
Node B: count = 3, parent = non-null;
Node C: count = 0, parent = non-null;
Node D: count = 1, parent = non-null;
Node E: count = 0, parent = null;
Node F: count = 2, parent = null;
Node G: count = 0, parent = non-null;
Node H: count = 0, parent = non-null.

**Answer:**

- (a) BDF;
- (b) ABCDGH;
- (c) ABCDFGH;
- (d) None of the above;
- (e) The information is insufficient.



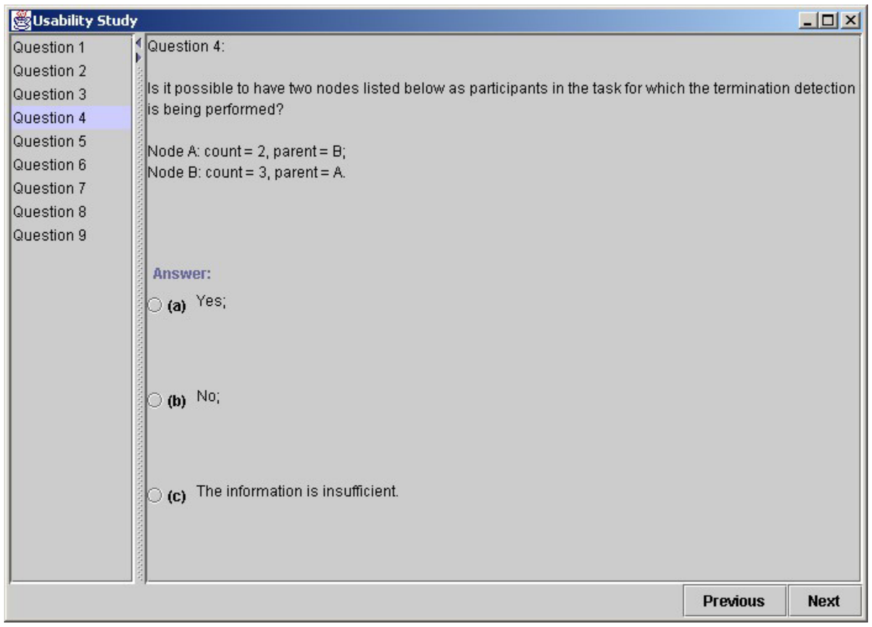
Question 3:

Given below is a list of all the nodes in the network and their parents and corresponding count values (same as in the previous question). Please tell which node is the root.

Node A: count = 0, parent = non-null;
Node B: count = 3, parent = non-null;
Node C: count = 0, parent = non-null;
Node D: count = 1, parent = non-null;
Node E: count = 0, parent = null;
Node F: count = 2, parent = null;
Node G: count = 0, parent = non-null;
Node H: count = 0, parent = non-null.

**Answer:**

- (a) E;
- (b) F;
- (c) E and F;
- (d) None of the above;
- (e) The information is insufficient.



Question 4:

Is it possible to have two nodes listed below as participants in the task for which the termination detection is being performed?

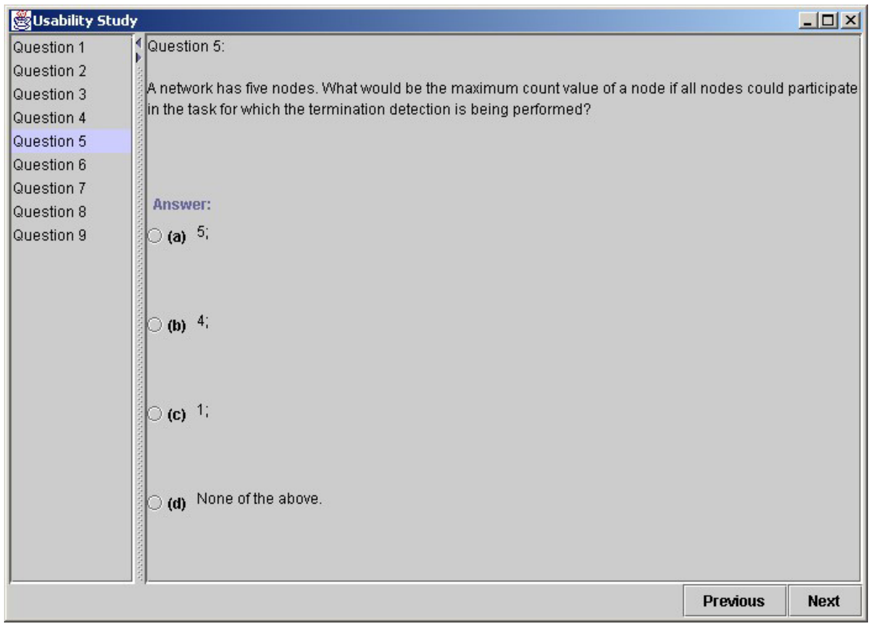
Node A: count = 2, parent = B;
Node B: count = 3, parent = A.

Answer:

(a) Yes;

(b) No;

(c) The information is insufficient.



Question 5:

A network has five nodes. What would be the maximum count value of a node if all nodes could participate in the task for which the termination detection is being performed?

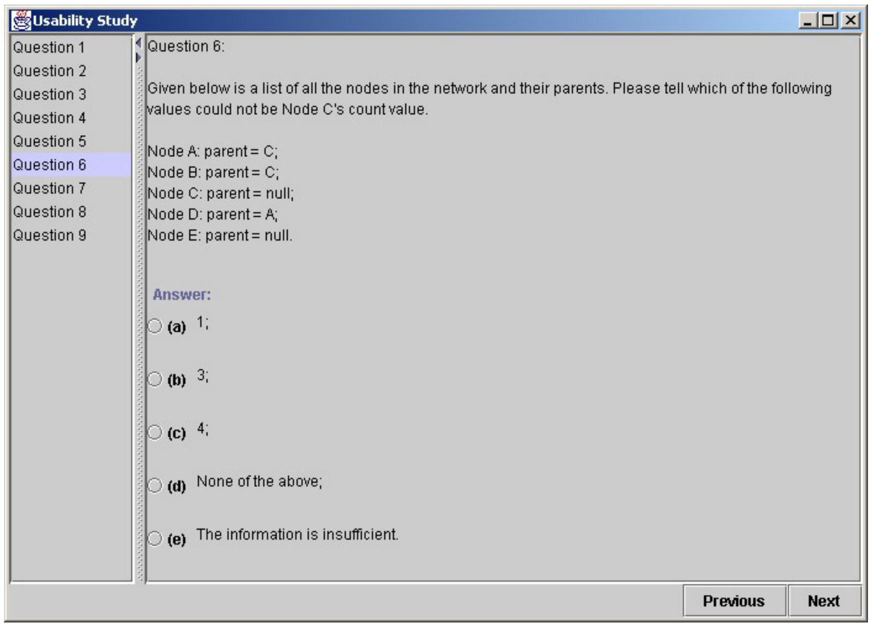
Answer:

(a) 5;

(b) 4;

(c) 1;

(d) None of the above.



Question 6:

Given below is a list of all the nodes in the network and their parents. Please tell which of the following values could not be Node C's count value.

Node A: parent = C;
Node B: parent = C;
Node C: parent = null;
Node D: parent = A;
Node E: parent = null.

Answer:

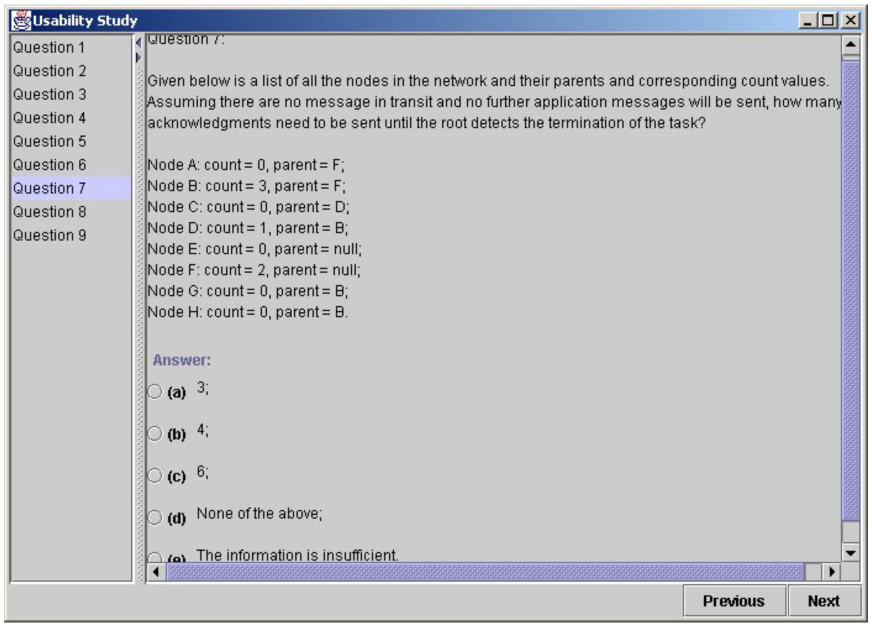
○ (a) 1;

○ (b) 3;

○ (c) 4;

○ (d) None of the above;

○ (e) The information is insufficient.



Question 7:

Given below is a list of all the nodes in the network and their parents and corresponding count values. Assuming there are no message in transit and no further application messages will be sent, how many acknowledgments need to be sent until the root detects the termination of the task?

Node A: count = 0, parent = F;
Node B: count = 3, parent = F;
Node C: count = 0, parent = D;
Node D: count = 1, parent = B;
Node E: count = 0, parent = null;
Node F: count = 2, parent = null;
Node G: count = 0, parent = B;
Node H: count = 0, parent = B.

Answer:

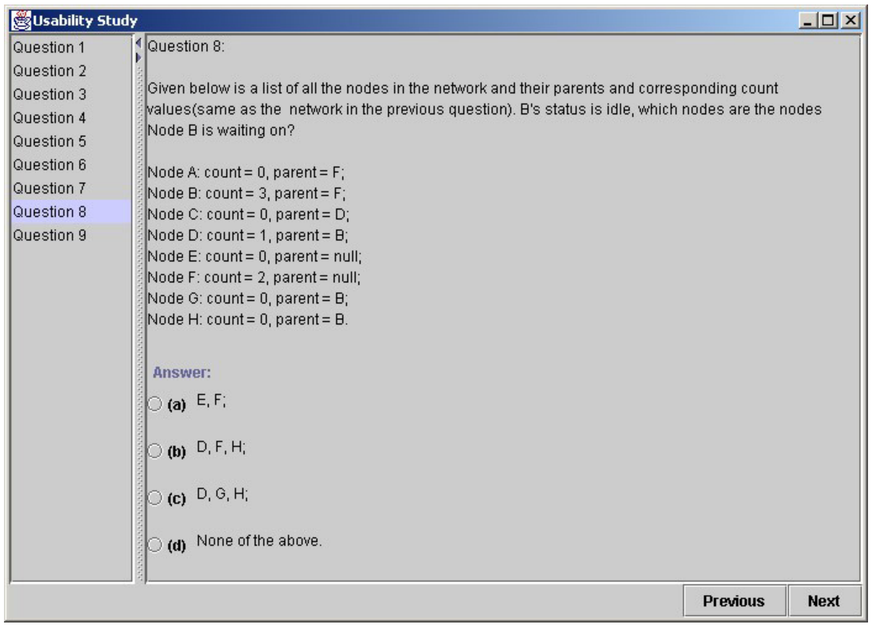
○ (a) 3;

○ (b) 4;

○ (c) 6;

○ (d) None of the above;

○ (e) The information is insufficient.



Question 8:

Given below is a list of all the nodes in the network and their parents and corresponding count values(same as the network in the previous question). B's status is idle, which nodes are the nodes Node B is waiting on?

Node A: count = 0, parent = F;
Node B: count = 3, parent = F;
Node C: count = 0, parent = D;
Node D: count = 1, parent = B;
Node E: count = 0, parent = null;
Node F: count = 2, parent = null;
Node G: count = 0, parent = B;
Node H: count = 0, parent = B.

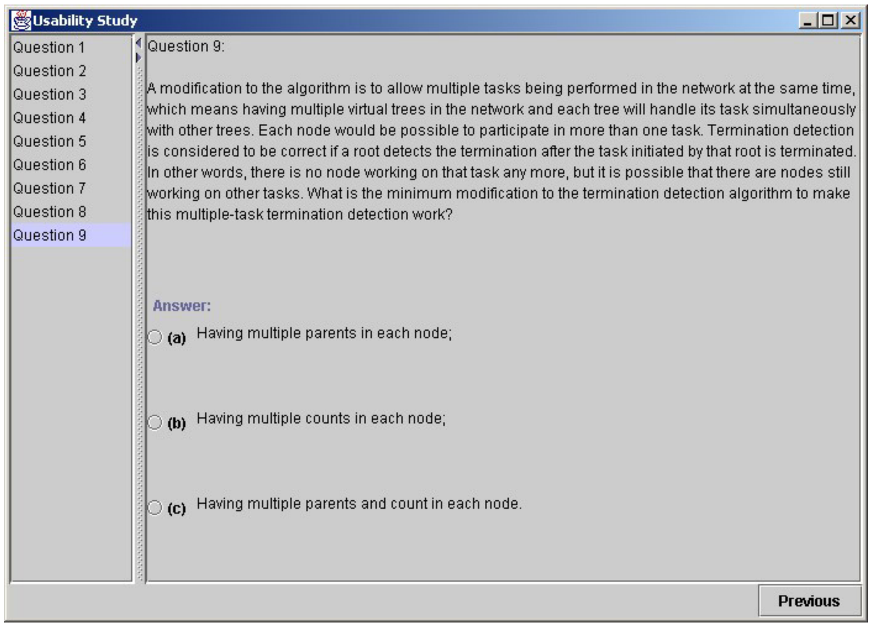
**Answer:**

(a) E, F;

(b) D, F, H;

(c) D, G, H;

(d) None of the above.



Question 9:

A modification to the algorithm is to allow multiple tasks being performed in the network at the same time, which means having multiple virtual trees in the network and each tree will handle its task simultaneously with other trees. Each node would be possible to participate in more than one task. Termination detection is considered to be correct if a root detects the termination after the task initiated by that root is terminated. In other words, there is no node working on that task any more, but it is possible that there are nodes still working on other tasks. What is the minimum modification to the termination detection algorithm to make this multiple-task termination detection work?

**Answer:**

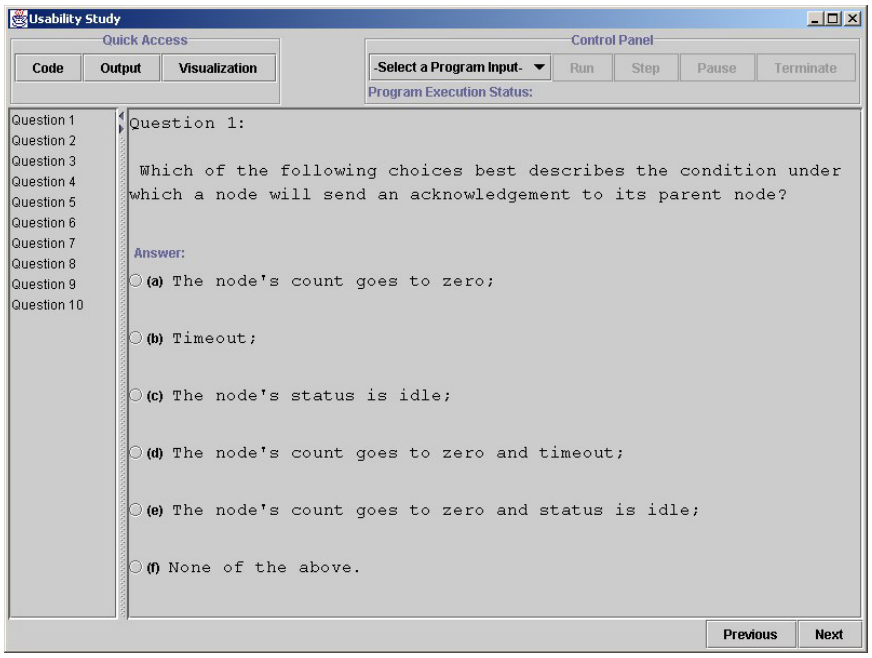
(a) Having multiple parents in each node;

(b) Having multiple counts in each node;

(c) Having multiple parents and count in each node.

# Appendix B

# Questions for Study $\mathcal{B}$



Question 1:

Which of the following choices best describes the condition under which a node will send an acknowledgement to its parent node?

Answer:

○ (a) The node's count goes to zero;

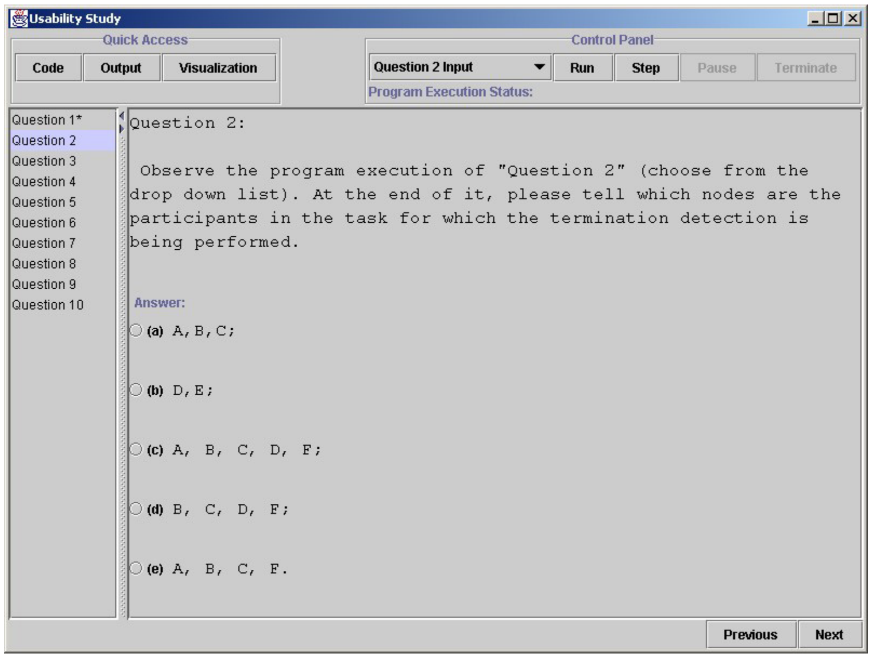
○ (b) Timeout;

○ (c) The node's status is idle;

○ (d) The node's count goes to zero and timeout;

○ (e) The node's count goes to zero and status is idle;

○ (f) None of the above.



Question 2:

Observe the program execution of "Question 2" (choose from the drop down list). At the end of it, please tell which nodes are the participants in the task for which the termination detection is being performed.

Answer:

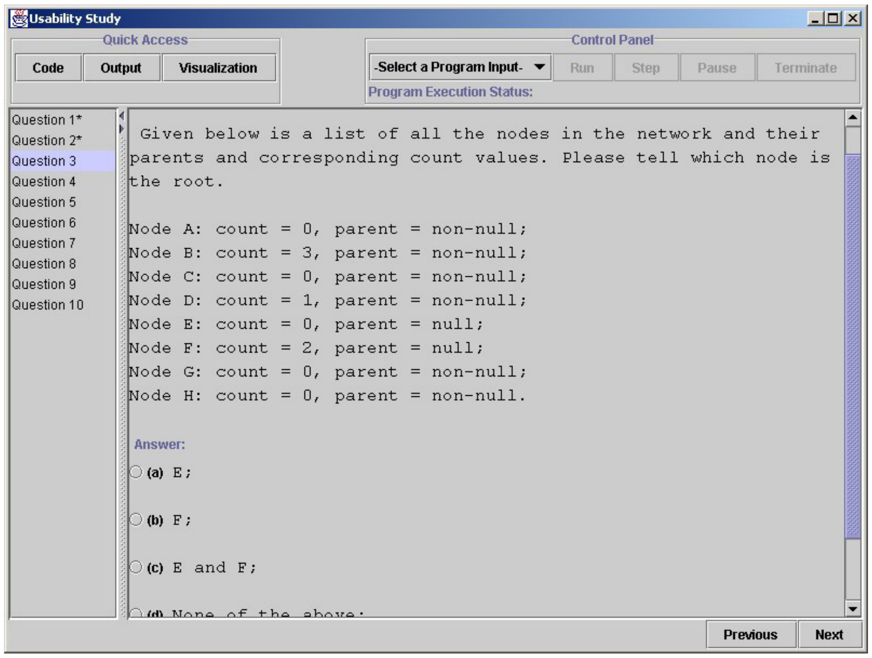
(a) A,B,C;

(b) D,E;

(c) A, B, C, D, F;

(d) B, C, D, F;

(e) A, B, C, F.



Given below is a list of all the nodes in the network and their parents and corresponding count values. Please tell which node is the root.

```
Node A: count = 0, parent = non-null;
Node B: count = 3, parent = non-null;
Node C: count = 0, parent = non-null;
Node D: count = 1, parent = non-null;
Node E: count = 0, parent = null;
Node F: count = 2, parent = null;
Node G: count = 0, parent = non-null;
Node H: count = 0, parent = non-null.
```

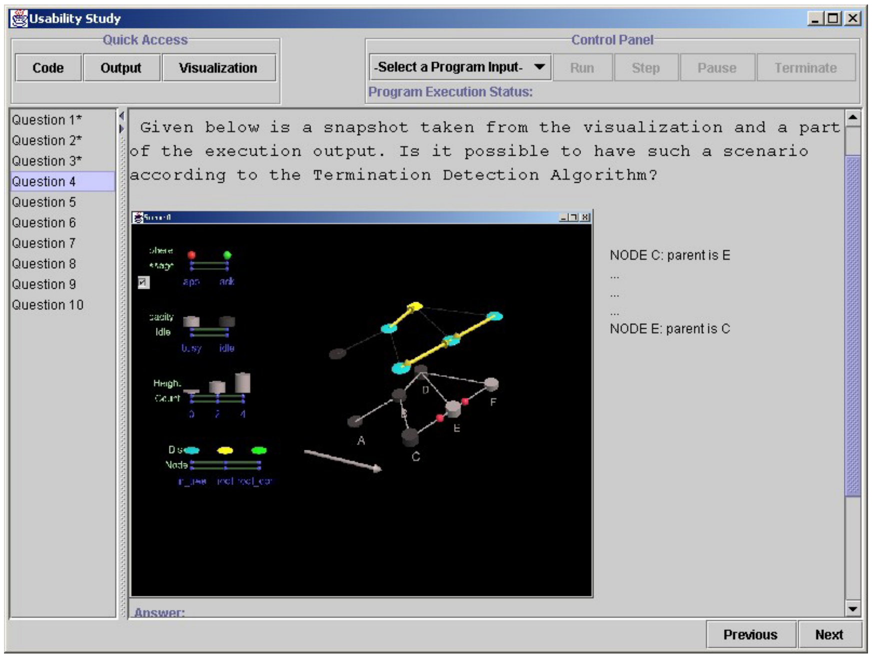
Answer:

(a) E;

(b) F;

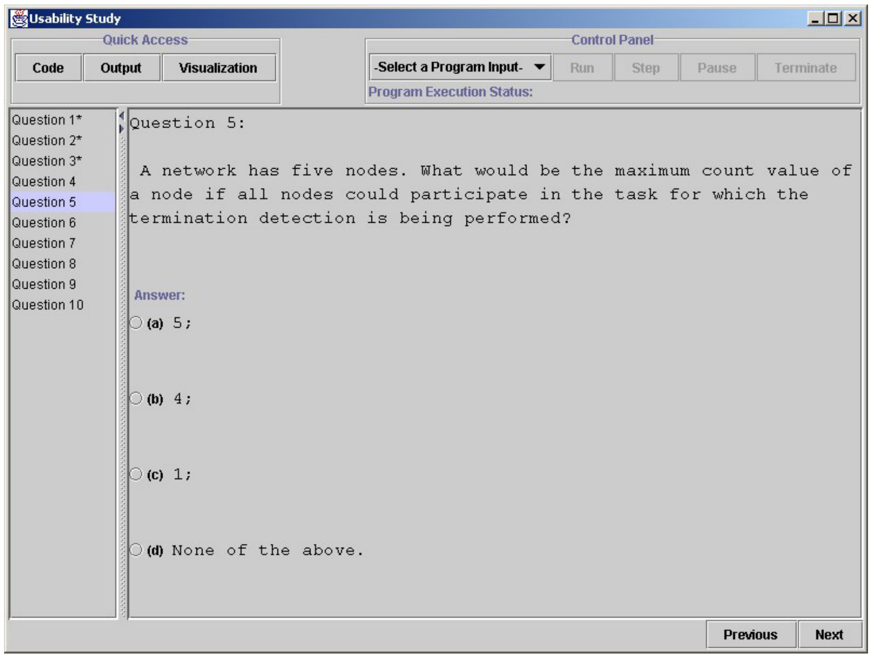
(c) E and F;

(d) None of the above;



Given below is a snapshot taken from the visualization and a part of the execution output. Is it possible to have such a scenario according to the Termination Detection Algorithm?

```
NODE C: parent is E
...
...
...
NODE E: parent is C
```



Question 5:

A network has five nodes. What would be the maximum count value of a node if all nodes could participate in the task for which the termination detection is being performed?

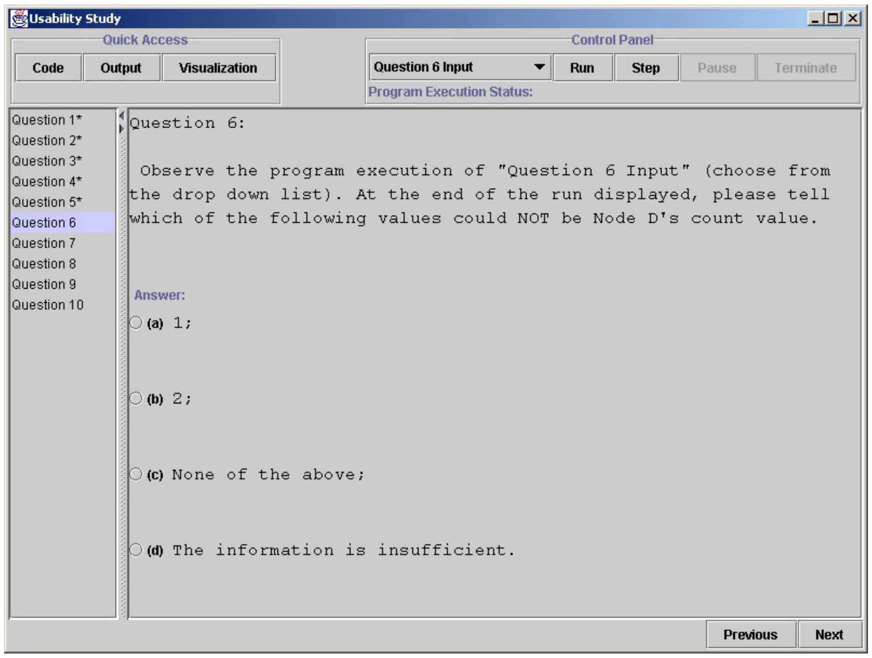
Answer:

(a) 5;

(b) 4;

(c) 1;

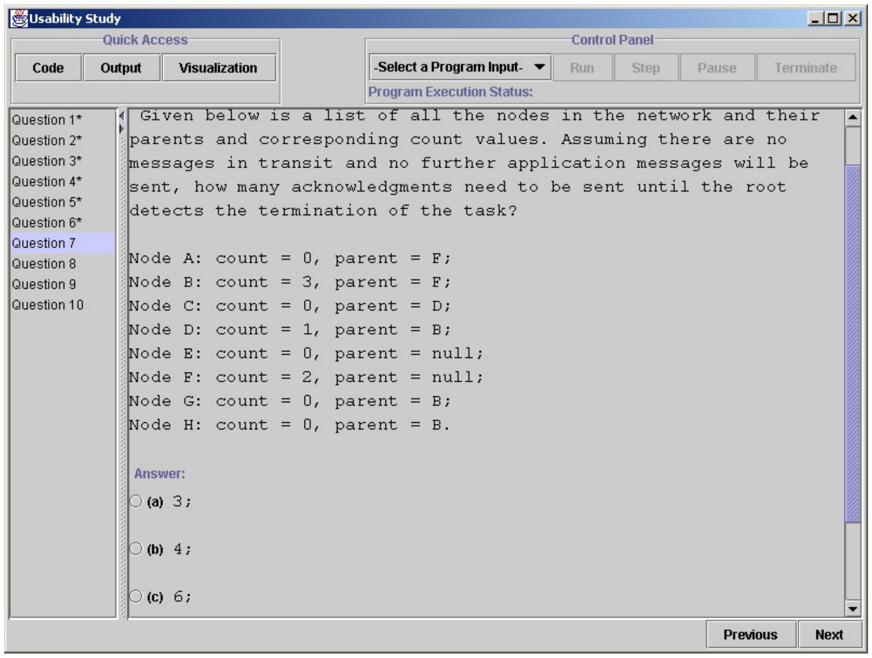
(d) None of the above.



```
Question 6:

 Observe the program execution of "Question 6 Input" (choose from
the drop down list). At the end of the run displayed, please tell
which of the following values could NOT be Node D's count value.
```

Answer:

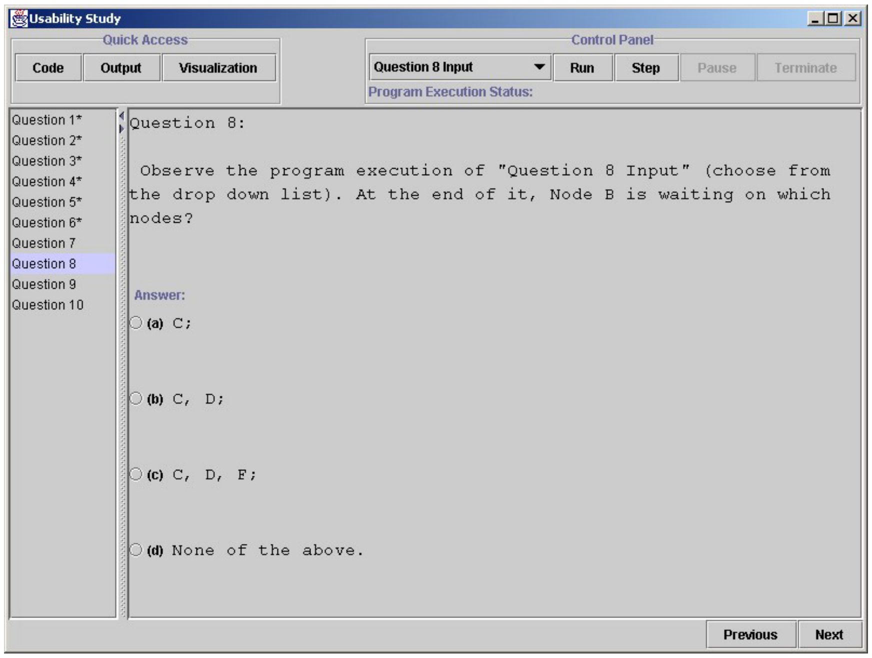
- (a) 1;
- (b) 2;
- (c) None of the above;
- (d) The information is insufficient.



```
 Given below is a list of all the nodes in the network and their
parents and corresponding count values. Assuming there are no
messages in transit and no further application messages will be
sent, how many acknowledgments need to be sent until the root
detects the termination of the task?

Node A: count = 0, parent = F;
Node B: count = 3, parent = F;
Node C: count = 0, parent = D;
Node D: count = 1, parent = B;
Node E: count = 0, parent = null;
Node F: count = 2, parent = null;
Node G: count = 0, parent = B;
Node H: count = 0, parent = B.
```

Answer:

- (a) 3;
- (b) 4;
- (c) 6;



```
Question 8:

 Observe the program execution of "Question 8 Input" (choose from
the drop down list). At the end of it, Node B is waiting on which
nodes?
```

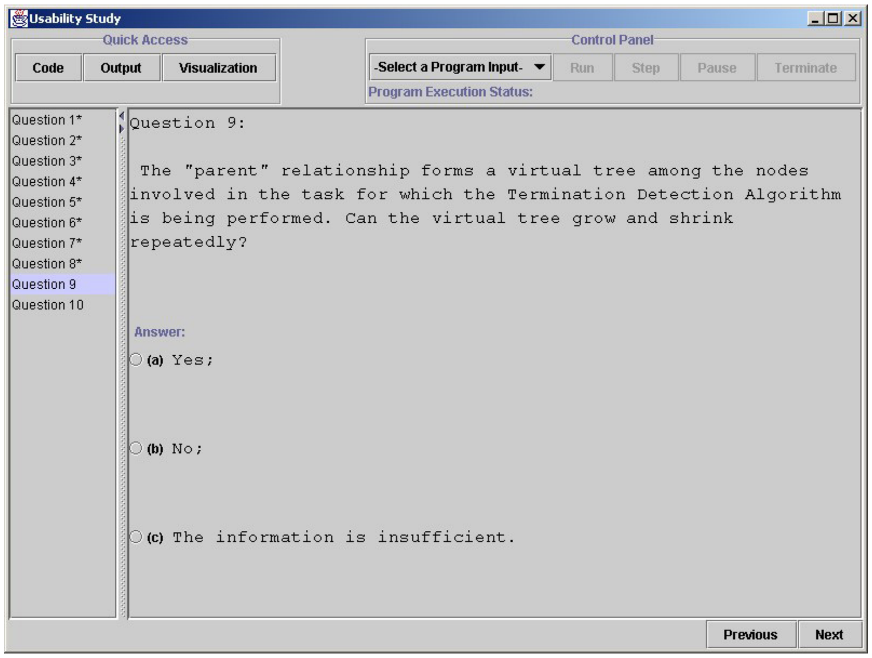
Answer:

(a) C;

(b) C, D;

(c) C, D, F;

(d) None of the above.



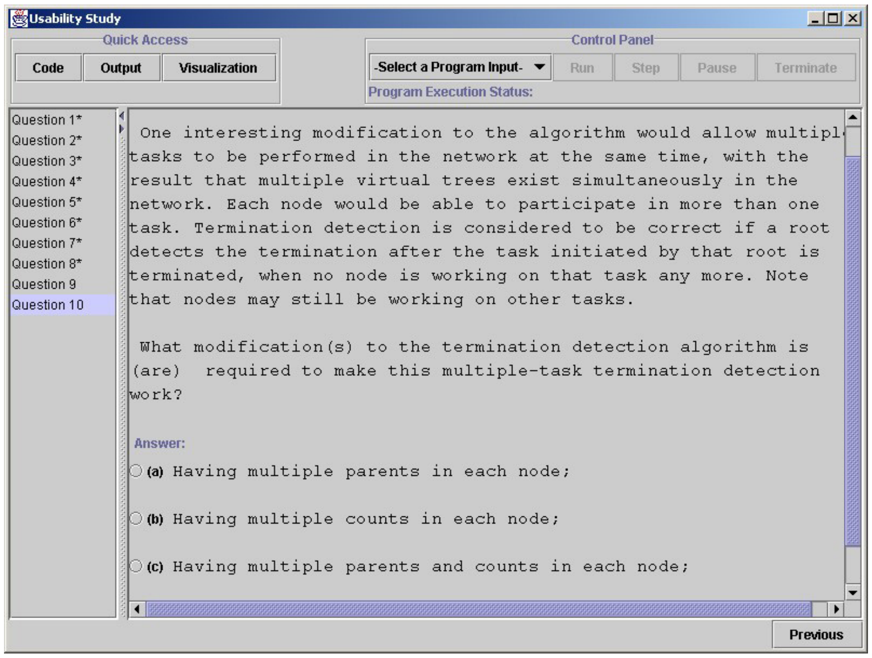
```
Question 9:

 The "parent" relationship forms a virtual tree among the nodes
involved in the task for which the Termination Detection Algorithm
is being performed. Can the virtual tree grow and shrink
repeatedly?
```

Answer:

(a) Yes;

(b) No;

(c) The information is insufficient.



```
 One interesting modification to the algorithm would allow multipl
tasks to be performed in the network at the same time, with the
result that multiple virtual trees exist simultaneously in the
network. Each node would be able to participate in more than one
task. Termination detection is considered to be correct if a root
detects the termination after the task initiated by that root is
terminated, when no node is working on that task any more. Note
that nodes may still be working on other tasks.

 What modification(s) to the termination detection algorithm is
(are)  required to make this multiple-task termination detection
work?
```

Answer:

(a) Having multiple parents in each node;

(b) Having multiple counts in each node;

(c) Having multiple parents and counts in each node;

# References

[1] *AVS Express*. http://www.avs.com.

[2] *VisiVue - Java Software Visualization Tool*. http://www.visicomp.com/product/visivue.html.

[3] *Visual Insights*. http://www.visualinsights.com.

[4] *Software Visualization: Programming as a Multimedia Experience*, chapter Software Visualization in Teaching at Brown University. M.I.T. Press, 1998.

[5] *Computational Science - ICCS 2001*, chapter Large-Scale Simulation and Visualization in Medicine: Applications to Cardiology, Neuroscience, and Medical Imaging. Springer-Verlag, 2001.

[6] Johnny Accot and Shumin Zhai. Beyond Fitts' law: models for trajectory-based HCI tasks. In *Proceedings of ACM CHI'97 Conference on Human Factors in Computing Systems*, pages 295–302, 1997.

[7] Johnny Accot and Shumin Zhai. Performance evaluation of input devices in trajectory-based tasks: an application of the steering law. In *Proceedings of ACM CHI'99 Conference on Human Factors in Computing Systems*, pages 466–472, 1999.

[8] Johnny Accot and Shumin Zhai. Scale effects in steering law tasks. In *Proceedings of ACM CHI'01 Conference on Human Factors in Computing Systems*, pages 1–8, 2001.

[9] Christopher Ahlberg and Ben Shneiderman. Visual information seeking: Tight coupling of dynamic query filters with starfield displays. In *Proceedings of ACM CHI'94 Conference on Human Factors in Computing Systems*, pages 313–317. ACM, 1994.

[10] L. Y. Arnaut and J. S. Greenstein. Is display/control gain a useful metric for optimizing an interface? *Human Factors*, 32(6):651–663, 1990.

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

[12] R. Balakrishnan and I. S. MacKenzie. Performance differences in the fingers, wrist, and forearm in computer input control. In *Proceedings of ACM CHI'97 Conference on Human Factors in Computing Systems*, pages 303–310, 1997.

[13] Benjamin B. Bederson and James D. Hollan. Pad++: A zooming graphical interface for exploring alternate interface physics. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, Visualization I, pages 17–26, 1994.

[14] Benjamin B. Bederson and Britt McAlister. Jazz: An extensible 2D+zooming graphics toolkit in java. Technical Report CS-TR-4015, University of Maryland, College Park, May 1999.

[15] J. Bertin. *Semiology of Graphics.* The University of Wisconsin Press, 1983.

[16] M. H. Brown. Exploring Algorithms using Balsa-II. *IEEE Computer*, 21(5):14–36, 1988.

[17] Marc H. Brown and Marc A. Najor. Algorithm animation using interactive 3d graphics. In John Stasko, John Domingue, Marc Brown, and Blaine Price, editors, *Software Visualization*, chapter 9, pages 119–136. The MIT Press, Cambridge, 1998.

[18] L. Buck. Motor performance in relation to controldisplay gain and target width. *Ergonomics*, 23(6):579–589, 1980.

[19] M. Byrne, R. Catrambone, and J. Stasko. Evaluating animations as student aids in learning computer algorithms. *Computers & Education*, 33(4):253–278, 1999.

[20] Stephen M. Casner. A task-analytic approach to the automated design of graphic presentations. *ACM Transactions on Graphics*, 10(2):111–151, 1991.

[21] Mei C. Chuah, Steven F. Roth, Joe Mattis, and John Kolojejchick. SDM: Selective dynamic manipulation of visualizations. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, 3D User Interfaces, pages 61–70, 1995.

[22] D. Brookshire Conner, Scott S. Snibbe, Kenneth P. Herndon, Daniel C. Robbins, Robert C. Zeleznik, and Andries van Dam. Three-dimensional widgets. In *Proceedings of the 1992 Symposium on Interactive 3D Graphics, Special Issue of Computer Graphics, Vol. 26*, pages 183–188, 1992.

[23] K.C. Cox and G.-C. Roman. A Characterization of the Computational Power of Rule-Based Visualization. *Journal of Visual Languages and Computing*, 5(1):5–27, 1994.

[24] Kenneth J. W. Craik. *The Nature of Explanation.* Cambridge University Press, Cambridge, 1943.

[25] P. Crescenzi, C. Demetrescu, I. Finocchi, and R. Petreschi. Reversible execution and visualization of programs with leonardo. *Journal of Visual Languages and Computing*, 11(2):125–150, 2000.

[26] H.A. Cunningham. Aiming error under transformed spatial mappings suggests a structure for visual-motor maps. *Journal of Experimental Psychology: Human Perception and Performance*, 15(3):493–506, 1989.

[27] Brian J. d'Auriol, Claudia V. Casas, Pramod K. Chikkappaiah, L. Susan Draper, Ammar J. Esper, Jorge López, Rajesh Molakaseema, Seetharami R. Seelam, René Saenz, Qian Wen, and Zhengjing Yang. Exploratory study of scientific visualization techniques for program visualization. In *International Conference on Computational Science ICCS 2001*, volume II, pages 701–710. Springer-Verlag, 2001.

[28] C. Demetrescu and I. Finocchi. Smooth animation of algorithms in a declarative framework. *Journal of Visual Languages and Computing*, 12(3), 2001.

[29] Edsger W. Dijkstra and C.S.Scholten. Termination detection for diffusing computations. *Inf. Proc. Letters*, 11(1):1–4, 1980.

[30] M. Eisenstadt and M. Brayshaw. The transparent prolog machine (TPM): An execution model and graphical debugger for logic programming. *Journal of Logic Programming*, 5(4):1–66, 1988.

[31] Ken Fishkin and Maureen C. Stone. Enhanced dynamic queries via movable filters. In *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems*, volume 1 of *Papers: Information Visualization*, pages 415–420, 1995.

[32] P.M. Fitts. The information capacity of the human motor system in controlling the amplitude of movement. *Journal of Motor Behavior*, 47(6):381–391, June 1954.

[33] Eugene C. Freuder. Partial constraint satisfaction. pages 278–283, Detroit, MI, 1989. Morgan Kaufmann.

[34] G. W. Furnas. Generalized fisheye views. In *Proceedings of ACM CHI'86 Conference on Human Factors in Computing Systems*, pages 16–23, 1986.

[35] C. Gibbs. Controller design: Interactions of controlling limbs, time-lags, and gains in positional and velocity systems. *Ergonomics*, 5:385–402, 1962.

[36] Yves Guiard, Michel Beaudouin-Lafon, and Deni Mottet. Navigation as multiscale pointing: Extending fitts' model to very high precision tasks. In *Proceedings of ACM CHI 99 Conference on Human Factors in Computing Systems*, volume 1 of *Vision and Fitts' Law*, pages 450–457, 1999.

[37] R. Haller, H. Mutschler, and M. Voss. Comparison of input devices for correction of errors in office systems. In *INTERACT '84*, 1984.

[38] Ashley Hamilton-Taylor and Eileen Kraemer. SKA: Supporting algorithm and data structure discussion. In John Impagliazzo, editor, *Proceedings of the Thirty-third SIGCSE Technical Symposium on Computer Science Education (SIGCSE-02)*, volume 34, 1 of *SIGCSE Bulletin*, pages 58–62, New York, February 27– March 3 2002. ACM Press.

[39] S. Hansen, D. Schrimpsher, and N. Narayanan. Learning algorithms by visualization: A novel approach using animation-embedded hypermedia. In *Proc. Third International Conference on The Learning Sciences, Atlanta, GA*, 1998.

[40] Delbert Hart, Eileen Kraemer, and Gruia-Catalin Roman. Consistency considerations in the interactive steering of computations. *International Journal of Parallel and Distributed Systems and Networks*, 2(3):171–179, 1999.

[41] Michael T. Heath and Jennifer A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29 –39, September 1991.

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

[43] William Hibbard, Charles R. Dyer, and Brian Paul. Display of scientific data structures for algorithm visualization. In *Proc. IEEE Visualization*, pages 139–146, 1992.

[44] C. D. Hundhausen and S. A. Douglas. Low fidelity algorithm visualization. *Journal of Visual Languages and Computing*, 2001. Under review.

[45] C.D. Hundhausen, S.A. Douglas, and J.T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing*. In press.

[46] Christopher D. Hundhausen. Integrating algorithm visualization technology into an undergraduate algorithms course: Ethnographic studies of a social constructivist approach. *Computers & Education*. Under review.

[47] National Imagery and Mapping Agency. Geo-spatial intelligence information visualization program (GI2Vis). http://www.nima.mil/.

[48] H. Jellinek and S. Card. Powermice and user performance. In *Proceedings of ACM CHI'90 Conference on Human Factors in Computing Systems*, pages 213–220, 1990.

[49] *Jinsight - Visualisation tools for Java*. http://www.research.ibm.com/jinsight/.

[50] Colleen Kehoe, John Stasko, and Ashley Taylor. Rethinking the evaluation of algorithm animations as learning aids: An observational study. *International Journal of Human-Computer Studies*, 54(2):265–284, 2001.

[51] Eileen Kraemer, Mihail E. Tudoreanu, and Ashley Taylor. Why johnny won't visualize? In *Workshop on Software Visualization at ICSE 2001*, 2001.

[52] G. Langolf and D. Chaffin. An investigation of fitts' law using a wide range of movement amplitudes. *Journal of Motor Behavior*, 8:113–128, 1976.

[53] J. Lave and E. Wenger. *Situated Learning: Legitimate Peripheral Participation.* Cambridge University Press, New York, 1991.

[54] Andrea Lawrence, Albert Badre, and John T. Stasko. Empirically evaluating the use of animations to teach algorithms. In *Proceedings of the 1994 IEEE Symposium on Visual Languages, St. Louis, MO*, pages 48–54, October 1994.

[55] I. S. MacKenzie. Fitts' law as a research and design tool in human-computer interaction. *Human-Computer Interaction*, 7(1):91–139, 1992.

[56] I. S. MacKenzie and S. Jusoh. An evaluation of two input devices for remote pointing. In *Proceedings of the Eighth IFIP Working Conference on Engineering for Human Computer Interaction - EHCI.* Springer-Verlag, 2001.

[57] I. S. MacKenzie and C. Ware. Lag as a determinant of human performance in interactive systems. In *Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems*, pages 488–493, 1993.

[58] Jock D. Mackinlay. Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics*, 5(2):110–141, 1986.

[59] Charles C. Mann. Why software is so bad. *Technology Review, Inc.*, June 2002. http://www.msnbc.com/news/768401.asp?0si=-&cp1=1.

[60] Sougata Mukherjea and John T. Stasko. Toward visual debugging: Integrating algorithm animation capabilities within a source level debugger. *ACM Transactions on Computer-Human Interaction*, 1(3):215–244, September 1994.

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

[62] Pu Pearl and Denis Lalanne. Interactive problem solving via algorithm visualization. In *Proceedings of IEEE Information Visualization*, pages 145–153, 2000.

[63] K. Perlin and D. Fox. Pad: An Alternative Approach to the Computer Interface. In *ACM SIGGRAPH'93*, pages 57–64, Anaheim, CA, 1993. ACM press.

[64] C. Prablanc, J.E. Echallier, M. Jeannerod, and E. Komilis. Optimal response of eye and hand motor systems in pointing at a visual target ii: Static and dynamic visual cues in the control of hand movement. *Biological Cybernetics*, (35):183–187, 1979.

[65] J. Preece, Y. Rogers, H. Sharp, D. Benyon, S. Holland, and T. Carey. *Human-Computer Interaction*. Addison-Wesley, 1994.

[66] Blaine A. Price, Ronald M. Baecker, and Ian S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, September 1993.

[67] Ramana Rao and Stuart K. Card. The table lens: Merging graphical and symbolic representations in an interactive focus+context visualization for tabular information. In Beth Adelson, Susan Dumais, and Judith Olson, editors, *Proceedings of the Conference on Human Factors in Computing Systems*, pages 318–322, New York, NY, USA, April 1994. ACM Press.

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

[69] Guido Roessling and Bernd Freisleben. The animal algorithm animation tool. In *ACM 5th Annual Conference on Innovation and T echnology in Computer Science Education (ITiCSE 2000)*, pages 37–40, New York, 2000. ACMPress.

[70] Guido Roessling and Bernd Freisleben. ANIMALSCRIPT: An extensible scripting language for algorithm animation. In *Proceeding of the Thirty-second SIGCSE Technical Symposium on Computer Sciense Education (SIGCSE-01)*, volume 33.1 of *ACM Sigcse Bulletin*, pages 70–74, New York, February 2001. ACMPress.

[71] G.-C. Roman, D. Hart, and C. Calkins. Visual Presentation of Software Specifications and Designs. In *Eighth International Workshop on Software Specification and Design*, pages 115–124. Washington University, Department of Computer Science, St. Louis, Missouri, 1996.

[72] G.C. Roman and K. Cox. A Taxonomy of Program Visualization Systems. *IEEE Computer*, 26(12):11–24, 1993.

[73] Gruia-Catalin Roman. Declarative visualization. In John Stasko, John Domingue, Marc Brown, and Blaine Price, editors, *Software Visualization*, chapter 13, pages 173–186. The MIT Press, Cambridge, 1998.

[74] Gruia-Catalin Roman, Kenneth C. Cox, Donald Wilcox, and Jerome Y. Plun. Pavane: a system for declarative visualization of concurrent computations. *Journal of Visual Languages and Computing*, 3(2):161–193, 1992.

[75] Steven F. Roth, Peter Lucas, Jeffrey A. Senn, Cristina, C. Gomberg, Michael B. Burks, Philip J. Stroffolino, John A. Kolojejchick, and Carolyn Dunmire. Visage: A user interface environment for exploring information. In *Proceedings of IEEE Information Visualization*, pages 3–12, San Francisco, October 1996.

[76] Steven F. Roth and Joe Mattis. Automating the presentation of information. pages 90–97, Miami Beach, FL, 1991.

[77] M. Scaife and Y. Rogers. External cognition: How do graphical representations work? *International Journal of Human-Computer Studies*, 45:185–213, 1996.

[78] Andrew Sears and Ben Shneiderman. High precision touchscreens: Design strategies and comparisons with a mouse. *International Journal of Man-Machine Studies*, 34(4):593–613, 1991.

[79] B. Shneiderman. *Designing the User Interface (3rd edition)*. Addison-Wesley, 1998.

[80] Ben Shneiderman. Dynamic queries for visual information seeking. *IEEE Software*, 11(6):70–77, November 1994.

[81] John Stasko. Samba. http://www.cc.gatech.edu/gvu/softviz/algoanim/samba.html.

[82] John Stasko. Smooth continuous animation for portraying algorithms and processes. In John Stasko, John Domingue, Marc Brown, and Blaine Price, editors, *Software Visualization*, chapter 8, pages 103–118. The MIT Press, Cambridge, 1998.

[83] John Stasko, Albert Badre, and Clayton Lewis. Do algorithm animations assist learning? an empirical study and analysis. In *Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems*, Understanding Programming, pages 61–66, 1993.

[84] John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors. *Software Visualization: Programming as a Multimedia Experience.* M.I.T. Press, February 1998.

[85] John T. Stasko. The Path-Transition Paradigm: A practical methodology for adding animation to program interfaces. *Journal of Visual Languages and Computing*, 1(3):213–236, 1990.

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

[87] John T. Stasko. Supporting student-built algorithm animation as a pedagogical tool. In *Proceedings of ACM CHI 97 Conference on Human Factors in Computing Systems*, volume 2 of *DEMONSTRATIONS: Programming with Less Programming*, pages 24–25, 1997.

[88] John T. Stasko and Eileen Kraemer. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing*, 18(2):258–264, 1993.

[89] John T. Stasko and Eileen Kraemer. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing*, 18(2):258–264, 1993.

[90] Sun Microsystems, Inc. Java 3D (TM). http://java.sun.com/products/java-media/3D/index.html.

[91] V.S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice & Experience*, 2(4):315 – 339, December 1990.

[92] Andrew S. Tanenbaum. *Computer Networks.* Prentice-Hall International, Inc., 1996. ISBN: 0-13-394248-1.

[93] Brad Topol, John T. Stasko, and Vaidy S. Sunderam. Pvanim: a tool for visualization in network computing environments. *Concurrency - Practice and Experience*, 10(14):1197–1222, 1998.

[94] Mihail E. Tudoreanu and Eileen Kraemer. Legends as a device for interacting with visualizations. Technical Report WUCS-01-44, Washington University in St. Louis, 2001.

[95] Lisa Tweedie, Bob Spence, David Williams, and Ravinder Bhogal. The attribute explorer. In *Proceedings of the CHI '94 conference companion on Human factors in computing systems*, pages 435–436. ACM Press, 1994.

[96] Colin Ware. *Information Visualization.* Morgan Kaufmann, 2000.

[97] Colin Ware and Jeff Rose. Rotating virtual objects with real handles. *ACM Transactions on Computer-Human Interaction*, 6(2):162–180, 1999.

[98] Kent Wittenburg, Tom Lanning, Michael Heinrichs, and Michael Stanton. Parallel bargrams for consumer-based information exploration and choice. In *Proceedings of the 14th annual ACM symposium on User interface software and technology*, pages 51–60. ACM Press, 2001.

[99] W. Woodson, B. Tillman, and P. Tillman. *Human Factors Design Handbook (second edition).* McGrawHill, 1992.

[100] Shumin Zhai, Paul Milgram, and William Buxton. The influence of muscle groups on performance of multiple degree-of-freedom input. In *Proceedings of ACM CHI 96 Conference on Human Factors in Computing Systems*, volume 1 of *PAPERS: Fingers*, pages 308–315, 1996.

# Vita

Mihail-Eduard Tudoreanu

| | |
|---|---|
| **Date of Birth** | February 13, 1974 |
| **Place of Birth** | Iasi, Romania |
| **Degrees** | D.Sc. Computer Science, August 2002 (expected)<br>B.S. Computer Science, June 1997<br>M.S. Computer Science, May 1999 |
| **Research Interests** | Program and algorithm visualization<br>Human-computer interaction<br>Automatic visual presentation of computations<br>Monitoring and steering of distributed computations<br>Mobile code |
| **Teaching Interests** | Information and program visualization<br>Human-computer interaction and user interfaces<br>Distributed computation and mobility<br>Software engineering |
| **Professional Societies** | Association for Computing Machinery<br>Institute of Electrical and Electronic Engineers (IEEE)<br>IEEE Computer. |
| **Publications** | Tudoreanu, M. E., Wu R., Hamilton-Taylor, A., Kraemer, E. (2002) Empirical Evidence that Algorithm Animation Promotes Understanding of Distributed Algorithms, *To appear in Empirical Studies of Programmers: Individual Symposium within 2002 IEEE Symposia on Human Centric Computing Languages and Environments (HCC'02).*<br><br>Tudoreanu, M. E.; Kraemer, E. (2001) Automatic Presentation of Running Programs, *Proceedings of the SPIE 2001 Conference on Visual Data Exploration and Analysis VIII*, p.143-155, San Jose, CA. |

Tudoreanu, M. E., Kraemer E. (2001) Legends as a Device for Interacting with Visualizations, *Washington University technical report*, WUCS-01-44, St. Louis, MO.

Hart, D.; Tudoreanu, M. E. (2001) Visualization Channels: Time Multiplexing on a Display, *Proceedings of IASTED International Conference on Visualization, Imaging and Image Processing*, p.95-100, Marbella, Spain.

Kraemer, E.; Tudoreanu, M. E.;Taylor A. (2001) Why Johnny Won't Visualize, *Workshop on Software Visualization at ICSE 2001*, Toronto, Canada.

Hart, D.; Tudoreanu, M. E.; Kraemer, E. (2001) Token Finding Using Mobile Agents, *Proceedings of the International Conference of Computational Science*, vol. 2, p.791-800, San Francisco, CA.

Hart, D.; Tudoreanu, M. E.; Kraemer, E. (2001) Mobile Agents for Monitoring Distributed Systems, *Proceedings of the Fifth International Conference on Autonomous Agents*, p.232-233, Montreal, Canada.

Tudoreanu, M. E.; Hart, D.; Roman, G.-C. (2000) Reshapeable Visualizations, *Proceedings of the IEEE Multimedia Software Engineering*, p.245-250, Taipei, Taiwan.

Tudoreanu, M. E., Kraemer E. A Study of the Performance of Steering Tasks under Spatial Transformation of Input, *To be submitted to Transactions of Computer-Human Interaction (TOCHI).*

August 2002