THE STUDY AND DESIGN OF ALGORITHM ANIMATIONS

by

ASHLEY GEORGE HAMILTON-TAYLOR

(Under the Direction of Eileen Kraemer)

ABSTRACT

Algorithm Animations (AAs) portray the high-level dynamic operation of an algorithm. The computer science education community held great expectations that AA would assist students in understanding algorithms. However, many studies of the instructional effectiveness of AAs have produced inconclusive results. We investigated a number of issues pertinent to AA effectiveness: the study of AA user needs, user-centered design and the role of perception in AA.

Existing algorithm animation systems typically have been designed without formal study of related teaching practices. We conducted an observational study of instructors teaching data structure and algorithm topics, focusing on activities involving the use of diagrams and algorithms. The results of this study were used to inform the user-centered design of SKA, the Support Kit for Animation. SKA combines interactive data structure diagram manipulation with flexible pseudocode execution, simple algorithm animation authoring support, a visual data structure library, and an animation engine designed for perceptual pacing and timing.

The role of perception in AAs had not been formally considered in the past. We collaborated on a number of empirical studies to investigate this role, and the design of software to be used to conduct these studies. We found that some animation techniques can assist user

perception and mapping in AA in some contexts, which will inform future AA design and

studies.

INDEX WORDS: Algorithm Animation, Software Visualization, Algorithm Visualization,
                 Program Visualization, Human-Computer Interaction, Educational
                 Technology, Perceptual Study, Perception, Perceptual Processing,
                 Empirical Study, Observational Study, Usability, User Interface Design

THE STUDY AND DESIGN OF ALGORITHM ANIMATIONS

by

ASHLEY GEORGE HAMILTON-TAYLOR

B.Sc., University of the West Indies, Mona, Jamaica, 1985

M.S., University of Illinois Urbana-Champaign, 1991

A Dissertation Submitted to the Graduate Faculty of The University of Georgia in Partial

Fulfillment of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

ATHENS, GEORGIA

2006

THE STUDY AND DESIGN OF ALGORITHM ANIMATIONS

by

ASHLEY GEORGE HAMILTON-TAYLOR

Major Professor: Eileen Kraemer

Committee: Robert Branch
                     Elizabeth Davis
                     Maria Hybinette

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
August 2006

DEDICATION

As I prepared this, I sat by the waters of the King Center, and a thought came to me: perhaps life is as much about what we learn on the journey as what we achieve.

May there be peace on this earth for all its inhabitants, and especially for all my family: my mother Nana Farika, wife Marie, children Kamilah and Gabrielle, brothers Marcus and Gize, sister Sahai, and my many relatives.

ACKNOWLEDGEMENTS

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.0   Introduction and Motivation

What is Algorithm Animation? An Algorithm Animation (AA) is a visualization of an

algorithm, usually intended for educational use [PBS93]. Algorithms are specified using static

textual code (or static graphical notation) that does not readily reflect the dynamic behavior of

the algorithms during execution. Algorithm animations are intended to portray the dynamic

operation of algorithms and, in some cases, their underlying concepts.

Algorithms and data structures have traditionally been regarded as a core area of

computer science that students find difficult [Denning89]. The advent of algorithm animation

created great expectations in the computer science teaching community, as animation was seen

as a better way than purely static media to illustrate the dynamic workings of algorithms. Marc

Brown , an AA pioneer stated that "multiple, dynamic, graphical displays of an algorithm reveal

properties that might otherwise be difficult to comprehend or even remain unnoticed"

[Brown87]. However, these expectations have been met only partially, at best.

In the early era of AA, researchers had a vision of the future that would see students and

instructors using these animations as an integral part of the learning process, perhaps integrated

with notes and using wireless notebooks, as in Alan Kay's Dynabook [Kay81]. They would

interact with animations shown by instructors, as in the interactive classroom at Brown

University [Bazik98]. While the technology and systems to support this vision have emerged,

AA use has not been integrated into the practice of instruction [Naps03] or study on a pervasive basis.

The early thrust of AA research emphasized AA system design. This focus was common for educational technology development during that period [Smith86], when it was assumed that the technology would be beneficial. This was followed by a period of reflection during which initial studies of AA effectiveness were conducted [HDS02].

However, while some studies of effectiveness have shown instructional benefits [LBS94][KST01][THRK02] for the use of algorithm animation, many studies are inconclusive [BCS96][Jarc98], and a few had negative results [Mulholland98]. In a meta-study [HDS02] of 24 AA studies, 11 had significantly positive results, 10 had no significant results, and 1 had a significantly negative result. However, the studies generally could not point to definitive reasons for their failure or success, and thus tend to speculate in their conclusions. The mixed results of these studies has prompted further studies and attempts to improve effectiveness, but a similar pattern of results has continued [Saraiya04].

1.1   Algorithm Animation Studies

Studies of AA effectiveness have examined a number of usage scenarios, including lecture presentations, laboratory use, self-study, and AA construction. Hundhausen notes that studies that involved active use of AAs, such as data set construction, prediction or interaction, had a higher rate of significant positive results than studies involving passive viewing [HDS02]. However, for every usage scenario, there are inconclusive studies. We contemplated some possible reasons for this in [KTH01].

Lawrence conducted studies that were set in all three scenarios and found that a group that used input data set construction with animation scored significantly higher on a post-test, compared to a control group that did not use animations [LBS94]. [Saraiya04] found that a group of students provided with a good predefined data set scored significantly higher on a post-test, compared to a group of students that constructed their input data sets.

Jarc found no advantage for use of prediction on tests comparing the use of animations with and without predictions [Jarc98]. Byrne, et.al. [BCS99] conducted a study that combined prediction and no prediction with animation and non-animation conditions. For a simple algorithm, both animation groups scored significantly higher on a post-test, but with a more complex binomial heap algorithm, a mix of significant and non-significant advantages was found for animation and prediction groups.

Kehoe, Stasko and (Hamilton-)Taylor [KST01] found that for the same binomial heap animation used by Bryne, using a textbook and AA as a resource in a problem-solving setting resulted in a significant advantage over the use of the textbook and a series of equivalent static figures. Hübscher-Younger found that a group that constructed an animation significantly outperformed (on a post-test) groups who used other media [HN03]. We discuss studies of AA and their learning contexts in further detail in Chapter 2.

1.2   Complexity of Authoring

Algorithm animation authors find that designing and implementing an AA can be lengthy and complex [Hund99][Naps03], despite attempts to address this in AA systems. Naps reports that two-thirds of instructors found that the length of time needed to create an AA was a limiting factor in their use of AA [Naps03]. Hundhausen found that students spent an average of 33.2

hours completing a Samba assignment [Hund99]. Creating an AA may require learning a complex system, with extensive APIs and new programming concepts [SK93]. Alternatively, it may be difficult or tedious to create a non-trivial AA using a simple system (such as a GUI-based system), which may not provide all of the capabilities needed [RF01][Stasko96]. An algorithm may have to be extensively modified to be animated. A significant amount of planning often has to be done to design an algorithm animation. This includes the conceptual design of the AA and the design of the display, the animation actions, mapping the actions to the algorithm operations, and the coordination and timing of the actions. These issues are discussed further in Chapter 3.

1.3   Our Analysis of AA Problems

We believe that a number of issues relevant to AA have not been sufficiently investigated, and that these issues potentially impact AA adoption and effectiveness at several levels. One objective of this dissertation is to identify and examine some of these issues to shed light on the perplexities of AA. We start with the assumptions that in order to be effective, AAs:

1. should support the needs of the users (the students/learners and the authors).
2. should be used in an appropriate learning context.
3. should be presented so that the student/learner can perceive and comprehend them.

This suggests that we identify and examine relevant issues in the following areas that relate to these needs: (a) human-computer interaction and user interface design, (b) perceptual, attentional, cognitive and engineering psychology, and (c) the learning sciences and educational

technology. We will address each of the above points, and preview relevant issues in the course of doing so.

1.4 HCI Design and Learning Context

Does the AA system support essential user needs and tasks? For example, the user may not be able to explore the algorithm effectively, or to control its execution. There may exist tasks that an instructor or student would do that the system does not support, such as annotation or modification of data structures.

Have AA system designers attempted to understand the existing learning processes employed in the teaching of algorithms and data structures? Do the animations affect the learning process in unexpected and unintended ways? For example, can an instructor show the same concepts and examples in the manner he or she is accustomed to with a black/whiteboard by using an animation? Are the animations being used in a manner conducive to the support of the learning process?

Is the learning context appropriate? What may be suitable for individual or lab use may not work in a lecture or discussion or vice-versa. For example, a particular AA might be suitable for use as a learning resource rather than as the sole means to teach an algorithm.

1.5 Perceptual Design

While AA aims to take advantage of human perceptual ability, we believe that some of the techniques used in AA may exceed our perceptual and attentional capabilities. Users may not be able to perceive the actions in the animations properly. The changes or pattern of changes

may be too fast or too complex to follow. The AA may not be portraying the algorithm effectively. The portrayal may be too complex for the user to absorb, obscuring the essential concepts, or it may not show these essential concepts.

Baecker, considered the founder of program visualization and AA, thinks that "timing is the key to effective motion dynamics and algorithm animation," but notes that it is "hard to find the right speed for a diverse audience" [Baecker98]. Marc Brown, an AA pioneer, begins a book chapter [BH98] by stating that "designing an enlightening software visualization is a tricky psychological and perceptual challenge". Stasko, a leading researcher in the field, considers "smooth, continuous motion" essential, and postulated that it would ensure that viewers could follow the animation actions, and thus improve AA comprehension [Stasko98a]. However, no studies were conducted to investigate such ideas prior to our research, which we discuss further in Chapter 6.

1.6   Goals and Framework of Dissertation Research

In this dissertation we will examine the relation of AA to a number of research areas that we believe have some bearing on AA problems. We will analyze a number of AA issues utilizing research perspectives of these areas, and drawing on research findings from these areas. We will then focus on particular issues that we believe to be of prime importance, and describe our research on these issues. We envision our research as proceeding in a number of phases:

1. The study of the problem
2. The design and implementation of appropriate tools
3. The evaluation of these tools in appropriate contexts

We feel that researchers have too often developed tools without sufficient or thorough study of the problem and related issues. The aspects of this study phase described in the dissertation are (a) a review of research in relevant fields, (b) a study of how experts discuss algorithms and data structures in the classroom, and (c) studies of how viewers perceive algorithm animations.

To this end, we conducted an observational study of computer science instructors [Hamilton-Taylor02b], in order to examine the context of learning in the classroom, and understand the tasks and needs of instructors. While a number of studies of students have been conducted [HDS02], including an ethnographic study of student AA construction [Hund99], similar naturalistic studies of computer science instructors for AA do not otherwise exist. The design and results of this study are described in Chapter 4. A description of SKA [Hamilton-Taylor02a] (Support Kit for Animation), a tool designed to support these tasks, is described in Chapter 5.

Our determination to investigate the role of perception in AAs led us to formulate a plan that specified (a) a number of potential perceptual problems and issues that we wanted to investigate, and (b) a strategy for investigating them from the lowest level of individual AA actions up to groups of AA actions. We wanted answers to a number of questions, including the following:

Are users perceiving the actions or changes in algorithm animations? How often do they miss these actions, and why? Can the user follow the animations perceptually? What perceptual techniques help? Can they relate the animation portrayal to the algorithm it attempts to represent? Do the animations convey the algorithm concepts and operations that the designer intended?

We initiated what became a large-scale project. An expert in visual and engineering psychology, Prof. Davis of Georgia Tech, became our collaborator. A joint project proposal between UGA and Georgia Tech, to which we contributed, was funded by the NSF [KD02]. This collaborative project proved quite fruitful. We designed an architecture for software to conduct perceptual experiments, VizEval. Other graduate students at UGA were selected to collaborate with us on the development of VizEval. Graduate students at Georgia Tech worked on the perceptual experiments. We collaborated on the design of a series of studies that examine the role of perception [DHKHR06] [RRKHDH06] [Rhodes06] [Hailston06] [Reed06] in AAs, and the design of software to conduct the studies [KRRH06] [RKHD06] [Reed06] [Thomas04] [Ross04]. We are not aware of any prior studies that investigate the perceptual aspect of AAs. These studies and the testing environments are described in Chapter 6.

We have described the main aspects of the study phase. The design phase includes the design of SKA and the VizEval perceptual testing environment. The evaluation phase includes current use of perceptual testing environments in studies of AAs and future evaluation of SKA.

1.7 Instructor Study

We examined one aspect of AA learning context and user needs by investigating the activities of instructors using AAs in a classroom. This is a common setting, but has not been previously studied, to the best of our knowledge. We conducted a naturalistic observational study of how instructors discuss algorithms in a lecture setting, with a focus on the use of diagrams and algorithm tracing. We videotaped the lectures, and employed a custom methodology/protocol based on activity analysis [Nardi96] and contextual inquiry [BH98] to describe activities, tasks and use of artifacts (entities). We then identified activities and artifact tasks that could be

supported by software tools. We found that the use of diagrams was prevalent, and was an integral part of algorithm execution tracing, derivations and discussion of theoretical algorithm issues. However, the tracing and diagramming tasks were often tedious and error-prone, and a prime candidate for task support.

1.8 SKA

We designed SKA, the Support Kit for Animation [Hamilton-Taylor02a], based on the analysis of user needs and tasks in our observational study of instructors. SKA is designed to support and enhance tasks such as diagram manipulation and tracing. It is intended to be a shared medium of discussion for instructors and students. SKA is a combination of an interactive data structure diagram manipulation environment, an algorithm animation system, and a visual data structure library. SKA supports interactive creation and manipulation of data structures via visual data structures, independently of algorithms, in a whiteboard-like environment. The visual data structures are active diagrams; operations on them manipulate the underlying data structure. SKA allows users to execute algorithms that appear as pseudocode on the data structure diagrams. Execution control is flexible and designed to emulate and support the manner in which instructors perform tracing tasks. SKA has a simple authoring system that portrays Java programs as pseudocode algorithms. New visual data structure diagram classes can be added by library developers. SKA also has a scripting interface that allows its animation and interaction capabilities to be used by other programs; this is used by our perceptual testing software, which is described in the following section.
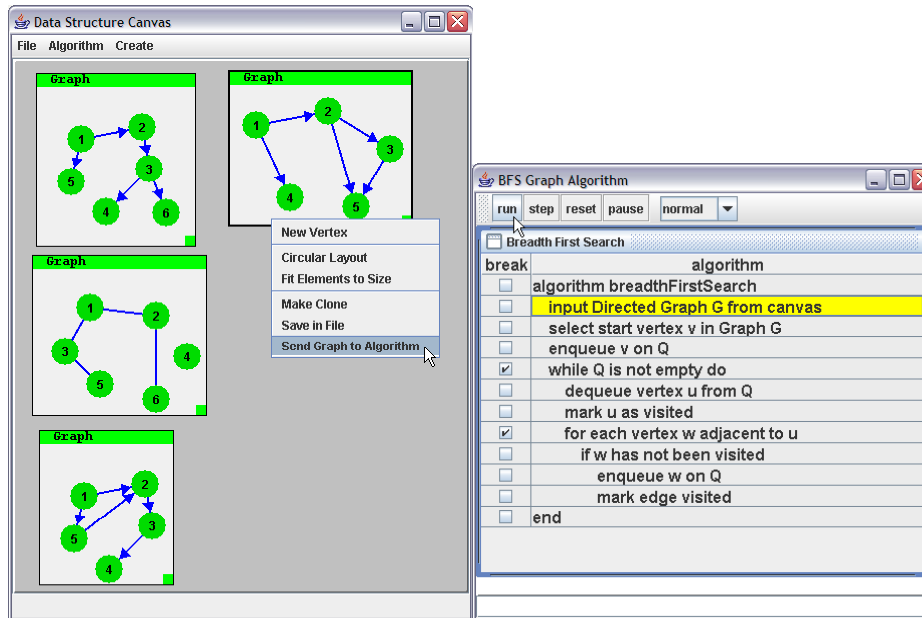
Figure 1.1:   SKA in action. The algorithm on the right is waiting for the user to select a directed graph from the canvas on the left.

1.9   Perceptual Studies

We proposed that AAs should be evaluated at the perceptual, mapping and comprehension levels, and that we should study the role of perception and perceptual design effects at each level. We then collaborated in the design of empirical studies to investigate the role of perception in algorithm animations. We designed software to be used to conduct these studies. We also examined research in perception, attention, memory, cognition, and cognitive engineering design and have analyzed AAs in light of relevant findings.

We contributed to the design of an experiment that evaluated the effect of cueing techniques and labeling objects on the detection of changes and locating the changed objects in an AA display [DHKHR06] [Hailston06]. Among other things, we found that cues and labels can help users to locate changes, especially for simultaneous changes, but did not affect their detection significantly. Also, larger numbers of objects on the display reduced the ability of observers to locate simultaneous changes significantly.

Another study we contributed to utilized a popup question methodology we developed to evaluate whether the viewer was able to perceive and relate (map) changes in an AA display to changes in values and to variables in the algorithm [Reed06] [RRKHDH06]. The cueing and animation techniques had a significant positive effect on perception and mapping questions. The study also evaluated the effect of different cueing and animation techniques on the comprehension of a quicksort algorithm animation, as measured by performance on a set of problems. These techniques did not help comprehension in the context of use, which allowed users to explore the AA while answering questions, and the availability of other redundant information sources, such as narrative descriptions.

A third, preliminary study [Rhodes06] evaluated the effect of the popup question methodology on AA comprehension, among other things. The popup methodology did not affect comprehension question scores if feedback to popup questions was provided. We found that performance on popup questions that mapped changes to algorithm variables was lower than that on questions involving mapping of changes to values. A change in color design also improved performance to some extent, compared to the second study.

In order to conduct perceptual studies of AA, we developed software to create and run the tests, to meet the specific needs of evaluating algorithm animations. We designed the VizEval software architecture [RKHD06] [Ross04] [Thomas04] to conduct perceptual studies of AA views, that facilitates:

- Specification of various types of trials, questions, interaction, and data collection
- Control of time-based animation of AAs, which is necessary to support trials involving perceptual tasks

The VizEval suite is comprised of software to create displays for trials, software to create trials and tests, and software to conduct tests and collect data. The scripting interface to SKA was used for VizEval, and also for a second system, SSEA (System for Study of the Effectiveness of Animations) [Reed06], which conducts comprehension-level AA tests and incorporates our 'homework' scenario testing methodology [KST01] and our popup methodology.

1.10 Contributions

This dissertation research explored a number of issues that relate to the three assumptions we stated earlier; namely that to be effective, AAs must support the needs and tasks of the users, must be used in appropriate learning contexts, and must be designed to support users' perceptual capabilities and limitations. This research has culminated in a range of contributions:

- A naturalistic study of instructors to examine their activities and use of artifacts whilst conducting algorithms and data structure lectures, and analysis of the results, used to suggest areas for potential technological task support.

- An AA system, SKA, informed by the instructor study findings, that supports interaction with active conceptual data structure diagrams, simple authoring and flexible execution of pseudocode algorithms, visual data structure library development, and an animation engine designed for perceptual pacing. SKA is designed to support instructors in the algorithm classroom context, discussion with and exploration by students.

- Collaborative design of a series of experiments that investigate the role of perception (and associated factors) in the use of AAs at various levels of perception and comprehension.

- Collaborative design and implementation of software to conduct perceptual AA experiments, that supports creation, running and data collection for a variety of trial types.

- Work on the analysis of AAs from various research perspectives and work on experimental methodologies to evaluate them in various contexts.

1.11 Structure of Dissertation

The structure of the remainder of the dissertation is as follows. Chapter 2 analyses algorithm animation concepts, the objectives of algorithm animation, and efforts to evaluate algorithm animation use, all from an HCI and educational technology perspective. Chapter 3 gives an overview of the historical development of algorithm animation systems and authoring paradigms. Our observational study of computer science instructors is described in chapter 4. The design and use of SKA is described in chapter 5. The perceptual studies, testing software, and related work are described in chapter 6. Finally, we provide a concluding discussion and a vision of future research in chapter 7. The relationship of the research components of this dissertation is shown in Figure 1.2.
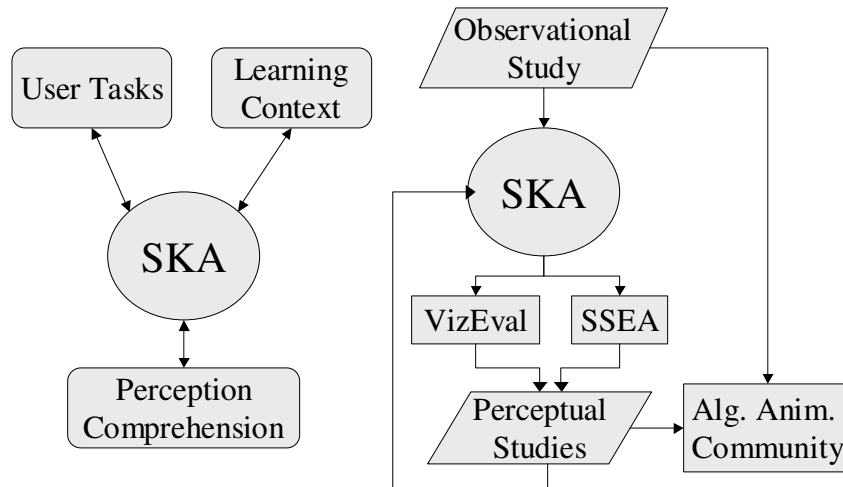
Figure 1.2:   Structure and relationship of the research components of the dissertation. On the left, SKA is informed by our three areas of concern, and is also designed to support them. On the right, the observational study investigates user tasks and the instructional learning context and informs the design of SKA, as well as the algorithm animation community. SKA provides capabilities that support perceptual design. The perceptual studies are run using VizEval and SSEA software, which use SKA. The perceptual studies will inform future versions of SKA and the algorithm animation community.

CHAPTER 2

ANALYZING ALGORITHM ANIMATIONS

2.0   Introduction

What is an Algorithm Animation (AA)? What do we think they can do, and why? How

have they been used in the learning context? What do studies show about AA use?

What is the fundamental rationale for using AAs and what are our objectives in so doing?

What is the nature of the artifacts and activities involved in the discussion and design of data

structures and algorithms?

Answers to questions such as these may provide useful insight into the design, use and

evaluation of algorithm animations.

Standard definitions for AA state that AA is a dynamic visualization of an algorithm

[PBS93]. However, we seek to understand the underlying nature of AA as an artifact and a tool.

In this chapter, we discuss these issues from a wider perspective, employing analytical

frameworks used in educational technology [Orey05] [BH05] and HCI [Nardi96]. Within this

context, we will also describe studies of AA use.

2.1   Educational Technology Processes

The Association for Educational Communications and Technology (AECT) definition of

Educational Technology is "the study and ethical practice of facilitating learning and improving

performance by creating, using, and managing appropriate technological *processes* and

resources" [BH05]. An educational technology *process* is an organized set of actions that aims to achieve some learning goal. Knowledge about an educational technology process or *product* can be categorized into concepts, theories, and practices [BH05]. An educational technology *product* is an idea, a framework of ideas used to develop technologies, or a concrete technology, used in a process. Branch notes that the AECT "assumes education is a process, technology can facilitate educational processes, educational technology projects are complex, and the sophistication of a process should match the complexity of the related situation" [BH05].

A *concept* is an idea or notion about an entity, process, phenomenon, or technology, for example, the idea of an algorithm animation. A *theory* is a formal proposal about an entity, process, phenomena, or technology. For example, researchers have theorized that AAs help learners by portraying the dynamic execution of algorithms. *Practice* describes the actual use of the entity, process, phenomena, or technology. For example, we can describe how AAs are used as learning tools in lab settings. The process classification of concept, theory, and practice represents a useful analytic framework to categorize the learning context of algorithm animations.

2.2   Algorithm and Data Structure Representation

We want to understand the concept of an AA, the theories ascribed to AAs, and the practice of using algorithm animations. Before we seek to examine AAs, we should try to gain an understanding of the entities that AAs attempt to visualize: algorithms and data structures. We should ask what these entities are, how they are represented, and how they are used in practice. We know that algorithms and data structures and their representations are used in design, debugging and instruction [AU85]. It is well known that there are common representational forms that are used in these contexts [CLR90][Dale03], as we shall discuss.

We will begin by describing algorithm and data structure artifacts and activities. This descriptive approach is derived from ethnographic description methods [FW96] and activity theory analysis methods [Nardi96]. In activity theory, artifacts are entities or external representations used in activities. Tools mediate or facilitate the performance of activities on artifacts.

```
Insert new value in list
   If we reach the end of the list
      OR the new value is less than the value in the node
      Create a new node with the new value
      Insert the new node in the list:
            Make the next link of the new node point to the list node
            Make the list pointer point to the new node
   Else
      Insert new value in the sublist starting at the next node
```

Figure 2.1:   Pseudocode algorithm to insert a new value in sorted place in a linked list.

```
template<class ItemType>
void Insert(NodeType<ItemType>*& listPtr, ItemType item)
{
  if (listPtr == NULL || item < listPtr->info)
  {
    // Save current pointer.
    NodeType<ItemType>* tempPtr = listPtr;
    // Get a new node.
    listPtr = new NodeType<ItemType>;
    listPtr->info = item;
    listPtr->next = tempPtr;
  }
  else Insert(listPtr->next, item);
}
```
Figure 2.2:   C++ program that implements the algorithm to insert a new value in sorted place in a linked list.

An algorithm is a set of steps that carry out a programmed task; this is the concept. Algorithms are described in a variety of notations ranging from arbitrary structured English (*pseudocode*) to algebraic expressions to flow diagrams and combinations thereof. An example of a pseudocode algorithm is shown in Figure 2.1. Algorithms are implemented as computer programs in various programming languages, as shown in Figure 2.2. An algorithm uses and

modifies scalar variables or elements, and data structures or objects. Data structures organize elements into groups, in a manner suitable for particular methods of storage, retrieval, deletion, and manipulation [AU85]. Common examples of data structures include arrays, lists, trees, and graphs.

We will now define a few terms to describe algorithms and data structures, as used in the context of this discussion. *Notation* refers to the language used to specify algorithms or computer programs and define how data is stored. *Representation* refers to the means of showing the state of a running algorithm or program. Representation can be static or dynamic, and often involves showing or annotating the corresponding notation. The *implementation level* refers to the lower level at which notation is implemented on a computer, i.e. the level of a computer program. The *conceptual level* refers to the higher level at which algorithms are designed.
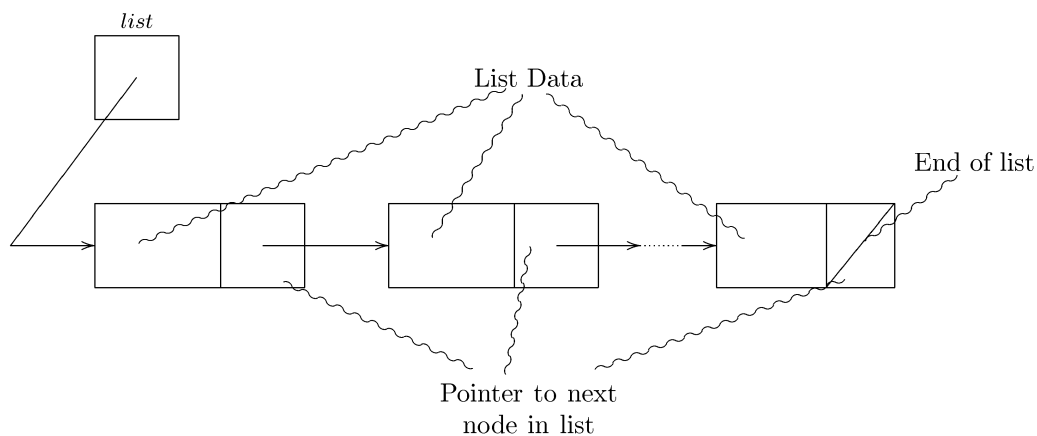
Figure 2.3: A Linked List diagram, using the de facto informal representation.

At the conceptual level, data structures are often described using a diagram representation, and algorithms perform tasks on these diagrams. In other words, the notation and representation are the same when using these diagrams. An established, though informal, set of diagram representations exists for data structures such as lists, trees, graphs, and arrays [CLR90]

[Dale03]. Examples of the de facto standard diagram representation are depicted in Figures 2.3 and 2.5. These diagrams are used in both high-level and lower-level discussion and design.

Data structures and data can also be represented by charts, such as a series of bars whose heights indicate the magnitude of array elements. These tend to be used to display values in a form suitable for easy visual comparison of magnitudes. Metaphoric representations can also be used, such as playing cards to represent data to be sorted, and towers of Hanoi diagrams to understand recursion; examples can be seen in textbooks such as [CLR90][Dale03]. Some experts use customized visual representations for specialized design, at a level not intended for novices [Petre02]. However, one finds that textbooks and instructors almost inevitably return to the de facto algorithm and data structure representations to discuss design, as these alternative representations tend to be used to illustrate very high-level concepts and behavior. Also, the alternative representations are not universally employed. We therefore believe it is important to examine the informal de facto algorithm and data structure representations and their relation to AA.

2.2.1 Notation and Representation Difficulties

Algorithm notation consists of static text (or static diagrams in visual languages), but the tasks an algorithm performs are dynamic. At the implementation level, data structures are specified (declared) using static textual notation, as shown in Figure 2.4. However, this implementation notation does not portray the changes in size, dynamic structure, and content that data structures undergo when modified by algorithms, as shown in Figures 2.3 and 2.5.

```
template < class DT >
class List
{
  public:
    // Constructor
    List ( int ignored = 0 );
    // Destructor
    ~List ();
    // List manipulation operations
    void insert ( const DT &newData )        // Insert after cursor
        throw ( bad_alloc );
  private:
    // Data members
    ListNode<DT> *head,      // Pointer to the beginning of the list
                 *cursor;    // Cursor pointer
};

template < class DT >
class ListNode
{
  private:
    // Constructor
    ListNode ( const DT &nodeData, ListNode *nextPtr );

    // Data members
    DT dataItem;       // List data item
    ListNode *next;    // Pointer to the next list node

  friend class List<DT>;
};
```

Figure 2.4: Part of a declaration of a List and a List Node in C++. The diagrams in Figures 2.3 and 2.5 are drawn manually by tracing a program that uses such a declaration, or by tracing an algorithm.

2.2.2 Conceptual Simulation versus Program Execution

Algorithms and data structures are generally designed and discussed at the conceptual level in research and learning environments. To show the effect of an algorithm at the conceptual level, one traces (i.e. simulates manually) its steps and their effects on data structure diagrams, as shown in Figure 2.5 [Dale03]. As we show in our observational study of instructors [Hamilton-Taylor02b], described in Chapter 4, tracing is a time-consuming, difficult, and error-prone process. Conceptual-level algorithms cannot be executed on a computer (unless an AA is used).

At the implementation level, programs show their effect on data structures by updating the data structure state, i.e. its variables. However, programs do not produce data structure diagrams from their internal representation (unless a program visualization system is used).
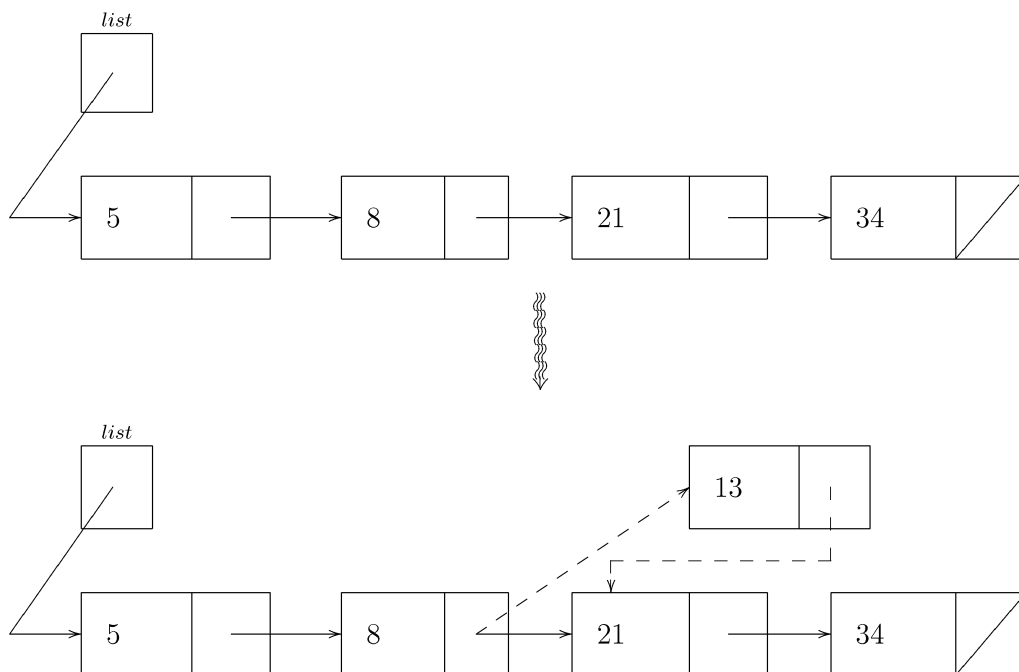
Figure 2.5.   A sorted linked list diagram, before and after a new value, 67, is inserted. This is drawn manually by tracing the insertion algorithm in Figure 2.1 or program in Figure 2.2 (from Dale03]).

2.2.3 Gulf Between Notation and Dynamic Representation

Because the behavior of an algorithm is not apparent from its notation, a fundamental gulf exists between the notation for algorithms and the tasks they perform, and between the notation for data structures and their representation. This is akin to what Norman calls the gulf of execution, which describes the gulf between a user's intention to do a task, and the user's ability to figure out how to instruct a system to perform that task [HHN85]. Cognitive dimension analysis [Green00] would classify these notations as lacking in role-expressiveness: the purpose

of the design of an algorithm is often not readily inferred from its notation. These are problems that AA and program visualization systems attempt to address.

2.2.4 Problems in Learning Situations

Novices face a number of difficulties in the course of learning programming: misconceptions about the model of execution, about programming concepts, and about notation [Pane02a]. The misconceptions in the mental models of novices, and the difficulties understanding the computational model (the "notational machine") have been identified in studies of novices and children learning introductory programming [DuBoulay89]. Mayer found that providing concrete models of abstract notational machines improved learning [Mayer89]. A primary goal of Boxer [diSessa86] and HANDY [Pane02a] was to provide an interactive environment for children and novices that reflected a concrete visual model of computation and data. These studies (and tools) involved only introductory programming. However, these difficulties extend to algorithms and data structures learning [HDS02], which may be classified as intermediate programming. Apart from anecdotal evidence, this is borne out by student performance in a number of AA studies [HDS02], including in conditions not involving animation use. We identified a number of student misconceptions about intermediate programming in Smalltalk in a study of sophomore-level computer science students at Georgia Tech [TG97].

In data structures and algorithms courses, learning situations often involve discussion, design, and exploration of conceptual algorithms and data structures. The instructors, or experts, often use high-level conceptual algorithms as their notation and data structure diagrams as their representation in high-level discussion and design [Hamilton-Taylor02a]. However, we believe

that the novice learner often lacks a sufficient grasp of the expert's mental model in order to use it accurately on the representation. Even when data structure diagrams are relatively simple, the actions performed on them by the algorithm are often quite complex. Thus, the novice often has difficulty recreating the behavior of the algorithm or exploring its effects on the data structure diagrams.

These challenges can provide motivation for the design of tools that can bridge this gulf and thereby assist novices, or learners. Though researchers have not analyzed the problem from this perspective, as far as we are aware, this was the motivation that led to the development of algorithm animation.

2.3   Algorithm Animation Concepts

Price [PBS98] distinguishes between software visualization (SV), program visualization (PV), and algorithm animation (AA). Software visualization involves the use of any form of visualization technique for software, either static or dynamic. Program visualization is the graphical (or multimedia) portrayal of the state of an executing program. Algorithm animation is the visualization of high-level and lower-level algorithms, usually for educational purposes, as opposed to the typically lower-level code shown by PV. Algorithm animation is also called algorithm visualization (AV). In practice, the terms AA and PV are often used interchangeably, as it can be hard to draw exact boundaries between high-level algorithms, lower-level algorithms and low-level code.

Algorithm animations, by definition, are intended to show the dynamic operation of an algorithm. In so doing, we believe they have the potential to bridge the gap between the static

notation and the dynamic behavior of algorithms, and between the static notation/representation and dynamic representation of data structures.

We differentiate between what we think an AA can show (the aim), and how it attempts to show it (the design). AA designs can be based on the informal standard algorithm and data structure representations, on charts, or use metaphoric representations, as discussed in the previous section. Adding a dynamic aspect provides additional display dimensions. Algorithm animation display designs are classified in [Brown98] and AA display techniques are surveyed in [Jeffrey98] [BH98].

An AA can show a dynamic representation of how the algorithm computes its results. This would not be visible from the algorithm. The AA can display dynamic representations of the strategies used by the algorithm in the course of execution (carrying out computations). Learning such strategies is an important objective for algorithms courses. Algorithm animations can also portray the performance or efficiency of an algorithm, which is often not obvious from the notation. This is another important learning objective for algorithms courses. Algorithms usually have sub-algorithms that carry out subprocesses. Algorithm animations can show the dynamic relationships between an algorithm and sub-algorithms, the data that is passed between them, and so on. This is important because this is often difficult to track and tedious to trace manually.

2.4 Algorithm Animation Theory

The fundamental theory of AAs is that they help learners by portraying the dynamic execution of algorithms [Brown87], thus bridging the gap between static algorithm notation and

their dynamic behavior. Hundhausen presented an analysis of AA (which he calls algorithm visualization or AV) using six learning theories, as follows (from [HDS02]) in Table 2.1.

Table 2.1:   Theories of algorithm animation/visualization (AA/AV) and software visualization (SV), as posited by six learning theories (from [HDS02]).

THEORY	SYNOPSIS	ROLE OF AV
EpistemicFidelity	Graphics have an excellent ability to encode an expert's mental model, leading to the robust, efficient transfer of that mental model to viewers	Encoding of knowledge
Dual Coding	Representations with dually-coded information (e.g., graphics and text) promote the most robust and efficient knowledge transfer	Encoding of knowledge
Individual Differences	Differences in human cognitive abilities and learning styles enable some to benefit from SV more than others	Unspecified, but "encoding of knowledge" fits in well with the theory
Cognitive Constructivism	Active engagement with SV enables one to construct one's own understanding	Resource for knowledge construction
Situated Action	AV is a communicative resource for building a shared understanding of algorithms	Communicative resource akin to speech, gesture, and gaze
Sociocultural Constructivism	AV enable people to participate in a community in increasingly central ways	Community artifacts that grant access to central community activities

Other AA researchers have put forward less formal theories, some of which relate to Hundhausen's analysis. Baecker proposed that timing and pacing are key in designing effective AAs, along with clear, uncluttered display layouts [Baecker98]. His experience with the AA film Sorting Out Sorting (SOS) provides anecdotal support for this theory. Stasko theorized that smooth, continuous animation makes movement easier to track than sudden movement, which allows the viewer to follow animation actions and thus facilitates AA comprehension [Stasko98]. These theories have not been confirmed, but we believe that AAs often have problems that are perceptual and attentional in nature that may contribute to the mixed results of studies of AA

effectiveness. We initiated and have collaborated on empirical studies to investigate these issues [DHKHR06] [RRKHDH06] [Rhodes06] [Hailston06], described in Chapter 6, and we are in the process of conducting further empirical studies.

Hundhausen believes that AA construction is more effective then AA viewing because of the nature of the design process and the higher level of engagement involved [Hund99]. Hübscher-Younger found that a group that constructed an animation outperformed groups that used other media on a post-test [HN03], supporting this position.

We agree with the cognitive constructionist view that using AA as a resource facilitates knowledge construction. We believe that AA use in a problem-solving context facilitates exploration of the algorithm, and promotes a higher level of engagement. Furthermore, we believe that active use of an AA to seek an answer to some question in the mind directs and sharpens cognitive focus and perception, particularly to aspects of the AA that relate to the question. This transforms the AA into an investigative tool, a virtual lab, an exploration tool. Prior studies we conducted, using AA as a problem-solving resource, provide some support for this notion [KST01] [TRHK02].

We also believe that AA environments can be used as a shared medium of discussion between expert and novices, or between novices. In particular, we propose that they can bridge the gap between the language of hand-drawn data structure diagrams and static notation that specifies the data structures, as well as between algorithms and their dynamic behavior. This is part of our design rationale for our interactive AA system, SKA, the Support Kit for Animation [Hamilton-Taylor02a], described in Chapter 5.

2.5   Algorithm Animation Practice

AAs have been used in a wide variety of learning situations. They include lectures, laboratories, tutorials, class discussions, office hour use, tests, assignments, and self-study. A survey of these is discussed in [HDS02] [Naps03]. We discuss some of the more common uses of AA and studies of their efficacy or use in those contexts.

The earliest use of AA was as a lecture demonstration tool. Intended for novices, Sorting Out Sorting (SOS) is a thirty-minute narrated film, similar in flavor to those produced for high school physics classes [Baecker98]. It presents a series of sorting algorithms at the conceptual and operational level. It goes on to present the efficiency of each algorithm, then explores their comparative efficiency using simultaneous animation. There is anecdotal evidence of the effective use of SOS.

The implicit learning model of many algorithm animations is that of passive transmission [Kosch96]. The objective of many existing AA systems is to provide a good graphical library. These systems generally support a 'video'/ batch processing paradigm – provide the input data and watch the video, with controls to slow it down, speed it up, pause it or replay it. Yet there is a school of thought in the education community that an active model of learning [Kosch96], involving exploration, problem solving, or construction, may be more appropriate and effective. Hundhausen suggests that active learning with AAs can be accomplished using a variety of activities, such as (from [HDS02])

- constructing their own input data sets;

- making predictions regarding future visualization states;

- programming the target algorithm;

- answering strategic questions about the visualization; and

- constructing their own visualizations.

The results of some studies of AA support the theory that interaction and active learning can improve learning [Lawrence93] [KST00] [HN03], though other studies have mixed or inconclusive results [BCS99][Jarc98]. In Hundhausen's meta-survey, 71% of the studies he categorized as involving active learning or cognitive constructionism [HD0S2] had significant positive results, versus 50% of the studies in the individual differences category, 50% in the dual-coding category, and 30% in the epistemic fidelity category [HDS02]. Algorithm animation system designers should therefore consider supporting interaction, active exploration, and other modes that facilitate active learning. Interaction control can affect outcomes. [Saraiya04] found that user-friendly execution controls provided a significant advantage (on a comprehension post-test) for a group using them, compared to a group whose system did not have these controls.

Generally, students are passive viewers when AA is used as a tool in lectures; however, AAs can be used to facilitate interaction between the instructor and the class. The electronic classroom at Brown University allowed instructors and students to pose and respond to questions, run animations and interact with them [Bazik98]. An examination of the tasks the instructors perform while discussing algorithms and data structures in a lecture in the absence of an algorithm animation system allowed us to identify some tasks that could be supported by a system [Hamilton-Taylor02b], as described in Chapter 4. We found that diagram use is an integral part of the discussion process, and used the results to inform the design of SKA [Hamilton-Taylor02a].

Algorithm animations are used as learning tools in laboratory settings; students may be asked to perform various types of exercises such as comparing the effect of input data sets on the

behavior and performance of the algorithm, or may be encouraged to explore on their own. A comprehensive study [LBS94] compared groups that observed an animation during a lecture to groups that viewed slides portraying the same algorithm, using the same data set. Each of these groups was divided into two sub-groups, one of which did a lab using an animation of the algorithm they were studying. The lab groups were subdivided into a 'passive' group, which ran the animation, and an 'active' group, which had to create their own data sets for the animation. They found that the group that constructed data sets while doing the lab exercise had significantly higher scores on a post-test than groups that just attended the lecture.

Algorithm animations are used as individual learning tools to observe or explore algorithms. Many studies of AA effectiveness, which are primarily of individual use, have produced mixed results [HDS02]. In some cases, this is so even when they included interaction.

Byrne, et.al. [BCS99] conducted a study that combined prediction and no prediction with animation and non-animation conditions. Students were asked to make predictions about the result of the next step of the algorithm. All groups were shown a short videotaped lecture. The study had two parts. In part A, a basic depth-first search algorithm animation was used and a post-test consisting of procedural questions was given to the students, who were non-CS majors. Both animation groups (prediction/no prediction) and the no animation/prediction group showed a significant advantage over the no prediction/no animation group. In part B, the students were advanced CS undergraduates, a more complex binomial heap animation was used, and the post-test had procedural and conceptual questions. They found a mix of significant and non-significant advantages for animation and prediction groups; no differences were found on conceptual questions. On the other hand, Jarc found no advantage for prediction on tests

comparing the use of animations with and without predictions [Jarc98]. A study by Rhodes and her colleagues [Rhodes06] had similar results to that of [Jarc98], described in Chapter 6.

In a study conducted at Georgia Tech in 1997, we had individual students use AAs as a resource while solving textbook exercises [KST01]. This has been called the "assignment" [HDS02] or "homework" scenario. The students were learning about a complex new data structure (a binomial heap), and a set of algorithms that use it. Students were also given the relevant section of the textbook, which included figures. We recorded detailed observations of students using these resources in this problem-solving process, and found that animations were used as heavily as a resource, in conjunction with other materials. Students in the animation group performed significantly better than those in another group who were given the same text and static diagrams from the algorithm animations. In post-session discussions, students pointed out specific instances where the animations helped to clarify operations. The same binomial heap animation was used in [BCS99], which makes for an interesting comparison.

In another study we collaborated on at UGA in 2001 [TRHK02], using this same "assignment" approach resulted in significant advantage in scores on a problem set for an animation group, as compared to a group that used cognitively equivalent non-animated material. The animation group also fared better compared to earlier groups that used a standard post-test approach, though there were a few differences in the experimental environments.

Another approach to using AAs in computer science education is to ask students to construct their own AAs, with the intention that in so doing, they will learn about the algorithm. Constructionist activity has been used in many subject areas [Pappert87]. Stasko found that students enjoyed using his Samba tool to do so, but it was quite time-consuming [Stasko96]. Hundhausen found that students took much less time to create AAs with ALVIS, a higher-level

tool, than with Samba [Hund99]. The design of ALVIS is based on his study of students using tangible material (paper cut-outs) to create AAs. Hübscher-Younger found that a group that constructed an animation outperformed those who used other media such as paper drawings, stories or anything else [HN03]. The comparison was done using pre- and post-tests of algorithm comprehension.

We participated in a team effort to conduct an ethnographic study of student perspectives on the learning of algorithm and data structures at Georgia Tech in 1998. This included investigating student views on algorithm animations developed by instructors, by themselves, by other students, and on the tools used by the students to develop animations (in this case, Samba). Our findings from this study (unpublished so far) revealed that students face a number of challenges in the construction process. Constructing an animation can help to understand an algorithm, but some students find they have to understand the algorithm first while others develop an understanding of the algorithm while developing the animation. Yet others report developing an animation (even with bells and whistles) without fully understanding the algorithm, and even receiving high grades. For example, a number of students reported that they fully understood the Shellsort algorithm they animated (which was not discussed in class) only after viewing a 'solution' animation during class. We also found that students spent a substantial amount of time in the pre- and early design phase on low-level details, and that the time required for constructing AAs using Samba could not be supported in a typical algorithms course for more than a few algorithms (and many algorithms need to be covered). The use of higher-level AA tools can help to reduce construction time, but may not necessarily address these other concerns.

2.6   Conclusion and Future Work

Much of the research in AA has concentrated on developing systems, proposing practices and analyzing the end results of studies of these practices, rather than examining the fundamental essential nature of the phenomena AA attempts to visualize, and considering the various aspects of AAs in the larger educational technology context. We have presented an analysis of the entities that AA is trying to represent and the problems their notation and representation imply for learning and AA system design. Rather than simply reviewing studies and definitions of AAs, we have discussed them in the framework of educational technology processes. Clearly, there are a number of AA theories that merit further thought and elaboration, including some of our own. Studies of the practice of AA use and construction reveal a complex picture of results, which points to the need for further examination of the relationship between AA theory, the aims of AA practice, the design of AAs, and the actual practice or use of algorithm animations.

We believe that our analysis represents a new approach to the study of the artifacts, activities, and related learning issues surrounding the subject of algorithms and data structures. We have employed this analysis to inform our framework for (a) the study of these artifacts and activities, (b) the design of tools for learning environment support, and (c) evaluation of factors relevant to their effectiveness. We are not aware of any research that describes and analyzes these issues as we have.

CHAPTER 3

THE HISTORICAL DEVELOPMENT OF ALGORITHM ANIMATION

3.0   Introduction

In the late 1960's, a graduate student at MIT named Ron Baecker had a vision. He would use the new graphics display terminals to portray data structures diagrams, and show dynamic changes to them as programs executed [Baecker68]. He anticipated that this could be useful for debugging, since programmers often drew these diagrams by hand and painstakingly traced changes on them in the process of debugging.

Thus was born software visualization (SV). In the late 1970's, Baecker pursued further work on the use of SV for educational purposes [Baecker75], resulting in a landmark work, Sorting Out Sorting [Baecker81] [Baecker98]. This film of animated sorting algorithms generated much excitement when it was shown at SIGGRAPH [Baecker81]. It was followed by algorithm animation (AA) systems such as BALSA [Brown87] and TANGO [Stasko98a].

Initially, it was assumed that AAs would be beneficial, as this seemed intuitive. Eventually, studies were conducted to investigate the benefits of algorithm animation. To the surprise of researchers, the benefits were not clear-cut. Further, we note that AAs seem not to be used as much in teaching as was expected [Naps03].

Why haven't AA systems had the type of impact that was widely anticipated? Why aren't they more widely used? Practitioners in the field continue to express concerns about these issues [Naps03]. One reason may be that AA systems designers, particularly in the early years, focused their efforts largely on technical design issues such as graphical capabilities, animation system

capabilities, and API (application programmer interface) design. Algorithm animation system capabilities have grown and become more sophisticated. For example, systems went from addressing serial computations to dealing with concurrency, from 2D graphics to 3D, and from single workstation to networked systems. No doubt exists that many of these systems are impressive technical achievements, and represent significant milestones in the development of AA system design. It took detailed planning to create abstractions for graphical layout, positioning, animation actions, etc. and the solutions embodied in the AA systems developed are quite elegant.

However, in this focus on technical design issues, we believe that a number of issues relevant to the ultimate goal of using AAs to effectively portray algorithms have been neglected. For example, with few exceptions [MS94][SH04] [Saraiya04], the capabilities described above have not been evaluated for usability nor shown conclusively to significantly increase AA effectiveness. Virtually all AA systems have been designed in the absence of preliminary studies to evaluate the tasks and needs of the authors and users.

A number of studies have been performed to evaluate the effectiveness of AA for learning, and the results have been varied [HDS02]. In general, these results have not been correlated to factors such as perceptual and cognitive abilities and limitations, AA design, or system capabilities that might offer an explanation of the results. In one of the few studies to look at system capabilities, [Saraiya04] found that providing forward and backwards step controls to one group resulted in them significantly outperforming (on a post-test) a group whose system did not have these controls.

In order to place the AA system design aspect of this dissertation into the proper context, we examine below the development of AA systems, their system goals and capabilities. We

categorize these systems according to their authoring paradigms, and discuss the design models, capabilities and limitations of these paradigms.

3.0.1 Evolution of AA Systems

One perspective on the evolution of AA systems is in terms of the specification paradigm employed. The various specification paradigms (methods of specifying or programming an animation) that have been used over time are discussed in the following sections and include:

- Customized specification (no specialized AA API)

- Event Annotation

- Declarative

- Programming by Demonstration (PBD)

- Scripting

- Automatic Animation

3.1 Customized Specification

Customized specification methods use general graphics or animation libraries for programming algorithm animations. This means that there is no specific API (application programming interface) developed for algorithm animation. The main advantage of this is that an author does not have to learn a new API. A secondary advantage is that the general graphics or animation API may have a wider set of capabilities than may be available through a particular animation system. However, the big disadvantage is it will almost certainly take much longer to implement an AA using a general library than it would using an AA system. Algorithm animation systems have specialized capabilities related to typical AA graphics and animation

needs that distinguish them even from general animation libraries. Because of this, customized

specification is not considered an important AA specification method. In a SIGCSE conference

discussion session we attended in 1998, instructors from Brown University reported that students

using Java libraries to do AA assignments took a great deal of time to implement their algorithm

animations. The Sorting Out Sorting film, discussed below, required a substantial amount of

time to create and produce, though only partially because of the use of a general graphics API.

3.1.1 Sorting Out Sorting

Although there were earlier efforts, it is widely recognized that the field of AA started

with the film Sorting Out Sorting (SOS) [Baecker98] [BP98] [Baecker81]. Developed by Ron

Baecker and a large team at U. Toronto between 1978 and 1981, SOS was a thirty-minute

animated film, similar in flavor to those produced for high school physics classes. Its debut at

SIGGRAPH 83 created quite a stir, and jump-started interest in algorithm animation

[Baecker81].

SOS showed a series of animations of sorting algorithms, accompanied by audio

narration. The film then moved on to an analysis of how fast they could sort data, accompanied

by animation with larger data sets, and finally a comparison of execution speed was made by

showing the large data set animation views simultaneously.

SOS Animations were programmed using batch-processing with a graphical API; there

was no AA API defined for it. SOS took three years to produce, even with a team of

programmers. Why so long? Baecker does not provide specific details about the development

process, though some insights may be gleaned from a paper describing its development [BP98].

Apparently, it was not just the lack of an API. There are at least three reasons we can identify.

The first reason is that much time was invested in the design of the animation content and layouts. SOS established some of the standard graphical display types for sorting, for example, the use of bar and dot displays. In addition, Baecker chose displays they believed would be best for particular algorithms. For example, numbers were used in the heapsort tree display because bars would not be easily comparable in a tree.

Secondly, Baecker recognized that a careful balance was needed between (a) showing a process at a pace that the viewer could follow and (b) boring the user by showing the process at a slow pace. He notes that timing was a central problem [Baecker98]. They decided to present the initial steps of the algorithm slowly to give the student an idea of what was going on. He points out that "totally literal and consistent presentations can be boring" (p.374) and that after slower initial step-by-step explanations he found he had to "add visual and dramatic interest" to "present more advanced material" (p.374). He also notes that timing requirements can vary with the nature of the particular algorithm. Hence, he used customized pacing and timing for different algorithms in SOS. In addition, the pacing often had to be changed during an animation. This seems to have taken some time to fine-tune.

Prior to SOS, Baecker designed the first system for cartoon animation, GENESYS [Baecker71], which incorporated Programming by Demonstration (PBD) techniques for controlling pacing and timing. The pacing and timing design of SOS is likely to have benefited from this experience. Given the large size of the production team acknowledged in the credits, Baecker may have also had advice from animators and filmmakers (SOS).

Thirdly, SOS had a well-defined lesson structure. SOS used audio commentary to accompany the animations. The commentary is used to introduce algorithms, the purpose and goal of the algorithm, give a conceptual overview of its high-level operations, explain its low-

level operations, comment on its performance, and summarize and review it. The use of audio narration may have an advantage over the use of on-screen narration such as accompanying text. In a series of studies, Mayer [MM98] found that this was the case for educational multimedia in some domains, presumably because audio uses a separate perceptual channel than visual presentation. In other words, SOS used classical educational multimedia design, perhaps before such guidelines existed. Baecker seems to have had input from someone on the large production team who knew about lesson design, although this is not mentioned. This no doubt added value to the AA, by presenting it almost as a structured lesson.

Baecker judges SOS as successful [Baecker98] based on general feedback, student interviews and "an informal unpublished experiment" (p.377). Baecker then makes a number of claims for animations, e.g. "we can perceive structure and relationships of causality and ultimately infer what the program is doing" (p.370), but does not substantiate them or propose to study them. However, Baecker notes [Baecker98] that pedagogical AA is "not a trivial endeavor" (p.370) because, among other things, we must "enhance relevant details, devise clear, uncluttered, and attractive graphic designs and choose appropriate timing and pacing" (p.370). However, he believes that "fancy rendering and smooth motion" are not "necessarily required" (p.378). In short, SOS was supported by

- Customized animation for each AA (pacing, control)

- Narrated Animations

- Carefully Planned Lesson Design

- Graphical Design (view design for large and small data sets, minimal number of colors used to indicate changes, side-by-side comparison of views)

SOS set a benchmark that was not recognized, at least at the time. Next generation AA system designers focused on the animation process rather than the design process. Their challenge was to write a system that one could use to author an AA for any algorithm, rather than creating a customized animation for each algorithm using a generic graphical API. Balsa [Brown87] was the first to attempt to fulfill this need. However, Balsa and many of the systems that succeeded it did not readily support the customized animation, customized pacing, and careful instructional design of SOS.

3.2   Event Annotation

The first popular AA specification method was event annotation. In this approach the author defines events, which correspond to functions that perform some graphical action. The events are generated by the algorithm, and serve to invoke graphical action that illustrates the current algorithm operation. The AA system receives the events and carries out the required action. The majority of AA systems use event annotation. The main advantage of this method is that events can be closely linked to source code. The main disadvantage is that extensive annotation may alter the structure of the source code.

3.2.1 Balsa/Balsa-II

Balsa was the first comprehensive AA system. It was developed at Brown University in the late 1980's by Marc Brown [Brown87] [Brown98]. This system was considered significant enough to merit the publication of a book, which attracted widespread attention, based on Brown's dissertation [Brown87]. Balsa introduced the event annotation method of specifying algorithm annotations. Brown calls this 'interesting event' annotation. Balsa had a set of primitives to create views (windows), graphical objects, and to animate these objects. It ran on

the then-new Macintosh, which was the only GUI platform available to the general public at the time.

Balsa-II replaced Balsa in fairly short order [Brown88]. The Balsa-II architecture added capabilities that allowed a programmer to produce comparative animations of the type shown in Sorting Out Sorting. For example, Balsa-II supported the visual comparison of multiple algorithms, allowing them to be run in separate windows. In order to compare the algorithms, AA designers were advised to mark operations of equivalent time complexity ($O(1)$ operations) as "single-step" events. Balsa-II would run each algorithm (in round-robin fashion) up to a single-step event. Algorithm animation designers could also designate other "interesting events" to be used in comparison, e.g. the completion of a loop, insertion of a node, etc.

Balsa-II facilitated execution control through two types of breakpoints – every single-step event was a breakpoint, and the other "interesting events" could also be made into breakpoints. However, these breakpoints had to be specified at compile time, and could not be changed during execution (we address this limitation in our system design in Chapter 5). Events were stored by the animation engine, and could be played in reverse, thus giving the impression of reverse execution.

A major limitation of Balsa and Balsa II was that they lacked authoring-level animation control or pacing. Animation speed was based on the speed of the system, which became a problem as computer system speeds increased exponentially over the years. Also, the animation was not guaranteed to be smooth. Balsa-II included other capabilities such as limited animation interactivity. Researchers at Brown University used Balsa and other tools in teaching practice in a variety of ways, which are described in [Bazik98].

3.2.2 TANGO/XTANGO

3.2.2.1 Main Ideas

TANGO, the next major AA system, also uses the event annotation specification model [Stasko98a]. TANGO was developed by Stasko at Brown University around 1990, followed in short order by an X-windows version, XTANGO. The main goal of TANGO was to allow simple creation of smooth animation of primitive graphical objects along user-defined paths. Stasko theorized that smooth animation would ensure that viewers could follow the animation actions, and thus improve AA comprehension. However, no studies were carried out or proposed to confirm this belief, nor was any supporting evidence presented.

At the time the development of these systems was occurring, it was common for pioneering educational systems such the Alternate Reality Kit [Smith86] to be designed based on similar assumptions. It was quite rare for empirical testing or any form of user testing to be done to confirm their beliefs. Such testing became seriously considered several years later as the educational community began to call for verification of beneficial claims.

On the technical side, TANGO introduced a path-transition paradigm for specifying animation actions. Stasko believes that this was the major contribution of TANGO, and we agree [Stasko98a]. The programmer could specify the path followed by an object (the image), and how the object changed (the transition) while following the path. TANGO established formal abstractions for locations, paths, and transitions, all of which could be defined independently of graphical objects. Each abstract type had a set of functions defined on it.

A path was modeled after a geometric 2D path, a series of (x,y) difference changes, but could also define changes of size, color, etc. It could also define the time between each animation 'frame'. Paths could be manipulated in a number of ways with functions relevant to

animations – iteration, concatenation, and composition, which are essentially list manipulations. Transitions use paths to specify how to modify images. Transitions could use path composition to perform actions concurrently.

Although the animation in TANGO was smooth, the overall rate of animation was machine dependent and also dependent on CPU load. The user was given a speed control slider, but it could be rather sensitive, so an animation could be too slow at one moment then, after a small adjustment, too fast at the next.

Performing complex animation took some planning on the part of the AA author, even with this framework. The paths could be a bit tedious to program if the actions were not straightforward. Layout had to be planned carefully. In particular, complex concurrent actions were generally difficult to specify, once they involved more than a few objects. Some of these limitations were addressed in the design of the next system, POLKA.

3.2.3 POLKA

3.2.3.1 Main Points

POLKA was developed by Stasko at Georgia Tech around 1993 [SK93]. POLKA's primary aim was to extend the XTANGO model to include parallel AA and revise its programming model. This would facilitate the portrayal of parallel/concurrent algorithms and visualizations. Also, in some sequential algorithms, many operations are conceptually concurrent, for example, the exchange of elements in a sort, or the comparison of several paths in a graph algorithm. While the addition of parallel animation is very useful for authors, little consideration was given to the perceptual/attentional implications of this capability.

POLKA's programming model incorporates a number of improvements over the XTANGO model. The path-transition model introduced in XTANGO was enhanced. It readily

supports parallel animation of objects within a view, a challenge for XTANGO. POLKA also supports multiple views.

The POLKA programming model is more modular, and since it is written in C++, it is object-oriented. A POLKA animation is separated into (1) headers of animation functions, (2) animation function bodies (scenes), and (3) algorithms, which invoke animation functions.

Despite the mature animation control design, POLKA did not have time-based pacing control – animation speed still varied with system speed and load. This was due in part to the fact that the popular operating systems of the day did not support certain types of time-based actions very well. However, POLKA does allow the definition of animation speeds in relation to other actions. For example, one action can be twice the speed of another, as defined using a common clock. However, we cannot control the actual pace of the clock – it is system-dependent.

3.2.3.2 Technical Description

To get a feeling for the concepts one must understand to be an AA author using an event annotation AA system of this generation; we therefore present a brief technical overview of POLKA.

POLKA defines an Animator class that performs animation on one or more views, which are members of Animator. The Animator is used to coordinate multiple views, to define common variables, and can contain either the algorithm to be animated or a main event-processing loop that receives the "interesting events" and then invokes the corresponding functions.

A view is a window, with a real-valued coordinate system. Scene functions can be defined on views to perform semantic actions, but these scenes are not required. Views can contain AnimObjects, the graphical objects in the view. Actions can be independently defined

and associated with one or more AnimObjects at a particular clock time by use of the function *program()*. Each view has a clock, which is independent of the clocks in other views. To define concurrent actions, one simply programs each of the multiple actions to start at the same time.

The animation is actually performed when the *animate()* function is called. This function also advances the clock time. The programmer specifies how many frames to generate in the animate call (e.g. animate(n)). Note that clock time is actually the number of frames (i.e. each clock tick is a frame). The animate function checks to see if any of the AnimObjects defined in the view have actions programmed on them for that time period and sends them update and draw messages.

The programmer can vary the relative pace of an Action by specifying how many frames to spread the action over. The relative pace is the pace of an animation action in relation to other actions (in the same program). The absolute pace is the speed of animation the user sees, which is governed by the number of frames per second. The absolute pace is system-dependent (depending on system speed and load) and cannot be controlled by the programmer.

When the viewer is running the animation, he or she can control the absolute pace by using the speed control bar. However, it can be quite a task to get an animation to run at the desired pace. For example, it is frustrating when the user desires that one section run quickly and the next section slowly, and has to try to quickly change the speed before the transition from the fast section to the slow section. Finding the right pace can also be a challenge as the speed can be quite sensitive to small changes of the speed control bar, especially on fast machines.

POLKA includes functions for creating various geometric objects, a variety of Action types (e.g. MOVE, GROW, COLOR), paths, and functions to manipulate them, and a few

functions to assist with basic layout. POLKA also has low-level interactivity support. It provides

a pick function that invokes a callback function defined on a visual object.

POLKA is a solid, well thought out system. However, the authors admit that it does not

completely meet their goal of being easy-to-use [SK93]. The concepts take a little getting used

to, the library takes some more time, and to actually use it to design and implement an AA is

quite a bit of work for the average time-strapped instructor or student. Also, there are no

predefined data structure display objects (e.g. trees and graphs and arrays), so all layouts have to

be designed from scratch.

POLKA is ideal for someone who wants to do a serious animation and has some time to

invest. POLKA continues to be used (a windows version is available) and is also the underlying

engine of the Samba scripting system.

3.2.4 POLKA-RC

POLKA-RC is an extension of POLKA that adds real-time control over pacing, i.e.

absolute pacing [SM95]. It is obvious that the lack of this capability was seen as a serious issue

for an entire system extension to be devoted to addressing it. POLKA-RC replaces the

fundamental notion of the clock as a frame, and uses actual time in its animation functions.

POLKA-RC also adds pacing controls such as slow-in/slow-out to improve the perception of

animation. The authors note that cartoonists use these techniques, and that real-world objects

hardly start and stop abruptly. However, the effectiveness of these techniques was not studied.

Although POLKA-RC is ostensibly an extension of POLKA, their API functions differ

quite a bit at times. While the temporal control capabilities of POLKA-RC are useful, even

simple actions can involve hand-tuned time constants, as shown in the following POLKA-RC

code, which animates two objects on an arc:

```
top_trans = new transition(
    icn_obj1,
    new arc_traj(250,303, 293, 3.8,5.0),
    start_in, MSec(500), duration, Sec(4));

bottom_trans = new transition(
    icn_obj2,
    new arc_traj(250,453,293,3.8,5.0,&slowinout),
    start_after_start_of, top_trans, MSec(0),
    duration, Sec(4));

transition_agent->schedule_transition(top_trans);
```

Higher-level timing and pacing abstractions such as those used in POLKA-RC can help

make authoring easier, but one still has to plan the design of AAs and adjust the original program

in relation to this plan. Detailed AA actions and temporal considerations are the chief issues. For

example, the sequence of actions to animate an operation that finds the shortest edge in a graph

might be:

1. Highlight or cue the group of edges to be considered (for some time period).

2. Move an object along one of more edges

3. Keep track of shortest edge found so far and show it graphically

4. Show the final minimum edge and change its appearance

5. Add it to a new set or tree

If one adds timing and pacing considerations, it is easy to see how the authors of Sorting

Out Sorting spent so much time working on AA design - if one wants to make the AA visually

clear and appealing. Obviously, some sort of support to reduce this design and implementation

burden would be useful. Declarative and Programming-by-Demonstration (PBD) specification

methods are attempts to do so.

3.3   Declarative Specification

Declarative specification methods [Roman92] work by allowing users to define actions that should be performed when certain changes occur or when user-defined conditions are met. Authors define rules that contain the conditions or changes. The AA system then monitors changes or accesses to data structures or variables and matches them against the rules. For example, changes to an element of an array could trigger an animation routine. Declarative specification methods are attractive because they promise to reduce the amount of annotation code needed to portray animations.

A major objective of declarative specification is to decouple or separate the program code from the animation invocation code as well as the animation code. Event annotation only separates the animation code. The declarative idea is to leave the original code relatively untouched, and thus more readable. However, this leads to possible complications. Suppose we have different animation actions at different points that we want to perform using the same rule, or suppose two or more rules are met simultaneously. Which action should be performed first? The order in which rules are fired is critical for the correct operation of the algorithm animation. However, this ordering can be difficult to specify, especially when the source code is in different routines and rules are attached to each routine. The separation of rules from code can make AA design/authoring and debugging harder.

3.3.1   PAVANE

PAVANE [Roman92], developed by Roman at Washington University around 1992, was the first well-known declarative AA system. In PAVANE, a visualization is formally defined as

a mapping from program states to a graphical representation. PAVANE specifies mappings using mathematical mapping functions, which can contain logical or set expressions, assertions or rules. These are specified via dialog boxes.

The main goals of PAVANE are to present a cleaner way of specifying AAs, and to support concurrent algorithm animation. Roman claims that declarative specification can simplify AA authoring. For example, it can reduce the need to save program state, in some cases. However, the declarative approach can get complex. PAVANE uses composite objects, in which one mapping defines a graphical object, which is sent to a second mapping that modifies it, and so on. For example, a node can be defined in one mapping, and the second mapping modifies its appearance or content. This is useful in handling recursion, which is a challenge in PAVANE as actions must figure out their recursion level. An AA author can design graphical objects that can be constructed from others, a form of inheritance that can be used to create a library of objects to simplify AA layout. Although he argues against event annotation, Roman recognizes that event detection is needed at times. One can declare mappings to detect and handle events. PAVANE also supports distributed computation and distributed AA by sharing state between processes**.** PAVANE supports 3D graphics, and the execution environment supports 3D navigation of views and interaction.

Roman states that an evaluation showed that "competent" programmers (with no previous visualization experience) could learn PAVANE and produce an AA within 3 to 6 hours, and that most AAs need less than 24 rules, with 3 to 10 rules being common. While PAVANE has some good features for authors, we believe that quite a bit of effort is needed to write reasonably complex algorithm animations.

Designers kept up the quest for easier AA authoring techniques, and created the Programming by Demonstration and Scripting paradigms.

3.4   Programming by demonstration (PBD)

Programming by demonstration (PBD) is a specification method in which the user demonstrates what the program should do, using a visual tool. The method has been used for general programming, and for end-user programming in particular. For example, many commercial multimedia systems use some form of PBD to specify animations. It has been employed for AA in only a few instances. Stasko's Dance system [Stasko98b] used PBD to specify objects and the paths they should follow, etc, and generated TANGO code. The goal was to reduce authoring effort, and in this regard it showed promise, especially for novice users or those who do not have time to learn other types of AA systems.

One limitation of PBD is that it is often challenging to use it to specify generalized AA actions, for example, a swap that can occur between any two elements in an array, not just two fixed locations. Moving an object along a graph edge should work for any edge of any orientation. Actions are often composed of smaller actions, and this must be accommodated. Timing and pacing can be challenging to specify as well. Even general multimedia animation tools can be quite tedious and time-consuming to use.

Specifying AA actions visually poses additional challenges. Predicting all actions may not be possible when these actions are under program control. Some actions are related to the semantics of a program, so they are hard to predict, e.g. drawing a pointer from one point to another in a tree algorithm. Specifying a generalized action, and relating the action to an algorithm operation can be difficult or tedious. For example, to swap bars, a bar has to be related

to an array value, then one must specify that a swap in the algorithm causes an animated swap and visually specify how the bars must move, perhaps using a template of bars. A usability study of the Lens animated debugger system [MS94] found that users had difficulty constructing mappings between code and the visual representation. Dance [Stasko98b] used dialog boxes to specify which functions would call the actions specified using PBD, and how the parameters would be used. We are not aware of any well-known system that uses PBD for general algorithm animation. Some AA scripting systems (e.g. ANIMAL [RF01] and ALVIS [HD02]) use visual tools that incorporate some aspects of PBD, or can be considered composite systems.

3.4.1 Dance

Dance [Stasko98b] was developed by Stasko at Georgia Tech around 1991 to allow the creation of animation routines via direct manipulation. It uses PBD to specify animation routines (called scenes) for TANGO. The scenes are specified using menus, dialog boxes, and a visual tool, and are then output as C functions. The algorithm then calls these routines with the appropriate parameters. However, AA construction using Dance involves a large number of steps. For example, to swap two bars in arbitrary positions that represent array values in a sorting routine, one would:

1. Define the parameters of the routine (e.g. p1, p2)
2. Create rectangles for use as templates (arbitrary position rather than fixed position) and name them (e.g. bar1, bar2)
3. Define an association between each parameter and the corresponding bar (e.g. p1 to bar1)
4. Position the bars on the display

5. Create a variable that names the location of the lower-left corner of each bar (e.g. loc1 and loc2)

6. Create and name a path (e.g. path1) from loc1 to loc2, and specify its direction (e.g. clockwise)

7. Create a second path (path2) from loc2 to loc1 (can be copied and modified from path1)

8. Create a transition (move1) that moves bar1 along path1, and move2 that moves bar2 along path2

9. Create a composition transition exchangeBars that combines move1 and move2

This is a large number of steps, and each step involves some use of menus, dialog boxes, and visual tools. While Dance does free users from learning the TANGO API and helps with layout, one still must understand the abstract concepts of locations, paths, and transitions. One also must plan what parameters will be needed and how they will map to the algorithm. And of course, the animation concept and layout has to be designed.

Dance leaves some AA construction problems unresolved. One difficult issue is the generalized mapping of algorithm entities to graphical entities. For example, even associating bar heights with variable values is tricky in Dance. However, Dance was a useful system that achieved some degree of success in the quest for easier AA construction.

3.4.2 LENS

LENS is an animated visual debugger [MS94] that can perform actions specified by PBD. LENS was developed by Mukherjea and Stasko at Georgia Tech around 1993. It is a visual tool

that works in conjunction with the Unix dbx debugger, using dbx to access variable values and program control info.

Using the PBD tool of LENS, one can create graphical objects and map them to variables. The authors identified actions and objects commonly used in AAs in a study [MS94]. This included actions such as flash, move, resize, and objects such as circles, lines and rectangles. One can associate one of these predefined actions with an object and specify visually that the action should be invoked when a selected line of source is run.

While LENS showed promise, it had a number of limitations, and one has the impression that further development was not pursued. For example, C expressions were used to specify some aspects of graphical objects and relationships between them (e.g. size, spacing). This aspect of LENS is declarative. The authors note that a direct manipulation interface would be better suited for this purpose. However, they state that it is quite difficult to use PBD to specify some graphical relationships, e.g. the mapping between a disk and a peg in a towers of Hanoi visualization. They also note that there are many visual representations and animations that cannot be constructed with PBD in LENS, and that programming must be used. LENS can also do limited automatic visualization, which we mention in the Automatic Specification section.

3.4.3 ANIMAL

ANIMAL was developed by Roessling, Schueler, and Freisleben at the University of Siegen around 2000 [RF01]. The main goal of ANIMAL is to make the creation of AAs easier using direct manipulation. ANIMAL is a visually oriented tool that allows users to create objects and animate them. It supports a number of object types, including arrays, list elements, and even pseudocode displays. A limited set of animation actions is supported (move, color, etc.), and

action durations may be specified. The ANIMAL visual tool allows users to create animation actions via PBD and dialog boxes. One selects elements previously created with the tool, and chooses actions to perform. It then generates scripts in its scripting language (described in the scripting section). These scripts can be examined and even modified if desired. Here is an example from the ANIMAL visual scripting tutorial [Rößling] that explains how to get a pointer to move from one element to another (italics are mine):

Click on the arc icon. Select a point directly next to the top right corner of the new list element as the arc center and click once.

Now move the mouse to see the outline of the current arc. *Try to manage* that this arc line touches both the tip of the first list element and the left side of the new list element at the same height as that element's tip. Figure 18 on page 30 shows an example of the result. *This may take some time* in trying out possible arc centers. However, using the figure, you can determine where to place the element to make it work.

Next, click on the first element's tip end resting next to the second list element to mark the arc start angle. The next mouse click then goes to the left side of the new list element, and should result in something resembling….

Apparently, specifying semantic actions using visual tools is not always easy. While ANIMAL is quite useful for constructing simple AAs, as the complexity increases, so does the work the user needs to do. ANIMAL has a scripting aspect as well, which is described later in this paper.

3.5   Automatic Visualization

Automatic visualization methods attempt to create an animation without any authoring effort, or minimal configuration. They do this by monitoring changes or accesses to data structures or variables. For example, changes to an element of an array could trigger an animation of some sort. This approach is more typical of program visualization (PV) than algorithm animation. Automatic visualization has been used for graphical debugging and performance monitoring in PV in systems such as ZStep 95 [LF98]. There is some crossover, as Baecker worked on some early graphical debugging systems, and LENS [MS94] used XTANGO [Stasko98a] as a basis for animated debugging displays.

Automatic visualization methods are better suited for PV than for AA because the visualization can be a direct representation of the data or data structures of the program. Even this is a challenge since variables (e.g. indices, stack top, etc) may need human interpretation to be presented in the proper context. The issue is semantics. It is difficult to interpret the meaning of a program from its code, and AA often tries to illustrate the conceptual level of an algorithm. For example, how do we know if an integer variable contains a simple value, or if it contains an index of an element in an array? In the latter case we may want to represent this graphically, say as an arrow to the element. With automatic visualization, actions taken are generally the same for same operations, e.g. all assignments look the same. Some systems such as EBBA [Bates89] attempt to derive higher-level events from low-level events using models and rules, but we cannot create rules for all possible events, and someone must compose these rules. Other systems like LENS have predefined visualizations and behaviors for each data structure. The weakness of this paradigm is that it is practically impossible to reflect all the semantics of an algorithm automatically. However, a number of very useful systems use the paradigm.

3.5.1 FIELD/MEADOW

Steve Reiss and his colleagues at Brown University have produced a number of AA

systems over the years. One of the earliest was FIELD [Reiss98]. FIELD is a programming

environment, somewhat ahead of its time, with automatic visualization capabilities. The goals of

FIELD were to support development activity with visual tools and visualization. It is essentially

an IDE that is a visual wrapper and an intercommunication framework for a number of Unix

tools such as make, profilers, source code control tools, and the dbx debugger. It adds additional

visual tools and an underlying message passing system used to exchange information between

the tools. It provides a visual editor, animated call tree views, a visual call graph, a visual class

browser, and a graphical source module view. FIELD can animate linked data structures using its

automatic visualization capabilities. MEADOW [Bazik98] is a scaled-down version of FIELD

that is designed for use by novice programmers. Some of the more complex features were

removed, existing features were simplified for novices, and the user interface was revamped.

Follow-up visualization systems have been produced such as Jive [Reiss03], though these have

been designed primarily for performance measurement and debugging.

3.5.2 LENS

As mentioned in the PBD section, LENS is an animated visual debugger [MS94] that can

do limited automatic visualization. LENS includes visual representations for variables and

common data structures such as arrays, linked lists and trees. One uses a template to define

which variables should be shown. To show a linked list automatically, the pointer variable, the

value field and the next field are selected. LENS uses this information to render and update the display.

3.5.3 Jeliot

Jeliot [MMSB04] provides automatic animation of Java programs. The main goal of Jeliot is to support novice Java programmers as they develop programs and learn. Jeliot was initially developed at the Universities of Helsinki and Joensuu around 1997 by a research team [HPSTTV97], and it has since gone through a number of versions and design teams. Jeliot2000 [LBU03] adds visualization of control structures, method calls, parameters and return values. Users can also evaluate Java expressions. Jeliot3 [MMSB04] adds visualization of object-oriented concepts, and uses an instrumented compiler to achieve this. Not all Java classes can be dynamically visualized. However, the system is extensible. JGRASP [CHB02] has similar features and functionality.

3.6 Scripting

Scripting systems use a scripting language to specify an algorithm animation. Generally, the aim is to create a simpler method of specification than that obtained with event annotation or declarative methods. Scripting systems generally employ one of two specification methods. In the first, the algorithm outputs the script, which is then read by an interpreter that renders the algorithm animation. This method is used by Samba [Stasko96] and JAWAA [Rodger02]. While the algorithm can use any specification paradigm to produce the output script, many scripting systems use the equivalent of event annotation by placing print statements at appropriate points in the algorithm.

In the second method, the script is the actual algorithm, or a simulation thereof, that is run by the AA script interpreter. The script becomes the conceptual version of the algorithm. Systems using this approach often incorporate a drawing tool to create the visual elements and the layout. Sometimes the visual tool is also used to help specify the animation. Examples of these systems are ANIMAL [RF01], ALVIS/SALSA [HD02], and JAWAA2 [Rodger02] as well. Some of these tools are more like multimedia authoring tools with limited computational ability. Such systems may be easier to use, and can reduce authoring time compared to other specification methods. However, creating complex AAs can prove difficult when there is no 'normal' algorithm. One can wind up writing the algorithm in the scripting language, which usually has limited computational facilities compared to a general purpose language.

3.6.1 Samba

Samba was developed by Stasko at Georgia Tech, and actually runs on top of POLKA [Stasko96]. The goal of Samba is to permit very easy scripting of animations, and its capabilities include naming and grouping of objects, and concurrent movement. Samba AAs are generally constructed from the output of other programs. Samba does not support interaction; these types of scripting systems are generally output only. In order to simplify the complexity the detailed path animation control in POLKA is not implemented in Samba. Otherwise, execution pacing control is limited by the same factors affecting POLKA. Also, the detailed path animation control in POLKA is not implemented in Samba. A Java version, JSamba, is also available that incorporates the POLKA animation engine.

In an unpublished study of students doing animation assignments, we found that one of the main limitations of Samba was that students spent too much time designing layouts and

getting animations to work correctly; another study reports similar findings [Hund99]. Samba is very easy to use and easy to learn, but because it supports only low-level graphical objects (and their composites) it requires authors to build more complex visual representations and layouts.

3.6.2 JAWAA

JAWAA was developed by Rodgers at Duke University around 1998, with a subsequent version 2.0 around 2001 [Rodger02]. The main goal is to allow simple and rapid development of algorithm animations. JAWAA can be considered an extension of Samba that includes common data structures such as binary trees, graphs, stacks, queues, linked lists and arrays. JAWAA also includes a visual layout editor and support for animation of linking arrows and visual markers. It therefore relieves AA authors from dealing with low-level layout issues for these predefined data structures. The scripting syntax for DS construction and animation commands is quite similar to that of Samba, with some extensions. The scripting model prohibits interaction, as with other systems of this type.

As with Samba, JAWAA AAs can be generated as the output of other programs. Rodgers also uses JAWAA as a standalone authoring language, which is feasible because of its support for libraries of data structure objects. JAWAA AAs can also be run within web pages. Naturally, JAWAA appears to reduce the authoring time for simple animations involving the supported data structures. However, it is hard to add features to these data structures, e.g. node names must be integers. JAWAA is not designed to be extensible and thus does not support the addition of new data structures. It also lacks built-in support for pseudocode displays. Overall, JAWAA has a very useful set of AA authoring capabilities.

3.6.3 ANIMAL

The ANIMAL visual AA development system (previously described in the PBD section) [RF01] also allows users to specify scripts with a language that looks somewhat like Samba, with a few notable additions. Any action command can have a duration and a delay, which offers some degree of pacing control. Pseudocode displays are supported and there are functions that do actions such as highlighting a line of code. Aspects of the location of an object can be referenced (e.g. SW corner). The language includes special functions for particular element types. For example, a list element has one function to move the element but not an attached pointer, and another for the pointer, but not the element. The default move action will move both. One peculiarity of ANIMAL is that it is not clear how one would generate scripting code from an external program (as one would in Samba) that can access/manipulate entities created in the visual tool. This would limit the expressiveness of the system since the ANIMAL scripting language has no variables, control structures, etc. The authors refer to an API for Java that is under construction. However, the language seems to be well-designed in many respects, including layout and timing.

3.6.4 ALVIS/SALSA

ALVIS is a visual AA development tool based on the idea of constructing paper prototypes of algorithms. SALSA is a scripting system that works with ALVIS. Hundhausen started the design of ALVIS and SALSA [HD02] at the University of Oregon as part of his dissertation research and completed its implementation at the University of Hawaii around 2001. In an ethnographic study, Hundhausen found that students constructing AAs concentrated on the conceptual level when they used tangible material such as paper cutouts, whereas students using

AA programming tools such as Samba focused on low-level details [Hund99]. ALVIS was designed to provide operations analogous to the manual operations performed by students using the cutouts. ALVIS supports pen-based interaction and construction of graphical objects, but these objects are graphics and not variables or data structures, although they may represent such conceptually. Execution is reversible, and one can change the objects during execution. Technically ALVIS is a PBD tool, but because of its tight integration with SALSA, we have included it here.

ALVIS uses the visual tool to construct a visual representation and generate SALSA scripts. SALSA is a scripting language that is also used to create animations that act on the generated visual representation (which is also stored as a script). SALSA is unusual among AA scripting languages because it has some control structures. SALSA essentially uses only string representations of data; for this and other reasons (e.g. the limitations of the control structures), it is difficult to use it to write general-purpose algorithms. However, the pair of systems does seem to support easier AA authoring, where they are suitable for the algorithm.

3.7   Summary and Conclusion

AA system designers face formidable challenges in the quest to create systems that effectively support the tasks of AA authors and are effective and usable for AA users. Over time, complex technical problems have been solved that relate to AA system design. However, many effectiveness and usability challenges remain. Specification is not easy for almost any non-trivial AA when we want to control timing, pacing, movement, interaction and layout.

Custom development is not constrained by the AA system, and thus offers complete flexibility for all aspects of the algorithm animation. However, since it does not use an API

designed to support (targeted to) AA tasks and therefore all development must be done from scratch.

Event annotation allows authors to map animation actions closely to exact points in the algorithm, yet separate the actual animation from the algorithm. However, Brown notes that interesting events can include things that are not relevant to the algorithm itself (e.g. previously unused else's), and that this annotation can change the structure of the original code significantly [Brown98].

Declarative specification allows the separation of rules that invoke actions and the animation from the source algorithm. Rules can be hard to write and complex. The order in which they are invoked can be hard to predict, which, along with their independent placement from the source, makes them hard to debug.

Scripting languages are intended to have a simple syntax to make authoring easy. If the script is generated by an external algorithm, interaction with the AA is usually impossible. Also, animations of complex algorithms can be difficult to design if the algorithm is written in the scripting language and that language does not support generalized computation.

Programming by demonstration (PBD) allows authors to do layout and/or animation using a GUI or dialogs. However, this process is not always as simple as it sounds, especially for generalized or complex routines.

Automatic visualization may seem ideal, requiring no authoring, but it is generally impossible to automatically animate the conceptual aspects of an algorithm as this would require automatic interpretation of the semantics of the algorithm and its variables. How does one automatically derive a custom portrayal or animation of a top pointer to a stack, for example? Hence, automatic visualization is used primarily for visual debuggers, employing standardized

visual representations. The strengths and weaknesses of each paradigm are summarized in Table

3.1.

Table 3.1:   Comparison of the AA specification paradigms.

Specification Paradigm	Strengths	Weaknesses
Custom	Flexibility; not limited by AA system capabilities	Lengthy development; no specialized AA support
Event annotation	Integrated coordination with algorithm; Separation of animation	Changes source algorithm/code structure
Declarative	Separation of rules and animation from source algorithm	Rules can be complex, difficult to formulate, hard to debug
Scripting	Simple syntax for easier authoring, can use any specification paradigm	Can be hard to write AAs of complex algorithms. Can limit interaction with AA
Programming by demonstration	Simplifies specification of layouts and/or animation	Specification of non-trivial actions or layouts is often difficult
Automatic visualization	No authoring needed (may have to mark the variables to be animated);	Generally impossible to automatically derive good animations of conceptual aspects of algorithm

Clearly all the specification paradigms face serious design challenges. In a recent study,

Naps reports that two-thirds of instructors found that the length of time needed to create an AA

was a limiting factor in their use of AA [Naps03]. This points to the complexity of writing an

AA using many systems. The question is whether this is inherent or to what extent we can make

authoring easier.

Some systems such as SALSA/ALVIS reduce authoring time. However, the

SALSA/ALVIS system and many like it have limitations with respect to authoring complex

animations, and non-trivial temporal control is also an issue.

The way that temporal control issues are addressed in POLKA-RC is a good example of

how designers can be tempted to focus on the technical aspects of algorithm animation. The

technical problem is solved, but specifying all but trivial temporal control remains complex for the author. In POLKA or POLKA-RC, all actions have to be planned to fit into an overall schedule for the animation. Not only is this tricky, but sometimes variables must be created to mark points in time, for example, the end of a swap, the start of a pass. This planning is not normal activity for a student or even an instructor just writing an algorithm.

Baecker proposed that timing and pacing are very important for AA design, and paid attention to this in his work on SOS. Brown begins a book chapter [BH98] by stating that "designing an enlightening software visualization is a tricky psychological and perceptual challenge" (p.81). Some systems such as ANIMAL and SALSA/ALVIS include some support for pacing and timing, but empirical knowledge about the role of perception would be useful information for AA systems designers, as well as AA authors.

There are few studies of the usability of AA systems [SH04] [Saraiya04]. The capabilities of AA systems have not been matched to user needs, with few exceptions [Hamilton-Taylor02a][HD02]. So even if the specification problems were solved, would we be specifying the right things, and showing them in the right way?

What is needed is to establish AA system design on a better knowledge foundation in terms of all the issues that impact on it, including the needs of learners and authors, perceptual and attentional issues, cognitive issues, and visualization design issues. It is our intention to examine what these issues are, and to initiate the process of investigation. We have in fact, begun this process, in our studies of instructors [Hamilton-Taylor02b] which informed our design of SKA (The Support Kit for Animation) [Hamilton-Taylor02a], and in our ongoing collaborative perceptual studies, which are described in Chapters 4, 5 and 6, respectively.

CHAPTER 4

A NATURALISTIC OBSERVATIONAL STUDY OF INSTRUCTORS

4.0   Introduction and Motivation

Algorithm animation systems are far from being a standard part of teaching

practice for relevant courses, as Naps found in a survey of instructors [Naps03]. In

general, the design of algorithm animation systems has not been based on the context and

practice of algorithm and data structure discussion. Instead, the focus has been primarily

on providing graphical capabilities for animation, as discussed in Chapter 2.

The study of user needs and tasks is an essential component of modern HCI

system design. This is the cornerstone of user-centered design methods [Norman88] such

as contextual design [Beyer98], in which the first step is to study the user's existing tasks

and workflow. Design proposals are made with the objective of enhancement the existing

system, while a careful attempt is made not to displace aspects of the existing workflow

process that enable the tasks to be accomplished. Systems are then designed to support

these tasks.

It seems self-evident that we need to examine the context in which students learn

algorithms before we propose learning models for algorithm animation, or design

algorithm animation systems. This context of learning has been practically ignored, apart

from [Hund99], which examines student construction of AAs. We decided to study one

aspect of this context of learning. It is our position that an appropriate starting point in the

design of an algorithm animation system to be used in a classroom setting is to examine the tasks instructors perform and the context of these tasks.

It is common knowledge that instructors, tutors, student peer groups and individual students are the participants in the learning process. We know that the participants and textbooks use an informal conceptual diagramming scheme to introduce and discuss data structures, and use variations of pseudocode to describe algorithms, as discussed in Chapter 3. However, little formal knowledge exists about the use of these diagrams and algorithms, and other entities in this process.

4.1   Related Work

Naturalistic inquiry methods [LG85], ethnographic inquiry methods [FW96] and contextual design inquiry [Beyer98] are all related. Naturalistic inquiry is the broadest term covering all these methods. Naturalistic observational methods suggest a passive approach, which aim not to influence or control the natural environment being studied. Ethnography involves highly detailed observation and inquiry. Contextual design basically employs a less intense form of ethnography inquiry, combined with a specific set of activities and diagram notations that were developed by the authors.

In our study, we draw on elements of various naturalistic design methods. Scaife and Rogers also found that they had to "adapt and combine well known user-centred design methods" in conducting naturalistic studies of users in order to fit the needs and "concerns specific to the design of virtual environments … to support learning" in a virtual theatre system for children [SR01].

Hundhausen conducted ethnographic studies of how students construct algorithm animations [Hund99], comparing the use of tangibles such as paper cutouts to development with the Samba system [Stasko96]. He found that the tangibles facilitated higher-level conceptual design, and used the results to design similar operations as the foundation of the design of SALSA and ALVIS [Hund99].

Pane studied how adults and children performed algorithmic problem solving and used this information to design HANDS, a programming language and environment for children [Pane02a][Pane02b]. Petre studied the mental imagery of software design experts and found that these experts use a language of diagrams, but generally don't use visualization software because these programs do not mesh with their tasks and needs [Petre99].

4.2   Overview and Objectives

The broad objective of our study was to investigate the activities performed by instructors in algorithms and data structure lectures, and how the instructors used artifacts in that process. The method of investigation we used for the study involved the following primary goals:

1. To identify and describe the activities and tasks performed by instructors

2. To identify and describe the types of artifacts used, and how they are used

3. To describe how the task environment is used

In the context of this dissertation, *activities* are carried out in the process of teaching a topic or subtopic, and include discussion, derivations or proofs, working

through examples, problem solving, and so on. *Tasks* are typically smaller units and include writing, drawing, showing a transparency, erasing, etc. Tasks can also more specific or may be a subpart or combination of other tasks such as constructing a diagram, annotating, making corrections, and so forth. *Behaviors* such as gestures of various types, facial expressions, and tone of voice are also involved in activities.

Artifacts are relevant entities used by instructors such as text and diagrams, formulas and algorithms. The task environment includes the facilities of the classroom or lecture hall, such as black/whiteboards, overhead projectors, markers, computers, and so on.

This method of investigation and analysis is derived from activity analysis [Nardi96], contextual analysis [Beyer98], ethnographic analysis [FW96], and other naturalistic observational analysis methods. In activity analysis, tools mediate or facilitate the performance of activities on artifacts. The tools, in this case, are found in the task environment.

4.3   Study Design

This study was conducted over a number of semesters in order to observe a number of instructors. Four instructors participated between 2002 and 2004 (three at UGA and one at UWI Mona, Jamaica), and five courses were involved, which included three offerings of one of the courses. The courses were sophomore-level data structures/algorithms and upper level-undergraduate/lower-level graduate algorithms. Three instructors taught the same sophomore data structures/algorithms course in different semesters at UGA, one of whom also taught the graduate algorithms course at

UGA, and one instructor taught a sophomore data structures/algorithms course at UWI. We identified a set of topics that included a number of commonly taught algorithms and data structures; these included topics in trees, graphs, and search lists. The relevant lectures were videotaped. In our videotape analysis we identify the activities and tasks performed in each segment and the time spent on each task, noting the use of various artifacts. Interim results were published in [Hamilton-Taylor02b], with further analysis continuing.

4.4 Results

The study generated a large body of data (over twenty hours of videotape), which took some time and effort to transcribe into an appropriate form, but was well suited for the activity/artifact analysis. The results were fascinating, especially as we probed the connections between the activities and artifacts.

4.5 Activity Analysis and Artifact Use

We found that instructors performed a similar set of activities and used similar artifacts, with some variation depending on the topic. The activities and artifacts are generally well known, so identifying them presented no great surprises. Activities included discussion, description, derivations or proofs, doing examples, problem solving, tracing, question and answer interactions, and so on. Tasks included writing, drawing, constructing diagrams, annotating them, making corrections, erasing, showing a transparency, and so forth. Artifacts included definitions, descriptions and other text, data structure diagrams, other diagrams, formulas and pseudocode. The artifacts and some

tasks performed on them are shown in Table 4.1. Making corrections and erasing was not considered integral to the artifacts, though common to all artifacts, and therefore not included in the table.

Table 4.1.   Artifacts used by instructors and common tasks on them.

Artifacts	Tasks
Data Structure Diagrams	Constructing Manipulating Annotating Performing operations Copying to show state
Pseudocode	Tracing Constructing Annotating
Formulas	Deriving Annotating
Definitions	Deriving Annotating
Descriptions	Annotating
Other Diagrams	Constructing Manipulating Annotating

An interesting finding was that data structure diagrams can be abstract, concrete, or both. An abstract diagram typically contains no data and is often a compact representation of a data structure. An example of this is the use of a triangle or rectangle to represent a subtree in an AVL tree (Figures 4.1a, 4.1b) or a binary tree with empty nodes (Figure 4.1c). Abstract diagrams are generally used to show some property of the data structure without requiring that all the details be shown, and thus save the instructor drawing time and space. A concrete representation shows all of the important data, for example all the key values in the data structure diagram. Some diagrams are part concrete and part abstract, for example, a concrete tree with an abstract subtree (Figure 4.1b).
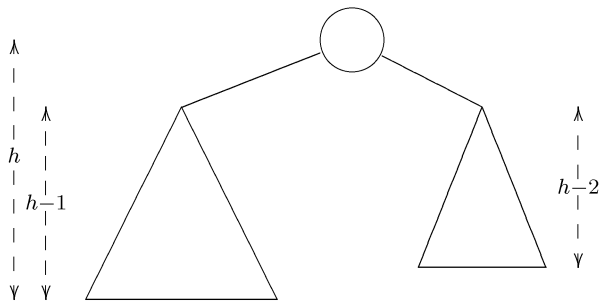
Figure 4.1a: Abstract AVL tree with height annotation on abstract subtrees.
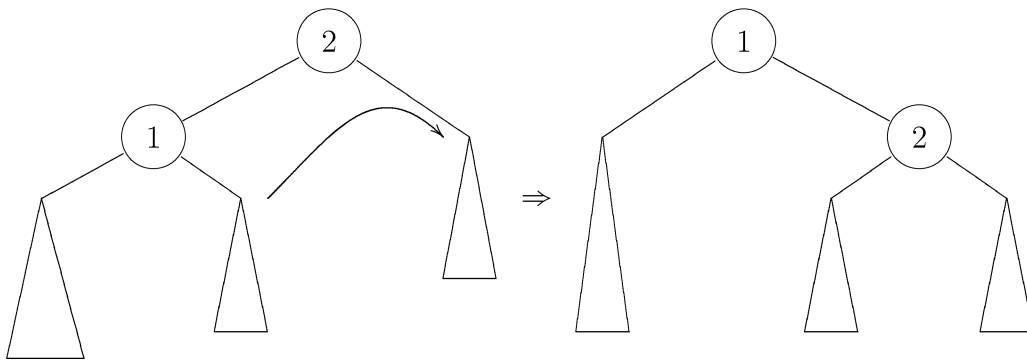
Figure 4.1b: Rebalancing operation on AVL tree with abstract subtrees.
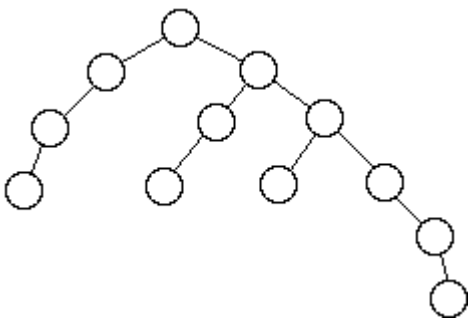
Figure 4.1c: Another form of abstract binary trees

In general, there was a high degree of use of multiple artifacts during activities. These artifacts were used both separately and in various combinations, for example, definitions and formulas, diagrams and formulas, diagrams and pseudocode, and so forth. What was particularly interesting was how the artifacts were used in coordination during

the activities, and how annotation and behaviors were used to connect the artifacts in the course of performing the activities. For example, a diagram might be used to derive a formula using diagram annotation to point out properties of the diagram that are generalized in the formula. Gestures between the diagram, the annotations and the emerging formula might form (imaginary) connections between them. For example, one instructor used AVL tree diagrams for various insertion cases, annotated them with their height, and derived the general formula for AVL tree insertion time from this.

What was particularly noteworthy was the high degree of data structure diagram (DSD) use in many activities. We witnessed instructors constructing DSDs from definitions, constructing definitions from DSDs, deriving formulas from DSDs, constructing algorithms from diagram operations, and performing tracing of pseudocode algorithms on DSDs, and other forms of cooperative use. We performed further analysis on this diagrammatic aspect of the data, as it seemed to be an integral component of the entire instruction process.

4.6   Data Analysis

Our approach to videotape analysis is to divide topics (e.g. binary search trees) into subtopics (e.g. binary search tree insertion), and divide the subtopics into segments - for example, an introductory discussion, examples, further discussion, and analysis. We identify the activities and tasks performed in each segment and the time spent on each, noting any use of artifacts and behaviors. We record the use of annotations and gestures used in conjunction with the artifacts. We were particularly interested in the use of data structure diagrams and their combined use with other artifacts, especially pseudocode.

An example of this is the summary of the binary search tree deletion video session in

Figure 4.2.

Duration (min)	Activity
Part 1	
1:00	Introduces topic.
0:30	Copies tree diagram.
0:30	Annotates tree diagram with keys to be deleted, traces leaf deletion.
1:00	Traces key deletion, constructs two possible resulting trees.
1:00	Discusses general rule for deleting interior nodes.
2:30	Traces key deletion, circles successor/predecessor nodes, writes notes on them, draws resulting trees. (All the traces above were done using DSDs only)
Part 2	
6:30	Writes deletion algorithm, describing fragments along the way. Annotates one algorithm fragment with explanation.
1:00	Continues writing algorithm, referring to tree diagram to explain what a certain fragment does.
1:00	Refers to diagram to show what the fragment about to be written will do. (Both references are gestures along path of tree that the algorithm fragment will follow).
0:30	Copies tree diagram.
2:00	Traces key deletion using algorithm. Annotates tree diagram with pointer variables from algorithm (Diagram becomes somewhat cluttered with arrow annotations).
3:00	Starts redoing example because of errors encountered. Needs to get rid of arrows, so erases and recopies tree diagram. Draws added arrows more carefully, traces very carefully, going back and forth between diagram and algorithm.

Figure 4.2.   Record of instructor activity

As we have noted, DSDs were employed in conjunction with other artifacts in

many activities and tasks, and we wanted to understand this usage. In order to analyze the

use of DSDs, we recorded their use in various contexts, as summarized in Table 4.2,

focusing on the tasks, gestures and annotations involved.

Table 4.2. A summary of task types for each subtopic, and the use of data structure diagrams and other artifacts.

Topic	Diagram			Diagram Usage			
	Type	tracing	analysis	discussion	drawing	gestures	Annotations
Binary Search on Arrays, part 1	Array	algorithm and diagram	None	Yes	construct array	point to statements of algorithm	index vars for array in trace
Binary Search on Trees, part 2	Binary Search tree	algorithm and diagram	Yes	Yes	construct tree	point to tree properties, path, nodes	
Interpolation Search	Array	algorithm and diagram	Yes	Yes	copy array	point to array area accessed	index variables for array in trace
Binary Search Tree Insertion	Binary Search tree	algorithm and diagram	None	Yes	construct, copy tree, update tree	point to tree path, nodes	add, cross out pointer variables during trace
Binary Search Tree Deletion, part 1	Binary Search tree	diagram only	None	Yes	construct, copy tree, update tree		mark subtrees. Add, cross out pointer variables
Binary Search Tree Deletion, part 2	Binary Search tree	algorithm and diagram	None	Yes	construct, copy tree, update tree	show path, affected nodes before writing algorithm	add, cross out pointers. Add arrows to show path followed
AVL Tree Insertion	AVL Tree, Abstract AVL tree	algorithm and diagram	Yes	Yes	construct, copy tree, update tree	point to tree path, nodes, rotation direction, subtree movements	add, update node balances, add rotation arrows
AVL Tree Deletion	AVL Tree, Abstract AVL tree	diagram only	Yes	Yes	construct, copy tree, update tree	point to tree path, nodes, rotation direction, subtree movements	add, update node balances, add rotation arrows
2-3 Tree Introduction and Insertion	2-3 Tree	algorithm and diagram	Yes	Yes	construct, update tree	movement of nodes with insertion in the tree	Label nodes
2-3 Tree Deletion	2-3 Tree	diagram only	None	Yes	construct, update tree	indicate direction of rotation	
Topological Graph sort	Directed graph	diagram only	None	Yes	construct graph, update graph	Indicate direction of new edges	Add edges

As shown in Table 4.2, we have observed a number of types of annotations and gestures. For example, annotations are used to track variable values during tracing and highlight sections of data structures while gestures are used to identify elements that are being accessed or compared, and to indicate properties such as the number of elements, path followed, etc., for an entire DSD or part thereof.

Table 4.3.   Percentage of time spent on each type of activity.

Topic	Time%				
	Trace	Analysis	Discussion	Joint	Total (hr:min:sec)
Binary Search, part 1	80%	0%	20%	0%	05:00
Binary Search, part 2	16%	77%	7%	49%	30:30
Interpolation Search	47%	24%	29%	0%	24:30
Binary Search Tree Insertion	25%	0%	75%	19%	08:00
Binary Search Tree Deletion, part 1	67%	0%	33%	0%	06:00
Binary Search Tree Deletion, part 2	49%	0%	51%	14%	17:30
AVL Tree Insertion	52%	18%	30%	33%	1:45:00
AVL Tree Deletion	24%	19%	57%	33%	21:15
2-3 Tree Introduction and Insertion	32%	30%	38%	29%	19:00
2-3 Tree Deletion	59%	0%	41%	27%	11:00
Topological Graph Sort	37%	0%	63%	27%	04:30

We recorded the time spent doing various types of activities, as summarized in Table 4.3. The time spent using diagrams (tracing or joint use) varied considerably over the topics observed thus far, but in many cases it is a substantial portion of total time spent, especially when part 1 and 2 of the topics are combined. Note that the percentages add up to more than 100% unless joint time is excluded. Joint time is time spent on discussion and diagrams, or analysis and diagrams. In our observational data, tracing

always involved diagrams. One interesting finding here is that tracing, which includes diagram operations, often occupies over half the time of the activity. Diagram construction, manipulation and tracing all consume large quantities of time.

4.7   Data Structure Diagram Use and Tracing

As we have previously noted, we found that data structure diagrams are frequently used by instructors. We have observed data structure diagrams being used primarily in (a) tracing algorithms, sometimes without a written algorithm, (b) discussing or describing properties of the data structures or algorithms, (c) deriving or confirming theoretical analysis of algorithms.

We found that there were certain common patterns of DSD use, particularly with tracing. An instructor will usually conduct an introductory discussion about an algorithm or data structure, then proceed to show relevant examples. In the case of a data structure, examples of diagrams, storage formats, and common operations are generally presented. The effect of the operations is sometimes shown without initial reference to an algorithm. For example, the effect of a height-balanced tree rotation might be shown prior to the discussion of the particulars of an algorithm to achieve this.

The vast majority of algorithms operate on a data structure. An instructor will typically construct a data structure diagram, then trace the execution of the algorithm, using the data structure diagram as input. If the algorithm modifies the data structure diagram, the changes are made manually to the diagram. Even if no changes are made, some recording of which parts of the data structure have been accessed may be done on the diagram. For example, we might record which vertices have been visited in a graph

search algorithm. Other algorithms construct a data structure as they execute. The initial data structure diagram is empty, but the instructor still has to show changes on the data structure diagram over time. Other algorithms require input data structures and also construct data structures.

A high-level pseudocode version of an algorithm is often used for discussion. The idea is to ignore the low-level details that do not relate to the main operations of the algorithm. An instructor will generally show the effect of each step of the algorithm on the data structure diagram(s), moving back and forth between the algorithm and the diagram. During this process of tracing, instructors may perform a variety of tasks. They may highlight part of a data structure diagram, such as a path in a graph. They may record particular states of a data structure by copying the diagram, perhaps at the end of each major phase of the algorithm. Students may be asked to predict the state of the data structure after the next operation or phase. An instructor may go slowly during the first few passes through a section of an algorithm, then go faster, just showing the net effect of that group of operations thereafter. Finally, instructors may need to use a particular data structure instance to show special case behavior of an algorithm, and need to trace quickly up to some point in the algorithm where the special case behavior exhibits itself.

Of note are the various types of annotation used. Some are part of the data structure such as an edge weight; others note properties of the data structure such as its height. Both vary somewhat widely by data structure. For example, an edge in a flow graph has multiple annotations. An AVL tree node has a balance as well as contents.

Every type of data structure diagram seems to have its own annotation needs, and many algorithms that operate on these diagrams have additional annotations they add to

the diagrams. Also, instructors can have an individual annotation style, though we did not observe much of this.

4.8 Instructor Style

There were some differences in instructor style worth noting, both in the use of artifacts during activities and in media use. We will briefly describe some of these differences. The instructors used either black/whiteboards or transparencies. None of the instructors used any computer-based media during our study; no animations or powerpoint slides.

Instructor 1 used transparencies of various artifacts: diagrams, definitions, algorithms, formulas and theorems. He would often perform a trace on example diagrams by annotating them with a transparency marker, and would frequently switch between them and transparencies of algorithms, definitions, formulas and theorems to show the relationship between them, derive properties, and so forth.

Instructor 2 used the black/whiteboard, and generally had a well structured order, giving definitions, followed by diagram examples, then tracing, then derivations. Her pace of presentation allowed time to observe, and seemed to suggest an even pace. The gestures between the artifacts also followed this pace. One might see an even, expansive vertical spreading of the arms to indicate the height of a tree and its relation to a formula.

Instructor 3 also used the black/whiteboard, and often constructed or derived the algorithm from the examples he had performed on the algorithms. For example, he performed various cases of AVL tree insertion and rebalancing, then derived the algorithm cases from them. His style tended to change by topic. He tended not to point

between the diagrams and other artifacts, but to alternate between them in short durations. However he often used gestures to connect them. He used annotations heavily, both on diagrams, and as connecting devices between them and formulas. He tended to do tracing by performing operations on DSDs rather than by using pseudocode as the driving source.

Many of the gestures use the black/whiteboard or screen as a backdrop, so one might see an expansive vertical spreading of the arms to indicate the height of a tree, for example. One advantage of the black/whiteboard is the space it offers. It does have the restriction that one has to write or draw all artifacts, and perform manual manipulations. On the other hand, it affords annotation and gestures. Classrooms that have an accompanying projector can relieve the instructor from writing and drawing everything, but manual manipulations remain, though one can show changes with static slides or even with animations.

4.9   Requirements Derived from Task Analysis

We found that data structure diagram use is an integral part of discourse and the learning process. Though this varies considerably by topic, as shown in Table 4.3, we observed that, on average, more than half the time spent on a topic involves the use of DSDs. One might almost posit that the process is diagram-centric, as many of the other activities are performed in conjunction with the use of these diagrams. To summarize, we found that:

1. DSDs are used primarily for tracing tasks, but also for analysis and general discussion.

2. DSD construction, manipulation and tracing using DSDs are often performed in the absence of a written algorithm.

3. Tracing is performed in a flexible manner, which varies from doing a single operation to showing the effect of a segment of an algorithm.

4. DSDs often have to be copied to perform a new trace or record a history of DSD states.

5. Annotations are used in a variety of ways. For example, they are used to record changes, show a path or series of accesses in DSDs, and to relate steps in the algorithm to actions on DSDs.

6. Gestures are used to connect spatially separated entities, give an intuitive indication of direction or size, etc. For example, a gesture can be used to show the direction of a search in a DSD.

However, we observed that these activities and tasks are time-consuming. It is very-time consuming to construct large, detailed diagrams. Diagrams often have to be redrawn to show different examples. Updating, correcting and erasing all consume time. Transparencies can be useful, but they can only be annotated, not changed.

Tracing has particular limitations. It can be a long process. It can be tedious and error-prone. To record a visual history of changes requires that diagrams be redrawn repeatedly, and the time to do so increases in proportion to the complexity of the data structure diagram. It can be difficult to reconstruct the intermediate states, which are

often recorded by erasing or overwriting the previous state. It is difficult to go over a

trace without redoing it, particularly without a visual history. Tracing special data cases

or algorithm variations takes yet more time. Even highlighting and removing the

highlighting notation (using different line types, etc.) can be a chore. Outside of this

study, we have seen well-regarded instructors spend multiple lectures discussing a

complex algorithm, with a significant portion of that time devoted to tracing.

Tracing can take so long that students lose concentration and lose track of the

flow of the algorithm and the discussion. In addition to this, it is quite difficult for

students to take notes, especially to show state changes to a data structure diagram. It is

hard for students to walk away with a detailed record of the trace to compare it to their

reconstruction later on. Instructors can also make mistakes as they act as a virtual

execution machine, draw and erase, talk, and answer questions simultaneously. In

summary, while tracing and diagram manipulation are integral activities in the instruction

process, they can be tedious and error prone. This makes these tasks prime candidates for

technological support. We have identified some activities, tasks and artifacts that could

be the target subject of this type of support, as summarized in Table 4.4.

Table 4.4. Tasks that are time-consuming that could be supported by technology. Making
corrections and updating are time-consuming tasks common to all artifacts,
and are thus not shown.

Artifacts	Tasks
Data Structure Diagrams	Constructing Manipulating Annotating Performing operations Copying to show state
Pseudocode	Tracing Constructing Annotating

A means of 'tying together' the data structure diagram, the algorithm and other artifacts is also a desirable feature of a system that intends to support the activities we observed. We note that support for gestures in software cannot emulate natural human expressiveness. However, a system can afford support for tasks such as highlighting, and might provide support for gestures (albeit limited compared to manual system), for example, by using a projector and a laser pointer.

4.10 Conclusion

We have observed that instructors share some common methodological ground in teaching data structures and algorithms. The process involves a complex web of multiple interrelated activities and use of artifacts. We were specifically interested in the use of data structure diagrams and pseudocode in the classroom, having found that instructors spend much time constructing, manipulating and performing simulations using these media. We also believe that it is difficult for students to record, replay, and explore these manipulations and simulations.

Our study shows that the manual processes of diagram manipulation and tracing are clearly important, but challenges and limitations make them ideal candidates for support of technology. An AA environment that intends to support these tasks and needs can enhance the user's ability to perform these tasks by providing operations that reduce the time and effort that a user would expend manually, such as copying, saving and retrieving diagrams to reduce manual drawing time, and automatically updating data structures during tracing. In Chapter 5, we describe how we used the results of this study to derive a set of requirements for an AA system, SKA (the Support Kit for Animation),

which provides enhanced support for such tasks. To the best of our knowledge, this represents a new approach to the identification of requirements for algorithm animation systems in the classroom context. We believe that this analysis will contribute to better algorithm animation system design.

In the future we plan to continue analysis of the data, examining annotation of data structures and other artifacts in depth, as well as other aspects of the data. We plan to conduct further studies of the learning process, involving students as well, as we believe that this will yield additional findings information about the activities and artifacts involved in the process, and contribute to further design improvements.

CHAPTER 5

SKA: THE SUPPORT KIT FOR ANIMATION

5.0 Motivation

Existing algorithm animation systems typically have been designed without formal study of related teaching practices. We believe that this has contributed to low adoption rates of these systems, since these systems may not have addressed the needs of these users. We conducted a naturalistic observational study of instructors teaching data structure and algorithm topics (described in Chapter 4), with a focus on the use of diagrams and tracing [Hamilton-Taylor02b]. The results of this study were used to inform the design of SKA, the Support Kit for Animation [Hamilton-Taylor02a]. Our methodology draws on user-centered design methods [Norman88] such as contextual design [Beyer98] and artifact analysis [Nardi96]. In a similar vein, Hundhausen conducted ethnographic studies of how students construct algorithm animations, and used the results in the design of SALSA and ALVIS [Hund99]. Pane designed HANDS, a programming language and environment for children, based on a study of algorithmic manual tasks performed by adults and children [Pane02a] [Pane02b].

Another important issue that impacts the adoption of algorithm animation systems is the time it takes to author an animation. The average instructor lacks the free time to create algorithm animations. Naps finds that two-thirds of instructors surveyed express this concern [Naps03]. Some systems, for example Samba [Stasko96], have attempted to address this issue, but it appears to be largely unresolved. One study found that students took an average of 33 hours to produce a Samba animation [Hund99].

One must write code to create, display, and modify data structures such as graphs and trees in all but a few systems, such as JAWAA [Rodger02] and Animal [RF01]. Where data structure objects are provided, systems often do not support general purpose programming operations on them, or semantic manipulation; they are not active objects. We have attempted to minimize authoring time in SKA by providing an extensible library of visual data structure objects, and a set of simple operations that facilitate access to these objects.

In order for users to comprehend animations, we believe they must occur at a pace that a user can follow perceptually. SKA attempts to support this by providing an action API (application programming interface) and animation engine that supports time-based pacing. Other systems such as Polka-RC [SM95] and Animal [RF01] also provide time-controlled actions.

We will describe how we derived the requirements of SKA from the observational study. We discuss how we used our tool design philosophy with these requirements to design SKA. We will then describe how to use SKA, and how to create an animation using SKA. An overview of the SKA architecture follows. Finally, we present future directions for SKA.

5.1   Requirements Derived from User Study and Task Analysis

An examination of the tasks the instructors perform while discussing algorithms and data structures in a lecture in the absence an algorithm animation system allowed us to identify some tasks that could be supported by a system, as shown in Table 5.1 (and previously in Table 4.3). We found that some tasks performed by instructors, particularly diagram construction/manipulation and tracing, can be tedious and error-prone. Tracing is the manual simulation of the execution of an algorithm.

Table 5.1: Tasks we identified in our study of instructors that are candidates for system support. Making corrections and updating are tedious tasks common to all artifacts, and are thus not shown.

Artifacts	Tasks
Data Structure Diagrams	Constructing
	Manipulating
	Annotating
	Performing operations
	Copying to show state
Pseudocode	Tracing
	Constructing
	Annotating
Other Diagrams	Constructing
	Manipulating
	Annotating

We observed that tracing takes place in a larger context. Part of this context is data structure diagram construction. Sometimes conceptual data structures diagrams (DSDs) are drawn just to show examples of the DSD, but they are not used for simulation later. Other DSD instances are manipulated to show an operation (e.g. add an element) without explicit reference to an algorithm. Yet other DSD instances are drawn because they cause special case behavior when an algorithm is executed on them. Arbitrary textual or visual annotations may be made on the DSD instances and/or the algorithm. Sometimes a history of selected DSD states and/or variable states is constructed, and the selected set can vary during a particular trace. We believe that this construction and free manipulation of data structures diagrams should be supported by technology.

Most algorithms have major phases or passes, which repeat until some goal or state is reached. A visual history of changes usually records the state at the start or end of major intervals or phases. Particular critical steps in each pass determine the course of action for the pass, or if

the algorithm has reached the final state. These steps may require close examination. After a number of cycles, a pattern may emerge and some simulation details in the remaining cycles may be skipped. This is another aspect of tracing we wish to support, by providing flexible execution and breakpoint controls, and a means of recording visual history. We note that user controls were found to be important in AA system use; [Saraiya04] found that user-friendly execution controls provided a significant advantage (on a post-test) for a group using them, compared to a group whose system did not have these controls.

5.2   Design Approach

The objective of our system is to support this set of activities we have just described with technology. We believe that technology can at least partially address the limitations we have noted. We want to create an environment that allows users to execute algorithms on conceptual-level data structures. To support these tasks, our system design includes a workspace for the visual creation and manipulation of DSD instances by the user (instructor, tutor, or student). These visual data structures are active diagrams that can be manipulated by the user, entirely independent of any algorithm, or they can be used as input to an algorithm, which can then manipulate it.

Just as the user interacts with a manual simulation, we believe that our system needs to be interactive. To support the notion of real execution, we believe that it is necessary that the data structures reflect their current value, i.e. that they are active. Moreover, we believe that visual changes to a DSD made by the user should be reflected in the underlying value of the DSD to retain the notion of reality, and that the interaction should generate a sense of direct engagement [HHN85]. In this respect our work is close in spirit to that of Morphic systems [SM95], to that of

'live' (active) debugging systems [ULF97], and even closer to systems such as the Alternative Reality Kit (an interactive physics simulation environment) [Smith86].

We believe the design of this environment will afford all users a common medium for discussion and exploration. It is our vision that students will be able to take tangible algorithm and data structure artifacts with them and experiment with them in this common environment. In this sense our design approach also aligns with Cognitive Constructivism, Situated Action, and Sociocultural Constructivism learning theories [HDS02], discussed in Chapter 2.

To merge these philosophical design perspectives with our findings and make them concrete, we derived the following requirements from analysis of our observational study (and the tasks we identified in Table 5.1):

1. Provide a workspace to construct and manipulate data structures

2. Provide diagram manipulation capabilities that change the underlying data structure

3. Support highlighting and/or selecting of elements or sections of data structure diagrams

4. Support arbitrary annotation of data structure diagrams

5. Provide ability to clone the diagrams to provide a visual history of changes

6. Provide ability to save and retrieve data structure diagrams to save construction time

7. Allow the user to run the pseudocode algorithm on the data structure diagrams

8. Provide flexible execution controls to facilitate user control observed in tracing

9. Support arbitrary pseudocode notation to allow authors to control the conceptual level of pseudocode and use individual pseudocode style

10. Provide authoring support by creating a library of commonly used visual data structures, and by supporting construction, manipulation, and interaction with minimal coding.

In Chapter 1, we proposed that an AA system should support users tasks and needs, be used in appropriate learning contexts, and should be designed so that users can perceive and attend to them. Our design thus far has addressed the first and second objectives. Based on our appreciation of human perceptual limitations with respect to visualization [Ware99] and other applications of animation [Bartram01], we decided that our system should provide time-based animation to support design for perception. Furthermore, we decided that it should also facilitate investigation of the perceptual issues by providing a scripting interface for external programs to use the animation capabilities.

Finally, we determined that the architecture should also provide a common low-level library of geometric objects as the underlying level for the visual data structures, and these objects should facilitate manipulation and interaction. We then proceeded to design an initial version of SKA.

5.3   Design Overview

SKA is a combination of a visual data structure manipulation environment, an algorithm animation system, and visual data structure library. New visual data structure classes can be added to the SKA library by library developers. SKA runs on any Java platform (PC, Mac, Linux), and a projection device is recommended for use in lectures.

When the SKA environment is started, instances of a visual data structure (an interactive data structure diagram) can be created and manipulated on the canvas (seen in the left of Figure 5.1), independent of any algorithm. Manipulations include adding and removing elements, highlighting parts of the visual data structure, and performing other operations specific to that data structure class, such as deleting a subtree in a tree data structure. SKA supports arbitrary annotation of visual data structures and their elements; a pen-based system is highly

recommended for this purpose. When an element is moved, its annotation moves with it, and deleting an element removes its annotation.

A visual data structure may be cloned (copied) at any point in time, and the clone can be manipulated independently of the original. In Figure 5.1, the directed graphs are clones. Each visual data structure instance has a corresponding model data structure instance. Cloning creates a separate copy of the visual/model data structure pair, but with identical elements and configuration. This allows the user to create a visual history of changes. A visual data structure may also be saved to a file and read in later.
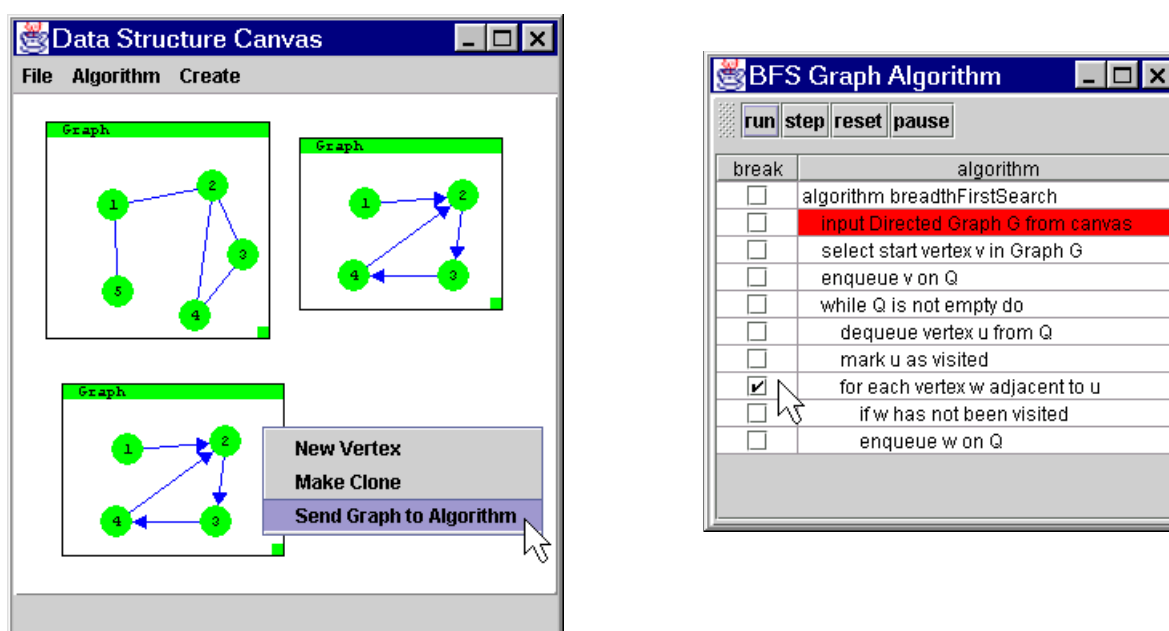
Figure 5.1:   The algorithm on the right is waiting for the user to select a directed graph from the canvas on the left.

The SKA environment can execute animated algorithms. The authoring process will be described in the next section. When invoked via a menu, a pseudocode algorithm appears in an execution panel, seen in the right of Figure 5.1. Through this panel, the user may run the algorithm, pause and resume, set and remove breakpoints before or during execution, or step

through the algorithm a line at a time. The user can also reset (terminate) an algorithm and run it again.

An algorithm can use, as input, a visual data structure on the canvas created by the user, as shown in Figure 5.1. This same visual data structure is also manipulated by the algorithm. Algorithms can also create visual data structure instances on the canvas. An algorithm can even request the selection of an element from a particular visual data structure instance. During a pause or break, a user may make clones of a data structure that is being used by (bound to a variable of) an algorithm, and manipulate either the clones or the original. Such a manipulation of the original might cause an error in the algorithm, but this can be used to illustrate erroneous states.

SKA supports recursion. Each recursive call instance is represented by a separate pseudocode subwindow, within the overall code window, as seen in Figure 5.2. The arguments can be shown for each call, and users can examine all calls and minimize them if desired, but can only control execution of the current call.
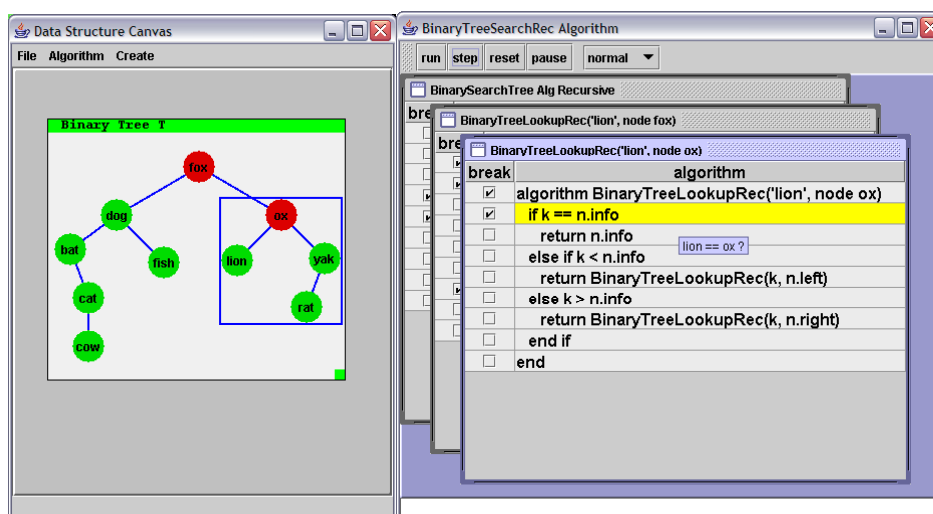

Figure 5.2:   A recursive binary search algorithm, showing the current call on the subtree highlighted in blue on the left. The current line, highlighted in yellow, is comparing the red node 'ox' to the search value 'lion', as also shown by the popup label.

Multiple algorithms can be run in parallel, even multiple instances of the same algorithm. However, the execution time of these algorithms cannot be relied on for comparative purposes because of the varying time each algorithm spends pausing, waiting on input, and performing animation.

5.4   Usage Scenarios

In this section, we describe a typical use of SKA by an instructor. Let us imagine that an instructor, Prof. Iani, is going to discuss the breadth-first search graph algorithm, and uses SKA in the following manner. Prior to the lecture, Prof. Iani prepares some sample input graphs by starting SKA, creating them on the canvas, and saving them. Alternatively, an algorithm could have been used to create sample input data structures, which can also be saved.
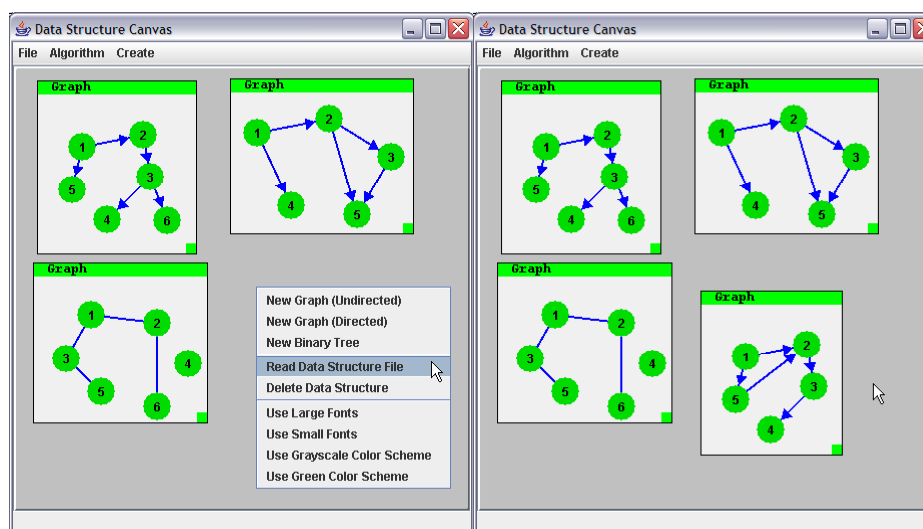
Figure 5.3a: The user reads a previously prepared directed graph, seen on the right.

In class, the instructor starts SKA, and creates a sample graph on the canvas, and reads in the saved graphs (Figure 5.3a). Prof. Iani discusses the properties of the sample graphs (acyclic, directed, undirected, DAG, tree, etc.), then opens a breadth-first graph search algorithm window, sets some breakpoints, and clicks the runs button to initiate execution. When execution reaches

the line that is highlighted in Figure 5.3b, the algorithm requests that a directed graph be input

from the canvas, and waits for the input.
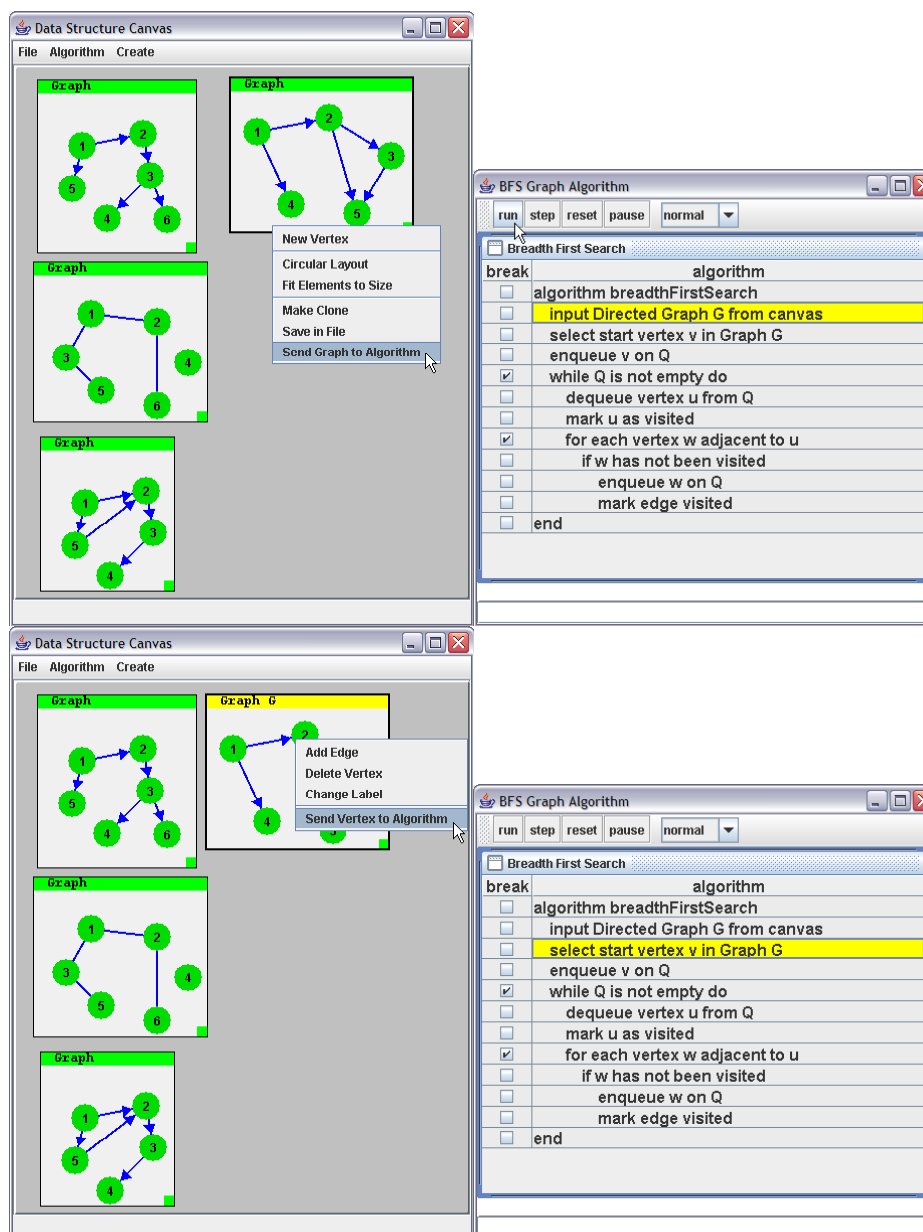
Figure 5.3b: The user initiates execution of an algorithm by clicking the run button (seen top
right). The algorithm (seen top right) requests input of a directed graph. The user
selects a directed graph (seen top left). The algorithm (seen bottom right) requests
input of a start vertex in that graph. The user selects a vertex (seen bottom left).

Prof. Iani selects an undirected graph, but the system will not allow the wrong type to be

sent as input. Prof. Iani then selects a directed graph, and the algorithm labels it with the bound

variable name. The algorithm requests the selection of a starting vertex from the selected graph,

and the instructor does this. At a breakpoint Prof. Iani makes a clone of the graph (Figure 5.3c).

He then asks students to predict the set of vertices to be visited by the algorithm, and highlights

them by dragging around them.
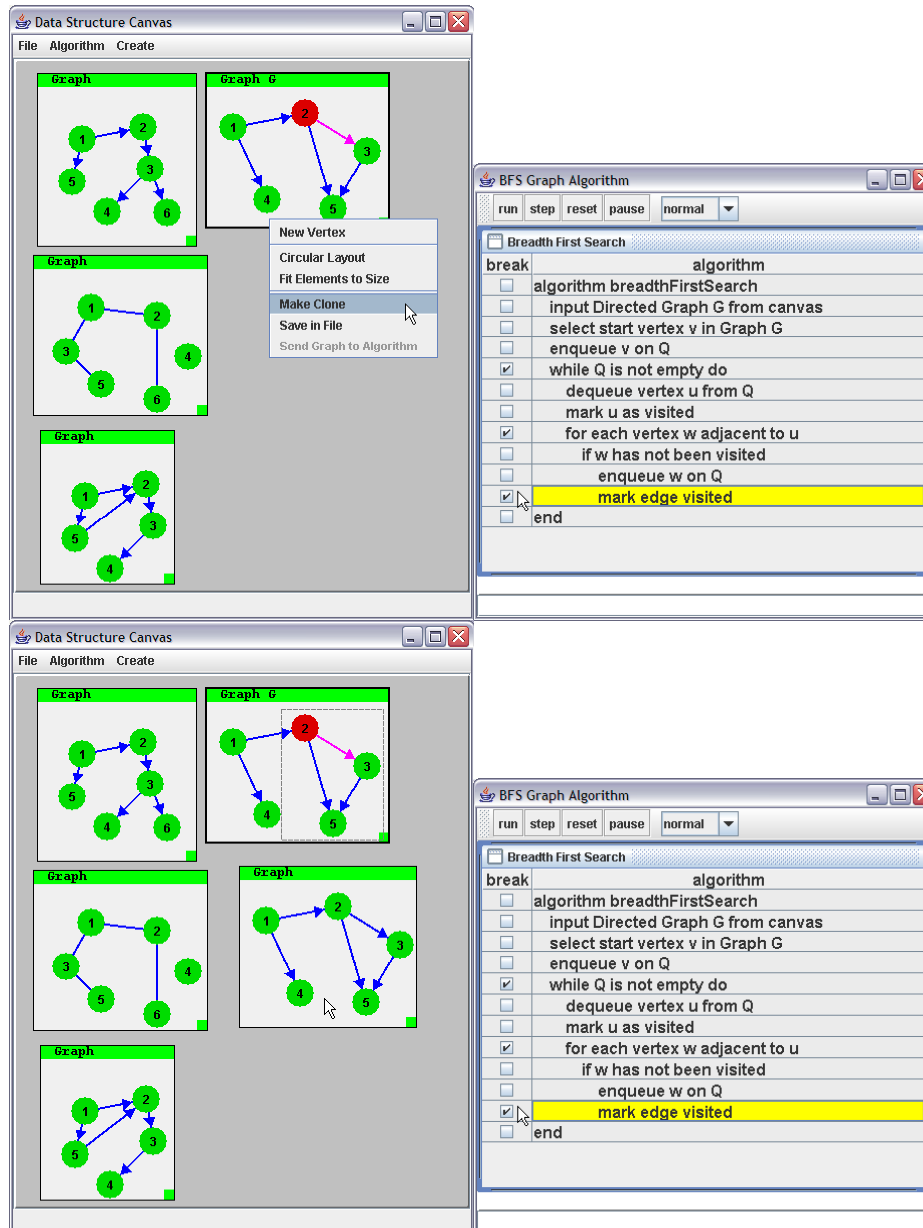
Figure 5.3c: After the algorithm runs to a breakpoint, the user creates a clone of the graph G
             being used by the algorithm, then highlights an area of the original graph.
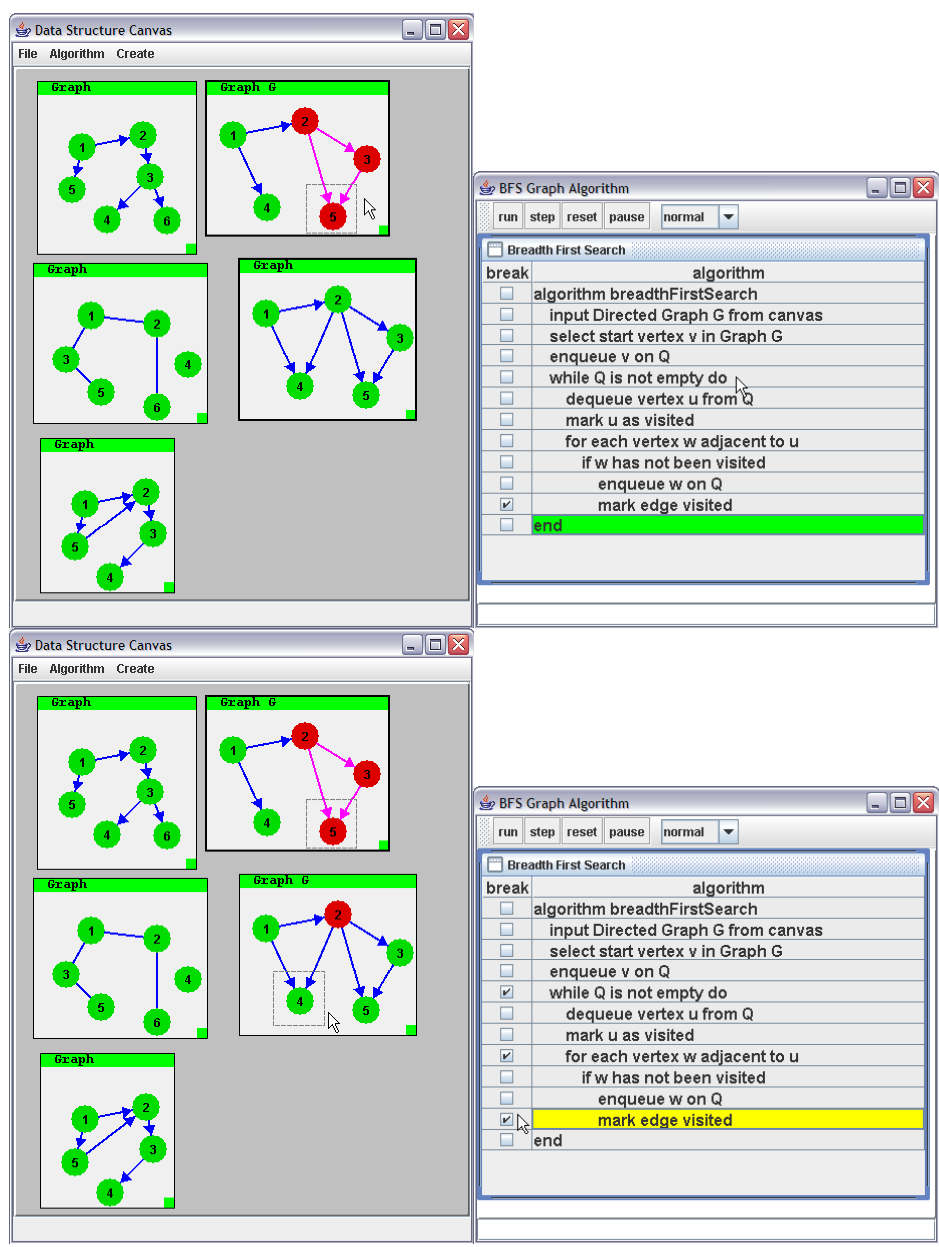
Figure 5.3d: After the algorithm runs to completion, the user adds an edge to the cloned graph (seen at top). The user runs the algorithm again on the modified clone of the graph, then selects a vertex at a breakpoint (seen at bottom).

Prof. Iani modifies the cloned graph by adding an edge to it, and asks students how that would affect the vertices visited by the algorithm (Figure 5.3d). He then runs the algorithm again, using the modified cloned graph. At a breakpoint, he asks students to predict the next

vertex to be visited by the algorithm, and selects the predicted vertex (by dragging around it) on the clone. He discusses the influence of the input data structure on the behavior of the algorithm.

Let us examine another possible scenario of use: illustrating the insertion of an element into an AVL tree, and the subsequent rebalancing. Prof. Iani constructs an AVL tree (Figure 5.4a), and inserts an element, 70, which makes the tree unbalanced (Figure 5.4b).
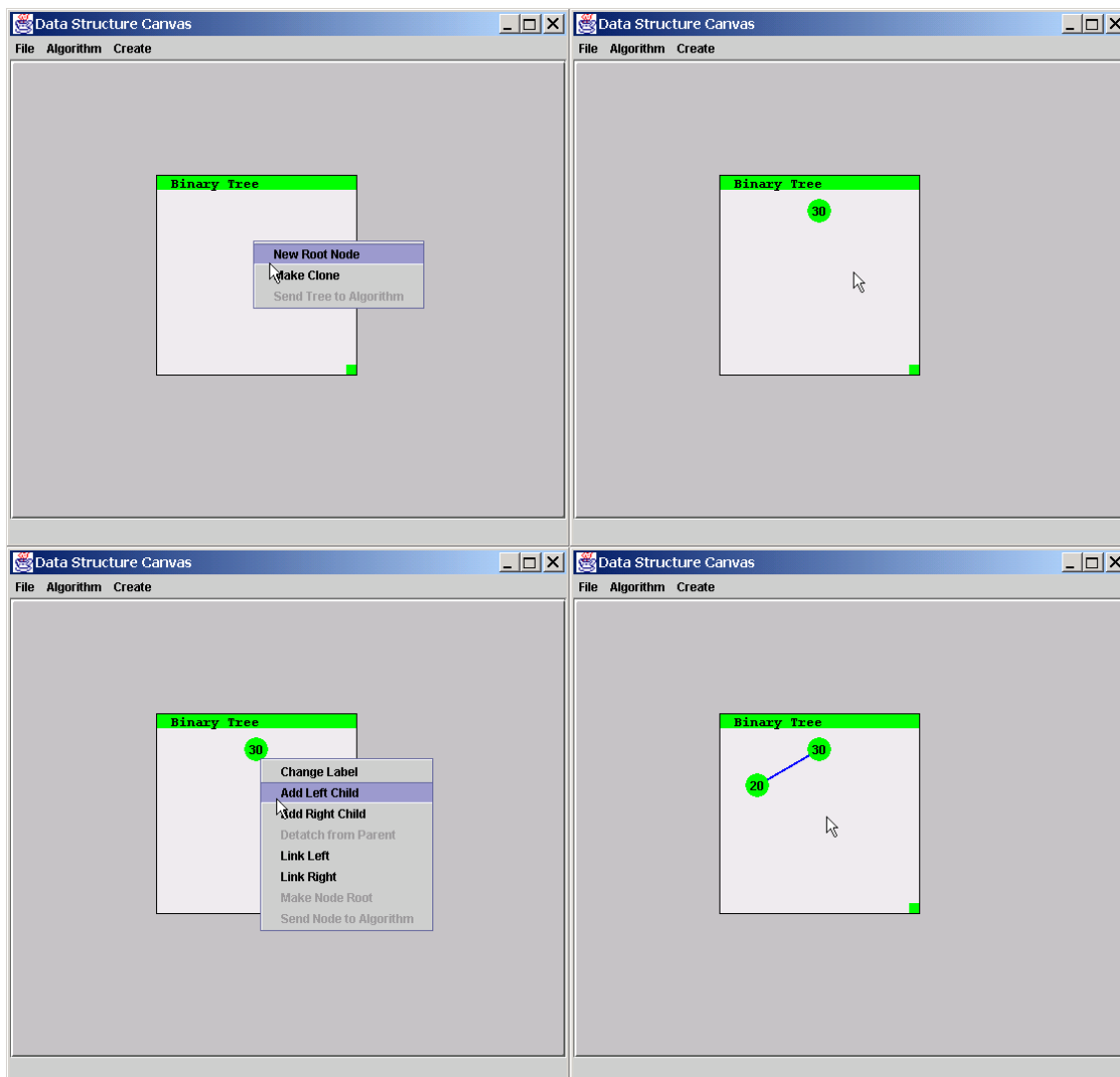
Figure 5.4a: The user is constructing an AVL tree.
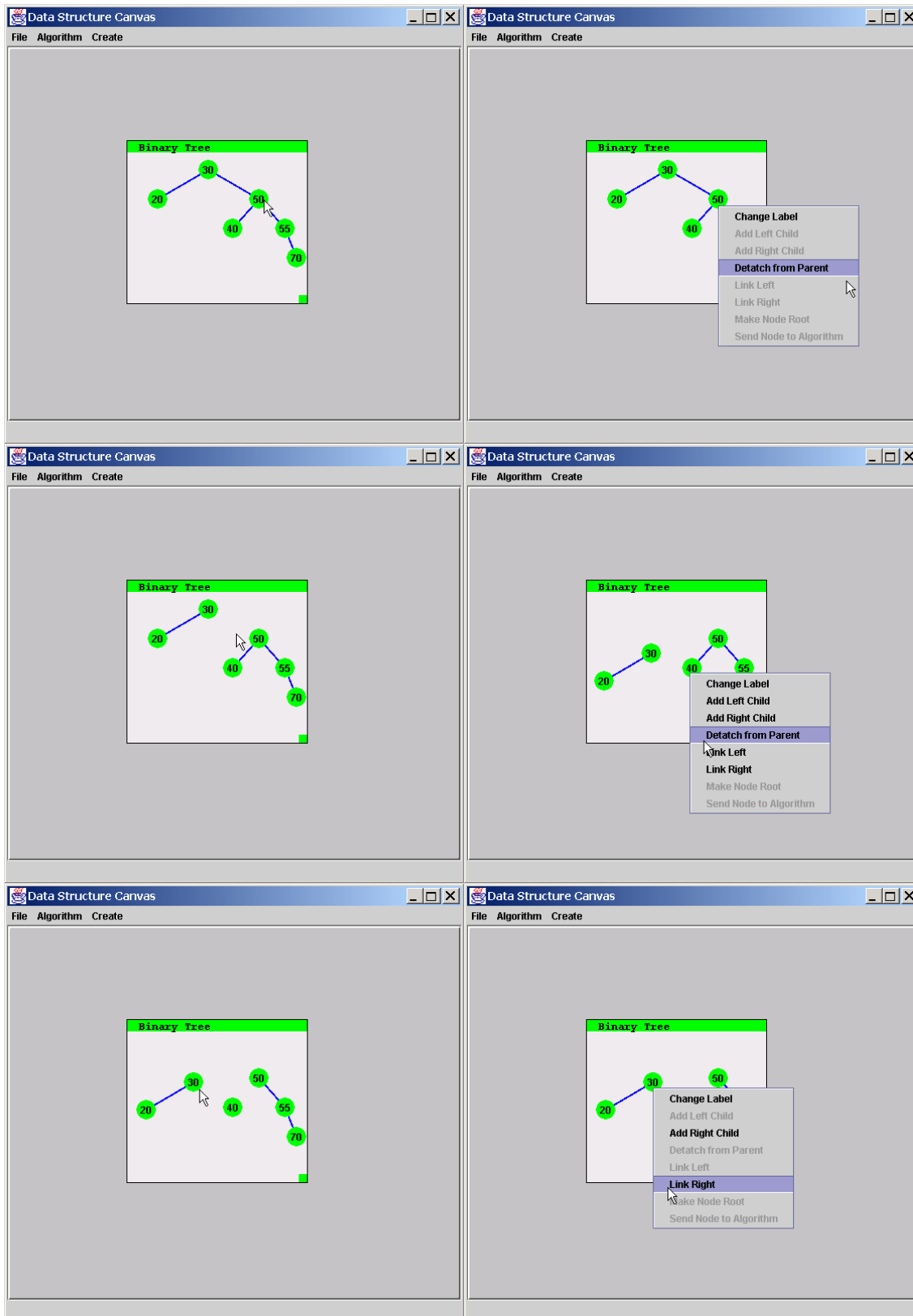
Figure 5.4b: Adding node 70 makes the AVL tree unbalanced, so the user starts a single rotation
by detaching the subtrees with root nodes 50 and 40, then linking nodes 30 and 40.

Prof. I then performs a single left rotation in a number of steps, shown in Figures 5.4b and 5.4c. In this rotation node 50 will become the root, with node 30 as its left child, and node 40 will become the right child of node 30.
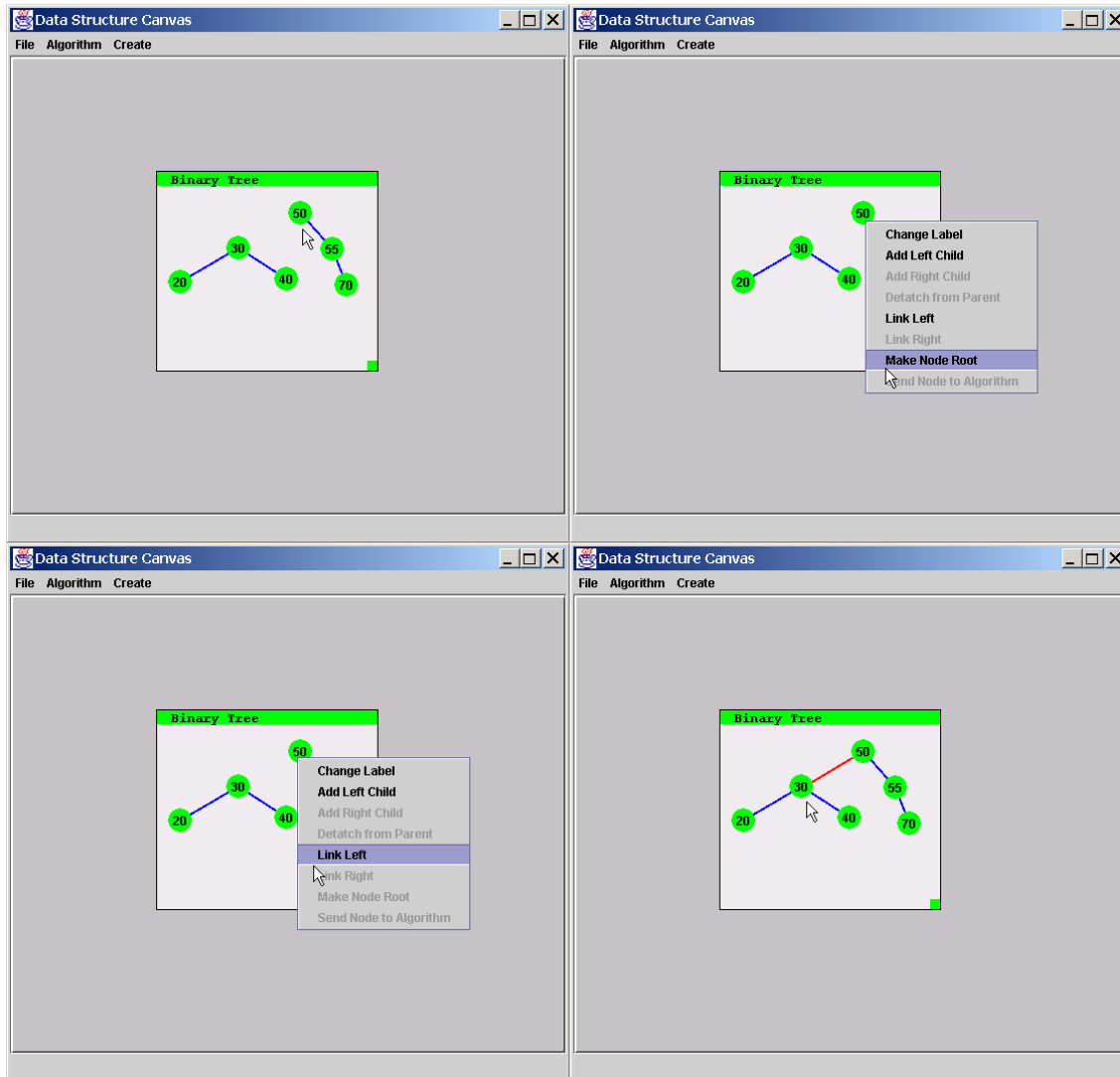
Figure 5.4c: The user makes node 50 the root, then attaches (links) node 30 to node 50. The rotation is complete.

The subtrees involved in the rotation are detached by right-clicking each of their root nodes (nodes 50 and 40) and selecting 'Detatch from Parent' (Figure 5.4b). Prof. Iani makes node 50 the root of the entire tree by right-clicking it and selecting 'Make Node Root', then

drags the subtrees into a desired layout (Figure 5.4c). Next, Prof. Iani makes node 30 the left

child of node 50 by right-clicking on node 50 and selecting 'Link Left' (at which point a

rubberband line appears), then clicking node 30. Finally, node 40 is made the right child of node

30 in a similar manner. The rotation is complete.

5.5   Authoring an Animation

How does a user, typically an instructor, prepare an animated pseudocode algorithm in

SKA? We walk through the process of authoring the BreadthFirstSearch algorithm example we

saw in the previous section. We start with the source code for the algorithm (henceforth called

the code), seen in Figure 5.5. The source code must be written in Java. The present

implementation language is Java because of its widespread use in computer science education,

and its support for user interface development. The instructor writes a pseudocode equivalent of

the code, shown in Figure 5.6. The level of detail or style of the pseudocode does not matter,

since the pseudocode is merely displayed, not executed.

```java
public BreadthFirstSearch() {
  DirectedGraph g = new DirectedGraph();
  int v, u, w;
  LinkedList Q = new LinkedList();
  // construct graph by some type of input, then…
  v = Integer.parseInt(
      JOptionPane.showInputDialog("Enter start vertex index"));
  Q.addLast(new Integer(v));
  while (!Q.isEmpty()) {
    u = ((Integer)Q.removeFirst()).intValue();
    g.getVertex(u).setVisited(true);
    System.out.println("Visted vertex "+u);
    for (w=0; w< g.nVertices(); w++) {
      if (g.edgeExists(u, w))
        if (!g.getVertex(w).isVisited()) {
          Q.addLast(new Integer(w));
          System.out.println("Visted edge(" +u +"," +v +")");
        }
    }
  }
```

Figure 5.5:   The original BreadthFirstSearch algorithm source code.

```
algorithm breadthFirstSearch
  input Directed Graph G from canvas
  select start vertex v in Graph G
  enqueue v on Q
  while Q is not empty do
    dequeue vertex u from Q
    mark u as visited
    for each vertex w adjacent to u
      if w has not been visited
        enqueue w on Q
        mark edge visited
```

Figure 5.6:   The pseudocode version of the BreadthFirstSearch algorithm.

```
public BreadthFirstSearch() {
  SkaDirectedGraph g = new SkaDirectedGraph();
  int v, u, w;
  LinkedList Q = new LinkedList();
  showAlgLine(1, "algorithm breadthFirstSearch");

  showAlgLine(2, "  input Directed Graph G from canvas");
  g = (SkaDirectedGraph) inputVizDS(SkaDirectedGraph.class);

  showAlgLine(3, "  select start vertex v in Graph G");
  v = (SkaVertex) inputVizElement(g, SkaVertex.class)

  showAlgLine(4, "  enqueue v on Q");
  Q.addLast(new Integer(v));
  showAlgLine(5, "  while Q is not empty do");
  while (!Q.isEmpty()) {
    showAlgLine(6, "    dequeue vertex u from Q");
    u = ((Integer)Q.removeFirst()).intValue();
    showAlgLine(7, "    mark u as visited");
    g.getVertex(u).setVisited(true);
    System.out.println("Visted vertex "+u);
    showAlgLine(8, "    for each vertex w adjacent to u");
    for (w=0; w< g.nVertices(); w++) {
      if (g.edgeExists(u, w)) {
        showAlgLine(9, "      if w has not been visited");
        if (!g.getVertex(w).isVisited()) {
          showAlgLine(10, "        enqueue w on Q");
          Q.addLast(new Integer(w));
          showAlgLine(11, "        mark edge visited");
          System.out.println("Visted edge(" +u +"," +v +")");
        }
      }
    }
  }
```

Figure 5.7:   The annotated code for BreadthFirstSearch, in which the DirectedGraph G is
replaced with a SkaDirectedGraph, and *showAlgLine*() calls are added to connect
the pseudocode lines with corresponding source code.

The first step in preparing an algorithm visualization is to replace the data structures we wish to display with the equivalent SKA data structures from the SKA data structure library. In this case we replace *DirectedGraph* with *SkaDirectedGraph* in Figure 5.7.

Next, we annotate the source code by placing *showAlgLine*() calls to mark which line of pseudocode corresponds to which line of code. If we want to get a data structure from the canvas, we replace the input routines with a canvas *inputVizDS*() call, as shown in Figure 5.7. Any extra animation calls (e.g. to highlight some element) are then added. The code is then compiled with the relevant SKA libraries.

5.6   SKA Object Scripting

We provide a scripting interface to the SKA animation engine so that external programs can use its animation capabilities. This has been used in the VizEval and SSEA testing environments for our perceptual experiments, as described in chapter 6. Although the syntax may be somewhat similar, one aspect of our scripting system that sets it apart from systems such as Samba [Stasko96] and JAWAA [Rodgers02] is that users can interact with the objects. The canvas on which the objects are displayed can be embedded in other Java applications and this interaction information can be made available to these applications. The applications can also create and manipulate objects on the canvas. This is how VizEval and SSEA use the SKA architecture.

Scripted objects can be defined in a manner similar to that in Samba. For example, a rectangle with an id of 0, an upper left coordinate of 0, 100, height 10, and rgb color (0,255,0) or green can be created using:

```
object 0 rectangle x 000 y 100 h 10 c 0 255 0
```

Actions such as movement and color changes can be performed on the objects. A move takes place over a specified time, and follows a path of n submoves, which are coordinate differences of the form (dx,dy). We can also use pacing techniques such as slowInOut for gradual movement. We can also group actions so they occur concurrently (in parallel), for example:

```
move object 0 slowInOut time 1500 path 2 200 100 250 100
startParallel
      move object 0 time 2500 path 2 200 100 250 100
      move object 1 time 4000 path 3 100 100 150 50 100 250
      move object 2 slowInOut time 5000 path 3 150 200 100 50 100 200
endParallel
```

where the first command moves object 0 by (200,100) to (200,200) then by (250,100) to (450,300) in 1500 milliseconds.

5.7 SKA Architecture

SKA uses three main components: the canvas, pseudocode windows, and actual programs represented by the pseudocode. A pseudocode window object displays pseudocode and contains the controls to control the execution of the underlying program (Figure 5.8). The canvas contains visual data structures. Common operations such as creating or deleting a visual data structure or reading one from a file are defined on the canvas.

Visual data structures are composed of visual elements such as nodes and edges. A visual container, which appears as a mini-window (also referred to as a tile or plate), contains the elements and defines the boundary of the data structure. This is necessary for data structures that may be disconnected visually, such as graphs. Interactions such as adding a node or highlighting part of a tree are defined on visual data structures. Operations common to all visual data structures such as mini-window operations are defined in a parent class.
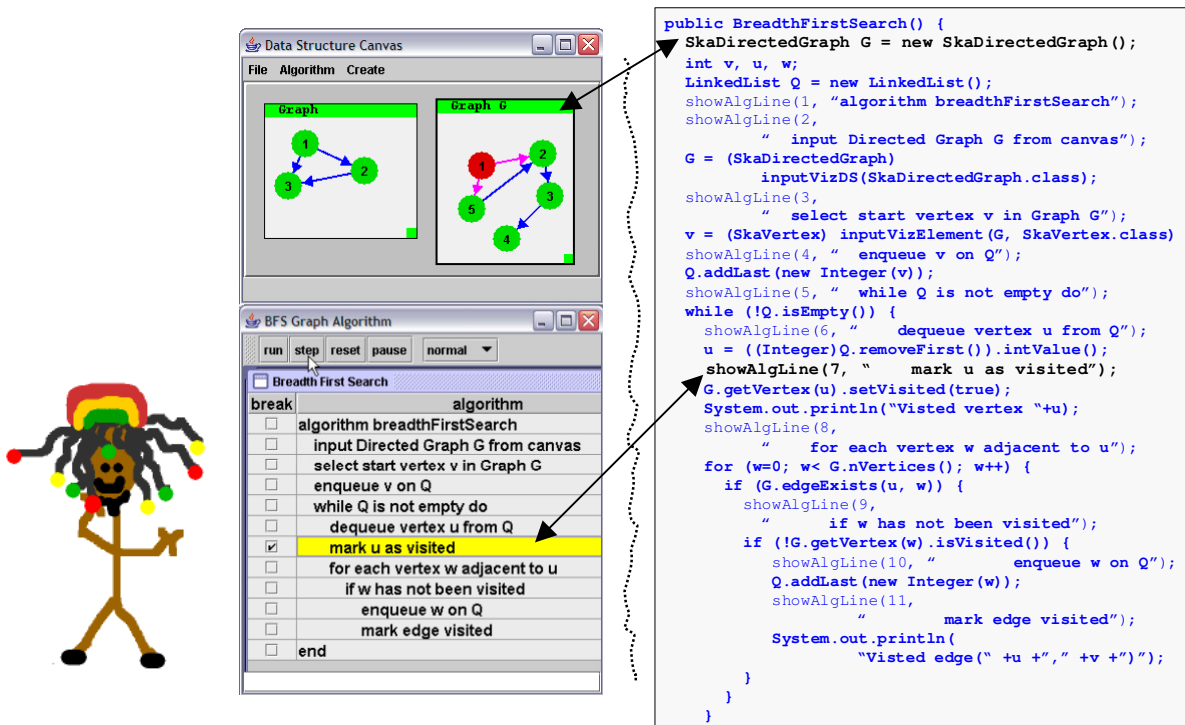
Figure 5.8: The SKA architecture in use; the user sees only the visual data structures and the pseudocode window to the left, not the actual program on the right. Changes made by the user to Graph G on the canvas on the left affect variable G in the program on the right, and vice versa. The showAlgLine calls in the program mark produce the pseudocode display. The user can control execution of the pseudocode on the left, which then controls execution of the program.

Visual data structures have corresponding model data structure instances. A change in one causes a change in the other. Similarly, visual elements have corresponding model elements. Visual elements use glyphs as their visual representation. Interactions such as changing the contents of a node or adding a child are defined on visual elements. Glyphs are basic objects such as rectangles, lines, strings, circles, etc., or composites of these basic objects. Operations such as movements and color changes can be performed on glyphs. The operations are designed to reduce the work the programmer needs to do to in specifying glyph layout and movement.

SKA uses the interesting event model [Brown98] when we annotate code, but also has an aspect of automatic visualization [LF98] as the visual data structures automatically reflect the

changes in the underlying data structure. The interactive and manipulative aspect of SKA does not fit into any of the established algorithm animation authoring models. It also supports the scripting model [Stasko96], but adds interaction.

5.8 Visual Data Structure Development

The SKA library currently contains visual data structures for undirected and directed graphs, binary trees, and linked lists. Developing a new visual data structure class is more difficult than authoring an animation, though we have attempted to provide a framework that simplifies their development, particularly for user interface operations. Essentially, one creates a subclass of the parent visual data structure class, VizDS, and designs any new visual element types that are needed as child classes of VizElement. One also creates a subclass of the parent model data structure class, SkaDS, and designs any new model element types that are needed as child classes of SkaElement. New interactions in either the new visual data structure or visual element classes must be designed as well. Creating a specialized subtype of an existing class, such as a new type of tree, can be accomplished by creating a subclass, and while this reuse can save time, it still requires design of any new aspects. Currently, library development is a task for serious developers, which requires one to understand the SKA architecture and the Java Swing API.

We expect to provide enough data structure classes to fulfill the most common needs in the near future, with more classes coming on stream over time, perhaps contributed by the wider computer science education community.

5.9   Conclusion and Future Directions

The capabilities of SKA have been specifically targeted to address many of the limitations instructors encounter in the classroom when teaching algorithms and data structures. Specifically, SKA supports and enhances the lengthy processes of data structure diagram creation and manipulation, and tracing algorithm execution on data structure diagrams. We believe that SKA will be a useful tool in the classroom, allowing instructors to better utilize time both in preparation and in the classroom.

Our initial implementation of SKA incorporates a number of features, some of which are unique, and others that have never been brought together. It facilitates simple mapping of programs to pseudocode algorithms and provides interactive algorithm execution control. Execution control is flexible and designed to emulate and support the manner in which instructors perform tracing tasks. It allows interactive data structure instance creation and semantic manipulation, and direct-manipulation of underlying data structures. It allows input of visual data structures to algorithms. SKA supports simple authoring for complex interaction and exploration. A basic library of interactive data structures is provided to simplify authoring, which we expect to expand in the future. SKA uses time-based animation to provide better temporal control. It provides pacing facilities and other animation techniques to aim to improve visual clarity. A scripting interface is provided that facilitates interaction, and use in other contexts such as perceptual experiments. The SKA architecture is extensible and designed for reuse.

We believe that our research is an important contribution to algorithm animation because it blends modern HCI design theory, Educational Technology theory, and Engineering Psychology with ideas from prior research in algorithm animation to create a system that takes

factors from all these fields into account in its design. Prior systems were not designed with the aim of supporting instructional or exploratory tasks in particular contexts.

We approach design trying to provide a virtual high-level execution environment to allow learners to explore, observe and verify or refute their notions about an algorithm. This notion may not be entirely novel, but we believe our implementation has unique attributes.

Reeves categorizes learning tools into those one can "learn from" and "cognitive tools" one can "learn with" [Reeves99]. He states that one requirement of effective "learn from" tools is that learners must "perceive and encode" the "messages encoded in media". Cognitive tools have been "intentionally adapted or developed to function as intellectual partners to enable and facilitate critical thinking and higher order learning." Examples include various applications, development and system tools from word processors to spreadsheets to computer languages to multimedia/ hypermedia construction software.

We describe algorithms and data structures as artifacts that are constructed, then observed and explored. Therefore, we believe that SKA can be described as a "cognitive tool". However, it has an aspect of the "learn from" software category when AAs are being observed during execution and exploration, and one of the goals of our empirical perceptual studies (described in Chapter 6) is to evaluate whether the "learners perceive and encode" the animations.

We plan to use the results of our perceptual studies to inform further work on the SKA perceptual animation libraries. We will then conduct a user evaluation study of SKA. Such a study would preferably involve a second video study to compare SKA use to manual tasks. Both of these projects are extensive and as thus will be pursued in future work. We plan to make SKA available when this phase is complete. We expect that some changes will be made to SKA when

we examine the results of our user studies. However, we believe that our research represents a

radical re-examination of algorithm animation system design.

CHAPTER 6

INVESTIGATING THE ROLE OF PERCEPTION IN ALGORITHM ANIMATION

6.0   Introduction and Motivation

Over the years, we have informally observed a number of problems that users

encounter with algorithm animations (AAs). Students in our algorithms and data

structures courses have complained that many AAs were hard to follow visually. We

watched instructors struggle to control AA speed and pacing. While AA implicitly

intends to take advantage of human perceptual ability (as do other forms of visualization),

we believe that some of the techniques used in AA may exceed our perceptual and

attentional capabilities, and may thus contribute to the mixed results of studies of AA

effectiveness [HDS02].

Algorithm animation requires the viewer to construct and maintain mental

mappings from the AA to an algorithmic process throughout the visualization. For

example, a value can be mapped to a bar or node, and a display may represent the current

state of an algorithm. Researchers have put forward possible explanations of what affects

the viewer mapping process [BCS99], but have not studied the perceptual and attentional

aspects of the phenomena directly.

The role of perception and attention in AA has not been formally considered, to

the best of our knowledge. We believe that this oversight needs to be addressed. We have

worked with our collaborators at Georgia Tech and the University of Georgia to conduct

a number of empirical studies to investigate the role of perception and attention in AAs

[DHKHR06] [RRKHDH06] [Rhodes06] [Hailston06] [Reed06]. We have designed

software to be used to conduct these studies [KRRH06] [RKHD06] [Reed06]

[Thomas04] [Ross04], which use SKA animation and interaction capabilities [Hamilton-

Taylor02a] [Hamilton-Taylor02b]. We have also examined perceptual and attentional

research and analyzed AAs in light of relevant findings.

6.1   Our Approach

As described in the previous section, we started with the informal observation that

AAs seemed hard to track perceptually. We speculated that viewers might not perceive

some of the AA actions. Furthermore, we assumed that if they did not see the changes

they would have difficulty mapping the graphical changes to the behavior of the

algorithm, and learning would suffer. We believe that perception and attention may affect

AA use at several levels:

1. Detection of changes/actions

2. Identification of the objects involved (and their spatial locations)

3. Monitoring or tracking actions if they continue beyond an initial change

4. Mapping between AA entities and algorithm entities

5. Mapping of AA actions to algorithm operations

6. Comprehension of algorithm animations

For brevity, we will refer to the first three levels as the perceptual level, although

we believe that perception, attention, and working memory all play important and

interconnected roles at these levels. Levels 4 and 5 will be referred to as the mapping

level, and level 6 as the comprehension level. We believe that each level depends to some

extent on lower levels, such that if a breakdown occurs at lower levels, it will affect the processing levels above. We speculate that mapping depends heavily on working memory as well as attention and perception. Finally, we believe that comprehension involves higher cognitive processes, but also relies on effective mapping.

Each level has tasks of varying degrees of complexity. Detection, identification and monitoring can vary in complexity, depending on the number of objects involved and the number and type(s) of actions being performed. The use of multiple views adds further complexity. Mapping a single visual entity to a simple variable is easier than mapping a visual data structure such as a linked list to the corresponding algorithm entities, or mapping a comparison of algorithm variables to its visual representation. One may comprehend part of an algorithm, but not another. Or one may comprehend the overall concept of an algorithm, but not the details of lower levels, or vice versa.

We propose that the design needs differ at each of these levels. At the perceptual level, we believe that perceptual techniques, whether they are animation techniques (such as cueing and motion type) or graphical object design techniques can impact performance, and the studies we describe support this. We believe that improving performance at the higher mapping and comprehension levels will require improved display design and conceptual design; we discuss related display design concepts in the following section, and we will return to this notion later in this chapter.

In order to establish the role of perception at each level, we worked with collaborators at Georgia Tech to conduct studies that would measure detection, identification of spatial locations of change, and monitoring of individual and group actions [DHKHR06] [Hailston06]. We also conducted initial studies that explored the

role of perception in the mapping of animation actions to algorithm operations, and its role in the comprehension of algorithm animations [RRKHDH06][Rhodes06]. This represents the beginning of a systematic investigation of the various factors involved at each level and of the relationships between the levels.

To carry out these studies, we had to develop software that facilitated the evaluation of AAs at these multiple levels, as existing software did not meet these specific needs. Two systems were developed. The VizEval suite [RKHD06] [Ross03] [Thomas04] is used to evaluate perceptual and attentional factors, primarily. SSEA (Software System for Evaluation of Animations) [Reed06][KRRH06] is intended to evaluate the comprehension, mapping and perceptual levels in a multi-window AA environment.

6.2   Related Work

6.2.1   Perception, Attention, and Memory

Humans have sophisticated perceptual capabilities; an overview can be found in [Coren04][Goldstein02]. These include various aspects of object recognition, as well as motion detection and processing capabilities. Bartram compared the capabilities and limitations of motion detection and color and shape change detection for applications such as industrial control panel monitoring. She found that motion changes were, with one exception, detected and identified at a much higher rate than color and shape changes [Bartram01]. The difference was more pronounced in the periphery of the user's field of view. Bartram found that a variety of motion changes measured at up to 52 degrees of visual angle were detected quite accurately ($< 4\%$ error rate). Color and shape changes, on the other hand, had respective detection error rates of 36% and 20% at 7 to 20 degrees,

and 64% each at 40 to 52 degrees of visual angle. The corresponding motion changes measured at up to 52 degrees of visual angle were identified at error rates from 0% to 1.8%, with the exception of the 'jumpy' linear motion (6.6% at 7 to 20 degrees). Corresponding color and shape changes were identified at error rates of 19% and 6.3% at 7 to 20 degrees, and 15% and 4.8% respectively at 40 to 52 degrees visual angle. Bennett found that using luminance changes to simulate motion was more effective than use of color changes to mimic animation in displays [Bennett97].

Attention and memory are also relevant to animation processing, and are closely associated with perception [Pashler98, Baddeley93]. A number of studies of visual search and attention have been performed in other domains [Davis06b] [Davis04] [Geisler95] [Palmer00] [Shaw80] [Treisman93], with varying results and interpretations. Attention can be described as the ability to concentrate on one or more stimuli (focused attention versus divided attention) to the exclusion of others. Animations that involve multiple changing elements or windows require divided attention. Though people can simultaneously attend to non-contiguous locations [Kramer95], divided attention is generally harder to maintain if the sources are not in close proximity [WC97]. This would typically involve spatial proximity, e.g. for multiple windows, though it is possible to cause confusion by crowding. Other forms of proximity (e.g. color, shape) can assist in maintaining divided attention, particularly where close spatial proximity is not possible or optimal [WC97].

Selective attention is directed to a stimulus that attracts attention [Pashler98]. It may be directed to one of several competing stimuli or may be involuntarily drawn to a source because of the type of stimulus. Overuse of a stimulus to attract selective attention

can lead to loss of sensitivity, or confusion in some cases [WC97]. For example, if many sources keep flashing, flashing will become a less effective stimulus.

Working, or short-term memory [Baddeley93] has a very limited capacity and duration that decreases with cognitive load. A stream of events that requires items to be stored in working memory, such as typically seen in an AA, is said to involve dynamic working memory [WC97]. Wickens points out that new information can interfere with the old if it is similar in meaning, or if working memory is updated too rapidly [WC97].

However, Baddeley posits that there are separate audio and visual short-term memory stores, and thus one can process more information if tasks involve a combination of audio and visual resources than if all the demand is on one type of resource. This position is supported by a number of studies of multimedia by Mayer [MM98].

6.2.2 Visualization

Improved design for perception has been shown to improve the effectiveness of systems in a number of domains. Bartram used perceptual principles to design tests for supervisory control systems: she was able to identify and classify a number of techniques to improve critical information display [BWC01]. Faraday and Sutcliffe [FS97] used eye-tracking to monitor overt shifts of attention to various aspects of a multimedia medical lesson. They were able to show significant improvement on a post-test using a redesigned lesson, in which they used a combination of perceptual techniques to direct attention to aspects of the lesson that the viewers missed. However, they did not attempt to use each technique in isolation, and thus did not assess the contributions of the individual factors.

McCrickard compared different schemes for presenting peripheral information using animation, with the goal of maintaining awareness with minimal distraction from the main task, and found that some animation techniques were less distracting and more effective than others [MSC01].

Bartram also showed that attention and cognitive load can affect identification and detection even when perceptual techniques are used that are effective in the absence of those cognitive tasks [Bartram01]. She asked subjects to perform a focused attention task (text editing, Tetris or Solitaire) while detecting changing icons. The cues and motion types that were effective in isolation were less effective as the cognitive load of the accompanying task increased. Subjects found the various types of changing icons disruptive, some more than others. In increasing order of disruptiveness, they were blinking, zooming (in-place growth and shrinking), and linear movement. However, Bartram was not trying to measure the influence of these techniques on comprehension or mapping.

6.2.3 Visual Display Design

A number of display design concepts from human factors engineering may be pertinent to AA design. We give a brief overview of these concepts.

In display design, one important analytical concept is that the quality of the relationships between the individual, the interface and the domain determine HCI quality, and these relationships are mediated by the correspondence and coherence of the interface [WR88]. Correspondence is the mapping between a problem domain and the system representation. This determines what information to include in the interface in

order for the intended tasks to be feasible. Coherence is determined by the mapping between the system and the user. It is a holistic measure of how well the interface conveys pertinent information to the user and how effectively the user can interact with the interface to carry out intended tasks.

The attention-based approach to display design [Bennett97] focuses on how the design of interface elements, their layout, and the encoding mechanisms used affect object perception and visual attention. The goal of the attention-based approach is to design displays that help to reduce the cognitive load of the interface and its tasks. Attention is viewed as a continuum from focused tasks that require attention to specific elements to *integration tasks* that require distribution of attention across multiple features that must be considered together. Examples of integration tasks include comparing objects visually (e.g. their color or size), or comparing them using other information (e.g. is x > y, where one has to look up the values of x and y in a table).

A number of graphical dimensions can be used for coding information in a display. These include (but are not limited to) object shape, size, orientation, color, texture, and motion [Ware99]. In some cases, these graphical dimensions can be processed rapidly by the pre-attentive visual system [Healey00], and can thus be efficiently analyzed and effectively used to code information in displays. There are a large number of coding guidelines and findings that could be useful in AA design. Ware provides a good overview of these guidelines in relation to information visualization [Ware99]. For example, it is recommended that the number of colors used for coding in a visualization display should be limited to around six to avoid visual confusion [Stokes88].

6.3 The Experiments

The evaluation of the perceptual and attentional factors in AA turned out to be a fairly large-scale project, which is ongoing. Two research groups collaborated on the project, one in Computer Science at the University of Georgia, headed by Eileen Kraemer, and the other in Engineering Psychology at Georgia Tech, headed by Elizabeth Davis. We describe three sets of experiments that these groups have participated in or conducted.

The first set of experiments [DHKHR06] [Hailston06] focused on the *detection* and *localization* of actions in a typical AA sorting display, and the effect of using cueing techniques and labeling. *Detection* is defined as noticing a change. *Localization* is defined as identifying the spatial location of a change (i.e. where did it change?), as distinguished from identification, in which one identifies the object that changed by some property (i.e. which object changed?). Specifically, we were interested in measuring and distinguishing between the ability of the viewer to attend to actions on individual objects and groups of objects using cueing and labeling.

The second set of experiments [RRKHDH06] focused on measuring the impact of using the effective cueing techniques identified in the first set of experiments to measure performance on a set of questions within the context of a multi-view algorithm animation. We also measure detection and identification of actions on objects, and perform initial measures of mapping AA changes to algorithm variable changes.

The third and final set of experiments [Rhodes06] evaluated the influence of the popup question evaluation methodology on comprehension performance. It also compared performance on a revised color design to that in the second study.

6.4   VizEval Suite

To support the running of the first set of experiments and a planned series of similar experiments, we developed the VizEval suite [RKHD06] [Thomas04] [Ross04]. The VizEval suite allows the experimenter to develop an animation experiment, to deploy that experiment and to collect and organize the output. Automation is important because of the size and complexity of these experiments, which may consist of hundreds of trials and complex ordering within a trial or across test participants. The experiment consists of a number of blocks. Each block contains some number of trials. In each trial, the participant views a short animation and is then asked a series of questions about what she saw and understood.

Figure 6.1 depicts the system architecture of the VizEval Suite, which consists of SKA (the Support Kit for Animation)[Hamilton-Taylor02a], and TestCreator, FileCreator, TestTaker, and Utility modules.

SKA is a combination of a visual data structure library, a visual data structure diagram manipulation environment, and an algorithm animation system, all designed for use in an instructional setting. In the context of the VizEval suite, SKA serves as the graphics and animation engine.   As a result, we are able to directly apply lessons learned in the VizEval environment to continuing refinement of SKA.
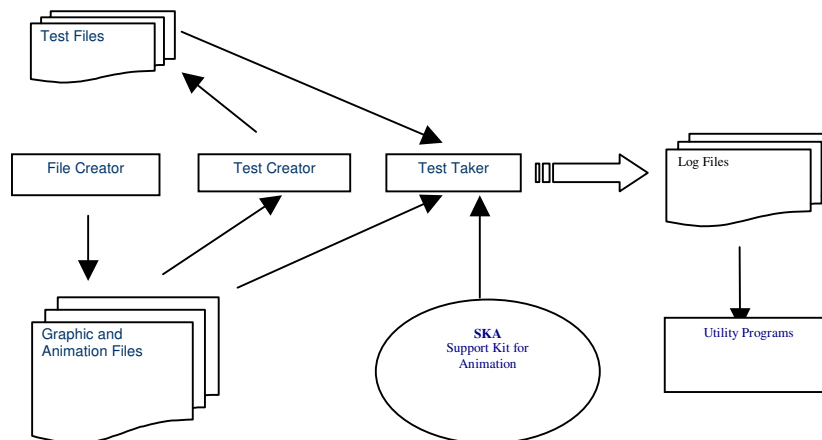
Figure 6.1: The VizEval Architecture, showing the dependencies of various subsystems and entities. TestCreator uses graphics and animation files and generates test files. TestTaker runs experiments using the test files and SKA, and output data to log files.

Graphical objects and their animations are specified in graphics and animation files, respectively. These are simple text files that are processed by SKA at run-time. While such files may be created manually using a text editor, it is desirable in the case of large experiments to use FileCreator to automate this process.

FileCreator is currently a custom application that specifies the graphical objects for the particular type of display used in this experiment; however, we plan to generalize it. It allows the user to create a single file in which the objects (bars, in this case) to cue and the objects to animate can be specified or randomly selected. When generation of the set of all files is requested, all possible combinations are created for each of the group of objects.

TestCreator, shown in Figure 6.2, facilitates the design and generation of experiment test files. It leads the experiment designer step-by-step through the process of specifying each block, the trials within each block, and the graphics, animations, and questions associated with each trial. TestCreator supports five different types of

questions: mouse (requires user interaction with a mouse), keyboard (requires interaction through the keyboard), Multiple Choice, N-Point (e.g., Likert Scale), and Yes/No (True/False). Additional customized question types may be created, by extending the "Question" class.

The experiment file generated by TestCreator contains or specifies all the information needed to run the experiment, including user instructions, questions, start method (enter key, mouse click, space bar), attractor (countdown timer, etc.), graphics and animation files to be used, as well as various timing and flow of control parameters. The application is also designed to generate random values within a range for certain parameters, thus saving the users from having to generate values for these. A batch processor module allows the experiment designer to specify parameters that repeat across blocks, trials or questions. The experiment file that is generated through the use of the batch processor can then be opened and customized. This feature is especially useful when the experiment consists of a large number of blocks or trials.
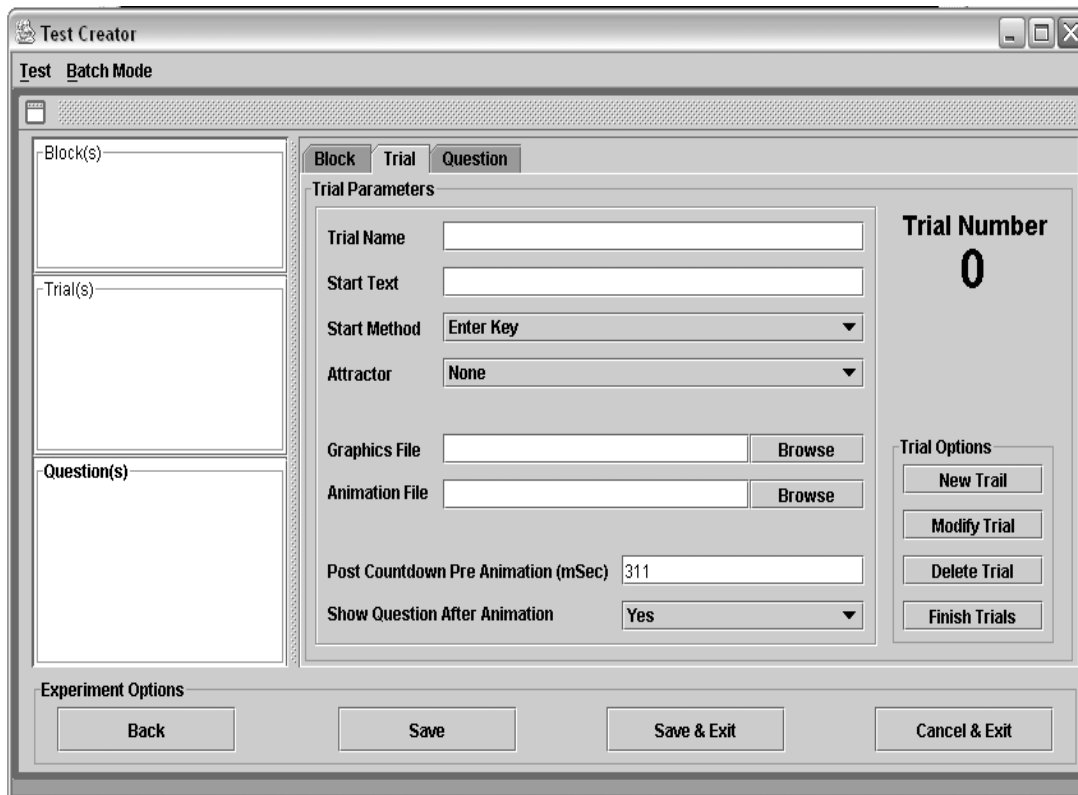
Figure 6.2:   TestCreator GUI. Using TestCreator, researchers specify an experiment to be performed.

TestTaker, seen in Figure 6.3, is the execution environment for the experiments. It keeps track of the user data, the date and time at which the experiment was conducted, and other metrics such as height and width of the screen, distance of the eyes from the screen and directory into which the log files are written.   Through TestTaker, animations and associated questions are displayed. User responses and other needed information are written into a log file. The utility programs parse these log files to extract and analyze the date.

Figure 6.3 depicts a sample user session. In this case eight bars are shown. One or two bars have been cued (by flashing) and one or two bars have changed height. The user is then asked a series of questions to determine if they noticed that something changed (detection) and can identify the location of the object that changed (localization).
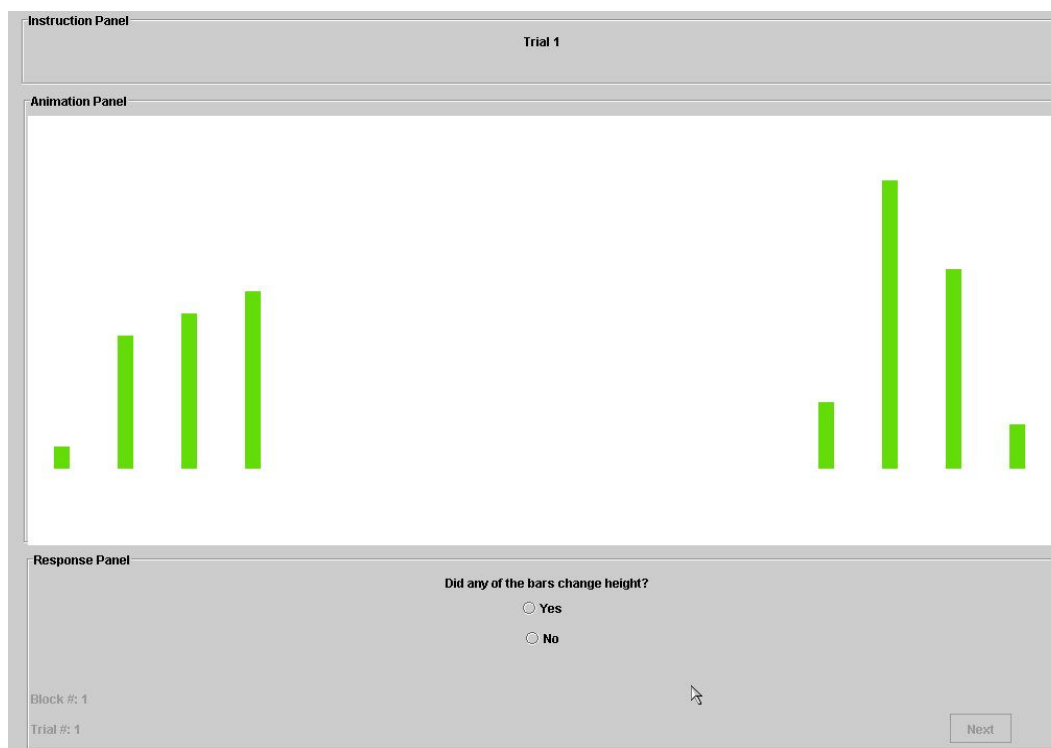
Figure 6.3:   The TestTaker interface during a session.

The VizEval suite has some features that are similar to those in E-prime [Eprime], a commercial software package for conducting psychology experiments. E-prime has extensive functionality, including visual and GUI creation of timed trials of various types, blocks, and sessions, the ability to collect data from the keyboard or mouse, and data logging and data output in various forms. It also has a scripting language, E-Basic, that appears to be similar to Microsoft Visual Basic. One can create displays similar to that in Figure 3, but to generate the combinations of display actions that we used in the first study would require writing scripts in E-Basic to generate them. Also, displays involving more general AA actions would require an experimenter to observe the AA and painstakingly copy its behavior using the E-prime scene construction GUI. Alternatively, they would have to try to rewrite the AA in E-Basic, which has no algorithm animation system or animation engine.

6.5   First Study: Evaluating Perception

Our initial focus was to measure the perception (*detection* and *localization*) of some basic actions in algorithm animations.

One of our initial speculations was that viewers were not detecting and locating actions in the periphery (or outside the area of immediate focus) while observing another action, perhaps because they were not looking in the required vicinity at the time. If this were true, we wanted to know if cueing could help to address the problem, as it did in [Bartram01].

We believe that localizing the actions was particularly important, as it would give a point of reference for an object in the AA display, which we thought was essential for the observer to form a meaning for the action.

Working with the Davis group at Georgia Tech, we investigated these speculations in the context of what users perceive in a 'conventional' bars display typically used in sorting algorithms. This study [DHKHR06] [Hailston06] sought answers to fundamental questions such as: (1) Can users detect that something has changed? (2) Can they accurately locate where a change occurred? (3) Does the number (set-size) of display elements (for example, bars) affect performance? (4) Can they simultaneously divide attention over two widely separated locations while monitoring activity in both regions? (5) Does cueing where a change may occur necessarily help performance? and (6) Does labeling locations help performance?

Many AA sorting displays use bar arrays (which look like bar graphs) where one or more bars may change their properties (e.g., bar height) to indicate a change in value. We used bar displays (as in Figure 6.3) in which (a) one bar changes its height, (b) two

bars changed simultaneously, or (c) no change occurs. For each trial, the observer must

detect if a bar changes and then identify the location(s) of the bar or bars that have

changed by clicking on the appropriate bars. We also employed catch trials, in which no

change occurs.

Each change could be cued (the bar flashed twice), or uncued. When two bars

changed, they were on opposite sides of the center of display, and one or both could be

cued, as seen in Table 6.1. The display contained 4, 8, or 16 bars. The intention was to

see if working memory limitations would affect detection and localization outcomes. The

bars were arranged so that they were on either side of central area of the display,

extending into the periphery of the display, as seen in Figure 6.3. The changes could

occur in the periphery only, or in varied locations (near central area, far periphery, or

intermediate locations). Participants were asked to focus on a countdown timer in the

center of the screen until a trial began.

Table 6.1:   Experimental conditions for the first study.

Within-Subject Factors for both Labeled and Unlabeled Blocks of Trials				
Set Size		Number of Bars That Change Height		
4	**Cues**	0	1	2
	1 Cue	X	X	X
	2 Cues	X	X	X
8	**Cues**	0	1	2
	1 Cue	X	X	X
	2 Cues	X	X	X
16	**Cues**	0	1	2
	1 Cue	X	X	X
	2 Cues	X	X	X

Thirty-six Georgia Institute of Technology students participated in this study. All

participants had 20/20 vision after any necessary refractive correction and all received

extra-credit in their courses for participating. Two Dell Dimension desktop computers

with Sony Trinitron 19" color monitors were used. Participants interacted with TestTaker of the VizEval suite, which managed the presentation of questions, graphics and animations, and logged participant responses.

6.5.1 Results and Discussion

This study yielded some interesting results. We found that factors that enhance localization performance often hinder detection, and vice versa.

Detection is significantly better if two bars change than if one bar changes $(F(1,33)=62.96, p<0.001)$, which implies that users can divide their visual attention across the display. This might be partially due to probability summation, as users have two chances to detect a change when two bars change.

Detection is worse in the periphery for the smallest set size of 4 when bars are only presented in far peripheral fixed locations. The difference between a far peripheral display with 16 bars and one with 8 or 4 bars is that some of the innermost bars are removed. The interbar spacing remains the same. Therefore, as we reduce the number of bars, the bars become more concentrated in the periphery. This points to the importance of studies such as this, as intuition would lead one to believe that smaller sets are always better in a learning context. However, when bars are presented in varied locations, performance was equivalent across all set sizes (4, 8, and 16 bars). Thus, detection performance was not hampered by working memory.

Cueing did not have a significant impact on detection. We can posit that, in this case, a good animation change technique was sufficient on its own for detection, and thus cueing did not offer much additional help.

Localization is significantly worse when there are two changes ($F(1,33)=89.18$, $p<0.001$). Perhaps localization performance suffers because while the observer is making responses about the first location, information is fading from working memory about the second location. Localization rates fall as set size increases, regardless of object position on the screen (whether in the periphery or not).

Cueing affected localization performance positively. Localization performance is much worse for an uncued change than a cued change ($F(1,33)=64.98,p<.001$)), especially when 8 or 16 bars are presented ($F(1,33)=11.563,p<.002$)). The set-size effect is larger when 2 bars change. For localization, it seems that that cues give an additional opportunity prior to the animation to note the localation of an object, and it becomes more important as set size grows. We can infer this from the data, which shows that when two bars change height and one change is cued and the other is uncued, the difference in average detection rates is more than 10% higher for the cued change, for set sizes of 8 and 16. When only 1 bar changes and it is cued, there is no set size effect to speak of. This again suggests that working memory limitations may negatively impact localization performance when there are simultaneous changes, an important note to AA designers. Generally, localization rates vary from below 85% for two changes and 16 bars to over 95% for smaller set sizes or for one change.

Labels apparently help to 'mark the spot', as they can improve localization. This is particularly notable for two changes, which is consistent with the notion that it is harder to remember two visual locations than one. Even with labels and cueing, localization performance for two changes decreases with set size from about 95% for 4 bars to approximately 85% for 16 bars. This indicates that animation designers should

avoid simultaneous changes with large set sizes until this issue can be addressed, unless they can live with less than perfect localization rates.

6.6   Second Study: Evaluating Multiple Levels

In the second set of experiments [RRKHDH06], we set out to simultaneously evaluate the effects of the animation techniques at three levels: perception (identification/detection) of changes, mapping of changes to algorithm operations, and comprehension of the algorithm depicted in the animation. In performing this multi-level evaluation, we wanted to answer the following questions:

1. Will the animation techniques we used in the first set of experiments result in improved performance on comprehension tasks?
2. Can the user map AA actions to algorithm operations? Do perceptual techniques help in this regard?
3. What are the relationships between performance on perceptual, mapping, and comprehension tasks as we vary animation techniques?

We investigated these issues in an experiment that uses a quicksort sorting animation in the SSEA testing environment [KRRH06] [Reed06], which we describe below. The study evaluates the effect of different cueing techniques and different types of change animations on user performance at various levels. Apart from comprehension, the study also evaluates the user's ability to perceive changes in the animation, and to relate (map) these changes in the animation to changes in the algorithm. Specifically, we measure whether the viewer perceived changes in the main bar display (in Figure 6.4) and was able to map these bar changes to the values they represent, and in some cases to

the variables they represent (e.g. firstHigh in Figure 6.4). Mapping bars and their changes to variables can be considered a simple integration task.

Perceptual effects and mapping are evaluated through responses to popup questions (questions that appear during playback of the animation, where the animation pauses its run until an answer is provided). By effects on comprehension we refer to the ability of viewers to understand an algorithm, as measured by the number of correct answers to questions about properties of the algorithm. The users' responses to such "traditional" questions measure the comprehension effects of the algorithm animation. For traditional questions, the "homework scenario" evaluation methodology we introduced in [KST00] was employed (also referred to as the "assignment scenario" [HDS02]), in which students can use the animation as a resource to solve the problems, and have no time limit to do so.

We vary two animation attributes. We label the first attribute as "cueing" or "no-cueing". The "cueing" depiction flashes the compared bars three times when the comparison event occurs. "No-cueing" simply does not flash. The second attribute is labeled as "grow" or "move". In the "grow" depiction of an exchange of elements, the bar representing the smaller element grows to the height of the larger element while the larger element simultaneously shrinks to the height of the smaller element. In the "move" depiction of an exchange of values, the two bars representing the data elements being exchanged are seen to move in an arcing motion, each one moving to the location of the other. Four animations were created: cueing with move, cueing with grow, no cueing with move, and no cueing with grow.

6.6.1 Method

6.6.1.1 Participants

Participants in the experiment were 59 volunteers drawn from the undergraduate computer science students registered at the University of Georgia in the fall semester of 2005 and spring semester of 2006. Participants received a five-dollar payment in return for their participation, which required approximately one hour.

6.6.1.2 Apparatus

Studies were conducted on Dell Dimension desktop computers with high-resolution 17-inch LCD flat-panel color monitors.

The empirical study we conducted used an algorithm animation framework called SSEA, a System to Study Effectiveness of Animations [KRRH06][Reed06]. SSEA integrates a visualization interface with a question panel, pop-up questions and a monitor. The monitor automates data collection of user interactions and events. A SKA [Hamilton-Taylor02a] module manages the display of the animation and adjusts the visualization to any users interactions. SSEA also supports our "homework scenario" evaluation methodology [KST00] via the question panel.

The main SSEA user interface is seen in Figure 6.4. It includes an animation area (A), a pseudocode display (B), animation controls (C), and a question area (D). The animation area and code area display the graphical representation of the underlying algorithm, and are synchronized by SKA. A high resolution monitor is required to view the multiple areas of SSEA in their entirety. Our empirical studies were conducted on such monitors.
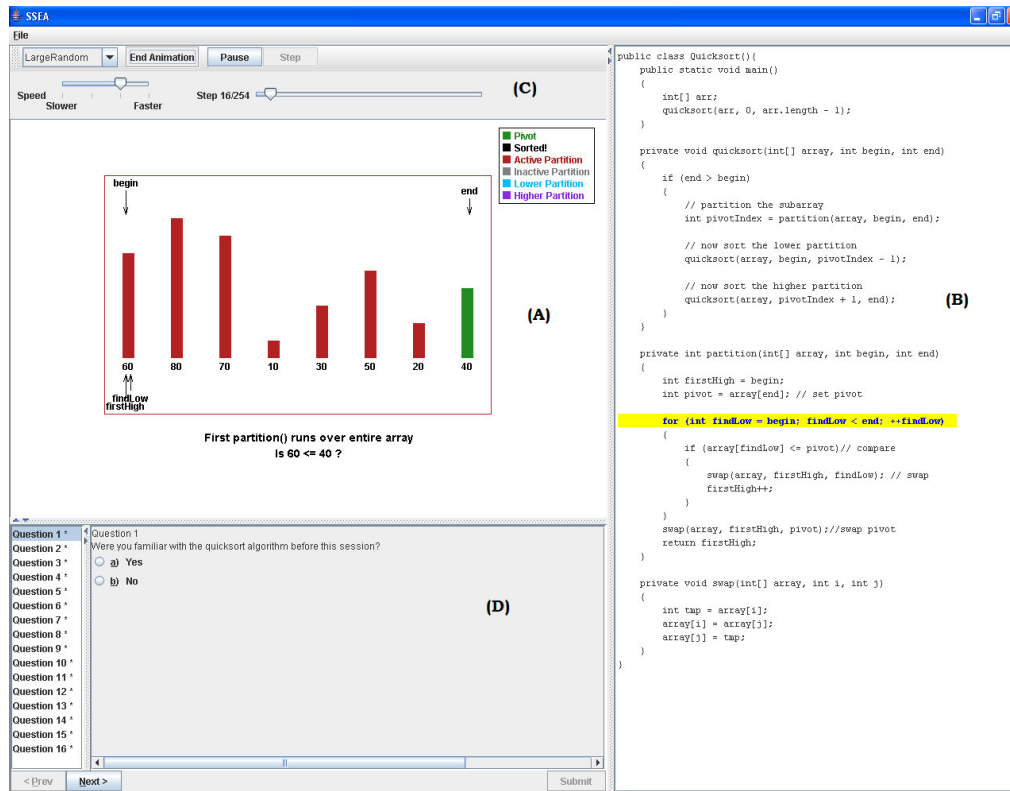
Figure 6.4: The SSEA Environment. (A) Animation area where visualization of underlying algorithm is displayed. (B) Pseudocode display: currently executing code is highlighted. (C) Animation controls: data set, start/stop/step/pause, animation speed and starting point. (D) Question list and questions.

The user can control the animated display. Specifically, the user can select the data set, control the speed, start, stop, pause and step the animation. In addition the user may move the slider seen in section C of Figure 6.4 to select a point in animation time at which to restart the animation.

The questions seen in area D of Figure 6.4 may be answered by the user at any time. An asterisk next to a question number in the list indicates that the question is still unanswered. The user may also return to a question and change his answer. After all questions have been answered a "submit" button becomes enabled. When the user clicks "submit" the session is complete. Both the intermediate choices and the final answers for

all questions are recorded in the log. In our studies we used this question mode to solicit answers to "traditional" questions.

The experiment designer may also specify "popup" questions, as seen in Figure 6.5. These questions are associated with the animation of the algorithm on a particular data set. The popup questions appear only the first time the user runs the animation of the algorithm with the associated data set. We used these questions to solicit answers about what the user just saw – which two bars were just compared, or which two bars just exchanged values, for example.

To prevent users from using the highlighted code to supply the correct answer to the popup questions, the popup box is positioned over the area containing the corresponding code and is not movable, nor does any action proceed, until a response is provided by the user. The popup text then updates to indicate the correct answer to the question. The user may then close the popup and continue with the animation. Users' answers to these popup questions are also recorded in the log.

6.6.1.3 Stimuli

Participants viewed a quicksort animation, as seen in Figure 6.5. The elements of the array of values are represented as filled rectangular bars. The value of the array element is indicated by both its height and a label that appears below the bar. The height of each bar is the value of the element multiplied by the "just noticeable difference" (95% discrimination threshold) of 0.65% of visual angle, which was equivalent to 22 pixels on our monitors when viewed at a distance of 28.5 inches. The "just noticeable difference"

or 95% discrimination threshold was determined through pilot studies, and indicates the

smallest bar height change that is noticeable at a 95% rate anywhere on the display.
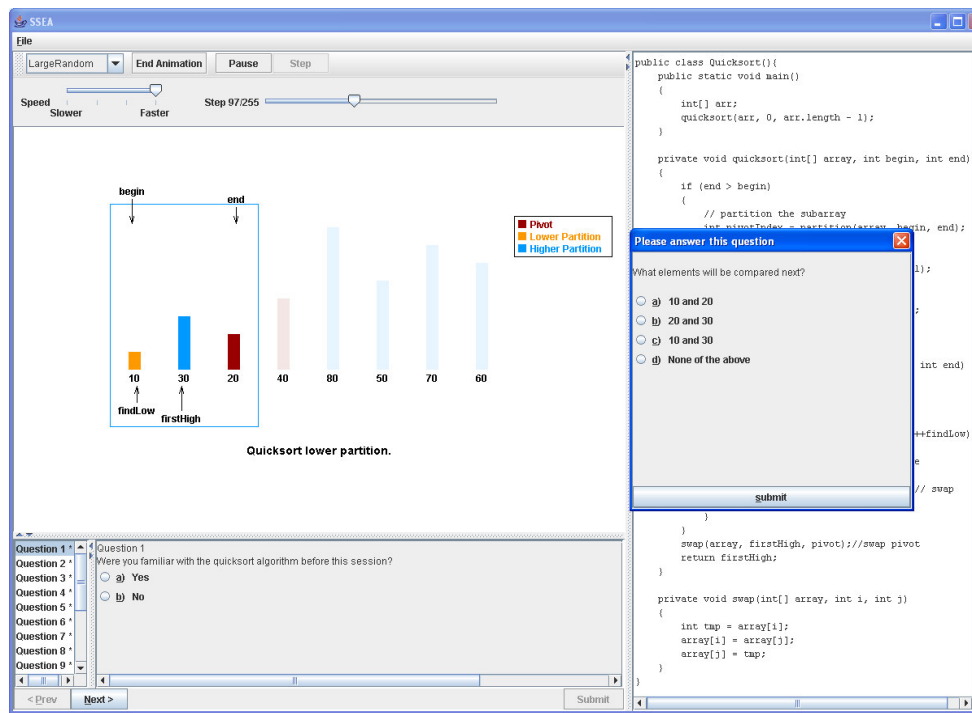
Figure 6.5:   Screen shot of the SSEA program with a pop-up question waiting to be
answered.

The quicksort algorithm is recursive and involves dividing and sub-dividing the

array into partitions. An outlining rectangle indicates the current partition. In addition,

labeled arrows marked "begin" and "end" indicate the boundaries of the current partition.

Two additional labeled arrows marked "firstHigh" and "findLow" represent the variables

that move within the current partition looking for values that are higher or lower than the

pivot.

We employed a color scheme that used six colors. Bars are colored to indicate the

state of the algorithm. The current pivot (a critical value) is colored green. Inactive

partitions are colored gray. The active partition is initially colored red. As the algorithm

proceeds the bars are sorted into a lower partition (values less than or equal to the pivot) which is colored light blue, and a higher partition (values greater than the pivot) which is colored purple.   The pivot value is then swapped with the rightmost value in the lower partition, which is its final location. It is then colored black to indicate that is in its sorted position. A legend is provided in the upper right-hand corner of the display.

Two lines of text below the array serve as captions, providing a description of the current step being performed. Lines in the pseudocode display of section B are highlighted as the animation is executing the corresponding event.

6.6.2 Design

Two between-subject factors were varied: (a) whether flashing (bars flash three times) was used to cue the comparison of the element at the "findLow" arrow with the pivot and (b) the method (move or grow) used to depict the exchange of values between two elements. In the "grow" method, the exchange is depicted by showing the larger bar shrinking to the height of the smaller bar at the same time the smaller bar grows to the height of the larger. In the "move" method the bars are seen moving in an arcing fashion from their original positions to their new positions.

Table 6.2:   2x2 Factorial Design – Letters in boxes indicate animation version.
Numbers in parentheses indicate number of participants assigned to that group.

	Move	Grow
Flash Cue	MC (14)	GC (16)
No Flash	MX (12)	GX (17)

6.6.3 Procedure

The participants were randomly assigned to one of the four groups (MC, MX, GC, GX), as seen in Table 6.2. Each participant initially completed a training exercise that required that the participants view a simple "Find Max" algorithm in SSEA, and answer one pop-up question and four traditional questions. The instructions led each participant through the use of each of the animation controls.

Participants then viewed the animation for the group to which they had been randomly assigned. Participants in each group were asked the same multiple choice questions, which reflected a range of concepts related to comprehension of the quicksort algorithm. The instructions directed participants to use the 'LargeRandom' input set. This ensured that participants would be asked all eight popup questions, which were categorized as either "cue-specific" (asking which two bars where just compared) or exchange-specific (asking which two bars just exchanged values). Upon completing the traditional questions and selecting "submit", participants were given the opportunity to comment on the animations they viewed, the SSEA system, or to give any general feedback, through a paper survey form.

6.6.4 Results

6.6.4.1 Traditional Questions

Participants had an average score of 61% on the traditional questions, some of which required detailed knowledge of the procedure or time complexity analysis. Average time to complete the study was 23 minutes. Table 6.3 shows the per-animation-group averages for score and time.

Table 6.3:   Average Time and Score per Group, Traditional Questions

Group	Average Score (%)	Average Time (minutes)
MC	62	26
GX	61	21
MX	59	25
GC	62	20

An ANOVA analysis was performed. No statistically significant effects were

found for performance on overall traditional questions for cueing, exchange type, or

interaction effects. The traditional questions were further sub-divided into groups based

on the type of knowledge the question tested: knowledge (recall/identification),

comprehension, and application. ANOVA analyses were performed on these subsets.

Again, no significant difference was found among the four animation groups.

6.6.4.2 Popup Questions

Overall performance on the pop-up questions was 66%.   Table 6.4 shows the per-

animation group scores for the pop-up questions. Times are as reported in the traditional

question section.

Table 6.4.   Average scores per group, popup questions

Group	Average Score (%)
MC	73%
GX	58%
MX	62%
GC	72%

An ANOVA analysis was performed. Cueing was found to have a statistically

significant effect ($F(1,57)=4.44$, $p < 0.04$) on participant performance on popup

questions. Further, cueing was also found to have a statistically significant effect in a subset of the popup questions classified as cueing-specific (1,2,3,8), $F(1,57)=10.39$, $p< 0.002$.

Exchange type was not found to have a statistically significant effect on performance on the overall set of popup questions. However, in a subset of questions classified as exchange-specific, a statistically significant benefit to "move" over "grow" was found ($F(1,57)=5.74$, $p< 0.02$).

6.6.5 Discussion

The results of the experiment make for interesting food for thought. Cueing significantly improved popup question scores, but not traditional question scores. However, there was a moderate correlation (0.466) between popup and traditional scores, which shows that those who did well on one type of question tended to do well on the other, and vice-versa. After the experiment was complete, we noted that there were several indicators that could supply the participant with needed information besides cueing and motion type: labels for the objects, arrows pointing to the objects before and after an action, descriptive text, and highlighted code. The design of the study permitted participants to use the animation as a resource and replay or explore it to find the needed information for traditional questions, thus providing several sources and paths to arrive at the answers. In this context, it is quite possible and reasonable that the effect of these perceptual techniques was too small to make a significant difference on those questions. The other available resources were much more likely to be important for conceptual development. In particular, the ability to replay the animation while answering questions without time constraints reduces dependence on memory and allows observers to perform

exploration and hypothesis testing. Perhaps we need to remove some of the other information sources and interaction capabilities (for experimental purposes) to isolate the effect of cueing and motion type.

However, the popup questions were shown only once, and cueing or motion immediately preceded the popup, which had to be answered then and there without the possibility of replay or exploration. This context makes these perceptual techniques important, hence its influence on the popup scores.

Some of the popup questions that involved variables require the user to map the variables to values and perform a comparison, which is an integration task. Integration tasks require the user to attend to and relate or consider multiple entities together [Bennett97]. Thus we find that cueing can help integration tasks in algorithm animations.

Motion type had no significant effect on traditional question scores, or overall popup question scores. In the case of traditional questions, this could be due to the same factors previously outlined. However, there was a significant advantage for the 'move' animation over the 'grow' animation for exchange-specific popup questions. This is not surprising, since exchanges are the only operations that utilize these animations in the context of that type of popup question. This may indicate that 'move' may be a more effective technique than 'grow' animation in this context of exchanges, perhaps because they cross the user field of view. More importantly, it confirms our hypothesis that good perceptual techniques can help users, where these techniques are relevant to the algorithm operation.

6.7   Third Study: Evaluating Popup Methodology

In experiment two, we evaluated mapping and perception using popup questions. We wanted to know if this aspect of our experimental methodology influenced comprehension. Specifically, we wanted to know if popups helped students by encouraging them to focus on the events queried by the popup questions, or if they were a distraction. This third study, designed and conducted by Philippa Rhodes and our research group [Rhodes06], was intended to investigate this and other issues. The study also evaluated the use of prediction for popups, and of feedback to popup questions. In addition, the numbers of colors used in the display design was reduced, as compared to experiment two. However, these results must be considered preliminary because of the small group of subjects.

Table 6.5:   2x3 Factorial Design - Number of Participants per group.

	NoPop-up	Without Feedback	With Feedback
Predictive	5	5	5
Responsive	4	5	5

Six groups were formed as shown in Table 6.5. The popup questions were either predictive (what will be the next step?) or responsive (what did you just see?). Half of the participants in each popup group received immediate feedback; they were shown the correct answer after each popup question. The two no-popup groups are actually one group, but the design simplifies analysis. The materials and procedure were essentially identical to that of the second study.

6.7.1 Results and Discussion/Conclusion

We were particularly interested in the influence of popup questions on participant scores, while noting that the results were preliminary. The average number of correct responses for the no-popup group was 76% on traditional questions. If we compare this to the average score of all popup groups (67%), we see that the use of popups may have made performance on traditional questions worse, but not significantly. However, the average traditional question score for the responsive-with-feedback group (76%) was the same as for the no-popup group. This suggests that the popup question methodology may have negligible effect on comprehension performance, if feedback is provided.

Prediction reduced scores on popup questions significantly, but did not affect overall traditional question scores. Feedback improved popup scores significantly (97.5% versus 77.5% without feedback), and also improved traditional scores, but not significantly. The gap in popup score averages for prediction (52.5%) versus responsive (87.5%) is understandable, given the difference in tasks. However, the gap in popup score averages for no feedback (58.8%) versus feedback (81.3%) is quite remarkable. Perhaps popups can be distracting, but the use of feedback allows the students to come to closure on each question.

Some mapping popup questions required identification of the variables that indexed the bars involved in an animation action. For the responsive group, the average score on these variable-mapping popup questions (73%) was much lower than that of mapping popup questions that just involved identification of values represented by the bars (96%). A similar difference was found for the predictive group: 48% for variable-mapping versus 60% for value-mapping. To answer the variable-mapping questions

correctly, the subject had to recall and had to keep track of the relationship of the variables to the graphical objects, because the variable labels were hidden. This could be considered a primitive test of the ability to track relationships in working memory in the context of a multi-view algorithm animation.

Surprisingly, the reduction in the number of colors used improved scores on traditional questions for the responsive-with-feedback group over the similar group in the second study (76% versus 62%), which made a marginally significant difference. This points to the need to examine various perceptual factors in AA design. Again, the results of this study must be considered preliminary because of the small group of subjects.

6.8 Comparative Study Discussion/Analysis

In the first study, cueing helped localization significantly, but did not help detection. Cueing had a significant effect on popup question scores in the second study, and these questions involved identification and localization. This indicates that cueing can be an effective perceptual technique for identification and localization, which was one of our original questions.

Of note is the fact that average localization performance for large set sizes (16 bars) was just under 85% for two cued changes versus over 95% for one cued change in the first study, while there was hardly any difference for smaller set sizes (4 and 8 bar displays). This supports one of our initial hypotheses: that viewers were not able to identify some changes, that this becomes worse as the number of changing objects increases and as the number of objects on the display increases. Even with cueing, this is still a concern, and animation designers should be aware of this. In the future, we plan to

test other cueing techniques to see if they can improve identification and localization of simultaneous changes with larger numbers of objects.

In our second study, we found that the use of perceptual techniques did not affect the comprehension level results significantly. We believe that viewers found some other path to the knowledge required to answer these questions, including the use of the replay controls to explore the behavior of the algorithm. This could indicate that this learning context and such interaction controls are important for AA comprehension tasks. Another possibility is that students may have used several other cues that were available, which indicate the same attribute or action. It may also indicate that the effect of the techniques used was too small to be significant.

In the second and third studies, we found that perceptual techniques such as cueing can help with identification and mapping in an animation (even when there are two changing objects). We also showed that users can perform simple integration tasks when mapping graphical representations to algorithm variables in a multi-view algorithm animation.

Study three suggested that experimental methodology might influence comprehension in unexpected ways, though we do not form firm conclusions due to the small number of subjects in the study. When popups were used with feedback, the methodology did not affect comprehension performance, but prediction and/or lack of feedback affected comprehension performance negatively. Perhaps the lack of feedback left students wondering if they were on the right track, and distracted them. Prediction may also be distracting. This suggests that we evaluate experimental methodologies

carefully, and examine the context of learning of AA and interaction design in future studies.

The third study also suggests that mapping graphical objects to variables may be harder than just identifying values represented by graphical objects, which is of relevance to the perception levels we proposed. Mapping tasks are apparently harder than identification tasks, which in turn are harder than detection tasks. Perceptual and object design techniques may help, as in the case of cueing in study two and color in study three. We plan to investigate this issue further. Finally, a change in color representation influenced comprehension performance, which points to the need to examine various perceptual factors in AA design, which we plan to do in future studies.

6.9 Conclusion

Our quest to thoroughly investigate the role of perception and associated psychological factors in AA has led us on an extensive journey, with contributions from a number of researchers and graduate students. This journey is ongoing. Thus far, our research has produced some fundamental results. We have shown that perceptual and attentional factors have an impact on AA viewing. Perceptual techniques can facilitate the ability to attend to animations at the perceptual and mapping levels, if used appropriately. We are not aware of any AA study that examines perception or any combination of perception, mapping and comprehension.

The VizEval and SSEA systems have shown their utility in facilitating the running of experiments that evaluate AAs at multiple levels of perception, mapping and comprehension. We are confident that they can be used as platforms for future

experiments that explore these issues further. We are not aware of any systems that facilitate this type of AA evaluation. Both systems are based on the SKA algorithm animation system, which means that the animation techniques we evaluate are portable across the systems. More importantly, this means that the techniques we verify as perceptually useful can be deployed in an actual AA system, SKA.

As a result of our analysis thus far, we can describe AA as follows: Algorithm Animation provides external representation of program entities and processes, but it also introduces perceptual tasks that need to be mapped to cognitive tasks, thus introducing new perceptual, attentional, and memory requirements. The perceptual and cognitive tasks include simultaneous tracking of data, processes, and goals as well as the interrelationships between them. The data can be complex, and are typically a combination of interrelated data structures and simple variables. The process description may or may not be shown. The processes usually have subprocesses (explicit or implicit) that have their own mental tracking requirements, but results, inputs and outputs have to be related to other parent processes or subprocesses. The cognitive agent has to track the changes not only in the algorithm, but in the AA visual mappings. In AA these tasks are often accompanied by frequent perceptual switching, which can lead to loss of tracking, attention, and orientation. This can occur within a view or between views. The continuous information and process updates in AA also introduces dynamic working memory issues and may exceed our short-term memory capabilities. The results of our studies support this analysis. All of these factors make AA design a serious challenge, which will need further exploration in the future.

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.0   Overview

In the quest for knowledge and understanding, those in a field of endeavor must, on occasion, engage in reflection on the wider universe of knowledge when faced with quandaries or a dilemma of a fundamental nature. We set out to conduct a holistic investigation into the factors affecting AA, given the puzzling results of many studies of AA effectiveness, and the questions arising from our own observations.

7.1   Contributions

We developed three guidelines that became a framework for investigation and design: namely, that to be effective, AAs must support the needs and tasks of the users, must be used in appropriate learning contexts, and that users must be able to perceive and attend to them.

To this end, we conducted studies to gain a deeper understanding of these issues, having chosen a subset of problems from these areas to examine. We conducted research to examine users needs and tasks, choosing a context that had not been previously investigated: the activities of instructors and their use of artifacts in algorithms and data structures classes.

We designed SKA to support the tasks performed by instructors, which represents a unique effort in AA system design to create an environment that supports the discourse of experts and novices, as informed by user studies.

We divided the study of the role of perception into an examination of the effects at the perceptual, mapping, and comprehension levels. We investigated the role of perception and associated factors in AA, in a series of collaborative studies. Some of these experiments tested the ability of users to perceive and attend to animations at the perceptual and mapping levels, using various animation methods and cues. The effect of these methods and cues on comprehension was also examined. We now have evidence that the use of these methods and cues can help users perceive and attend to algorithm animations, but can also have complex effects. We are not aware of any AA study that measures either perception or any combination of perception, mapping and comprehension.

We developed a software architecture to be used in the running of perceptual AA experiments, that facilitates creation of trials, running trials and data collection. We collaborated on the development of the VizEval testing suite and contributed the popup trial methodology aspect of the SSEA testing environment. These software support testing at multiple levels, and use the SKA scripting interface to specify perceptual animation tasks and to monitor user interaction.

We have analyzed AA from a number of research perspectives including HCI, educational technology and engineering psychology. We have worked on multi-level experimental methodologies to investigate AA from these perspectives. We believe that this combination of analytical perspectives has not been previously brought to bear on AA issues.

7.2   Future Work

We will continue research on all three aspects of the framework that we hold to have bearing on AA, as we aim to gain further understanding of relevant issues, and widen the horizons of AA research by incorporating research from other fields of endeavor.

We will continue and expand our user studies, including evaluation of SKA in various contexts. The results of the perceptual experiments will be examined, to seek to improve animation in the SKA libraries. We will seek to make authoring simpler, including for library developers. Additions will be made to the SKA libraries. We will continue to collaborate on perceptual experiments and develop additional experiments involving design for perception and associated factors.

It is our position that AAs demand a dynamic mix of focused, selective and divided attention, as well as perceptual tasks and cognitive tasks. This makes the processing of AAs at these multiple levels a complex task, and this difficulty generally increases with the complexity of the algorithm. Of course, it is often the more complex algorithms that we want to visualize.

There is a large leap between the mapping and comprehension levels. Comprehension involves factors related to learning such as learning style and background knowledge, as well as factors related to perception, attention, and memory. At the mapping level, effective designs must establish the connection between the AA entities and the algorithmic entities, and facilitate visual and cognitive tracking of these entities. At the comprehension level, a good design needs to portray the essential concepts that the AA designer wants to portray about the algorithm, and facilitate exploration of the algorithm where relevant or useful.

In the future, we plan to extend our present series of experiments with our collaborators. We also plan to update the VizEval suite, based on their feedback. We propose to examine how cognitive engineering design and analysis concepts such as knowledge representation, attention-based display design, correspondence and coherence [Norman88] [Bennett97] [WR88] relate to algorithm animations. How much information do we represent in the animation ("knowledge in

the world") versus how much we assume the user knows ("knowledge in the head")? Too much information can lead to display clutter and overwhelm the user, while too little can leave the user with insufficient resources.

We will consider how to organize complex displays, such as typically found in multi-view AAs so that users find it easier to attend to them, i.e. how to make these displays more coherent. We will also consider how to design the animation entities and actions to optimally represent the relationship, or correspondence between them and the algorithm entities and operations. We will investigate the role of pacing and timing. We believe that these and other design techniques have the potential to improve algorithm animation comprehension.

We believe that the use of AA and similar educational technology has a bright future, provided that we create conditions that give researchers of all origins the means to thoroughly investigate the issues that impact on the design and use of these technologies, and if we invest in those who have creative insight and innovative ideas, enabling them to fulfill their visions, which will ultimately benefit all.

REFERENCES

[AECT05]   *What is the Knowledge Base?* Association for Educational
Communications and Technology. AECT (2005). Retrieved September
30, 2005 at http://www.aect.org/standards/knowledgebase.html

[AU85]   Alfred Aho and Jeffrey Ullman. *Data Structures and Algorithms*.
Addison-Wesley, 1985.

[Baddeley93] Baddeley, Alan. Working Memory or Working Attention? In Baddeley,
Alan (Ed.). *Attention: Selection, Awareness, and Control*. Ch.8.
Clarendon Press: Oxford (1993).

[Baecker68] Baecker R., "Experiments in On-Line Graphical Debugging: The
Interrogation of Complex Data Structures," *Proceedings of the First
Hawaii International Conference on the System Sciences*, January 1968,
pp. 128–129.

[Baecker71] Baecker, R. A demonstration of the Genesys Computer Animation
System, 1971. In *The Dynamic Image*. Baecker, R.M. 60 minute colour
sound film, Dynamics Graphics Project, University of Toronto, 1987.
(Distributed by Morgan Kaufmann, Publishers.)

[Baecker75] Baecker R., "Two Systems which Produce Animated Representations of
the Execution of Computer Programs," *SIGCSE Bulletin*, Vol. 7, No. 1,
February 1975, pp. 158–167.

[Baecker81]   R.M. Baecker, Sorting out Sorting (video), in *SIGGRAPH Video Review 7*, 1983. (see http://kmdi.utoronto.ca/RMB/ for sample clips).

[Baecker98]   Baecker, R. Sorting Out Sorting: a Case Study of Software Visualization for Teaching Computer Science. In Stasko, J., Domingue, J., Brown, M. H., & Price, B. A. (1998). *Software visualization: Programming as a multimedia experience*, Ch.24, pp.369-382. Cambridge, MA: MIT Press.

[BP98]   Baecker, R, and Price, B. The Early History of Software Visualization. In Stasko, J., Domingue, J., Brown, M. H., & Price, B. A. (1998). *Software visualization: Programming as a multimedia experience*, Ch.2, pp.29-34. Cambridge, MA: MIT Press.

[Ball88]   Ball, K.K, Beard, B.L., Roenker, D.L., Miller, R.L. and Griggs, D.S. (1988). Age and visual search: Expanding the useful field of view. Journal of the Optical Society, A, 5, 2210-2219.

[Bartram01]   Linda Ruth Bartram. *Enhancing Information Visualization with Motion*. Unpublished Ph.D. thesis, School of Computing Science, Simon Fraser University, Canada, 2001.

[Bates89]   Peter Bates, Debugging heterogeneous distributed systems using event-based models of behavior, *ACM SIGPLAN Notices*, v.24 n.1, p.11-22, Jan. 1989.

[Bazik98]   Bazik, J., et.al. Software Visualization in Teaching at Brown University. In Stasko, J., Domingue, J., Brown, M. H., & Price, B. A. (1998).

Software visualization: Programming as a multimedia experience. Ch.25, pp.383-398. Cambridge, MA: MIT Press.

[Bennett97]   Bennett,K.B., Nagy,A.L., Flach,J.M. Visual Displays. In Salvendy, G. (Ed.) (1997). *Handbook of Human Factors and Ergonomics*, 2nd ed. Ch.20, pp.659-696. New York: John Wiley & Sons.

[Benyon05]   Benyon,D., Turner,P., Turner,,S . *Designing Interactive Systems*. Addison-Wesley (2005).

[BH05]   Robert Maribe Branch and Christa L. Harrelson. Processes. Unpublished Report, Department of Educational Psychology and Instructional Technology, The University of Georgia, 21 July 2005.

[Brown87]   Marc H. Brown, *Algorithm Animation*, ACM Distinguished Dissertations, MIT Press, 1987

[Brown88]   Marc H. Brown, Exploring Algorithms Using Balsa-II, *Computer*, v.21 n.5, p.14-36, May 1988.

[Brown98]   Marc H. Brown. Interesting Events. In Stasko, J., Domingue, J., Brown, M. H., & Price, B. A. (1998). *Software visualization: Programming as a multimedia experience*. Ch.12, pp.155-172. Cambridge, MA: MIT Press.

[Brown98b]   Marc H. Brown. A Taxonomy of Algorithm Animation Displays. In Stasko, J., Domingue, J., Brown, M. H., & Price, B. A. (1998). *Software visualization: Programming as a multimedia experience*. Ch.12, pp.155-172. Cambridge, MA: MIT Press.

[BH98]   Marc H. Brown, and John Hershberger. Fundamental Techniques for Algorithm Animation Displays. In Stasko, J., Domingue, J., Brown, M. H., & Price, B. A. (1998). *Software visualization: Programming as a multimedia experience*. Ch.7, pp.81-102. Cambridge, MA: MIT Press.

[Beyer98]   Hugh Beyer and Karen Holtzblatt. *Contextual Design: Defining Customer-Centered Systems*, San Francisco: Morgan Kaufmann (1998).

[BCS99]   Michael D. Byrne, Richard Catarambone, and John T. Stasko. Evaluating animations as student aids in learning computer algorithms. *Computers & Education*, Vol.33, No.5, pp.253-278, 1999.

[Coren04]   Coren, S., Ward, L.M. & Enns, J.T. (2004). *Sensation and Perception, 6th ed*. Wiley.

[CLR90]   Cormen, T. H., Leiserson, C. E., & Rivest, R. L. (1990). *Introduction to Algorithms*. Cambridge, MA: The MIT Press,

[CHB02]   Using the Debugger as an Integral Part of Teaching CS1 - by James H. Cross II, T. Dean Hendrix, Larry A. Barowski. *Proceedings of FIE 2002*.

[Dale03]   Nell Dale, Daniel Joyce & Chip Weems. *Object-Oriented Data Structures Using Java*. Jones and Bartlett, 2003.

[Davis04]   Davis, E.T., & Palmer, J. (2004). Visual search and attention: An overview. *Spatial Vision, 17*, 249-255.

[DHKHR06]   Davis, E. T., Hailston, K., Kraemer, E., Hamilton-Taylor, A., Rhodes, P., Papadimitiou, C., and Garcia, B., 2006. Examining Perceptual

Processing of Program Visualization Displays to Enhance Effectiveness, in: *To appear in 2006 Human Factors and Ergonomics Society (HFES).*

[Denning89]   Peter J. Denning, A debate on teaching computing science, Communications of the ACM, v.32 n.12, p.1397-1414, Dec. 1989.

[diSessa86]   A. A diSessa, H. Abelson. Boxer: a reconstructible computational medium. *Communications of the ACM*, Vol.29, No.9, September 1986 .

[DuBoulay89]   Du Boulay, B. Some Difficulties of Learning to Program. In E. Soloway and J. Spohrer (eds.), *Studying the Novice Programmer*, New Jersey: Lawrence Erlbaum Associates (1989), 431-446.

[Eprime]   E-prime website. http://www.pstnet.com/products/e-prime/

[FS97]   Peter Faraday and Alistair Sutcliffe. Designing Effective Multimedia Presentations. *Proceedings CHI'97* ACM, 1997, pp.272-278.

[FW96]   Jeremy Ford and Larry Wood. An overview of ethnography and system design. In D. Wixon & J. Ramey (Eds) *Field methods casebook for software design*. New York: Wiley (1996).

[Geisler95]   Geisler, W.S. & Chou, K.L. (1995). Separation of low-level and high-level factors in complex tasks: Visual search. *Psychological Review*, 102, 356-378.

[Goldstein02]   Goldstein, E. B. *Sensation and perception, 6th ed.* Wadsworth: Belmont, California.

[Green00]   Green, T. R. G. (2000) Instructions and descriptions: some cognitive aspects of programming and similar activities. Invited paper, in Di Gesù,

V., Levialdi, S. and Tarantino, L., (Eds.) *Proceedings of Working Conference on Advanced Visual Interfaces (AVI 2000)*. New York: ACM Press, pp 21-28.

[GC96]   Judith S. Gurka and Wayne Citrin. Testing effectiveness of algorithm animation. *Proceedings of the 1996 IEEE Symposium on Visual Languages*, pages 182-189, Boulder, CO, September 1996.

[Hailston06]   Hailston, K., et.al. In preparation.

[Hamilton-Taylor02a]   SKA: Supporting Algorithm and Data Structure Discussion. Ashley Hamilton-Taylor and Eileen Kraemer. *Proceedings of SIGCSE 2002*, pp.58-62, February 2002.

[Hamilton-Taylor02b]   Designing an Algorithm Animation System to Support Instructional Tasks. Ashley Hamilton-Taylor and Eileen Kraemer. Interactive Multimedia Electronic Journal of Computer-Enhanced Learning (IMEJ), October 2002.

http://imej.wfu.edu/articles/2002/2/04/index.asp

[HPSTTV97]   Jyrki Haajanen, Mikael Pesonius, Erkki Sutinen, Jorma Tarhio, Tommi Teräsvirta, Pekka Vanninen. Animation of User Algorithms on the Web. *IEEE Symposium on Visual Languages* (VL'97), 360–367.

[Healey00]   Healey, C. G. (2000) Building a Perceptual Visualisation Architecture. *Behaviour and Information Technolog,y* 19(5), 349-366.

[HN03]   Hübscher-Younger, T., N. H. Narayanan. Dancing Hamsters and Marble Statues: Characterizing Student Visualizations of Algorithms. ACM Symposium on Software Visualization, 2003.

[HDS02]   Hundhausen, C.D., Douglas, S.A., & Stasko, J.T. (2002). A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages and Computing*, 13(3), 259-290.

[HD02]   Hundhausen, C.D. & Douglas, S.A. (2002). Low Fidelity Algorithm Visualization. *Journal of Visual Languages and Computing,* 13(5), 449-470.

[Hund99]   Christopher Hundhausen. *Toward Effective Algorithm Visualization Artifacts: Designing for Participation and Communication in an Undergraduate Algorithms Course*. Unpublished Doctoral Dissertation (Tech Rep. No. CIS-TR-99-07), Dept. of Computer and Information Science, University of Oregon, 1999.

[Healey00]   Healey, C. G. (2000) Building a Perceptual Visualisation Architecture. *Behaviour and Information Technolog,y* 19(5), 349-366.

[HHN85]   Edwin Hutchins, James Holland and Donald Norman. Direct manipulation Interfaces. Human-Computer Interaction, Vol.1, pp.311-336, 1985.

[HDS02]   Hundhausen, C.D., Douglas, S.A., & Stasko, J.T. (2002). A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages and Computing*, 13(3), 259-290.

[Jarc99]   Duane Jeffrey Jarc. *Assessing the Benefits of Interactivity and the*
                  Influence of Learning Styles on the Effectiveness of Algorithm Animation
                  using Web-based Data Structures Courseware. Dissertation, George
                  Washington University, 1999. http://student.seas.gwu.edu/~idsv/

[Jeffrey98]   Clinton L. Jeffrey. A Menagerie of Program Visualization Techniques.
                  In Stasko, J., Domingue, J., Brown, M. H., & Price, B. A. (1998).
                  Software visualization: Programming as a multimedia experience.
                  Ch.12, pp.155-172. Cambridge, MA: MIT Press.

[Kay81]   Ryan, Bob. "Dynabook Revisited with Alan Kay". Byte. vol 16,
                  February 1991. Retrieved July 11, 2006, from the World Wide Web:
                  http://www.squeakland.org/school/HTML/essays/dynabook_revisited.ht
                  m

[KST01]   Colleen Kehoe, John Stasko, and Ashley (Hamilton-)Taylor. Rethinking
                  the evaluation of algorithm animations as learning aids: An
                  observational study. *International Journal of Human-Computer Studies*,
                  Vol.54, No.2, pp.265-284, February 2001.

[Kosch96]   Timothy Koschmann. Paradigm Shifts and Instructional Technology. In
                  Koschmann (Ed.), *CSCL: Theory and Practice of an Emerging*
                  Paradigm, pp. 1-23. Lawrence Erlbaum, NJ.

[KRRH06]   Kraemer, E., Reed, B., Rhodes, P. and Hamilton-Taylor, A. (2006).
                  SSEA: A System for Studying the Effectiveness of Animations. *To*

Appear in Proceedings, Fourth Program Visualization Workshop, 2006. Florence, Italy.

[KD02]   Kraemer, E., Davis, E. *Program Visualization: Using Perceptual and Cognitive Concepts to Quantify Quality, Support Instruction, and Improve Interactions.* NSF proposal#0308063.

[KTH01]   Why Johnny Won't Visualize. Eileen Kraemer, Mihail Tudoreanu, and Ashley (Hamilton-)Taylor. *Software Visualization Workshop at the International Conference on Software Engineering (ICSE01)*, Toronto, 2001.

[Kramer95]   Kramer, A.F. & Hahn, S. (1995). Splitting the beam: Distribution of attention over non-contiguous regions of the visual field. *Psychological Science*, 6, 381-386.

[Lawrence93]   Andrea W. Lawrence. *Empirical Studies of the Value of Algorithm Animation in Algorithm Understanding.* Unpublished Ph.D. thesis, 1993, Georgia Institute of Technology, Atlanta, GA.

[LBS94]   Andrea W. Lawrence, Albert M. Badre, and John T. Stasko. Empirically Evaluating the use of Animations to Teach Algorithms. *Proceedings of the 1994 IEEE Symposium on Visual Languages*, October 1994, pp. 48-54.

[LF98]   Leiberman, H., and Fry, C. ZStep 95: A Reversible, Animated Source Code Stepper. In Stasko, J., Domingue, J., Brown, M. H., & Price, B. A.

(1998). *Software visualization: Programming as a multimedia experience*, Ch.19, pp.277-292. Cambridge, MA: MIT Press.

[LG85]   Lincoln, Y., & Guba, E. (1985). *Naturalistic inquiry*. Beverly Hills, CA: Sage.

[LBU03]   Ronit Ben-Bassat Levy, Mordechai Ben-Ari, and Pekka A. Uronen. The Jeliot 2000 program animation system. *Computers & Education*, 40(1), 1-15, 2003.

[Macmillan91]   Macmillan, N. and Creelman, C.D.(1991). *Detection Theory: A User's Guide*. New York: Cambridge University Press.

[Mayer89]   Mayer, Richard. The Psychology of How Novices Learn Computer Programming. In E. Soloway and J. Spohrer (eds.), *Studying the Novice Programmer*, New Jersey: Lawrence Erlbaum Associates (1989), 431-446.

[MM98]   Richard Mayer and Roxana Moreno. A Split-Attention Effect in Multimedia Learning: Evidence for Dual Processing Systems in Working Memory. *Journal of Educational Psychology*, Vol. 90, No. 2, pp.312-320.

[MCS01]   D. Scott McCrickard, Richard Catrambone, and John T. Stasko. "Evaluating Animation in the Periphery as a Mechanism for Maintaining Awareness." In *Proceedings of the IFIP TC.13 Conference on Human-Computer Interaction (INTERACT '01)*, Tokyo Japan, July 2001, pp 148-156.

[MMSB04]   Moreno, A., Myller, N., Sutinen, E., and Ben-Ari, M.: Visualizing

                Programs with Jeliot 3. To appear in the *Proceedings of the*

                International Working Conference on Advanced Visual Interfaces AVI

                2004, Gallipoli (Lecce), Italy, 25-28 May, 2004

[MS94]   Mukherjea, Sougata and John T. Stasko, "Toward Visual Debugging:

                Integrating Algorithm Animation Capabilities within a Source-Level

                Debugger," *ACM Transactions on Computer-Human Interaction*, Vol. 1,

                No. 3, Sept. 1994, pp. 215-244.

[Mulholland98] Mulholland, P. A principled approach to the evaluation of SV: a case

                study in Prolog. In *Software visualization: Programming as a*

                multimedia experience. The MIT Press, Cambridge, MA, (1998), 439 -

                452.

[Naps03]   Thomas L. Naps , Guido Rößling , Vicki Almstrum , Wanda Dann ,

                Rudolf Fleischer , Chris Hundhausen , Ari Korhonen , Lauri Malmi ,

                Myles McNally , Susan Rodger , J. Ángel Velázquez-Iturbide, Exploring

                the role of visualization and engagement in computer science education,

                ACM SIGCSE Bulletin, v.35 n.2, June 2003.

[Nardi96]   Bonnie A. Nardi. Activity Theory and Human-Computer Interaction. In

                Context and Consciousness: Activity Theory and Human-Computer

                Interaction. MIT Press (1996).

[Norman88]   Norman, Don. (1988) *The Psychology of Everyday Things.*New York:

                Basic Books.

[Orey05]   Orey, M. (2005). Preface. In M. Orey, J. McClendon & R. Branch

                (Eds.), *Educational media and technology yearbook* (Vol. 30, pp. ix-x).

                Englewood, Colorado: Libraries Unlimited, Incorporated.

[Pappert87]   Seymour Pappert. Computer Criticism vs. Technocentric Thinking.

                Educational Researcher, Vol. 16, No. 1, pp. 22-30.

[Pashler98]   Pashler, Harold. *The Psychology of Attention.* MIT Press (1998).

[Pane02a]   J.F. Pane, "A Programming System for Children that is Designed for

                Usability," Ph.D. Thesis, Carnegie Mellon University, Computer

                Science Department, CMU-CS-02-127, Pittsburgh, PA, May 3, 2002.

[Pane02b]   J.F. Pane, B.A. Myers, and L.B. Miller, "Using HCI Techniques to

                Design a More Usable Programming System," *Proceedings of IEEE

                2002 Symposia on Human Centric Computing Languages and

                Environments (HCC 2002)*, Arlington, VA, September 3-6, 2002, pp.

                198-206.

[Petre02]   Petre, M. Mental imagery, visualisation tools and team work. Keynote

                paper, in proceedings of: *Second Program Visualisation Workshop*,

                June 2002, Hornstrup Centret, Denmark.

[PBS93]   Price, B. A., Baecker, R. M., & Small, I. S. (1993). A principled

                taxonomy of software visualization. *Journal of Visual Languages and

                Computing*, 4(3), 211-266.

[PBS98]   Blaine Price, Ronald Baecker and Ian Small. An Introduction to
               Software Visualization. In Stasko, J., Domingue, J., Brown, M. H., &
               Price, B. A. (1998). *Software visualization: Programming as a
               multimedia experience*, Ch.1, pp.. Cambridge, MA: MIT Press.

[Proctor97]   Proctor,R.W., Proctor,J.D. Sensation and Perception. In Salvendy, G.
               (Ed.) (1997). *Handbook of Human Factors and Ergonomics*, 2nd ed.
               Ch.3, pp.43-89. New York: John Wiley & Sons.

[Pylylshyn00] Pylylshyn, Z.W. and Sears, C.R. Multiple object tracking and
               attentional processing. Canadian Journal of Experimental Psychology.
               2000, 54(1), 1-14.

[RKHD06]   Rhodes,P., Kraemer,E., Hamilton-Taylor,A., Thomas,S., Ross,M.,
               Davis,E., Hailston,K., Main,K., 2006. VizEval - An Experimental
               System for the Study of Program Visualization Quality, in: *To appear in
               IEEE Symposium on Visual Languages and Human-Centric Computing
               VL/HCC 06*, Brighton, UK.

[Rhodes06]   Philippa Rhodes, Eileen Kraemer, Bina Reed. The Importance of
               Interactive Questioning Techniques in the Comprehension of Software
               Visualizations. *To Appear in ACM Software Visualization SoftViz06*,
               Brighton, England.

[Reed06]   Reed, B., 2006. *Investigating Characteristics of Effective Program
               Visualizations: A Testing Environment and the Effect of Cueing and*

Swapping Techniques in Algorithm Animations. Unpublished M.S. Thesis, Computer Science, University of Georgia, Athens, GA.

[RRKHDH06]   Bina Reed, Philippa Rhodes, Eileen Kraemer, Ashley Hamilton-Taylor, Elizabeth Thorpe Davis, and Kenneth Hailston. The Effect of Comparison Cueing and Exchange Motion on Comprehension of Program Visualizations. *To Appear in ACM Software Visualization SoftViz06*, Brighton, England.

[Reiss98]   Reiss, S. Visualization for Software Engineering – Programming Environments. In Stasko, J., Domingue, J., Brown, M. H., & Price, B. A. (1998). *Software visualization: Programming as a multimedia experience*, Ch.18, pp.259-276. Cambridge, MA: MIT Press.

[Reiss03]   Reiss, S. Visualizing Java in Action, *Proc. IEEE Conference on Software Visualization*, pp. 123-132, May 2003.

[Reeves99]   Reeves, Thomas. A Research Agenda for Interactive Learning in the new Millennium. ED-MEDIA 99 Keynote Address Paper.

[Roman92]   G.-C. Roman, K.C. Cox, D. Wilcox and J.Y. Plun. Pavane: a system for declarative visualization of concurrent computations. *J. Visual Languages*, 3(2): 161-193, 1992.

[Rodger02]   S. H. Rodger, Using Hands-on Visualizations to Teach Computer Science from Beginning Courses to Advanced Courses, *Second Program Visualization Workshop*, Hornstrup Centert, Denmark, (6 pages) June 2002.

[RF01]   Guido Rößling, Bernd Freisleben. AnimalScript: An Extensible
                  Scripting Language for Algorithm Animation. *ACM 32nd SIGCSE
                  Technical Symposium on Computer Science Education (SIGCSE 2001)*,
                  Charlotte, NC. pp. 70-74, ACM Press, 2001.

[Rößling]   ANIMAL visual scripting tutorial.
                  http://www.animal.ahrgr.de/usingAnimalVisually.pdf

[Ross04]   Ross, Matthew. *A testing environment for the evaluation of program
                  visualization quality*. Unpublished M.S. Thesis, Computer Science,
                  University of Georgia, Athens, GA.

[Saraiya04]   Purvi Saraiya, Clifford A. Shaffer, D. Scott McCrickard, Chris North.
                  Effective Features of Algorithm Visualizations . *ACM 35th SIGCSE
                  Technical Symposium on Computer Science Education (SIGCSE 2004),*
                  Norfolk, Virginia, pp.382-386.

[SR01]   Scaife, M. and Rogers, Y. Informing the design of a virtual environment
                  to support learning in children. *In Int. J. Human-Computer Studies*,
                  (2001), 55, 115-143.

[Shaw80]   Shaw, M.L. (1980). Identifying attentional and decision-making
                  components in information processing. In R. Nickerson (Ed.), *Attention
                  and Performance*, VIII. Hillsdale, NJ: Erlbaum.

[Smith86]   Randall B. Smith, The Alternative Reality Kit: An Animated
                  Environment for Creating Interactive Simulations, *1986 IEEE Computer*

Society Workshop on Visual Languages, June 25 - 27, 1986, Dallas
Texas, pp. 99 - 106.

[SM95]   Randall Smith and John Maloney. Directness and Liveness in the
Morphic User Interface. *Proc. UIST '95*, Pittsburgh, pp. 21-28.

[Stasko96]   John Stasko. Using Student-Built Algorithm Animations as Learning
Aids. Technical Report GIT-GVU-96-19, Georgia Institute of
Technology, August 1996.

[Stasko98a]   Stasko, J. Smooth, Continuous Animation for Portraying Algorithms and
Processes. In Stasko, J., Domingue, J., Brown, M. H., & Price, B. A.
(1998). *Software visualization: Programming as a multimedia
experience*, Ch.8, pp.103-118. Cambridge, MA: MIT Press.

[Stasko98b]   Stasko, J. Building Software Visualizations through Direct Manipulation
and Demonstration. In Stasko, J., Domingue, J., Brown, M. H., & Price,
B. A. (1998). *Software visualization: Programming as a multimedia
experience*, Ch.14, pp.187-204. Cambridge, MA: MIT Press.

[SH04]   Stasko, J.T. & Hundhausen, C.D. (2004). Algorithm Visualization. In S.
Fincher & M. Petre (eds.), *Computer Science Education Research* (pp.
199-228). Lisse, The Netherlands: Taylor & Francis.

[SK93]   John Stasko and Eileen Kramer. A Methodology for Building
Application-Specific Visualizations of Parallel Programs. *Journal of
Parallel and Distributed Computing*, Vol.18, No.2, June 1993, pp.258-
264.

[SM95]   John Stasko and Scott McCrickard. Real Clock Time Animation Support for Developing Software Animations. *Australian Computer Journal*, Vol. 27, No.4, November 1995.

[Stokes88]   Stokes,A., Wickens,C., Kite,K. Color and Pictorial Displays. In Stokes,A., Wickens,C., Kite,K., *Display Technology: Human Factors Concepts*. Ch.7,pp.65-87. Society of Automotive Engineers (1988).

[TG97]   Debugging in Smalltalk: Identifying Student Difficulties. Ashley George Taylor and Mark Guzdial. Poster, *Seventh Workshop on Empirical Studies of Programmers*, Alexandria, Virginia, October 1997.

[Thomas04]   Thomas, Sujith. *An experiment designer tool for evaluation of program visualization quality*. Unpublished M.S. Thesis, Computer Science, University of Georgia, Athens, GA.

[Treisman93]   Treisman, A. (1993). The perception of features and objects. In A. Baddeley & L. Weiskrantz (Eds.), *Attention, Selection, Awareness, and Control* (pp. 6-34). Oxford: Clarendon.

[TRHK02]   Mihail Tudoreanu, Rong Wu, Ashley Hamilton-Taylor, and Eileen Kraemer. Empirical Evidence that Algorithm Animation Promotes Understanding of Distributed Algorithms. *IEEE Symposia on Human-Centric Languages and Environments (HCC'02)*, Virginia, September 2002.

[ULF97]   Ungar, D., Lieberman, H., and Fry, C. (1997). Debugging and the Experience of Immediacy. *Communications of the ACM*, 40(4), 38-43.

[WC97]   Wickens,C.D., Carswell,C.M. Information Processing. In Salvendy, G.

                (Ed.) (1997). *Handbook of Human Factors and Ergonomics*, 2nd ed.

                Ch.4, pp.90-129. New York: John Wiley & Sons.

[Ware99]   Colin Ware. *Information Visualization: Design for Perception.*

                Academic Press (1999).

[Williams85]   Williams, L. 1985. Tunnel vision induced by a foveal load manipulation.

                Human Factors 27, 2, 221–227.

[Woods84]   Woods,D.D. (1984). Visual momentum: A concept to improve the

                cognitive coupling of person and computer. *International Journal of

                Man-Machine Studies*,21 ,229-244.

[WR88]   Woods,D.D., and Roth,E.M. (1984). Cognitive systems engineering. In

                M. Helander, Ed., *Handbook of Human-Computer Interaction* (pp.1-41).

                Amsterdam:Elsevier Science Publishers.