A Framework Proposal for Algorithm Animation Systems

Chih Lung Yeh

Submitted in partial fulfilment of
the requirements for the degree of
MAGISTER COMMERCII
in the Faculty of Business and Economic Sciences at the
Nelson Mandela Metropolitan University

January 2006

Supervisor: Prof. J.H. Greyling
Co-supervisor: Dr. C.B. Cilliers

Department of Computer Science and Information Systems

Acknowledgements

First and foremost, massive thanks to my supervisors, Jeán and Charmain, for the confidence they had in me, and for their inputs and ideas towards the project. This dissertation is as much their effort as it is mine. Their ever-present sense of humour kept me sane when things were getting difficult. I thank them for helping me over the many challenging miles (literally and figuratively).

Thanks also go to Dieter for the help and advice he has provided me with during the implementation of the project.

Finally, I would like to thank Lee-Ann for assisting me by proofreading this dissertation, and for her incredible patience and attention to detail.

Table of Contents

Summary

The learning and analysis of algorithms and algorithm concepts are challenging to students due to the abstract and conceptual nature of algorithms. Algorithm animation is a form of technological support tool which encourages algorithm comprehension by visualising algorithms in execution. Algorithm animation can potentially be utilised to support students while learning algorithms.

Despite widespread acknowledgement for the usefulness of algorithm animation in algorithm courses at tertiary institutions, no recognised framework exists upon which algorithm animation systems can be effectively modelled. This dissertation consequently focuses on the design of an extensible algorithm animation framework to support the generation of interactive algorithm animations.

A literature and extant system review forms the basis for the framework design process. The result of the review is a list of requirements for a pedagogically effective algorithm animation system. The proposed framework supports the pedagogic requirements by utilising an independent layer structure to support the generation and display of algorithm animations. The effectiveness of the framework is evaluated through the implementation of a prototype algorithm animation system using sorting algorithms as a case study.

This dissertation is successful in proposing a framework to support the development of algorithm animations. The prototype developed will enable the integration of algorithm animations into the Nelson Mandela Metropolitan University's teaching

model, thereby permitting the university to conduct future research relating to the usefulness of algorithm animation in algorithm courses.

Keywords: Algorithm animation framework; algorithm animation system; sorting algorithm; technological support tools

List of Figures

List of Tables

List of Definitions

Chapter 1

Research Context and Background

1.1 Introduction

Introductory algorithm curricula encompass various concepts, one of which is the study of fundamental computing algorithms. In the study of computing algorithms, specific predefined algorithms are investigated, and the computational efficiency related to these algorithms is discussed. Students learn to analyse and evaluate algorithms based on certain criteria (IEEE and ACM 2001). This learning process can be simplified by the use of algorithm animations.

The studying and teaching of algorithmic concepts present a constant challenge for both students and educators. Kehoe, Stasko and Taylor (2001) put this problem in perspective:

> *"There is something difficult about understanding and analysing algorithms; ask any computer science student. What that something is and how to reduce the difficulty are two problems whose solutions are anxiously awaited by many students and instructors".*

Unlike structures and processes of other disciplines, computer algorithms and data structures are implementations of abstract and conceptual phenomena that have no physical, concrete form. As a result, the algorithms are hard to follow, and are thus difficult to understand and learn (Lattu, Meisalo and Tarhio 2003). Furthermore, students are expected to cope with the programming notations, syntax, semantics, structure and style of the language in which the algorithm is implemented, compounding an already difficult problem (Cilliers 2005). Without a full understanding of an algorithm, students are not able to apply and implement it to solve a given problem, and consequently perform poorly.

Tertiary educational institutions worldwide continually face the challenge of maintaining satisfactory performance rates for their students (Lister and Leaney 2003). These institutions are constantly pursuing the use of new strategies to improve the throughput rate of their students, including the integration of technological support tools to support the curricula. The availability of computers has improved the accessibility of technological support tools to educational institutions and students.

The remainder of the chapter outlines the research leading up to the adaptation of algorithm animations in an educational environment (Section 1.2) and the role played by algorithm animations in improving learning (Section 1.3). The focus of research (Section 1.4) discusses the issues concerning the implementation of an algorithm animation system and the related research questions.

1.2 Background and Prior Research

Various instructional aids are employed by instructors of algorithm courses to help students in better comprehending and applying the algorithm concepts (Baldwin and Scragg 2004). A commonly used method is to provide the algorithm source code or pseudo-code to the students, accompanied by textual explanations and lecture interactions that discuss the algorithm in varying levels of detail. Students also use informal collaborative methods for learning. The students, for example, work in small groups to complete assignments together or to explain difficult concepts to each other (Hübscher-Younger and Narayanan 2002). These methods, however, still rely much on the students to construct mental models of the abstract concepts for themselves, thus increasing the students' cognitive load and reducing learning effectiveness (Tudoreanu 2002).

Related research on software visualisation has focused on making use of the power of the human visual system and its ability to effectively take in a large amount of information, detect visual patterns, and absorb pictorial representations (Roman and Cox 1992). Tools and techniques were created to assist understanding by providing visual form to abstract program concepts.

"Sorting Out Sorting" (Baecker 1981) is a 30-minute algorithm animation video which demonstrated the characteristics and operations of nine sorting algorithms using animations and audio commentary. The video is regarded as the first attempt to bring to life and successfully exhibit the dynamic nature of algorithms to students. Definitive work done by Brown (1988a) explored the educational opportunities

offered by algorithm animations. The research output also led to the first computerised interactive algorithm animation system, BALSA-II (Brown 1988b).

Numerous algorithm animation systems have since been developed with the purpose of aiding students in their study of algorithms (Wilson, Aiken and Katz 1996; McCauley 1998; Wiggins 1998). More recently, algorithm animation has gained acceptance as a valuable educational tool in algorithm courses (Garner 2003; Costelloe 2004).

Research into the educational use of algorithm animations has focused generally on three interrelated processes which iterate cyclically. These research directions are illustrated in Figure 1.1.

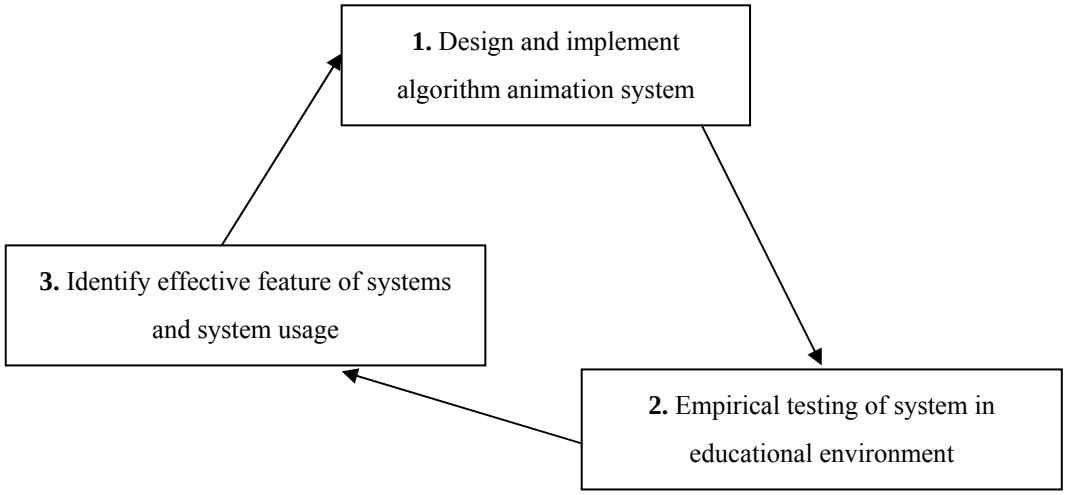


Figure 1.1: The interaction of current research focus on algorithm animation

Initially, algorithm animation systems were developed with the simple hypothesis that they could help students to better understand algorithms (Stasko and Lawrence 1998). The systems were developed as instructional tools for computer science courses. The early algorithm animation systems relied primarily on intuition to guide design, as no

conventions existed on what features and usage techniques were conducive to learning (Hundhausen 1993).

The implemented algorithm animation systems act as platforms to formulate and validate new hypotheses. Whilst all algorithm animation systems seek to aid students to better understand algorithms, markedly different approaches and methods are utilised to attempt to achieve the results[1]. Each system is designed to meet the requirements of a given learning environment or research focus.

Empirical testing of effectiveness of given systems are conducted in educational environments. Various methods are employed to identify and collect information, including pre- and post-tests to measure learning outcomes (Lawrence, Badre and Stasko 1994), and ethnographical studies to observe and gain feedback on students' perceptions of the systems (Hundhausen 2002).

Finally, supported by experiences gained from the abovementioned research foci, effective features and methods of integrating algorithm animation systems into educational usage can be identified (Saraiya, Shaffer, McCrickard and North 2004). This in turn promotes further research in the abovementioned research directions by guiding designs towards the formulation of a system for algorithm animation.

The following section outlines the role algorithm animations play in algorithm courses, and the issues leading up to the research focus.

[1] (Brown and Sedgewick 1984; Dann, Cooper and Pausch 2001; Rößling and Freisleben 2002; Tudoreanu 2003)

1.3 Relevance of the Investigation

In algorithm courses, students are taught to select particular algorithms from a range of alternatives. Students must also be able to justify the selection of the algorithms, and implement the algorithm in a programming context. The categories of algorithms typically studied include merging, sorting and searching algorithms (IEEE and ACM 2001).

One of the goals of teaching algorithms is thus to give the students the ability to select and apply algorithms appropriate to particular purposes, with strong emphasis on the issue of comparative efficiency and feasibility of implementation (IEEE and ACM 2001). A number of skills must be learnt by the students to achieve this result, including understanding the range of algorithms that address an important set of well-defined problems, recognising the strengths and weaknesses of each algorithm, and determining the suitability of the algorithms for any given scenario.

Various methods have been devised to address the problem of teaching algorithms and comprehension of their effectiveness, such as the use of graphical materials or demonstrations on laboratory computers. These methods, however, have limitations in their capability to demonstrate algorithms (Section 1.3.1). The limitations have stimulated the use of algorithm animations by educators as an instructional aid in their courses (Section 1.3.2). Background is provided of existing local research into integrating technological support tools into introductory algorithm courses, thus highlighting the context of algorithm animation as a new area of local research (Section 1.3.3). The relevant factors discussed in the section are then summarised (Section 1.3.4).

1.3.1 Demonstrating Algorithms

Educators and textbooks often make use of static visualisations, such as blackboard or textbook frame-by-frame illustrations, to aid in the teaching of algorithms and algorithmic concepts. The first set of images (Figure 1.2) shows an instructor presenting a once-off demonstration on the concepts of the Mergesort algorithm to students. The second image (Figure 1.3) shows an extract from an algorithm textbook, *Data Abstraction and Problem Solving with C++* (Carrano and Prichard 2002), illustrating and explaining a partial sequence of a Quicksort.

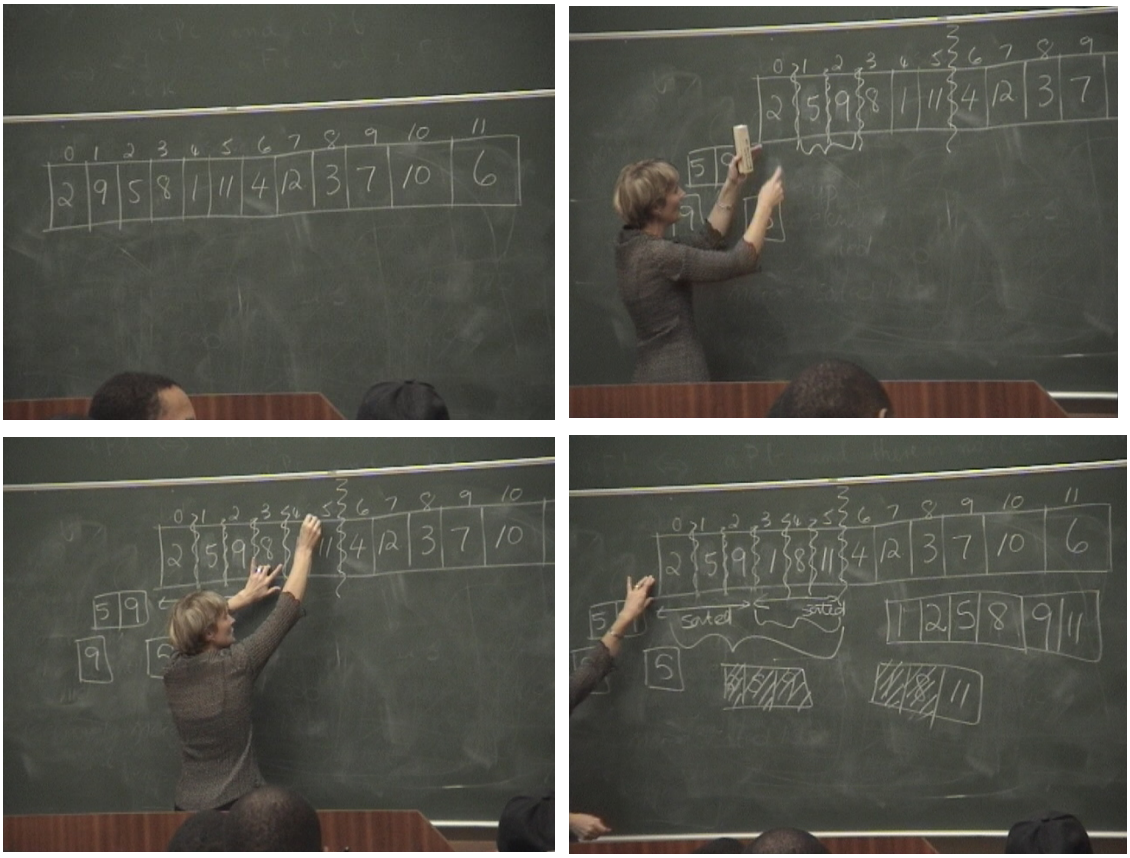


Figure 1.2: Once-off blackboard demonstration of a Mergesort in class.

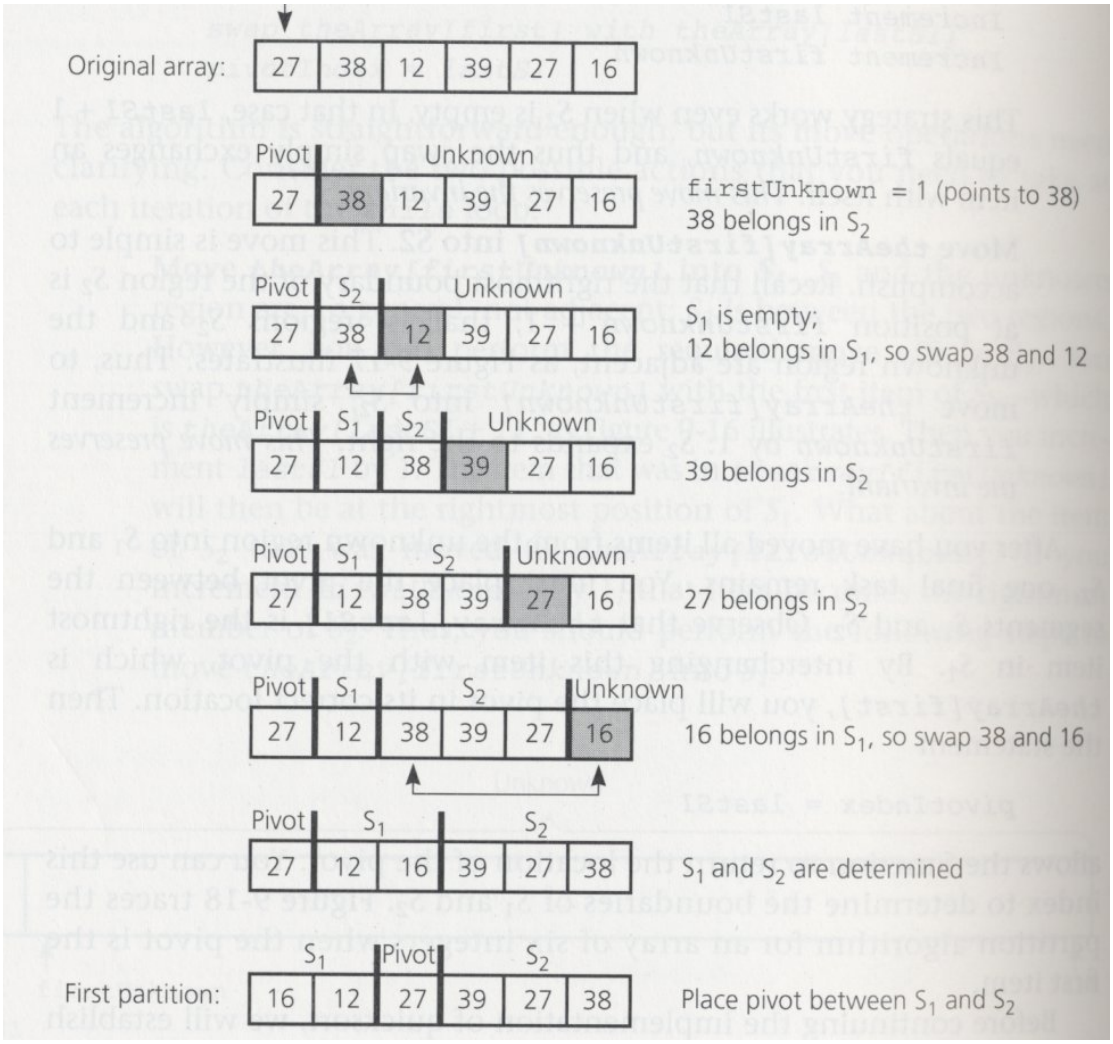


Figure 1.3: Textbook illustration of a sequence in Quicksort (Carrano and Prichard 2002).

Static illustrations are, however, not capable of capturing the dynamic movement of data and complex data structures, nor are the materials capable of exhaustively illustrating algorithm examples (Stern, Søndergaard and Naish 1999). Limitations of static illustrations are examined in this section, supported by Figures 1.2 and 1.3.

Once-off demonstrations, such as illustrations presented on a blackboard (Figure 1.2), mean that lecturers must constantly erase parts of the image and add new objects to reflect any changes made during each operation of the algorithm. This gives students less chance to absorb the material, since the constantly changing output of algorithms

mean that any work demonstrated cannot easily be noted down or reproduced for later self-study (Rößling and Freisleben 2000b).

Furthermore, mistakes can easily be made during classroom explanations by lecturers, since they must simultaneously act the role of a "virtual machine" by interpreting the algorithm in real-time, render the illustration, comment on and explain important concepts and events, and answer any queries posed by students (Hamilton-Taylor and Kraemer 2002).

Blackboard demonstrations and lecturer explanations also present another problem, as they can never be optimally paced to accommodate the entire class. At any average pace used, the smarter students are left bored and uninterested, whilst the weaker students are frantically taking notes, possibly without even fully understanding the concepts being discussed (Stern, Søndergaard and Naish 1999).

As illustrated in Figure 1.3, discrete steps of an algorithm are often omitted from textbooks due to space constraints. Students consequently have to figure out the missing details for themselves based on their own understanding of the algorithm. The historic nature of static materials also means that only preset scenarios are illustrated, thus not allowing students to test or enhance their understanding of an algorithm by trying different cases and examples (Kann, Lindeman and Heller 1997), reinforcing the learning thereof.

Evaluating and contrasting the performance and other characteristics of different algorithms, or that of an identical algorithm under different conditions are challenging due to the non-visible nature of the operations performed. Ironically, the power of

modern computers actually adds to the problem. Students implementing and executing different sort algorithms see the final result of the execution seemingly instantly. Without being given a chance to appreciate the effect, it is not surprising that students do not understand why or how one algorithm might be more or less efficient than another (Laxer 2001). Despite much discussion in class and in textbooks about the number of operations and the time per operation required to perform sorts, the students would still be left wondering as to how much worse a Bubblesort of an array really performs compared to a Quicksort (Rasala, Proulx and Fell 1994) of the same array. This is a significant drawback, since algorithm courses place a strong emphasis on the students' ability to analyse and compare the performance of sorting algorithms.

1.3.2 Use of Animation to Illustrate Algorithms

Algorithms are time-based in nature, consisting of elementary processes that are executed through time. Animations, as a medium of visual communication, are well suited to portray how the tasks of an algorithm are performed and how the state of its data structure evolves over time. Specifically, some sorting algorithms make use of iteration or recursion techniques to perform repetitive computations, which can be displayed more efficiently using motion pictures shown in algorithm animations (Baecker 1998). Furthermore, experiments have shown possible benefits in providing simultaneous display of algorithm animations to provide a contrast of algorithm performance.

Tudoreanu (2003) explains the effectiveness of algorithm animations in terms of how they aid in the viewers' cognitive economy. Cognitive economy comprises of reduction of cognitive load and user tasks, and increase of visualisation of information.

Firstly, the load on the cognitive system of the viewer is minimised by reducing the amount of information handled by the user and reducing user tasks which do not have direct relevance to the computations being observed. Secondly, cognitive economy attempts to maximise the visual information received by the viewers which are relevant to the study of the algorithm. While the two goals mentioned might seem contradictory (the first aims to reduce information load, whilst the second increases it), Tudoreanu argues that the design of effective algorithm animation systems involves a balance between the two factors.

Another approach explains that algorithm animations aid in the formation of the mental model due to the short cognitive distance between a concept and its corresponding visualisation (Bazik, Tamassia, Reiss and van Dam 1998). The more directly the visualisation matches the mental model, the more obvious and understandable it is to students. This allows them to focus on the ideas illustrated rather than having to put effort into disseminating the medium of presentation.

An empirical study has suggested that using algorithm animations, even without considering the effectiveness they have in aiding algorithm understanding, might possibly result in faster learning (Byrne, Catrambone and Stasko 1996). At the same time, it is pointed out that one of the advantages of algorithm animation over a lecturer demonstration is that, unlike a lecturer, the algorithm animation will illustrate its examples for as many times as needed, effectively allowing the students as much time as they require to understand the material.

A number of studies have observed that in anecdotal feedback, the students were unanimous in saying that the use of animations to learn sorting algorithms resulted in

a much more interesting and entertaining experience than traditional methods, and thus students were better motivated to learn and understand the coursework (Kann, Lindeman and Heller 1997; Rößling and Freisleben 2000b; Kehoe, Stasko and Taylor 2001). Learning from animations has thus made the learning of sorting algorithms an intrinsic motivator, in that students learn because they are interested to find out something, rather than an extrinsic motivator, where students learn in order to answer examination questions and pass the course. This is an important factor in improving the effectiveness of a learning experience (Alessi and Trollip 2001).

1.3.3 Overview of Existing Local Research Activities

The Department of Computer Science and Information Systems (CS&IS) at the Nelson Mandela Metropolitan University (NMMU) is actively conducting research aimed at increasing the throughput of students in algorithm courses (Calitz 1997; Greyling 2000; Cilliers 2005). The past and current research makes use of two approaches in attempts to achieve the objective, namely the identification of potentially successful students, and the modification of teaching models. The latter approach has led to the development of a number of experimental tools within the department (Cilliers 2005). The technological support tools developed are classified under specific categories depending on the area of research they support.

Technological support tools that support research on modifying teaching models undergo continuous development to incorporate new research topics and ideas. The tools have thus far targeted two areas, namely experimental integrated development environments (IDE) and visual programming languages. The research areas outlined in this section are presented in Figure 1.4.

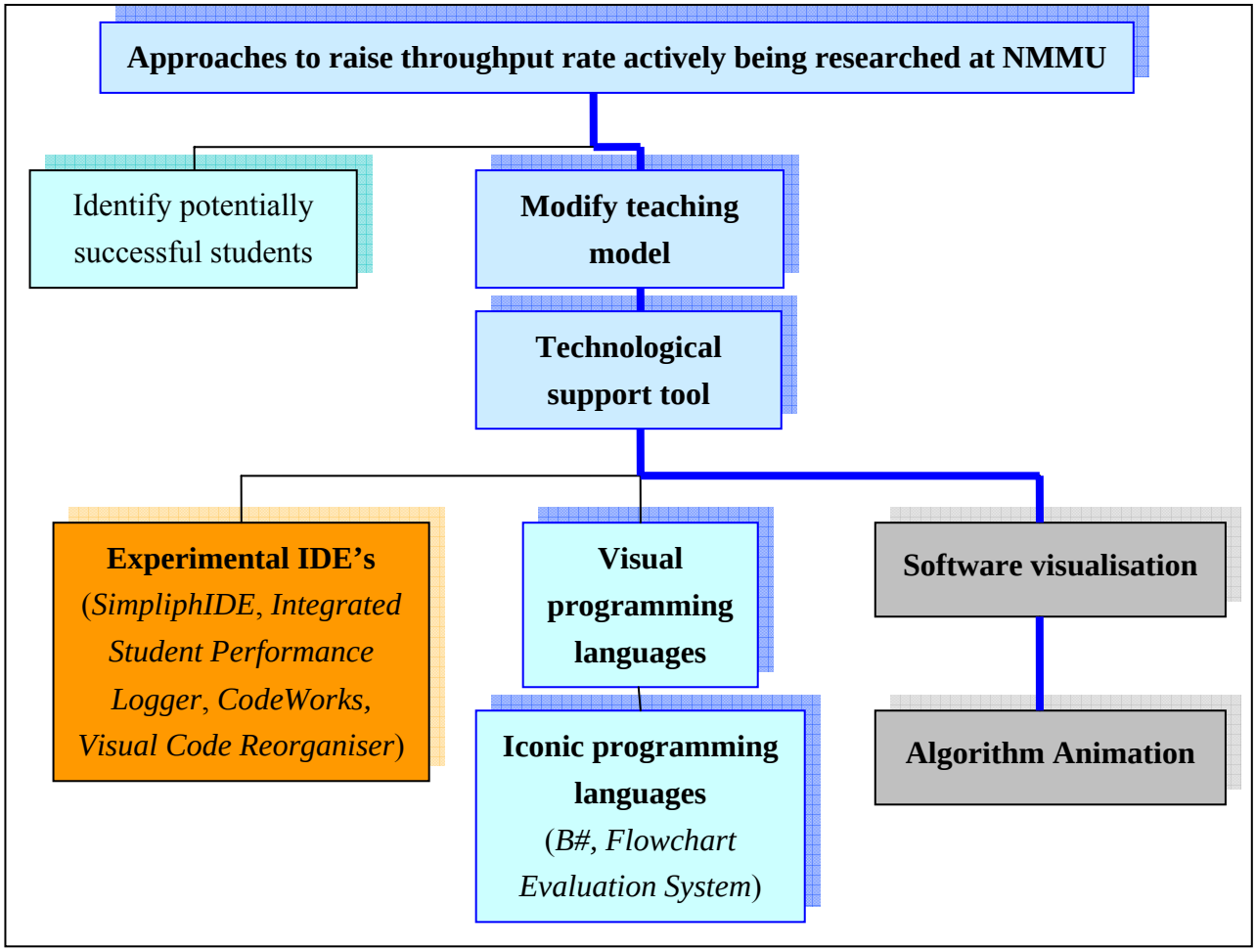


Figure 1.4: Hierarchical presentation of research in the department of CS&IS, NMMU – (adapted from Cilliers 2005)

Experimental IDE's include specific tools which are not available in commercial software development environments. The tools are designed to assist students in developing their algorithm syntax and logic skills, and allow instructors to capture and analyse data on the students' usage of the IDE's. The experimental IDE's developed by the Department are:

- SimpliphIDE (Christians 2003; De Jager 2004)

  A simplified version of the Borland© Delphi™ 7.0 IDE, designed to minimise complexities faced by novice programmers during program creation

- Student performance logging (Gamieldien 2003)

An event logging module designed to integrate with the SimpliphIDE system. This tool allowed for the capturing of selected programming activities of students who use the IDE.

- CodeWorks (van Tonder 2003)

  A lightweight Java™ IDE with integrated tools for building GUI Java applications. CodeWorks is the first IDE for Java to utilise the SpringLayout™ layout manager introduced in JDK 1.4.

- Visual Code Reorganisation Tool (Henning 2004)

  The system presents partially completed programs to the students, who are then expected to identify and manipulate missing code pieces to correctly complete the program.

Iconic programming languages are integrated into the initial phase of the introductory algorithm course to teach students introductory programming concepts. This approach allows the initial instructional focus to be placed on problem solving strategies rather than the notational mechanics of a given programming language. A number of interactive environments have been developed by the Department to make iconic languages more accessible to students:

- B# Iconic Language (Brown 2001; Thomas 2002; Yeh 2003; Cilliers 2005)

  The B# system allows students to create algorithm program solutions using iconic flowcharts, with visual links between relevant program source code and the flowchart icons. B# also integrates a virtual flowchart interpreter, allowing students to execute and trace flowchart programs which they have created.

- Flowchart and Textual programming evaluation System (Mamtani 2004)

  The system is used to construct questionnaires regarding the learning preference of students' between iconic and text-based programming languages. It also presents the questionnaire and captures the responses for later analysis.

The current study is on a third type of technological support tool which may be integrated into the NMMU's existing teaching model, namely in the area of algorithm animation. Algorithm animation, a subset of software visualisation, is a discipline which supports higher level understanding of algorithms by employing visual displays of algorithm concepts (Price, Baecker and Small 1998).

1.3.4 Concluding Remarks on Relevant Factors

Instructors traditionally employed blackboards and textbook diagrams to help visualise algorithms and data structures to students, which provides for more intuitive learning compared to simply reading the algorithm code. The discussion in Section 1.3.1 presents a number of limitations associated with this method:

- Once-off blackboard drawings are difficult to copy or reproduce for later reference;
- Instructors may err while attempting to simultaneously draw diagrams on-board and present the lecture;
- Textbook illustrations cannot represent all algorithm steps exhaustively, nor present novel scenarios which students are interested in, and
- Illustrations do not demonstrate the performance and characteristics of algorithms.

The utilisation of animations is presented as a possible method to address such limitations, offering advantages over static illustrations through a number of factors highlighted in Section 1.3.2:

- Motion images are more efficient at illustrating iterative operations of algorithms;
- Animation reduces the cognitive load on the student whilst increasing relevant visual information;
- Animations are more accessible to students for review outside the classroom, and are not limited on the number of times they may be used, and
- Animations are more enjoyable than text or static illustration explanations, thus offering an intrinsic learning motivator.

Section 1.3.3 outlines existing NMMU research, which aims to improve existing teaching models of algorithm courses through the use of experimental IDE's or iconic programming languages (Figure 1.4). The limitations of static illustrations (Section 1.3.1) serve to underscore the potential benefits of using algorithm animations in teaching algorithm courses (Section 1.3.2). These discussions thus support a further area of research, namely the use of algorithm animations as a technological support tool to complement current teaching methods. This forms the focus for the current investigation.

1.4 Focus of the Investigation

Research into algorithm animations is to be conducted in a similar approach to that of the previous areas of work (Figure 1.1). The research will provide a foundation for the proposal of an algorithm animation framework. A demonstration of the framework's

effectiveness by means of the implementation of a prototype algorithm animation system will act as a vehicle for conducting further research within the university. This section first outlines the goals and objectives of the research (Section 1.4.1). The research scope is provided to define the research areas to be covered (Section 1.4.2). Various research questions are then posed to guide the investigation (Section 1.4.3).

1.4.1 Goals and Objectives

Since the demonstration of "Sorting out Sorting" (Baecker 1981), much research has been done in the studying of the effectiveness of algorithm animation tools in teaching (Hundhausen 1997; Hundhausen, Douglas and Stasko 2002). Experimental evaluations have shown that the use of algorithm animations in learning environments has had positive effects on the students' understanding of algorithms (Hansen, Narayanan and Schrimpsher 2000; Hundhausen, Douglas and Stasko 2002).

The purpose of this study is the design of an extensible algorithm animation framework and the evaluation thereof through the implementation of a prototype system based on the framework design concept. Algorithm animations of sorting algorithms will be created using the implemented prototype system as part of a case study to evaluate the framework design. The case study will be based on the quadratic and $O(N \log N)$ sorting algorithms commonly taught in introductory algorithm curricula (IEEE and ACM 2001).

The algorithm animation framework will be designed to support a specific list of pedagogic requirements. A preliminary list of requirements is identified based on their potential effectiveness in complementing the learning strategy of the students. An

extant system evaluation will be performed by using the requirements identified. The evaluation will aid in deriving the final list of requirements to be supported by the framework.

Extensibility of the framework will be supported through an independent layered design. This will allow functionality and case study extensions to be made to the framework.

In support of the development of the prototype, an extensive literature review will be performed on extant algorithm animation systems and methodologies for creating algorithm animations. The prototype system will enable educators to create customised, interactive algorithm animations. The algorithm animation system will employ visual elements to help students understand algorithms, and analyse differences among algorithms through exploratory learning and interaction with the algorithm animation. This facility will allow students to directly compare and contrast algorithms utilising different scenarios. This feature is supported by having the system run multiple animations in parallel, thereby letting students contrast performance differences visually.

1.4.2 Scope

The study will focus on designing a framework for algorithm animation, and evaluating the framework through the implementation of the prototype. The study

includes research into identifying pedagogically effective features of algorithm animation systems (Chapter 3). A final system requirements list will be drawn up based on the identified features. The framework will then be designed to support the requirements.

The framework design and prototype implementation will communicate information through the visual channel (visual metaphors, motion, colour). Audio elements are briefly introduced as a technique for complementing visual displays in algorithm animations; however, the technique falls outside the scope of the project and will thus not form part of the framework design. The prototype will be designed to run on the PC client platforms within the NMMU CS&IS department, on a lecture hall data projector or in a closed laboratory environment.

The main deliverables of the study include an algorithm animation framework design, a prototype system implementation based on the framework, and animations of the sorting algorithms created using the prototype. The framework will not consider support for interactive dialogs, such as presenting interactive questions and quizzes. The evaluation of the framework will focus on the framework's capability to provide for the requirements of the system. Therefore a usability evaluation of the system is considered to fall outside the scope of the current study. The documentation will also focus on providing a list of algorithm animation requirements and an extant system evaluation, which form a key complementary deliverable of the research in support of the framework and prototype.

The case study consists of the building of algorithm animations based on sorting algorithms taught in NMMU's introductory and intermediate algorithm courses,

namely the Bubblesort, Insertion sort, Selection sort, Mergesort and Quicksort algorithms.

1.4.3 Research Questions

A number of research questions have been identified to guide the investigation. Table 1.1 lists the questions, the method to be used to answer each question, as well as the chapter which will address the questions.

	Research questions	**Method(s)**	**Relevant chapter**
1.	What is software visualisation?	Literature Study	Chapter 2
2.	What is algorithm animation?	Literature Study	Chapter 2
3.	What elements are used to form an algorithm animation?	Literature Study	Chapter 2
4.	How are algorithm animations used in teaching and learning algorithms?	Literature Study	Chapter 2
5.	What are the issues to be considered in	Literature Study	Chapter 3

	the design and specification of an algorithm animation framework?		
6.	What are the criteria for the design of effective algorithm animation systems?	Literature Study	Chapter 3
7.	How do extant algorithm animation systems match the criteria?	Literature Study	Chapter 3
8.	What does an algorithm animation framework look like?	Framework Proposal	Chapter 4
9.	What are the implementation issues faced by developers of algorithm animation systems?	Iterative Prototyping	Chapter 5
10.	How effective is the proposed framework?	Iterative Prototyping	Chapter 5
11.	How does the algorithm animation prototype developed match the identified measurement criteria?	Heuristic Evaluation	Chapter 5
12.	What are the limitation and contribution of the framework and prototype?	Conclusions and Summary	Chapter 6

Table 1.1: Research questions of the dissertation

1.5 Structure of Dissertation

This dissertation consists of six chapters. Figure 1.5 provides a research roadmap to show the flow of information within the dissertation. The figure illustrates the relevance of each chapter's investigations and contributions in relation to its subsequent chapters.

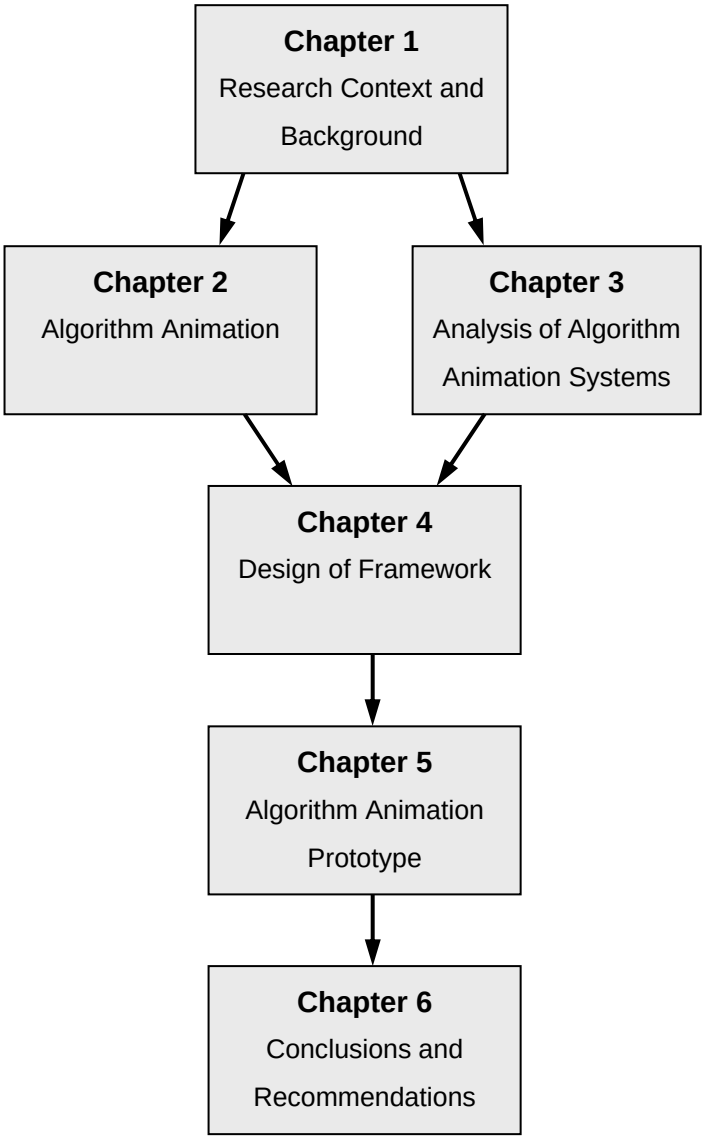


Figure 1.5: Research Roadmap

Chapter 1 provides background information on the problem domain, and a discussion of the factors leading up to the current research. The objectives and scope are then formulated in the context of the background discussion. A number of research questions are proposed to guide the investigation.

Chapter 2 provides an overview of algorithm animation. The chapter will investigate the context of algorithm animation within software visualisation. Discussion is

provided to define the concept of algorithm animation, and the various techniques of communicating information it employs.

Chapter 3 focuses on information related to the design and implementation of an algorithm animation system. It will discuss the various user types and system components of an algorithm animation usage environment. This is followed by an investigation into paradigms for linking algorithms to visualisations. A list of requirements for pedagogically effective systems is established through literature and extant system review.

Chapter 4 provides a detailed discussion of the proposed framework design. Motivations are provided for the selection of visual paradigms and the framework structure. Each of the framework components are examined in detail.

Chapter 5 documents the prototype implementation of the framework proposed in Chapter 4. The implementation methodologies decisions are motivated. Discussions are provided on the implementation of system components, system interfaces and the sorting algorithm animation case study. Observations gained from the implementation are also noted.

Chapter 6 concludes the dissertation by highlighting the theoretical and practical achievements, contributions and implications of the research. A number of future research projects based on the dissertation are also identified. A bibliography and appendix is provided at the end of the dissertation.

Chapter 2

Algorithm Animation

2.1 Introduction

Chapter 1 introduced the focus of the current investigation, which is the creation of an algorithm animation framework. The presentation of algorithms in an animated form harnesses the human visual perception to absorb and process visual information (Roman and Cox 1992), thereby decreasing the cognitive load of students learning and analysing algorithms (Tudoreanu 2003).

Questions thus arise concerning how an algorithm animation is defined, what elements they utilise to convey information, and how they are generally used. In order to understand the requirements for the proposed algorithm animation framework and sorting algorithm case study, a number of issues are discussed. The chapter first identifies the classification of algorithm animation in the broader context of software visualisation (Section 2.2). This is followed by defining the concept of algorithm animation, and the elements that make up an algorithm animation (Section 2.3). A discussion on a number of environments where algorithm animations are used follows

(Section 2.4). Section 2.5 discusses the concept of illustrating algorithms using different levels of abstraction.

2.2 Software Visualisation

Computer software is becoming increasingly difficult to create and to understand due to its increasing complexity. Software engineers thus develop and employ a number of approaches to enhance the comprehensibility of software. One approach is software visualisation, which focuses on improving the representation, presentation and appearance aspects of a program (Baecker and Price 1998). Background is first provided on software visualisation (Section 2.2.1). An overview of the taxonomy of software visualisation is then provided to place the context of algorithm animation in the field of study (Section 2.2.2).

2.2.1 Background and Definition

> Software Visualisation is defined as "*the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software"* (Price, Baecker and Small 1998)

Software visualisation thus focuses on presenting the bigger picture of a program system by making software visible using graphical representations. Software visualisation modularises information hierarchically to enhance structural

understanding, which in turn supports the process of large-scale software development and maintenance (Eick 1998; Marcus, Feng and Maletic 2003).

In brief, software visualisation involves the using of a variety of sensory inputs to cause the user to form a mental picture of logical structures or concepts, such as software source code (Price, Baecker and Small 1998). Familiar examples of software visualisation tools include Computer-Aided Software Engineering (CASE) tools (Chikofsky and Rubenstein 1988) and Nassi-Shneiderman (1973) diagrams.

2.2.2 Taxonomies of Software Visualisation

A number of taxonomies are available which characterise and categorise software visualisation using various attributes. These taxonomies serve to identify types of software visualisation suited to particular environments, based on the aspect of a program's information which is displayed or revealed by the visualisation.

Myers (1990) presented a basic taxonomy categorising software visualisation systems based on the level of abstraction (code, data or algorithm) and degree of animation (static or dynamic) of the visualisation. However, the taxonomy mainly focused on what the visualisation shows, passing over the issues relating to the design and construction of the visualisations. The shortcoming of Myer's work was addressed by the taxonomy presented by Roman and Cox (1993). This taxonomy classified software visualisation systems along five axes, namely *scope*, *abstraction*, *specification method*, *interface* and *presentation*.

- *Scope* defines the aspect of the program visualised. In other words, which part of the program is enhanced or represented visually. The focus can be placed at the

lower level, such as program source code statements, or at a higher level, such as program behaviours.

- The level of *abstraction* of a visualisation defines the kind of information visualised. Visualisations can be a direct representation of a program by directly mapping to a particular aspect of the program, or a synthesised representation which involves visualising the program based on derived program data that has no explicit representation. The concept is discussed in Section 2.5.
- The s*pecification method* describes the aspects of a program to be extracted, and how the visualisations are to be displayed. The method also affects the level of automation and design flexibility in creating visualisations. This issue is further explored in Section 3.3 and Section 4.2.1.
- *Interface* characterises how visual information is presented to the viewer in terms of the tools available for the presentation, such as the visual actions and graphic objects (Section 4.8.1). Another interface issue is the level of interactivity available for the viewer to control various aspects of the visualisation display. This forms an important issue, further discussed in Section 3.4.
- *Presentation* defines the methods used by the visualisation to communicate information. Presentation focuses on how the visualisation is designed using, amongst others, graphical objects, motion and colour (Section 2.3). The aim is to allow the viewers to interpret the visualisation to gain or complement their understanding of the program being visualised.

The abovementioned taxonomies focus on defining the characteristics of extant software visualisation systems. However, no definitive categorisation of the different classes of software visualisation has been offered. Stasko and Patterson (1993)

presented a taxonomy which focused on placing each of the classes of software visualisation into definitive categories. A number of classes of software visualisation are presented along with their characteristics in Table 2.1, namely data structure display, program state visualisation, program animation, algorithm visualisation and algorithm animation. These terms are explicitly presented to highlight the context of algorithm animation. Furthermore, similar terms, such as *data structure*, *display*, *program*, *algorithm*, *visualisation* and *visualisation of algorithm*, are often used throughout latter chapters to discuss algorithm animation framework related issues. The table will serve to separate such terms from the actual definition of software visualisation classifications. The criteria for Stasko and Patterson's taxonomy (1993) can be mapped to Roman and Cox's taxonomy (1993) as follows – *Aspect* to *Scope*, *Abstractness* to *Abstraction*, and *Automation* to *Specification Method*. The *Animation* criterion is discussed in Section 2.3.1.

	Aspect	**Abstractness**	**Animation**	**Automation**
Data Structure Display	low	Low	low	high
Program State Visualisation	medium	Low	low	high
Program Animation	medium	medium	medium	high
Algorithm Visualisation	high	High	low	low
Algorithm Animation	**high**	**high**	**high**	**low**

Table 2.1: Software Visualisation terms (Stasko and Patterson 1993)

Stasko and Patterson's taxonomy, represented in Table 2.1, defines algorithm animation as a *high aspect* visualisation focused on demonstrating program behaviour by employing *animation* techniques. Algorithm animation demonstrates algorithms at a *high level of abstraction*, focusing on particular events of interest rather than all

program activities. The process of designing algorithm animations has *low propensity for automation* due to the inputs required from the animation creator. The next section discusses algorithm animation in more detail.

2.3 Algorithm Animation

Algorithm animation is defined as "*the process of viewing the underlying logic of a computer algorithm through a series of pictures that are strategically chosen to illustrate the algorithm in execution*" (Hundhausen 1993). A further explanation of algorithm animation is to describe it as the dynamic visualisation of high level abstractions describing software (Price, Baecker and Small 1998), used to communicate the workings of algorithms by graphically or aurally representing its fundamental operations (Brown 1998).

Algorithm animation is thus concerned with the representation of specific algorithms and their characteristics. It attempts to encourage understanding of algorithms by visualising their runtime behaviour and their properties and consequences thereof (Ball and Eick 1996). The algorithm behaviours are often represented in abstract, artificially highlighting or concealing certain aspects and activities of the algorithm to enhance its explanatory value.

The presence of the word *visual* in software visualisation can be misleading, since software visualisation is not restricted only to visual elements. The primary meaning of visualisation is the process of forming a mental image of concepts which have no visual presentation, thus visualisation includes both visual and aural elements (Price, Baecker and Small 1993). Algorithm animations employ two channels of

communication, the visual channel, and the acoustic channel. The visual channel (Section 2.3.1) uses graphical shapes, quantitative presentations and colours whilst the acoustic channel (Section 2.3.2) uses pitch, volume and moving spatialised sound to convey information (Baloian and Luther 2001). The audio element forms part of the discussion on defining algorithm animation. However, it will not be considered in the remainder of the dissertation due to project scope constraints (Section 1.4.2).

2.3.1 The Visual Aspect

Algorithm animation depicts the logic of an algorithm by visualising two aspects of the algorithm - the data structures and the operations which manipulate the data structures. The following briefly describes the primary components which form the visual presentation of an algorithm animation, namely *visual metaphors*, *animation* and *colour*.

Visual metaphors

Visual metaphors can be drawn from specific application domains, or non-technical symbols which are familiar or easily inferable by the animation viewer (Giannotti 1987; Jeffery 1998; Arik 2005). The visual objects used to represent data structures should be as self-explanatory as possible. Educational algorithm animations favour the use of data structures that can be represented visually in an intuitive manner (Gloor 1998).

As an example, the most commonly employed metaphor in demonstrating a list of elements is to map each element to a rectangle, with the rectangle's height being proportional to the element's size (Baloian and Luther 2001).

Figure 2.1 gives an example of good and bad visual metaphors. The good metaphor presents the data using numerical elements, for which relative size can easily be induced. In addition, numerical elements have a value of magnitude, and thus map well to the rectangle visual metaphor. The bad metaphor presents data as letters of the alphabet, for which the relative size are more difficult to induce. Alphabetical values also lack any form of dimension with which to effectively present as a visual metaphor.

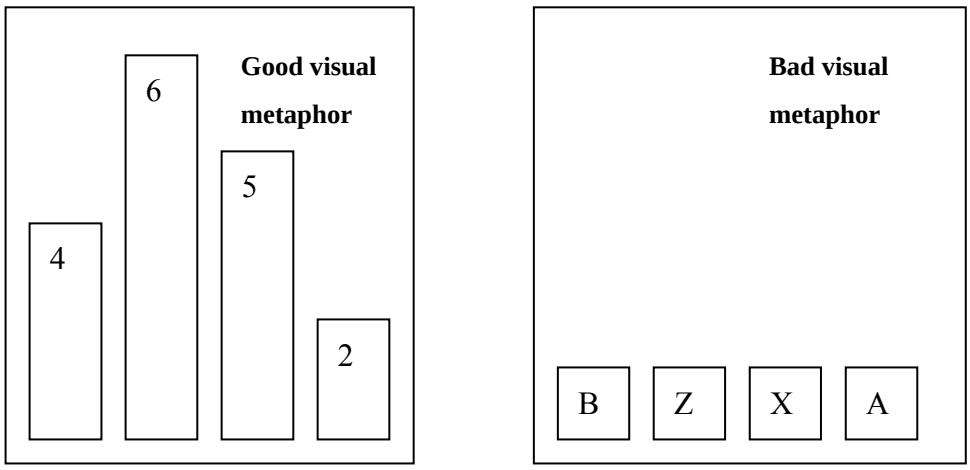


Figure 2.1: Examples of good and bad metaphors in algorithm animations (Gloor 1998)

More original and unconventional forms of visual metaphors can be applied to construct algorithm animations, providing that the metaphors support the concept being illustrated. An unconventional metaphor involves the use of metaphoric content beyond those conventionally found in textbook illustrations. The use of unconventional metaphors (Figure 2.2) has been shown to make an algorithm

animation more enjoyable to students. More interesting algorithm animations can thus better capture the students' attention, resulting in improvement in learning and comprehension of the algorithm illustrated (Hübscher-Younger and Narayanan 2003).

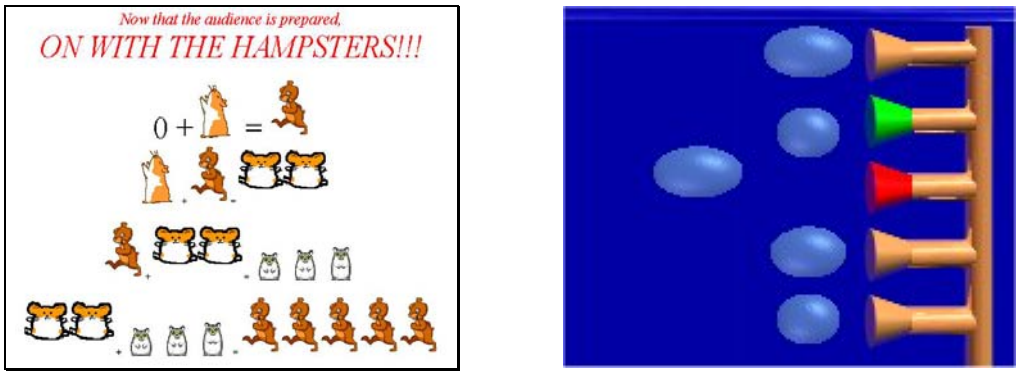


Figure 2.2: Example of unconventional metaphors: Fibonacci Hamsters (Hübscher-Younger and Narayanan 2003), and "Bubble"-Sort (Barbu, Dromowicz, Gao et al. 2001)

Animation

Traditional methods of demonstrating algorithm processes have involved the use of static images. A series of images can be created to show changes in the data structure after each step in the algorithm's operation. However, updates in static images take place instantaneously, making it difficult for learners to keep track of the operations which are occurring.

Animation is a technique for conveying visual information through motion, which is a perceptively efficient, low cognitive overhead visual dimension that is well suited for expressing change and activity (Bartram 1997). Based on the motion applications taxonomy (Bartram 2001), various purposes for utilising motion with specific relevance to algorithm animation are identified, namely awareness, emphasis and transition. *Awareness* is concerned with attracting and directing visual attention to a

specific viewing area. *Emphasis* is achieved by drawing attention to a particular visual object or process. *Transition* guides viewers through intermediate processes between non-temporal states.

Algorithm animation makes use of smooth, continuous animation to illustrate the transition which occurs between each state of an algorithm, such as two values being exchanged by seeing two representative blocks moving towards each other's original positions. This method of portraying each individual algorithm operation allows for the viewers' visual systems to easily perceive and track changes (Stasko 1998b). Smooth animations are especially effective in illustrating the processes of more complex ideas and algorithms (Sonnier and Hutton 2004).

Colour

Colour is capable of communicating large amounts of information to the viewer efficiently (Brown and Hershberger 1991). The role of colour in algorithm animations is especially useful since it provides another visual dimension to help illustrate concepts. Colour is used in algorithm animations in five ways (Brown and Hershberger 1998a):

1. Encoding the state of data structures – Colour is an effective tool for encoding state information, since few pixels and less space are needed as compared to using distinct shapes.
2. Highlighting activities – Alterations of colour can be used to focus viewers on temporal or transient operations taking place.

3. Uniting multiple views – Algorithms and data structures can often be represented using different views. Colours are useful in visualising corresponding features and thus help integrate the views.
4. Emphasising patterns – Colours used to encode information can collectively highlight certain trends and commonalities of a visualisation.
5. Making history visible – Colour sequences can be used to show an algorithm's history by representing a linear time order of past events or states using colour hues.

Figure 2.3 provides an example of the concepts described above using two views of a Quicksort algorithm[2], which uses a divide-and-conquer approach to sorting. The pivots which separate each sub-list are encoded in red, items which currently form the ignored sub-list are encoded in black, whilst the list currently being processed is encoded in blue. As the algorithm seeks for a new pivot within each sub-list, the examined item is highlighted in yellow. The unified colour scheme means that viewers can see how the two different views correspond in turns of operations in progress. The emphasis of the unique Quicksort pattern is clearly demonstrated in the corresponding dot plot, where each grouping of sub-list items (encoded in black) are boxed in by pivots (encoded in red) on each side.

[2] The design of these views form part of the implementation discussion in Chapter 5. Each of the concepts were utilised except for using colour hues to make history visible. The case study algorithms were unsuited for utilising such a colour technique, and thus Figure 2.3 will not illustrate the technique.

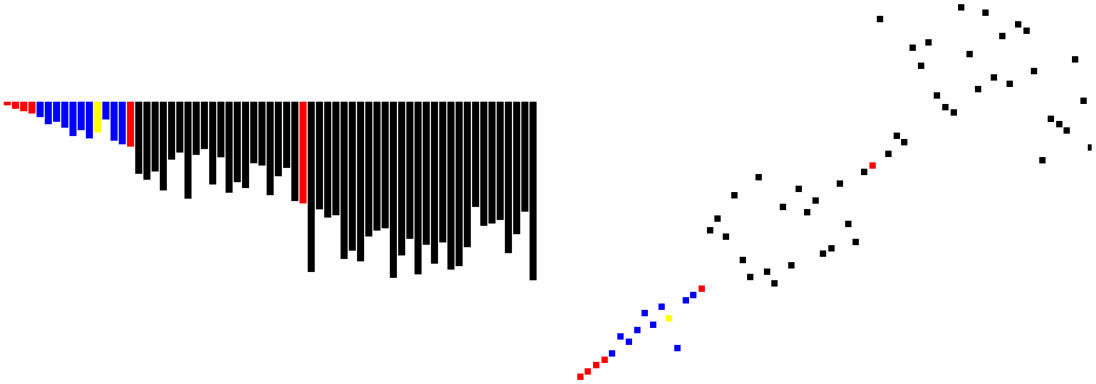


Figure 2.3: Example of colour usage techniques in algorithm animation

2.3.2 The Audio Aspect

Sounds are usually employed as a supporting element to imageries in animations, and are useful for conveying information which cannot be or are difficult to visualise. In support of visual presentations, sounds can help reduce visual clutter by providing an alternative medium for information presentation (Brown and Hershberger 1991). Vickers and Alty (2003) argue that while vision provides excellent spatial perception, auditory senses can absorb many properties simultaneously and in considerable volumes.

The element of sound can be used in algorithm animations to reinforce the visual elements, convey patterns, and signal exceptional conditions (Brown and Hershberger 1998b). Whilst researchers have looked into ways to replace visuals with audio or increase the use of audio in algorithm animations in general, this aspect still receives considerably less attention due to the difficulties of mastering the medium (Baloian and Luther 2001).

2.4 Algorithm Animations in Instructional Environments

Whilst the earliest algorithm animation, "Sorting Out Sorting" (Baecker 1981), was designed purely as a demonstrative tool, much research has since taken place in developing new algorithm animations and finding new ways to integrate them into the learning environment. Algorithm animations are used for demonstration in lectures, exercises in the computer laboratory, and remote viewing over the internet.

2.4.1 Lecture Demonstrations

Algorithm animations, with their illustrative capabilities, make for an ideal replacement for the typical in-class blackboard or slide demonstration (Rößling and Freisleben 2000b). The use of algorithm animations as a presentation aid in the classroom makes the lecturer's task of explaining the concepts of the algorithms easier for a number of reasons. The instructor can concentrate on explaining algorithm concepts without having to render and re-render the blackboard, and examples can be re-illustrated with minimal effort. When algorithm animation systems are used in lectures, different scenarios can be explored with minimal fuss. The use of algorithm animations also benefits students' learning due to the ability of animations and graphics to capture attention and maintain interest on the algorithm being studied.

2.4.2 Laboratory Usage

Integrating interactive algorithm animations into laboratory computers gives students the opportunity to further their understanding of the course work outside of the classroom. Students can interact with the animations at their own pace, performing

activities such as customising the data structure used, specifying alternative animation views, or analysing algorithms empirically (Gloor 1998). In addition, the students are able to perform these tasks as often as is required to achieve understanding of the algorithms taught.

2.4.3 Web-based Algorithm Animation

A number of algorithm animation systems have been designed to work over the internet architecture, typically implemented in the Java™ platform (Brown and Raisamo 1997; Dershem and Brummund 1998). These systems are often made available on public domains and thus provide the same benefits as the interactive algorithm animations in a laboratory environment. In addition, web-based systems are platform-independent, thus increasing their ease of accessibility to students.

2.5 Abstract Representations in Animation Algorithms

Algorithms are dynamic sequences of actions which, although implemented in a programming language, still make use of abstract concepts. As seen from examples of the sorting algorithms in Appendix A, the actual code of the algorithm and the accompanying explanation are presented at different levels of abstraction.

Brown (1988c; 1998) defines the content of an algorithm animation as either being *direct* or *synthetic*. *Direct* content shows isomorphic pictures representing the data structure of a program. *Synthetic* views show operations which change data or abstractions of the data, rather than map directly to any program variables. To be effective, animations must typically use a mixture of both techniques of presentation.

The issue of constructing direct and synthetic animation views are examined further in the visualisation paradigm discussion in Section 3.3.

In creating algorithm animations, the visualisation designer (Section 3.2) must consider the level of abstraction to display to allow students to see the operations of the algorithm at work. Detailed views may show how values are stored and moved around, while generalised views present the bigger picture, such as patterns or performance trends of the algorithm (illustrated in Section 5.5). Showing an animation with too much detail will lead to distractions and irrelevant information been shown, whilst abstracting away too many details may hide the important execution processes of an algorithm (Wilhelm, Müldner and Seidel 2001). Although the construction of algorithm animations can be supported by algorithm animation frameworks, the level of abstraction of the animation is still very much influenced by how the visualisation designer sees and chooses to express an algorithm (Bazik, Tamassia, Reiss and van Dam 1998; Fleischer and Kučera 2002). The high level of abstractness also provides designers with a certain degree of artistic freedom in creating algorithm animations (Stasko and Patterson 1993).

2.6 Conclusion

An overview of several taxonomies has shown methods of categorising software visualisations. Algorithm animation is defined as a form of software visualisation which animates a customised high abstraction view of algorithmic operations, which convey information through three visual elements, namely visual metaphors, animation and colour.

A brief discussion was presented on the possible uses of algorithm animations in an instructional environment. This leads up to the discussion, in Chapter 3, of the various key roles of such algorithm animation environments, which consist of the algorithm animation components and algorithm animation system users. The previous discussion highlights the fact that the high-level concepts of each algorithm often do not necessary correspond to its low-level source code. Thus, the animation creator must decide on the concepts and operations to be animated, and in how much detail, in order to allow effective transfer of knowledge.

This chapter has provided an overview of the various aspects of algorithm animation, including its context and definition, its methods of communicating information, and its uses. The following chapter will focus on functional issues relating to the design of algorithm animation systems.

Chapter 3

Analysis of Algorithm Animation Systems

3.1 Introduction

Students are faced with the challenge of comprehending the abstract concepts of algorithms, such as operational procedures and performance characteristics, and applying the concepts successfully to solve computational problems. The contributions offered by algorithm animations as a technological support tool, and the techniques used by algorithm animations for demonstrating algorithm concepts were highlighted in Chapters 1 and 2.

An algorithm animation framework is regarded as a generic design which is capable of supporting the requirements of an algorithm animation system. An algorithm animation system is intended to meet the needs of different user types whilst conforming to the concepts and goals of the framework design. Each component in the system is expected to provide technical features and functionalities which will allow various users to perform their tasks within the system (Rößling and Freisleben 2002). The components and user roles which make up an algorithm animation system are discussed in Section 3.2.

Various high-level paradigms have been devised in order to connect an algorithm to its related visual representation, with each paradigm having a number of available methods of implementation. The selection of paradigms will affect the design and characteristic of the framework. Each of the paradigms and associated methods of implementation are first examined in Section 3.3. The next chapter will then evaluate and select a paradigm as part of the framework design (Section 4.2.1).

A literature study identifies features that increase the instructional value of algorithm animation systems. The identified features are organised into an evaluation criteria for analysing algorithm animation systems (Section 3.4). This is then followed by an overview of a selected number of extant algorithm animation systems utilising the evaluation framework (Section 3.5). Based on the information gathered, a comparative study is performed to characterise the features of the systems and compile a list of requirements to be supported by the proposed framework (Section 3.6).

3.2 Algorithm Animation System - Users and Components

An algorithm animation system contains a collection of components which promote the efficient production of algorithm animations for the creators, and easy accessibility of the animations by the students. When designing an algorithm animation system, it is important to understand the role of each component within the final system, and the user types served by each component or collection of components.

Figure 3.1 illustrates the interactions among the system components and user types. The software visualisation software developer implements each of the algorithm animation system components. The algorithm programmer and visualisation tool developer implement classes within the algorithm repository and graphical repository, respectively. The visualisation designer constructs algorithm animations using the classes in the algorithm and graphical repository. Instructors and students utilise the learner interface and animation player to view and control the constructed animations.

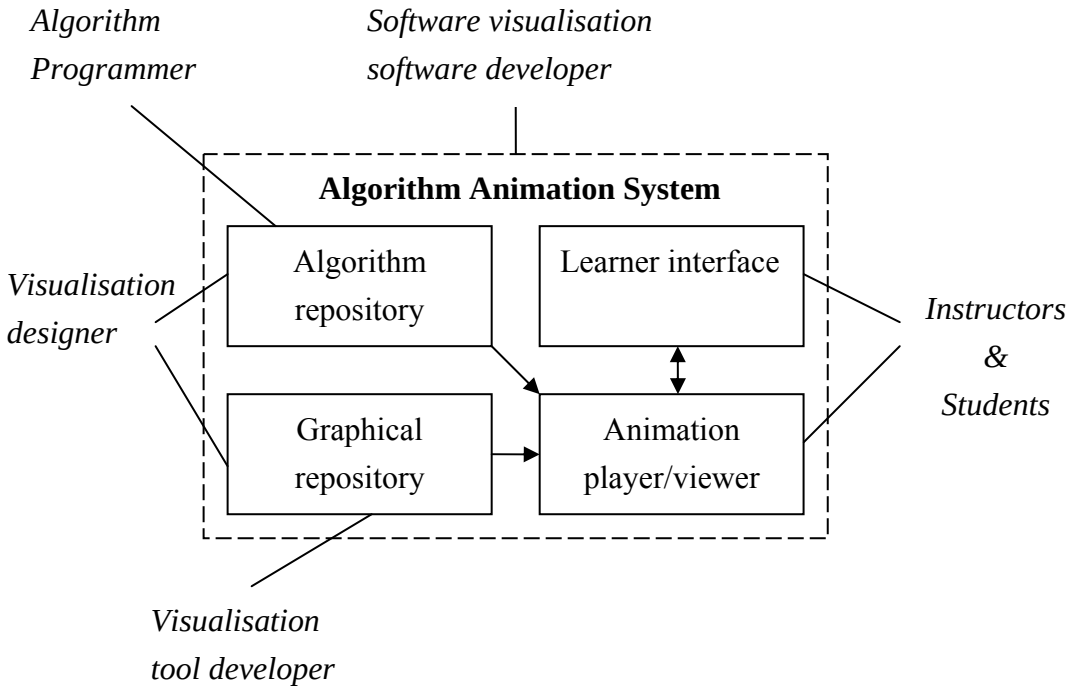


Figure 3.1: Interaction among system components and users

In this section, an investigation is performed to understand the role of the different user types within an algorithm animation usage environment (Section 3.2.1). The functionalities of the core components of an algorithm animation system are then discussed (Section 3.2.2).

3.2.1 Users within the Context of Algorithm Animation

A number of users are recognised in the context of using algorithm animations in a learning environment. Each of these six user types is briefly discussed, based on the roles identified by Price *et al* (1998) and Naps, Rößling, Anderson *et al* (2003).

- The *software visualisation software developer* is the user who implements the framework into a functional system, thus allowing the given framework to effectively support the activities of other users.
- The *algorithm programmer* is responsible for creating the algorithm which will be animated by the system. Depending on the method of animation creation used by the framework, the programmer may or may not need to know that the algorithm will be visualised.
- The *visualisation tool developer* creates the tools required to give a visual representation to program content, such as algorithms and data structures. The tools include visual elements (Section 2.3.1) and the mechanisms used to link program content to the visual elements (Section 3.3).
- The *visualisation designer* takes the algorithm to be studied, identifies the abstract concepts which require visualisation, and maps these concepts to an animated visual presentation. The visualisation of algorithms is done by utilising the visualisation tools provided in the system. The visualisation designer must possess proficient understanding of an algorithm in order to create animations which will effectively support the comprehension of the algorithm.
- *Instructors* integrate the use of the algorithm animations into their teaching materials to aid students in their understanding of algorithms.

- *Students* view and interact with the algorithm animation created by the visualisation designer, with the aim of enhancing understanding of the algorithm under study.

This overview of the different user types in the instructional use of algorithm animations provides an understanding of the users' expectations of an algorithm animation system. The components which support the user types are discussed in the following section.

3.2.2 Components of an Algorithm Animation System

The common mechanisms within an algorithm animation system are derived based on a number of examined extant designs obtained from the literature study. The mechanisms discussed consist of the algorithm repository, graphical repository, animation player/viewer and learner interface.

The *algorithm repository* stores the algorithms which can be expressed in an animated format (Baker, Cruz, Liotta and Tamassia 1996; Döllner, Hinrichs and Spiegel 1997). Depending on the techniques used by the system to identify and visualise the relevant concepts and data structures, the algorithms in the repository could either consist of the original, unedited source code, or code annotated with specific output or event calls to highlight interesting events.

The *graphical repository* stores visual objects and animation functionalities which can be utilised by the algorithms, either directly through API-type calls, or indirectly

through pre-scripted calls, to give visual representation to the algorithms (Rößling, Schüler and Freisleben 2000; Najork 2001).

The *animation player/viewer* makes use of the graphic components within the graphical repository, and the operations from the algorithms in the algorithm repository, from which algorithm animation may be dynamically rendered (Brown and Sedgewick 1984; Colombo, Demetrescu, Finocchi and Laura 2003).

The *learner interface* (usually implemented in the form of a GUI) allows end-users (the students and instructors) to interact with the system, allowing for input and control of the various aspects of the algorithm animations being viewed (Baker, Cruz, Liotta and Tamassia 1996; Syrjakow, Berdux and Szczerbicka 2000). Inputs may include selecting algorithms and views, setting up views, and inputting customised data structures.

An algorithm animation system is designed with a number of components, with each component or collection of components supporting a subset of user types to accomplish specific tasks within an algorithm animation usage environment.

3.3 Techniques for Creating Algorithm Animations

A number of paradigms are available for connecting algorithms to visual representations (Price, Baecker and Small 1993). This section provides an overview of the paradigms, and the techniques generally employed by each paradigm to link algorithm actions to visual presentations.

3.3.1 The Imperative Paradigm

The imperative (or event-driven) paradigm makes use of the *interesting event* concept to create animations. The concept consists of identifying events within an algorithm that have relevance to the visualisation (Brown and Sedgewick 1998). Program commands are placed in relevant sections of the algorithm to capture interesting events. The events are then conveyed to the visualisation component to produce the animations. Two techniques for creating visualisations can be categorised under the imperative approach, namely *API calls* and *scripting language*.

The *API* technique consists of a collection of pre-defined functions for generating visualisations, stored in a function library. The algorithm is then animated by embedding the algorithm code with calls to the API functions (Figure 3.2). Typically, the algorithm must be written in the same programming language as the API. ANIMAL [3] (Section 3.5) and JAL (SiliconGraphics 1999) [4] are examples of API-based animation systems.

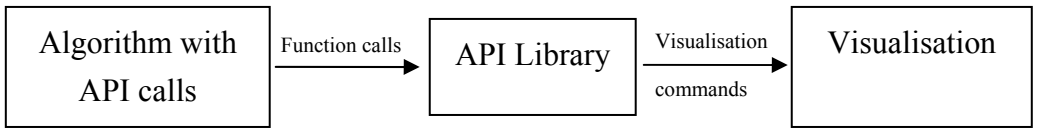


Figure 3.2: Using API library calls to generate visualisation

Animations can be generated using text commands defined by a *scripting language* (further discussed in Section 4.8.1). An algorithm is annotated with output statements at interesting events. The statements are then parsed and interpreted by the visualisation component to produce the relevant visual output (Figure 3.3). JSAMBA

[3] The ANIMAL API classes are undocumented at the time of writing this dissertation.
[4] Cited in (Rößling and Freisleben 2000a).

(Section 3.5.5), JAWAA (Section 3.5.6) and ANIMAL (Section 3.5.7) are examples of script-based animation systems.

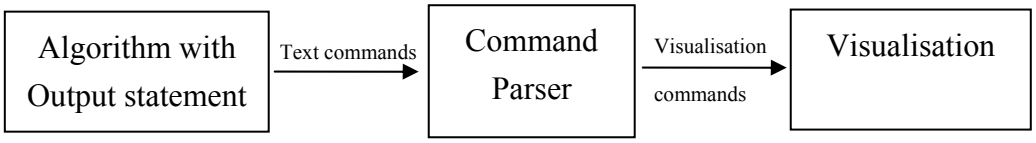


Figure 3.3: Using scripting language to generate visualisation

3.3.2 The Declarative Paradigm

The declarative (or data-driven, state-mapping) paradigm represents algorithm operations visually by defining mappings between program states and visual objects (Roman and Cox 1993; Roman 1998). Visualisations are generated based on data structure related events of the algorithm (Figure 3.4). Relevant data structures are specified and monitored, and changes in the state of the data structures trigger events which update the visualisation.

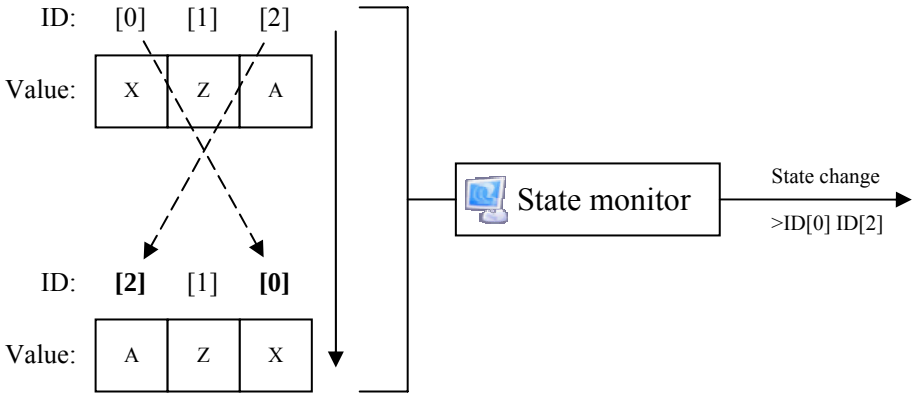


Figure 3.4: The declarative paradigm monitors state changes in the data structure

In the example of the declarative paradigm shown in Figure 3.4, changes in the data structure state are detected by the state monitor through changes in element ID flags.

Two techniques for creating visualisations can be categorised under the imperative approach, namely *comment embedding* and *direct animation*.

The *comment embedding* technique involves embedding animation commands into an algorithm without affecting its program structure. Commands which specify the data structures to be monitored are written as comments within the algorithm. The algorithm animation system's interpreter then extracts the commands from the comment code (Figure 3.5). LEONARDO (Crescenzi, Demetrescu, Finocchi and Petreschi 2000) uses its own declarative language called Alpha, which is embedded as comments into C-based algorithms .

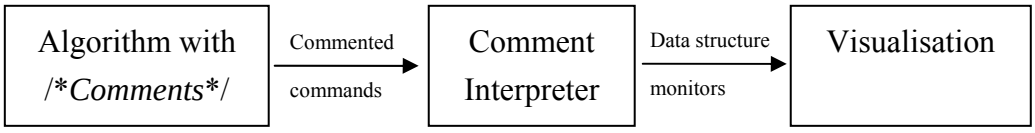


Figure 3.5: Interpreter extracts the comments which describe the data structures being monitored

Direct animation by code interpretation presents the most convenient and rapid method for creating animations. The system uses a virtual machine to interpret an algorithm and identify all data declarations (Figure 3.6). The data declarations are then automatically linked to visual elements. Any changes to the data state detected results in appropriate visualisation updates. Jeliot (Haajanen, Pesonius, Sutinen *et al*. 1997) supports direct interpretation and animations of Java™ code. The system's interpreter in effect replaces the role of the visualisation designer.

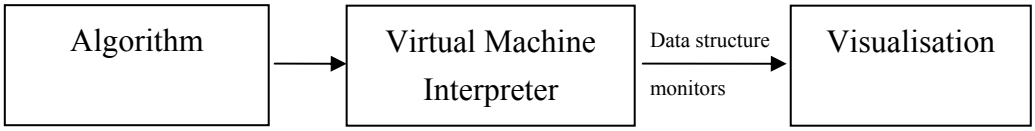


Figure 3.6: The virtual machine interprets the algorithm source directly to monitor data structure changes

3.3.3 Other Approaches

A number of alternatives to the approaches discussed under the imperative and declarative paradigms exist for constructing animations of algorithms. These alternative approaches allow rapid creation of animations without any form of programming implementation (Stasko 1998a; Rößling and Freisleben 2000a). No formal mapping is thus needed between the visualisation and the illustrated algorithm. The two approaches discussed are *visual editing* and *manual scripting language*.

Visual editors allow the visualisation designer to build animations using pre-defined tools and graphical objects. JAWAA and ANIMAL are examples of algorithm animation frameworks which include a visual editor (Section 3.5).

Manual scripting languages offer another method for creating animations, where the visualisation designer writes animation commands manually using a text editor. This approach will utilise the same grammar structure of the *scripting language* approach discussed under the imperative paradigm (Section 3.3.1), but differ in that no algorithm implementation is linked to the created animation.

3.4 Desirable Pedagogical Requirements for an Algorithm Animation System

The availability of modern computers allows for rapid design and real-time generation of animations, and the ability to facilitate interaction with students. The focus of algorithm animations has moved beyond merely showing students an algorithm animation in the hope that they will understand and retain some of the algorithm concepts illustrated. Current emphasis is placed on identifying factors which will

increase the instructional value of algorithm animations (Stasko, Badre and Lewis 1993; Saraiya 2002). Studies and observations (Roman and Cox 1992; Hundhausen 1993) have shown that the use of algorithm animations does not in itself guarantee improvements in algorithm understanding. This is akin to how an immaculately printed textbook cannot produce effective transfer of learning if important features (taken for granted in any decent textbook) such as well written explanations, relevant case studies and informative exercise questions are not included. An important step in designing an algorithm animation system is thus to understand what features effectively complement the students' learning strategy (Kehoe, Stasko and Taylor 2001).

The supported features of algorithm animation systems are generally fixed once the system has been implemented (Rößling and Freisleben 2001). Adapting and extending existing source code to support additional features are often a time-consuming and impractical process. This further highlights the need to define a clear specification of requirements when designing and implementing a system. A process is thus required to identify and motivate specific features derived from the research community.

In this section, discussions are given on algorithm animation features shown to increase the pedagogic effectiveness of the system or broaden its usefulness within a teaching environment. The first part discusses a number of features based on a system's interaction with the students (Section 3.4.1). This is followed by a discussion of system features which can further complement the learning effectiveness of an algorithm animation system (Section 3.4.2). The findings from the discussions are then summarised and presented (Section 3.4.3).

3.4.1 Requirements based on Levels of Engagement

An interactive environment in an algorithm animation system is believed to more effectively attract a student's cognitive attention and engage the mind of the student (Hansen, Narayanan and Hegarty 2002). Studies have suggested that rather than just letting students view an algorithm animation passively, better learning results may be obtained by allowing students to engage interactively with the animation (Hundhausen 2002; Naps, Fleischer, McNally *et al*. 2003). The observations suggest that the method and extent in which students are engaged with the algorithm animation and related learning activities have significant influence on the effectiveness of employing such teaching support tools (Faltin 2001; Grissom, McNally and Naps 2003). It is thus important to take into account the approaches of active engagement within the context of algorithm animation when identifying requirements for the system. Furthermore, a recent study has produced a taxonomy to define the level of engagement of students with an algorithm animation system (Naps, Fleischer, McNally *et al*. 2003). As a result, empirical studies within the research community can now uniformly evaluate the effectiveness of algorithm animation systems and features based on an established level-of-engagement framework (Grissom, McNally and Naps 2003).

Naps *et al* (2003) defines a taxonomy of students' interaction with algorithm visualisations based on six levels of engagement:

1. *No viewing* – no algorithm visualisation is utilised
2. *Viewing* – from students viewing a visualisation passively, to having them make adjustments on various aspects of the visualisation display

3. *Responding* – students answering questions concerning the animation during its execution
4. *Changing* – modification of the visualisation by the student in order to increase understanding of an algorithm
5. *Constructing* – creating a new visualisation of a given algorithm
6. *Presenting* – presenting visualisations to other students to stimulate discussions on the given topic

The taxonomy is designed to allow educators to develop systems which will take advantage of these various forms of engagement, and allow the developed systems to be evaluated using a standardised engagement level definition. Each of the requirements identified is organised based on the taxonomy of engagement levels presented by Naps *et al* (2003). The list of requirements is then used as a common framework for examining and evaluating a number of extant algorithm animation systems. The requirements are identified and organised based on four levels of engagement, namely viewing, changing, responding and constructing. The first and sixth level of engagement – *No viewing* and *Presenting* – are not discussed further. *No viewing* is essentially the absence of algorithm animations. *Presenting* involves the learner demonstrating an algorithm animation. It is thus an activity generally performed by the instructor to aid learners. However, since *Presenting* is inherently supported through *Constructing*, instructors can ask students to demonstrate animations which they have created to liven up lecture discussions.

Viewing

Algorithm animations can be viewed as a surveillance video that records and displays the execution of an algorithm[5]. When users investigate an algorithm, they may slow the video down to better examine a particular event, speed through events which offer no further contribution to the investigation, or step through key events one at a time. Speed and stepping controls allow algorithm animations to adapt to the learning pace for a given environment, whether it is instructors demonstrating in a classroom, or students self-studying in the computer laboratory. The users should have a unified set of functions for controlling the behaviour of the algorithm animation displayed (Gloor 1998). The system should include functionalities to pause and replay the animation. The ability to speed up animations will allow students and instructors to move over sections which are already understood (Rößling 2002).

Algorithm operations may be missed by students, which may occur if the animation is displayed too fast. Students may also be confused by certain operations if they were shown while students were still mentally processing previous operations, or if the operations were not anticipated by the students due to unfamiliarity with a newly introduced algorithm. Allowing students to backtrack the animation a specified number of steps will allow reviewing of past operations as needed, rather than having to restart the animation from scratch (Naps, Eagan and Norton 2000; Rößling 2002).

[5] Albeit a display with a high-level of abstraction.

Responding

Algorithm animations provide immediate visual feedback of algorithm operations, and can thus support students in testing predictions to enhance understanding. In a closed-lab study performed by Byrne *et al* (1999), students were required to pause algorithm visualisation at specific points and make predictions orally. This has shown to significantly increase algorithm understanding. Students may be asked to make predictive answers independently of the animation system during the display of the animation (Byrne, Catrambone and Stasko 1999), or answer text-based questions integrated into the animation system (Jarc, Feldman and Heller 2000). The system should support the activity of letting students make predictive answers by running animations in discrete steps, thus allowing the students to pause before each interesting event in the animation to predict the next algorithm action (Anderson and Naps 2000)[6]. Allowing animations to run one step at a time also acts as a method for slowing the animation down if students have difficulty in understanding certain operations of an algorithm.

Changing

Students and instructors should be allowed to input custom data into the algorithm. Instructors will thus be able to demonstrate algorithm specific characteristics to students, such as best-case and worst-case performance scenarios (Naps, Eagan and Norton 2000; Saraiya 2002). Students will also be able to input their own data set into the algorithms to test cases beyond those offered in lectures or textbooks. An important advantage of an algorithm animation system over traditional static teaching

[6] Cited in (Rößling and Naps 2002).

materials is a system's support for real-time generation of animations using user-specified input. Users can utilise this feature to examine various cases to improve algorithm understanding. Most importantly, the system should support real-time generation of datasets, based on a provided population size and list pre-sort percentage level. This will allow students to create and examine case studies relating to complexity and performance characteristics of different sorting algorithms by generating and utilising meaningful test data (Section 4.4).

Constructing

Creating an animation of the algorithm under study would induce students to have a deeper understanding of the algorithm's operations, since students must learn the algorithm with the intent of sharing their understanding of the algorithm concepts to an audience (Hübscher-Younger and Narayanan 2003).

This concept was tested by Hundhausen (2002) in his observational study, where students were asked to construct and present an algorithm of their choosing using existing algorithm animation tools. In effect, the students were actively engaging with the algorithm animation by performing the roles of *algorithm programmer*, *visualisation designer* and *instructor* (Section 3.2.1). Hundhausen observed that while engaging students in such activities took significantly more time than conventional teaching methods, these activities did contribute to the students' understanding of the algorithms studied.

3.4.2 Complementary Requirements

A number of algorithm animation system features are identified which are believed to enhance the pedagogic effectiveness and usefulness of the system. Each of these features is discussed below.

Smooth animation aids the student in tracking changes between discrete steps of an algorithm (Stasko 1998b). This feature forms a fundamental part of algorithm animation. In certain cases, such as when large datasets are being viewed, students should be able to disable animations and view discrete steps of the algorithm (Rößling and Naps 2002).

Analysis features can aid students in better understanding the efficiency of an algorithm and the relative performance differences among various algorithms (Gloor 1998). Algorithm efficiency can be illustrated by means of performance statistics collected from the animations or generated independently of the animation. Relative performance can be illustrated by running several algorithms simultaneously, thus letting the students compare the differences visually (Naps, Fleischer, McNally *et al*. 2003). Using animations to demonstrate sorting algorithm races have shown to be very convincing in illustrating performance differences (Baecker 1998).

Multiple views of an algorithm may be used in different approaches to aid students. Students may find the use of certain metaphors easier to understand, and thus prefer a certain approach of animation (Gurka and Citrin 1996). Different views may also be used to illustrate algorithm executions at different levels of abstraction, or demonstrate different characteristics, such as operational or performance trends

(Wilson, Katz, Ingargiola *et al*. 1995; Naps, Fleischer, McNally *et al*. 2003). In addition, alternative views also include information relating to the running algorithms, such as total execution time elapsed, and the number of operations performed.

Additional materials accompanying algorithm animations may increase the instructional effectiveness of the animation. The materials may include simple textual explanations, pseudo-code or source code views (Rößling, Schüler and Freisleben 2000). Alternatives include using multimedia elements, such as audio and video of instructors explaining the algorithm (Stasko, Badre and Lewis 1993). The materials may be presented separately, or integrated as part of the animation system.

Students may utilise algorithm animations in a self-study environment, without narrations and explanations from instructors. This may decrease the instructional value of the animation, as students are expected to figure out the plot of the algorithm unaided. Whilst it is possible to provide static text related to an algorithm by using printed supplementary material, the text will not be able to highlight specific details of the algorithm or to explain the workings of the algorithm as the animation is running, like an instructor in a demonstration environment could. A system which has some form of data awareness will be able to provide and store dynamic output as the algorithm runs. Static information such as "*the algorithm is now performing a swop of items*" can then provide more context sensitive information, such as "*swopping value 7 in position[1] with item with value 4 in position[2]*", thus offering a more specific explanation to the students viewing the animation (Naps, Eagan and Norton 2000; Sumner and Banu 2003).

A general animation system is designed to create animations in any domain of knowledge, whilst domain-specific animation frameworks are limited to animating a specific type of algorithm and data structure. However, it has been noted that general purpose systems are inherently more processor intensive than topic specific animations (Brown and Sedgewick 1984). A balance must be found that allows a given animation system to construct animations for a specific domain, whilst still offer some degree of support for non-domain specific uses. The system should be designed to allow creation of animations not necessarily restricted to sorting algorithms, allowing for the animation of other list-based algorithms, such as merges and searches, without need for modification (Akingbade, Finley, Jackson *et al*. 2003). Furthermore, a system which can create animations of various associated topics can offer a common user interface for students studying the different topics (Rößling and Naps 2002).

3.4.3 Summary of Requirements

From the preceding discussion, a number of preliminary features are identified as the requirements for a pedagogically effective system. The identified requirements appear in Table 3.1. The effectiveness of an algorithm animation system to complement the students' study of algorithms is determined by the system's ability to engage the students in an active learning process (categorised as requirements R1 through R6), and system features which either provide additional information to enhance comprehensibility of the animation, or increase its usefulness in an educational environment (requirements R7 through R11).

Requirements for Algorithm Animations	Section
R1: Allow speed control of algorithm animation	3.3.1 - Viewing
R2: Allow rewinding of the animation	3.3.1 - Viewing
R3: Accept user input data for the algorithm	3.3.1 - Changing
R4: Provide questions to predict algorithm behaviour	3.3.1 - Responding
R5: Allow stepping control of algorithm animation	3.3.1 - Viewing 3.3.1 - Responding
R6: Support construction of animation by students	3.3.1 Constructing
R7: Support for smooth motion	3.3.2
R8: Include capabilities for comparative algorithm analysis	3.3.2
R9: Provide multiple views of an algorithm	3.3.2
R10: Provide additional instructional material	3.3.2
R11: General purpose framework	3.3.2

Table 3.1: List of identified requirements

The list of requirements established in Table 3.1 is used as criteria for assessing a number of extant algorithm animation systems in the following section. The requirements identified, in conjunction with the extant systems overview (Section 3.5), will support the discussion of the finalised list of requirements (Section 3.6) for the proposed algorithm animation framework.

3.5 Overview of extant systems

The purpose of the extant system analysis is to use the requirements identified in Section 3.4 as a platform to evaluate existing system implementations. This section offers an overview of seven extant algorithm animation systems. General characteristics and the requirements supported by each extant system are identified, with the specific aim of guiding the creation of the scope of requirements for the proposed framework. The overview of these systems[7] includes a tabular summary of the requirements supported by the systems (Tables 3.2-3.8), based on the list identified in Section 3.4. Explanations are provided for requirements indicated as partially supported. Support for requirements which cannot be conclusively derived from available literature are treated as unsupported. Information is derived from a combination of literature studies and actual system reviews.

3.5.1 Sorting Out Sorting

The "Sorting Out Sorting" video (Baecker 1981; Baecker 1998), although not strictly defined as a system, is nevertheless, worthy of mention. The video employed a number of features which were unprecedented at the time in demonstrating sorting algorithms (Figure 3.7). These features include the use of various visual metaphors, including animation, colour, audio, and voice-over commentary. The operations of nine sorting algorithms were illustrated, followed by time versus data size performance graphs typically found in textbooks. The nine algorithms were then run simultaneous in a race to compare and contrast their performance characteristics.

[7] Comprehensive reviews of actual systems are sometimes not possible due to systems being no longer available, legacy, or incompatible with available hardware and software resources.

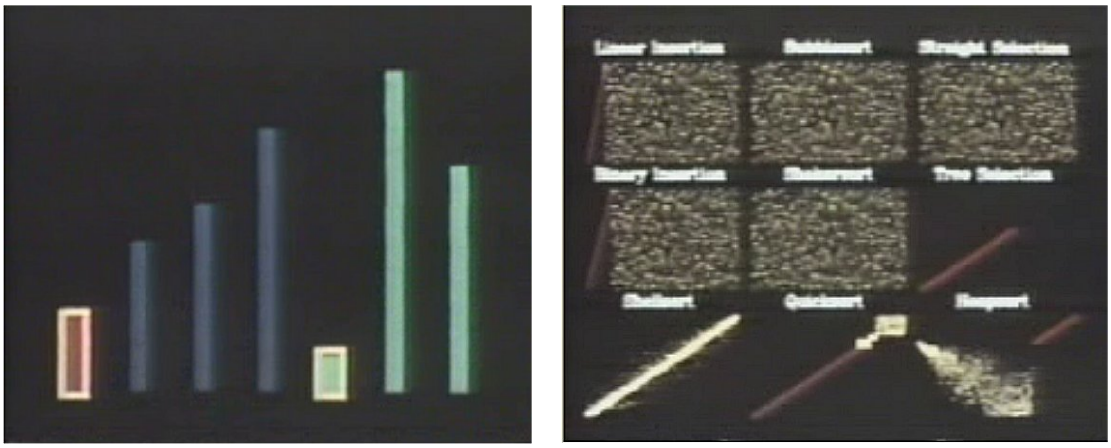


Figure 3.7: Sorting Out Sorting – Demonstrating a sorting algorithm / Race of nine sorting algorithms using a cloud representation (Baecker 1981)

Table 3.2 illustrates the support provided by "Sorting Out Sorting" based on the list of requirements derived in Section 3.4. The video is identified as supporting smooth motion (R7). Comparative algorithm analysis and multiple views of an algorithm are included (R8, R9). Additional material is provided through audio narratives (R10). Because the animations are produced as a video, all requirements are essentially of historical nature and thus regarded as partially supported.

Requirements for Algorithm Animations	Supported
R1: Allow speed control of algorithm animation	
R2: Allow rewinding of the animation	
R3: Accept user input data for the algorithm	
R4: Provide questions to predict algorithm behaviour	
R5: Allow stepping control of algorithm animation	
R6: Support construction of animation by students	
R7: Support for smooth motion	(✓)
R8: Include capabilities for comparative algorithm analysis	(✓)
R9: Provide multiple views of an algorithm	(✓)
R10: Provide additional instructional material	(✓)
R11: General purpose framework	

Table 3.2: Requirements supported by Sorting Out Sorting
✓ Full support. (✓) Partial support.

3.5.2 Brown University Algorithm Simulator and Animator II (BALSA)

The BALSA animation system (Brown and Sedgewick 1984) can be regarded as a concept prototype for all current systems due to the novel design concepts utilised by the system. The system was designed and implemented to integrate into Brown University's electronic classroom concept (Bazik, Tamassia, Reiss and van Dam 1998). BALSA dynamically generates algorithm animations by annotating Pascal algorithms with interesting events (Section 3.3.1), which are then used to notify and update animation views. BALSA is installed on workstations and displays animations based on scripts created by the instructors. BALSA-II (Brown 1988a, 1988b) added support for colour displays. BALSA-II also included the ability to run multiple algorithms in synchronised displays to illustrate algorithm races (Figure 3.8), similar to the demonstration done in "Sorting Out Sorting".

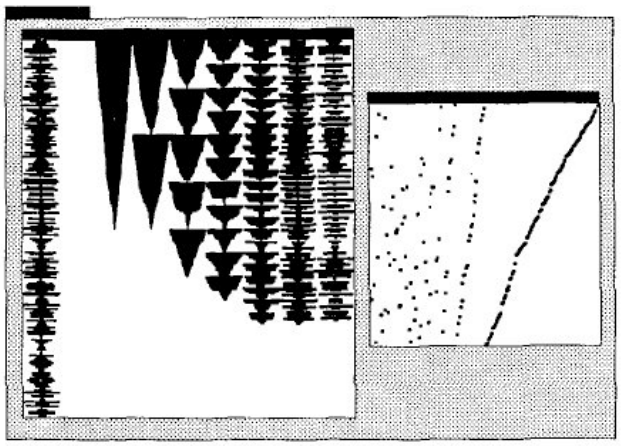


Figure 3.8: BALSA-II – Illustrating a mergesort using a clouds view, and a bar chart to show consecutive states of the data (Brown and Sedgewick 1984)

Table 3.3 illustrates the support provided by BALSA-II based on the list of requirements derived in Section 3.4. The system supports dynamic input to generate animation (R3), however, literature did not specify if the feature is directly accessible

by students. No details were available on the level of animated motion support by BALSA. The system allows for speed control of algorithms (R1) and stepping through animations (R5). The system also supports capabilities to compare algorithms (R8), and show alternative animation views (R9).

Requirements for Algorithm Animations	Supported
R1: Allow speed control of algorithm animation	✓
R2: Allow rewinding of the animation	
R3: Accept user input data for the algorithm	(✓)
R4: Provide questions to predict algorithm behaviour	
R5: Allow stepping control of algorithm animation	✓
R6: Support construction of animation by students	
R7: Support for smooth motion	
R8: Include capabilities for comparative algorithm analysis	✓
R9: Provide multiple views of an algorithm	✓
R10: Provide additional instructional material	
R11: General purpose framework	

Table 3.3: Requirements supported by BALSA-II
✓ Full support. (✓) Partial support.

3.5.3 Generalised Algorithm Illustration through Graphical Software (GAIGS)

GAIGS (Naps and Swander 1994) generates discrete snapshot visualisations of an algorithm's data structure at interesting events. The transitions between each frame of the visualisation are not animated (Figure 3.9). GAIGS uses a scripting language which specifies visualisations based on data structures rather than graphical objects, thus employing the declarative approach (Section 3.3.2) for creating visualisations.

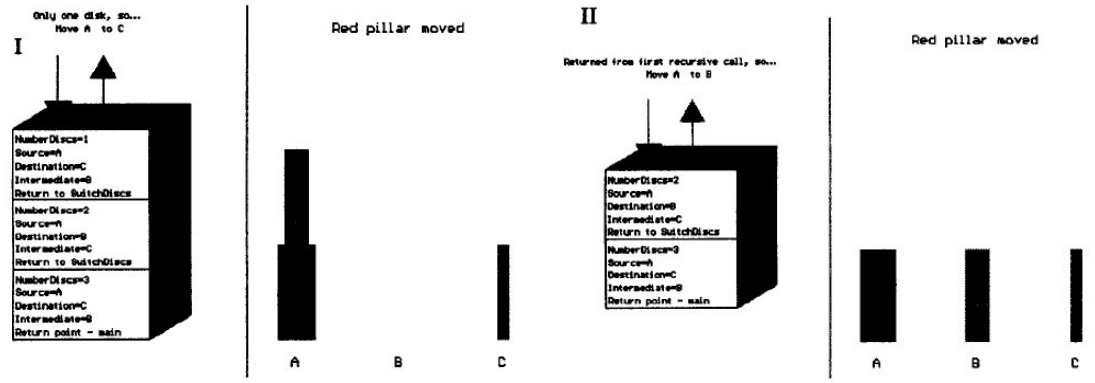


Figure 3.9: GAIGS – Two consecutive snapshots (Naps and Swander 1994)

Table 3.4 illustrates the support provided by GAIGS based on the list of requirements derived in Section 3.4. The system supports the feature of rewinding the algorithm to a previous state and replaying sequences (R2). Support for customised data is included (R3). Students can step through the frames at their own pace (R5). Multiple representations of an algorithm are supported (R9). GAIGS supports a limited static display of the algorithm code been visualised, with no support provided to mark-up the code displayed (R10).

Requirements for Algorithm Animations	Supported
R1: Allow speed control of algorithm animation	
R2: Allow rewinding of the animation	✓
R3: Accept user input data for the algorithm	✓
R4: Provide questions to predict algorithm behaviour	
R5: Allow stepping control of algorithm animation	✓
R6: Support construction of animation by students	
R7: Support for smooth motion	
R8: Include capabilities for comparative algorithm analysis	
R9: Provide multiple views of an algorithm	✓
R10: Provide additional instructional material	(✓)
R11: General purpose framework	

Table 3.4: Requirements supported by GAIGS
✓ Full support. (✓) Partial support.

3.5.4 Java Collaborative Active Textbook (JCAT)

JCAT (Brown and Raisamo 1997; Najork 2001) combines passive multimedia materials with algorithm animations through the use of HTML pages (Figure 3.10). The system supports the simultaneous control and display of algorithm animations on different workstations using Java™ applets. The integrated system allows the instructor to control the animation from a centralised control panel, and allow the students' client to view the animation from a remote workstation. Students can also run JCAT animations in a "solo" mode, where animation controls are directly available to the user (Ramshaw 1997). Animations are created by adding interesting events to algorithms, similar to BALSA.

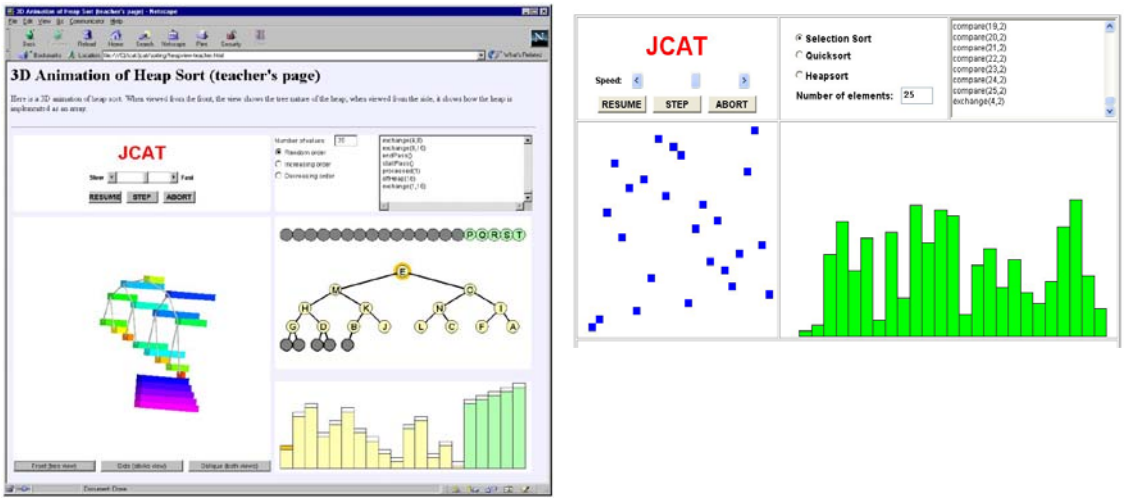


Figure 3.10: JCAT (Ramshaw 1997; Najork 2001)

Table 3.5 illustrates the support provided by JCAT based on the list of requirements derived in Section 3.4. Students have speed and stepping control of animations (R1, R5). Smooth motion is supported (R7). Display of multiple views representing one algorithm (R9) and display of static and context-sensitive text materials (R10) are supported.

Requirements for Algorithm Animations	Supported
R1: Allow speed control of algorithm animation	✓
R2: Allow rewinding of the animation	
R3: Accept user input data for the algorithm	
R4: Provide questions to predict algorithm behaviour	
R5: Allow stepping control of algorithm animation	✓
R6: Support construction of animation by students	
R7: Support for smooth motion	✓
R8: Include capabilities for comparative algorithm analysis	
R9: Provide multiple views of an algorithm	✓
R10: Provide additional instructional material	✓
R11: General purpose framework	

Table 3.5: Requirements supported by JCAT
✓ Full support. (✓) Partial support.

3.5.5 SAMBA/JSAMBA

SAMBA (Stasko, Badre and Lewis 1993) is a front-end application for the POLKA general purpose algorithm animation system. SAMBA takes in ASCII formatted script commands and generates animations based on the commands (Section 3.3.1). This allows for the algorithm to be written in any programming language, providing that appropriate outputs are generated for input into SAMBA. The scripting language allows students to construct and test algorithms by generating their own animations. JSAMBA is a Java-based applet which provides for an internet accessible and platform independent version of SAMBA[8] (Figure 3.11).

[8] The Java version of SAMBA (JSAMBA) is evaluated due to its easy of accessibility over the original SAMBA, which runs on an Unix X11 Window System.

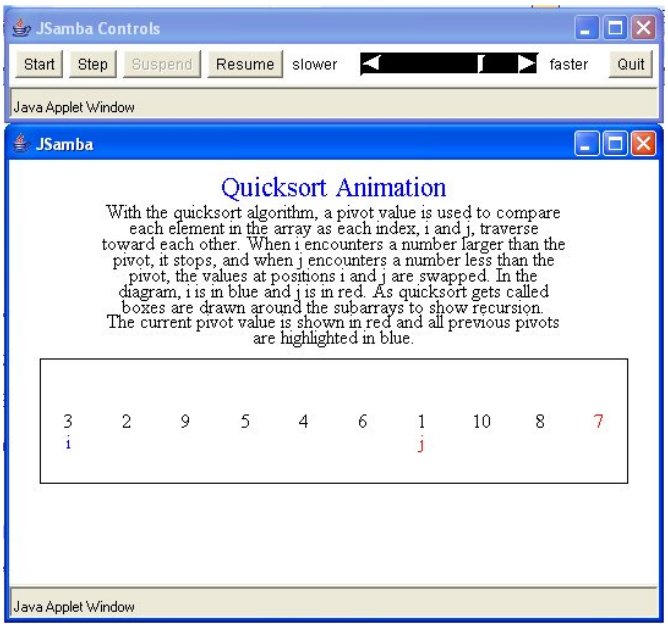


Figure 3.11: JSAMBA (Duskis undated)

Table 3.6 illustrates the support provided by JSAMBA based on the list of requirements derived in Section 3.4. Students have speed and stepping control of animations (R1, R5). JSAMBA allows only static text to be displayed within the animation view (R10). The scripting system gives more accessibility for constructing animations (R6), including general animations (R11). Smooth motion is supported (R7).

Requirements for Algorithm Animations	Supported
R1: Allow speed control of algorithm animation	✓
R2: Allow rewinding of the animation	
R3: Accept user input data for the algorithm	
R4: Provide questions to predict algorithm behaviour	
R5: Allow stepping control of algorithm animation	✓
R6: Support construction of animation by students	✓
R7: Support for smooth motion	✓
R8: Include capabilities for comparative algorithm analysis	
R9: Provide multiple views of an algorithm	
R10: Provide additional instructional material	(✓)
R11: General purpose framework	✓

Table 3.6: Requirements supported by JSAMBA
✓ Full support. (✓) Partial support.

3.5.6 Java And Web-based Algorithm Animation (JAWAA)

JAWAA (Pierson and Rodger 1998; Akingbade, Finley, Jackson *et al*. 2003) is an algorithm animation system which employs a scripting language, much like SAMBA. The visual objects and associated commands available in JAWAA are designed for the animation of algorithm operations, with specific support for data structure objects like arrays, stacks, queues, pointers and linked lists (Figure 3.12). A JAWAA Editor allows users to create an animation without having any knowledge of JAWAA's scripting language.

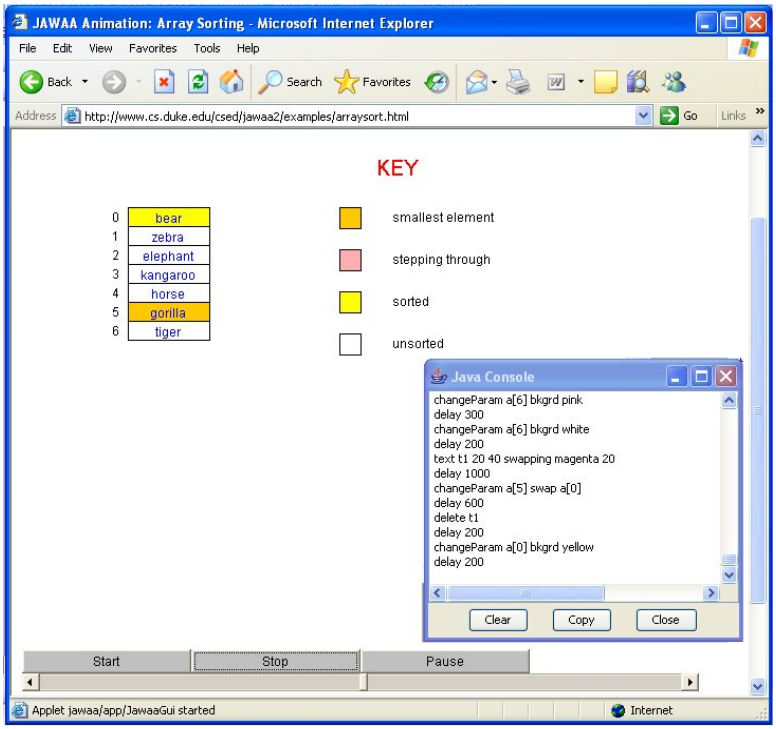


Figure 3.12: JAWAA (Rodger 2002)

Table 3.7 illustrates the support provided by JSAMBA based on the list of requirements derived in Section 3.4. Students have speed control of animations (R1). The scripting language graphic system gives more accessibility for constructing

various animations (R6, R11). The system is implemented in a Java™ applet, the embedding of static material into the client webpage is thus possible (R10). Smooth motion is supported (R7).

Requirements for Algorithm Animations	Supported
R1: Allow speed control of algorithm animation	✓
R2: Allow rewinding of the animation	
R3: Accept user input data for the algorithm	
R4: Provide questions to predict algorithm behaviour	
R5: Allow stepping control of algorithm animation	
R6: Support construction of animation by students	✓
R7: Support for smooth motion	✓
R8: Include capabilities for comparative algorithm analysis	
R9: Provide multiple views of an algorithm	
R10: Provide additional instructional material	(✓)
R11: General purpose framework	✓

Table 3.7: Requirements supported by JAWAA
✓ Full support. (✓) Partial support.

3.5.7 A New Interactive Modeller for Animations in Lectures (ANIMAL) and Java-Hosted Algorithm Visualisation Environment (JHAVE)

The ANIMAL (Rößling, Schüler and Freisleben 2000) system allows the visualisation designer to create animations using either the AnimalScript scripting language (Section 3.3.1), API calls (Section 3.3.1), or through a visual editor (Section 3.3.3). ANIMAL allows animations to be rewinded and replayed. ANIMAL includes the ability to display and mark-up text, making the framework suitable for "tracing" program code as part of the animation (Figure 3.13). The JHAVE (Naps, Eagan and Norton 2000; Rößling and Naps 2002) platform can be integrated on top of ANIMAL to add certain features, such as accepting user-custom input and presenting interactive questions.

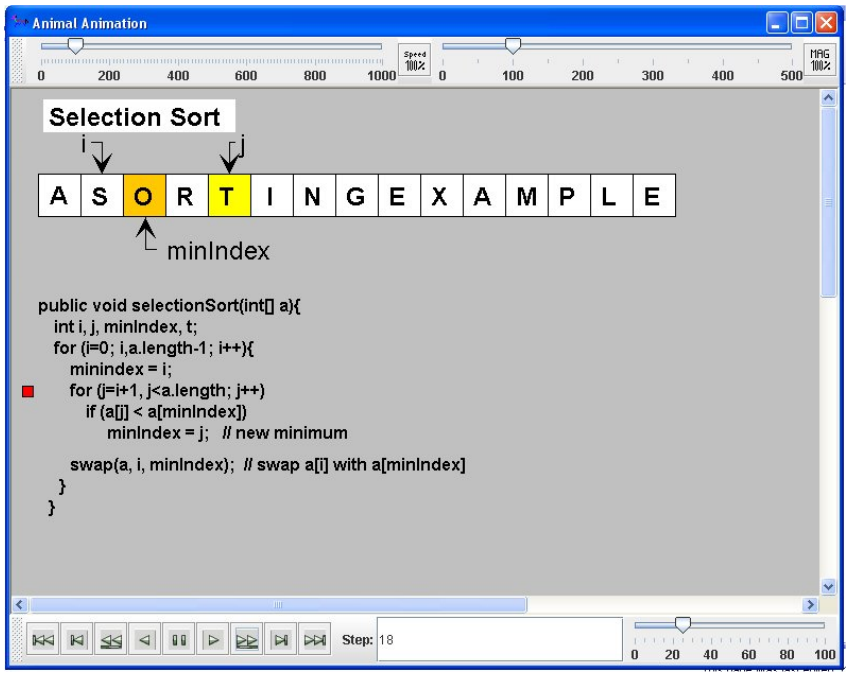


Figure 3.13: ANIMAL and JHAVE (Schüler and Rößling 2001)

Table 3.8 illustrates the support provided by ANIMAL and JHAVE based on the list of requirements derived in Section 3.4. Related literature indicates that all requirements are supported except for the ability to support multiple animation views and comparative animations. The requirements met by ANIMAL and JHAVE include support for controlling the animation display (R1, R2, R5), accepting input data (R3), interactive questions (R4), smooth animation (R7), and display of algorithm information during animation (R10). Animations can be constructed using a number of approaches (R6, R11).

Requirements for Algorithm Animations	Supported
R1: Allow speed control of algorithm animation	✓
R2: Allow rewinding of the animation	✓
R3: Accept user input data for the algorithm	✓
R4: Provide questions to predict algorithm behaviour	✓
R5: Allow stepping control of algorithm animation	✓
R6: Support construction of animation by students	✓
R7: Support for smooth motion	✓
R8: Include capabilities for comparative algorithm analysis	
R9: Provide multiple views of an algorithm	
R10: Provide additional instructional material	✓
R11: General purpose framework	✓

Table 3.8: Requirements supported by ANIMAL and JHAVE
✓ Full support. (✓) Partial support.

3.6 Scope of Requirements

A number of extant algorithm animation systems were evaluated using the requirements framework established in section 3.4. This section highlights particular requirements that the various systems have or have not adequately addressed, in so doing emphasising support for a number of features in the proposed framework resulting from this study (Section 3.6.1). A number of alternative requirements were excluded from the final list of proposed requirements. Motivations for these decisions are provided (Section 3.6.2).

3.6.1 Proposed Requirements for the Framework

"Sorting Out Sorting" (Section 3.5.1) and BALSA-II (Section 3.5.2) provide a unique analytical tool by supporting running and displaying of multiple algorithms simultaneously to provide a visual contrast in relative efficiency. However, this idea has been largely ignored by consequent research into new systems. BALSA-II is a

system designed for a legacy Macintosh OS and is thus unsuitable for use on currently available platforms. Two extant systems, namely WebGAIGS (Naps and Bressler 1998) and ANIM (Bentley and Kernighan 1991), provide functionalities for **demonstrating multiple algorithms in parallel** (R8). However, only static snapshots are supported. It has been suggested that there is a lack of existing systems which provide analytical capabilities through simultaneous animated visualisation of multiple algorithms (Petre, Blackwell and Green 1998).

"Sorting Out Sorting", BALSA-II, GAIGS (Section 3.5.3) and JCAT (Section 3.5.4) support the **display of multiple views of algorithms** (R9). The other extant systems reviewed only support the display of a single animation view.

Systems such as JCAT, SAMBA (Section 3.5.5), JAWAA (Section 3.5.6) and ANIMAL+JHAVE (Section 3.5.7) are capable of generating **smooth transitional animations** (R7) between discrete steps of an algorithm, a display technique initially used in "Sorting Out Sorting". JCAT allows the demonstrator of the animation to centrally adjust the display of visualisations, whilst JSAMBA, JAWAA and ANIMAL allow the students to **control the speed, playing and stepping through of the visualisations** (R1, R5) during viewing. ANIMAL+JHAVE is the only reviewed system which supports the **rewinding of animations** (R2), and **providing interactive questions** (R4) during the animation.

GAIGS, JSAMBA, JAWAA and ANIMAL+JHAVE each make use of a scripting language approach to generate animations. The scripting language is utilised as a protocol of communication to convey algorithm event information to the animation display. An algorithm generates a script containing details of the algorithm's

operation. Based on the given script, the system then constructs the relevant animation. JSAMBA, JAWAA, and ANIMAL+JHAVE are each developed around self-defined scripting languages. The languages provide commands for defining graphic objects and actions. Using the defined scripting language of the systems, **general purpose animations** (R11) can be created using the defined scripting languages.

JSAMBA provides a simple interface which reads and animates prewritten scripts. JAWAA and ANIMAL+JHAVE have a visual editor which can be used to specify visual objects and animation sequences, from which the equivalent animation scripts can automatically be generated. JSAMBA, JAWAA and ANIMAL+JHAVE thus allow the visualisation designer to **create animations** (R6) without necessarily possessing extensive programming knowledge. However, due to the scripting-language animation generation approach used by the systems, animations must be recreated when a new dataset is to be demonstrated or tested. ANIMAL+JHAVE is the only system to adequately support **direct input of datasets** (R3) by the user without having to re-script the animation sequences manually or through a visual editor.

"Sorting Out Sorting" provides audio commentary to guide viewers on the operations of the animations being demonstrated. GAIGS, JSAMBA and JAWAA allow static text to be displayed as part of the animation, whilst ANIMAL+JHAVE and JCAT allow the additional feature of **displaying text based on algorithm events** (R10) occurring during the animation.

Various possible requirements for algorithm animation systems were identified and considered based on various factors discussed above. From this, a final list of requirements is thus presented (Table 3.9).

Requirements for Algorithm Animations	Proposed Support
R1: Allow speed control of algorithm animation	✓
R2: Allow rewinding of the animation	
R3: Accept user input data for the algorithm	✓
R4: Provide questions to predict algorithm behaviour	
R5: Allow stepping control of algorithm animation	✓
R6: Support construction of animation by students	✓
R7: Support for smooth motion	✓
R8: Include capabilities for comparative algorithm analysis	✓
R9: Provide multiple views of an algorithm	✓
R10: Provide additional instructional material	✓
R11: General purpose framework	✓

Table 3.9: Scope of Requirements
✓ *Support*

3.6.2 Motivation for Excluded Requirements

The support for the rewinding of algorithm animations during playback was considered. However, various literature (Brown 1988a; Rößling 2002; Colombo, Demetrescu, Finocchi and Laura 2003) in conjunction with the evaluation performed in Section 3.5, have suggested that the ability to rewind an animation is a feature rarely implemented due to technical issues.

A number of methods for supporting animation rewinding were briefly investigated. Leonardo (Crescenzi, Demetrescu, Finocchi and Petreschi 2000) creates a virtual execution environment which compiles and executes its own logic visualisation

language using a reversible virtual CPU. ZStep95 (Lieberman and Fry 1998) stores an incremented history of the program execution and outputs. The implementation of the abovementioned methods are, however, beyond the scope of the current study.

The feature for providing questions concerning the algorithm during the animation was also considered. However, the focus of the framework is on creating animations to aid in algorithm analysis (Section 1.4.2), and thus interactive questions are not considered part of the project scope.

3.7 Conclusion

This chapter investigated the requirements of an algorithm animation system, which necessarily covers a fairly wide scope. Issues looked at include understanding the user types of an algorithm animation system and the basic structure of a system. An investigation was then performed on available methods of transferring the actions of algorithms to a corresponding visualisation. An evaluation of the methods will be conducted in the next chapter, with motivations for the selection of the paradigm used for the framework.

A list of desirable features for the proposed algorithm animation system was compiled by means of a literature review, supported by an evaluation of extant algorithm animation systems. The evaluation resulted in a comparative study which identified the features supported by the extant systems. There is currently no extant system that supports all of the identified requirements. “Sorting Out Sorting” (Section 3.5.1), BALSA-II (Section 3.5.2), GAIGS (Section 3.5.3) and JCAT (Section 3.5.4) support comparative analysis and multiple views of algorithms. However, support for

interactivity and animation construction flexibility is limited. JSAMBA (Section 3.5.5), JAWAA (Section 3.5.6) and ANIMAL+JHAVE (Section 3.5.7) utilised a scripting-language approach to support a flexible animation design process, with ANIMAL+JHAVE in particular providing the closest match to the list of requirements proposed. However, these systems do not provide adequate support for comparative analysis and alternative animation views.

Based on the evaluation conducted, a final list of requirements to be met by the proposed algorithm animation framework design is established. The information gathered from this chapter builds up to the design of an algorithm animation framework, discussed in Chapter 4.

Chapter 4

Design of Framework

4.1 Introduction

Algorithm animation systems are created with the aim of supporting the pedagogic goals of an algorithm course. It has been highlighted that the effectiveness of the animations are highly dependent on the particular features supported by the system (Section 3.4). The interaction among the various user roles and animation system components in producing and displaying an algorithm animation was also presented (Section 3.2). All of the abovementioned factors must be taken into account when implementing an algorithm animation system. Thus, a conceptual structure is required to describe the systematic support for the processes of creating and viewing algorithm animations, in order to guide the implementation of a system.

This chapter proposes an algorithm animation framework that provides instructors with a structured method to produce algorithm animations, whilst taking into account the pedagogic requirements previously identified. The chapter first outlines the fundamental concepts employed by the framework, and at the same time introduces the layered framework structure (Section 4.2). Sections 4.3 through 4.8 provide a

detailed discussion of each of the framework components. The discussion will follow the logical flow of the framework's structure. The methods of communication used within the framework are explained as part of the component discussions. The interface components are discussed in Section 4.9.

4.2 The Proposed Framework – an Overview

A framework designed for interactive algorithm animation must consider a number of high-level issues. Each of these issues is briefly introduced. Different paradigms have been devised to connect algorithm concepts to visualisations (Section 3.3), with each paradigm typically employing certain approaches. The selection of a paradigm and supporting technique for the proposed framework, and the motivation thereof, are covered in Section 4.2.1. The structure of the framework should support the requirement goals and allow for practical implementation. The framework is thus systematically divided into component groupings called *layers*. The layers are categorised based on their generated data inputs and outputs, which are used as the method of communication within the framework (Section 4.2.2). The creation and display of multiple animations mean that synchronising the speed of the display is an important issue. Furthermore, the animation is used as a representation of algorithm speed. A structure is thus required to control the timing of algorithms (Section 4.2.3).

4.2.1 Selection of Visualisation Paradigm

The imperative and declarative paradigms were previously introduced as methods for connecting algorithms to visualisations (Section 3.3). The imperative paradigm typically utilised the API or scripting language approach for generating animations

(Section 3.3.1), whilst the declarative paradigm utilised the comment embedding or code interpreter technique (Section 3.3.2). The following section evaluates the techniques, which are grouped according to paradigms, by first categorising them based on two factors, namely level of automation and visualisation design flexibility (Figure 4.1).

A high level of automation means that the visualisation tool developer pre-designs visualisations by specifying preset parameters during implementation of the visualisation component. In other words, once the parameters are set, the animation system will autonomously decide what the animation will look like. In contrast, a low level of automation will require the visualisation designer to provide more input into the design of the visualisation. Thus, as the level of automation decreases, the visual design decision shifts away from the visualisation component toward the visualisation designer.

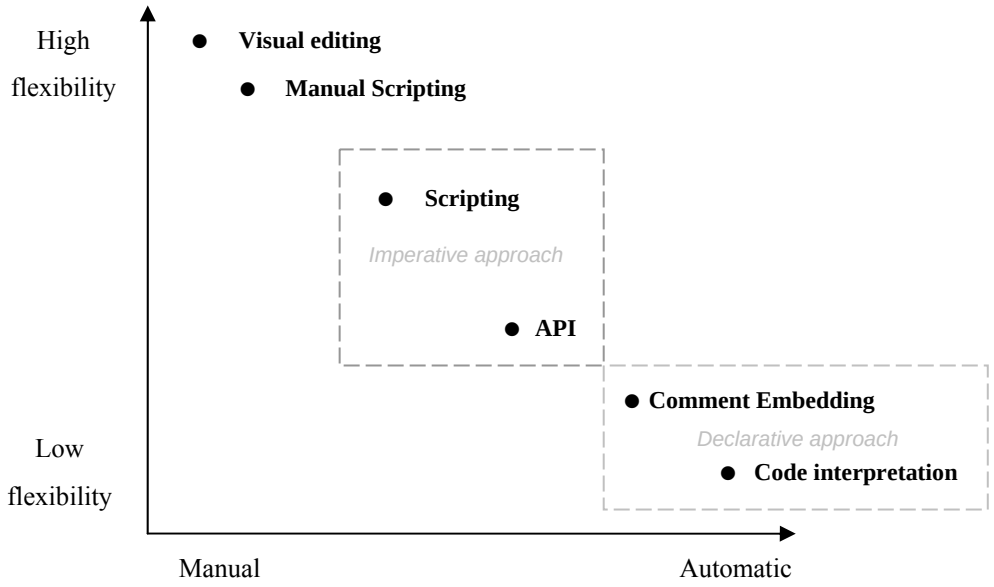


Figure 4.1: Level of automation versus Visualisation design flexibility

Automated visualisation is supported by the declarative approach, which visualises algorithms by graphically reflecting the state of their data structures. Since a common data state monitor and graphical interpretation are used to create visualisations (Section 3.3.2), the approach supports automated visualisation of any algorithms which utilise a supported data structure (Demetrescu, Finocchi and Stasko 2001). However, this also limits the flexibility to customise the visualisation of algorithms. The imperative approach visualises the interesting operations of an algorithm (Brown and Sedgewick 1998). This approach requires a more involved role from the algorithm programmer and visualisation designer, who are responsible for determining the events of interest and designing the visualisation.

The proposed framework will utilise concepts from the imperative approach, employing a hybrid of the scripting language-based and API-based animation generation techniques (Figure 4.1). API libraries offer the ability to efficiently capture and store interesting events generated from calls embedded within algorithms (Section 3.3.1). Script languages allow animation commands to be written manually through a text editor, allowing users to rapidly create custom animations without needing comprehensive knowledge of programming concepts. Alternatively, algorithms can also be annotated with output statements to generate the commands, in an approach similar to using API calls. Thus, by using a scripting approach, animations can be generated through a structured process (Section 3.3.1), or a manual but more flexible process (Section 3.3.3).

Typical API-based methods let the visualisation designer annotate the algorithm with function calls which invoke visualisation instructions (Rößling and Freisleben 2000a). However, the function calls require parameters which define the details of the

visualisation. Thus, as the function calls are annotated into the algorithm, additional support code is needed to specify and maintain abstract information, such as visual object placement and movement co-ordinates. The proposed framework seeks to utilise the API approach to rapidly capture interesting events (Sections 4.5 and 4.6), and employ an intermediate process to manage and specify the abstract visual information (Section 4.7), which is presented using a simple scripting language defined for the framework (Section 4.8.1).

4.2.2 Framework Structure

The framework is separated into various layers, namely the *data interface layer*, *data layer*, the *interpreter layer*, the *animation layer*, and the *animation interface layer*. The independent layer design creates a flexible framework structure by allowing for modular changes and extensions to be made to the framework. Figure 4.2 shows an overview of the framework design, upon which the structure of the chapter is based. The data layer interface allows input to be made to the data layer components (Section 4.9). The data layer consists of the data generator (Section 4.3), data structure definition (Section 4.4), algorithm repository (Section 4.5), and event API (Section 4.6) components. The interpreter layer consists of the interpreter component (Section 4.7), and the animation layer consists of the animation component (Section 4.8). The animation layer interface allows inputs to be made to the animation component (Section 4.9). The layers are defined based on the functions performed by each layer, and the inputs and outputs of the components within each layer.

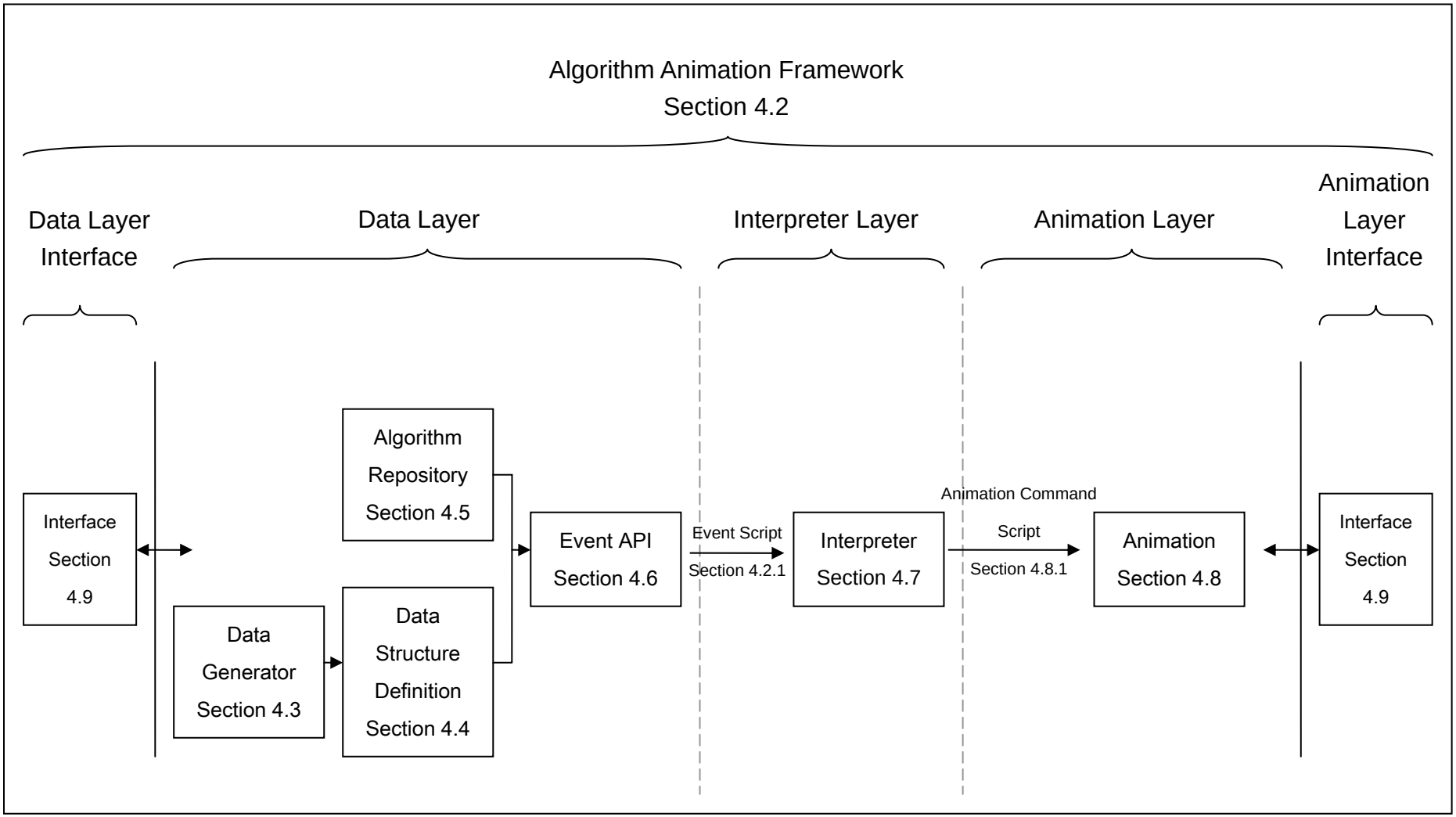


Figure 4.2: Framework structure

The logical separation of the framework into layers assists in defining the activities of each user type within the algorithm animation framework (Section 3.2). In such a mapping, the algorithm programmer interacts with the data layer, the visualisation designer with the interpreter layer, the visualisation tool developer with the animation layer, and the students and instructors with the data and animation layer interfaces. Mapping the framework layers and user types according to their functions thus simplifies the process of creating animations. The algorithm programmer can focus on writing an algorithm and identifying events for animation without considering how the algorithm is to be animated, or the code used to generate animations. The visualisation designer can then design and implement an animated representation of the algorithm events without detailed knowledge of the original algorithm's implementation.

The modular design of the framework also provides for easier implementation. The code within each of the components can be implemented independently, which will allow for later modifications or upgrades to be made to individual components without affecting the rest of the system, providing that the component's input and output remain compatible with its associated components.

Each layer within the framework consists of a single component or multiple components working together to produce specific outputs. These outputs are utilised as inputs by the next layer, resulting in the generation and display of animated algorithms. The information for creating animations (in the form of scripts and commands) are passed down the respective layers in the process of creating and viewing an algorithm animation. The data layer interface is linked to the data layer. The purpose of the data layer interface is to provide the end-users (students and

instructors) with a consistent method of specifying algorithm, data structure and visualisation information. The data layer provides a list of interesting events, referred to as an *event script* (Section 4.6), which give a high-level description of the relevant operations of an algorithm. The interpreter layer takes the event script and converts it into graphical representation commands which provide low-level animation instructions (Section 4.8.1). The animation layer processes the animation scripts from the interpreter layer and produces the visualisation. The actions of the visualisation are controlled through input from the animation interface layer.

The interpreter component is an abstract structure that allows extension by designers to animate different algorithms. In addition, multiple instances can be created of the algorithm, data structure, interpreter, and animation components within each of the layers. The layer instances can then combine to form different scenarios. This method of structuring the framework allows multiple animations to be created to support parallel viewing and analysis of an algorithm using different data (Figure 4.3a), or multiple algorithms using the same data (Figure 4.3b).

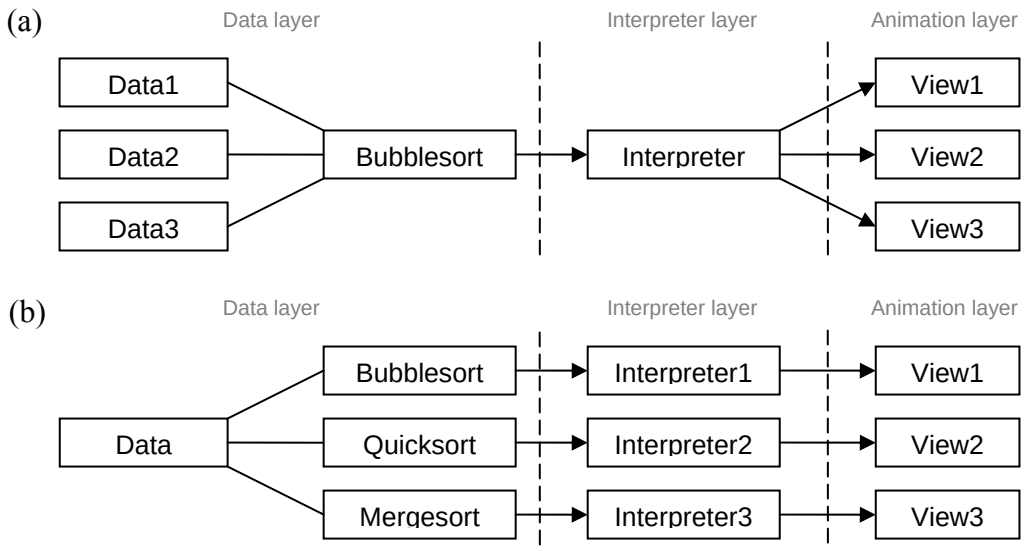


Figure 4.3: Structure allows for parallel analysis of algorithms and data

The separation of the animation functionality into its own layer also allows for it to be utilised independently (Figure 4.4). The animation layer takes input in the form of a graphical command script. This input may be based on output from the layered processes of the framework. The input may also be generated manually in support of two requirements. Firstly, accepting manual inputs allow students to create animation without needing to implement an algorithm. Secondly, general purpose animations, or animations which are unsuitable for creating through the framework's processes, may be animated directly through the animation layer using the graphical commands (Section 4.8.1).

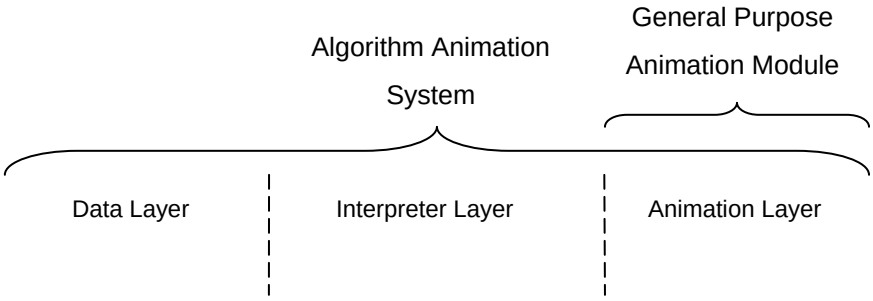


Figure 4.4: Modularisation of the Animation layer

4.2.3 Timing and Parallel Animations

An animated visualisation of an algorithm is seen as an abstract representation of the program states and operations of the algorithm. Beyond simply demonstrating the inner workings of an algorithm, animated visualisations can also make algorithm performance characteristics more apparent. Animations can demonstrate the speed of an algorithm by showing the quantity of operations needed to complete a given scenario. Furthermore, visual displays of multiple algorithms processing data lists with identical values, or the same algorithm processing different lists, provides a convincing contrast of performance and efficiency differences.

Animated algorithm demonstrations have been employed to demonstrate the time required for running certain algorithms (Baecker 1998). Therefore, the students viewing the animation will perceive it not as a representation of only the algorithm's operations, but also its speed. Thus it is important for the animation to be a true reflection of the algorithm's performance. However, executing and animating an algorithm in real-time will result in an animation which is too fast for demonstration purposes. In fact, if the visualisation is coupled to a "live-mode" execution of an algorithm, a disproportionably large amount of the total execution time will be used in rendering the animation. In addition, there are inherent synchronisation complexities associated with coordinating the execution of multiple algorithms for sorting races, and at the same time providing a common benchmark of performance across the different animations (Brown 1988a).

A method is thus needed to present animations at a practical speed, while at the same time still act as a (reasonably) true representation of each underlying algorithms' operational performance. The framework provides for the generation of an animation by first executing algorithms to completion, and at the same time recording interesting events of each algorithm in an event script (Section 4.6). The event script is then converted to animation graphical commands. The framework's animation component thus generates visualisations independent of an algorithm's real-time execution. The *post-mortem* approach (Diehl, Görg and Kerren 2002) simplifies the generation and coordination of animations, allowing the framework to simulate multiple algorithms running in parallel. The approach also allows the visualisation designer to create a standardised measure of performance across all algorithm animations.

The use of scripted events enables a property of time to be associated with each algorithm operation. Animated operations can be specified to complete based on a standardised time interval, rather than being dependant on the computer's true speed. The time interval is calculated based on the type of operations performed and the data structure used (further discussed in Section 4.4.2). Time interval information is retained in the graphical commands and reflected through the final animation display. Standardising time intervals allows for a fair race when comparing algorithms, since each algorithm's speed is a direct result of the cumulative time taken to perform all its operations, which are pre-specified with a time interval. Unpredictable factors such as individual computer performances, thread processing priorities, and memory caching methods which affect algorithm performance can thus be disregarded.

The remainder of the chapter looks at each of the framework components in turn (Figure 4.2), discussing each component's functionality requirements and design concepts.

4.3 Data Generator

Sorting is the process of systematically arranging elements into an ascending or descending order. When students study sorting algorithms, they are expected to understand the mechanical workings of the algorithm. Furthermore, students must also have an understanding of how efficiently (or inefficiently) each sorting algorithm would perform, based on the population size and pre-arrangement properties of the list. This understanding would allow students to utilise the optimal algorithm for any presented scenario, taking into account worst case and average case performances.

The proposed framework is well suited to teach the issue of relative efficiency, since students can easily generate an animation to visually prove and contrast how efficiently algorithms run. Students can experiment by trying an algorithm on a number of different lists, or a single list using a number of different algorithms. In such experiments, the framework may allow students to create lists by manually specifying values for each element. However, in cases where students would want to run tests on several sequentially dissimilar lists, each requiring >50 elements, manually specifying elements will become time-consuming and impractical. Furthermore, it would seem unreasonable to expect students to input good test data without some guidance (Grissom, McNally and Naps 2003). Thus, a requirement arises in the algorithm animation framework for the capability to let users create meaningful test lists in an efficient manner.

A scenario often examined when learning sorting algorithms is on how the algorithm will perform on an unbiased random list. Thus, this section will first identify a method for effectively creating quasi-random ordered lists (Section 4.3.1). In order to generate a data list by specifying the list's permutation attribute, a common benchmark for measuring data list order must first be established. Section 4.3.2 investigates methods for measuring data list order, based on which a definition is established for measuring sortedness within the framework (Section 4.3.3). The actual shuffling of lists to create the specified permutations is fairly simple, and is thus not discussed further.

4.3.1 Random Permutation of Lists[9]

A method of observing sorting algorithm behaviour and performance is to experiment using order-unbiased lists. The use of lists arranged in a random permutation will ensure that there is no intentional bias towards the mechanisms of any particular sorting algorithm. One method of generating a list of elements is to obtain values from a seeded random number generator. However, such a method does not guarantee lists with a value distribution suitable for visualisation, an example of a bad distribution being [1, 2, 2, 2, 500, 99999…]. The framework's list generator approaches the problem by generating an evenly spaced, ascending ordered list, for example [1, 2, 3, 4, 5…]. The list is then systematically shuffled to create a randomised list. This section outlines the theory for creating quasi-random lists.

Given a list of *n* unique elements, the number of possible permutations for the list can be worked out (Definition 4.1):

$${}_nP_k \equiv \frac{n!}{(n-k)!}$$ *where n is the total number of items and k is the number of items to be selected to form the new list.*

∴ The number of possible permutations of n-sized list is $${}_nP_n = \frac{n!}{(n-n)!} = n!$$

Definition 4.1: Number of possible permutations for an n-sized list

A method of creating a new random list sample is to have an *original* list containing values (the ordering of the values is unimportant) and a *new* list containing no values.

[9] Adapted from (Stephens 1998; Pallier 2002)

The random list is created by randomly selecting a previously unselected element from the original list, and appending a copy of the element to the new list. The process is then repeated until all element options from the existing list are exhausted. Using this method, the probability of acquiring any particular permutation is thus known (Definition 4.2).

Probability of selecting any element from an n-sized list is $\frac{1}{n}$.

Probability of any permutation occurring when selecting n unique elements from an n-sized list is $\frac{1}{n} \cdot \frac{1}{(n-1)} \cdot ..... \cdot \frac{1}{1} = \frac{1}{n!}$

Definition 4.2: Probability of creating any particular permutation

The method thus ensures that the random sample list is an equiprobable selection from all possible permutations. The pseudo-code for creating a random list sample based on the given method is presented in Figure 4.5.

```
For i = FirstIndex to (LastIndex – 1) do
    j = i + (LastIndex - i) × RandomFloat(Range 0 to 1)
    j = Round( j )
    Swop(List[i], List[j])
```

Figure 4.5: Pseudo-code for randomising a list

The code presents an in-place method of randomising the list, where all values before position *i* form part of the new randomised list, and values after position *i* are available for selection.

4.3.2 Approaches for Measuring Sortedness

Any list *L* would have three properties, namely population size (*n*), element values, and a defined arrangement sequence. Arrangement sequences (or permutations) and randomness of lists play a critical role in the efficiency of sorting algorithms (Chen and Carlsson 1991; Hwang, Yang and Yeh 2000). This section identifies and evaluates two methods for quantifying a list's level of sortedness.

Step-Down-Runs (Measure of Disorder)

Step-Down-Runs measure disorder by the number of adjacent element pairs which are in inverse order in a list. $M(X)$ is the measure of disorder of sequence X, thus a higher value indicates a less sorted list (Knuth 1973). The maximum possible value of $M(X)$ is $n-1$ where n is the list length and 0 represents the minimum possible value. The approach for working out $M(X)$ is illustrated in Definition 4.3. Examples: $M(1,2,3,4) = 0$; $M(4,1,2,3) = 1$; $M(2,1,4,3) = 2$.

$$Let\ X = X_1, X_2, ..., X_{|x|},\ then\ M(X) = \sum_{i=1}^{|x|-1} C_i$$
$$Where\ C_i = 1\ if\ \ X_i > X_{i+1}\ \ and\ \ C_i = 0\ if\ \ X_i \le X_{i+1}$$

Definition 4.3: Step-Down-Runs

Measure of Presortedness

Measure of Presortedness (or *mop*) is calculated by taking all possible combination pairs in a list, and counting the number of combinations which are correctly ordered

(Definition 4.4). *Inv*(*X*) is the measure of presortedness of sequence *X*, thus a higher value indicates a more sorted list (Brodal, Fagerberg and Moruz 2005). Examples: *Inv*(*1,2,3,4*) = 6; *Inv*(*4,1,2,3*) = 3; *Inv*(*2,1,4,3*) = 4.

$$Inv(x_1.....x_n) = |\{(i, j) \mid i < j \wedge x_i > x_j\}|$$

Definition 4.4: Measure of Presortedness

4.3.3 Defining Array Sortedness

Two approaches have been identified to define the sortedness of an input list for the framework. It can thus be seen that the concept of sortedness is not a universal one. For example, the list [4, 1, 2, 3], which is given as an example in Section 4.3.2, is a list with one item out of order. Whilst the results from the Step-Down-Run method are only minimally impacted, the *mop* result is significantly changed. However, an investigation into the relationship among the sorting algorithms and sortedness measurements is beyond the scope of this discussion.

The framework will measure sortedness by adapting the *mop* method, since it accounts for the overall effect of each item relative to all other items in the list. The framework's measurement adapts the *mop* value to a range between -100 and 100 percentage sorted (where -100 is reverse sorted, 100 is perfectly sorted, and 0 indicates an unknown random order), which will be more intuitive for users than a standard *mop* value. The number of all possible combination pairs is thus needed (Definition 4.5).

$_nC_k \equiv \frac{_nP_k}{k!}$ *where n is the total number of items and k is the number of items to be selected from the list as a unique combination*

∴ The number of possible combination pairs of n-sized list is $_nC_2 = \frac{_nP_2}{2}$

Definition 4.5: Possible combination pairs

The new measurement takes the *mop* value (the total item pair combinations that are out of order) as a fraction of the total possible pair combinations, and then maps the value to a range of between -100 to 100. The calculation is performed as $((\frac{mop}{_nC_2} \times 100 - 50) \times 2)$, producing a percentage value which will be used as a measure of sortedness.

4.4 Data Structure

Algorithms spend a considerable portion of execution time manipulating data structures, using them as a means to compute a solution, and in certain cases also presenting the data structure as part of the solution. In the framework, the data structure component is used to store data which drive an algorithm's operations with the aim of producing interesting events (Section 4.4.1), to use the data structure's data values to construct visual representations (Section 4.4.2), and to provide a common reference for the cost of time associated with each event (Section 4.4.3).

The data structure component plays an important part in the framework for obvious reasons. Algorithms in general, whether implemented specifically to be animated or

not, make use of data as an input for processing, for performing intermediate operations, and for presenting the resultant output. Algorithms must necessarily operate on a data structure in order to accomplish its algorithmic objectives, such as a sort algorithm sorting operating on an integer array to organise the elements into an ordered state. Thus, a data structure must support the basic read/write operations utilised by algorithms, allowing them to extract data for evaluation, and copy into or overwrite existing data.

4.4.1 Accessing Data Structure from Interesting Events

The data structure in the framework must complement algorithms in the process of producing interesting events, which describe the operations performed by the algorithms. Most of the interesting events that occur during algorithm operations involve some usage of the data structure. As a result, information relating to the data structure is often needed in the process of converting the events into an informative visualisation.

The data structure used by the algorithm often forms a part of the input parameter for the capturing of interesting events. For example, when data elements are compared or swopped, or when a new data structure is declared, the associated interesting event must record the data list, and the positional index of the data accessed within the list. This information in turn allows the values of the data involved in the event to be queried when required (Figure 4.6). The event will thus contain adequate information for visualisation purposes. Furthermore, the information contains enough detail to support the display of context-sensitive text to narrate the operations of an algorithm

animation. The feature was discussed in Section 3.4.2, and forms a part of the requirements supported by the framework.

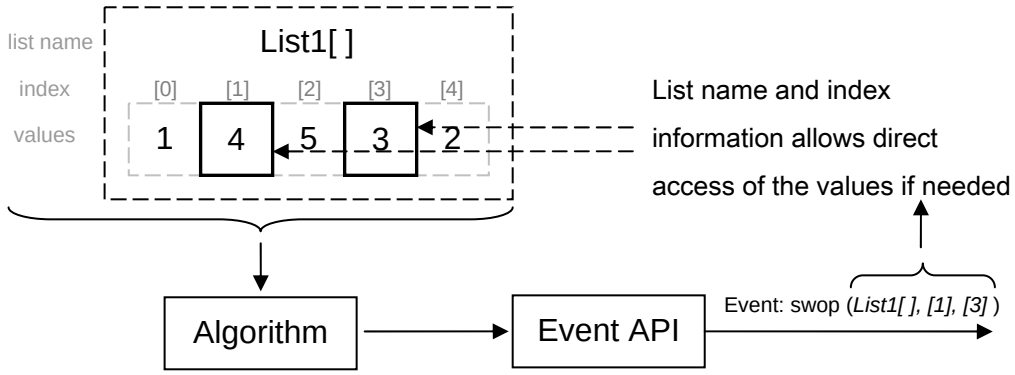


Figure 4.6: Information captured by the algorithm as part of an interesting event

4.4.2 Visual Mapping of Data

Experience and literature (Section 2.3.1) have also shown that numbers are easier to map to a visual representation due to their scalar nature (Figure 4.7), as opposed to other datatypes, such as character, string and Boolean values, which have no intuitive visual representation that offer an easily perceptible contrast of relative size differences. The framework will exclusively support the use of integers as the primitive datatype used in its data structure for algorithms. Also, for practical reasons, the use of only integers will allow for a simpler implementation process. Other datatypes may be incorporated as *virtual* data, which will affect the time-cost associated with various operations. However, the data will contain no values, and will not be used by the algorithm or represented in animations. This concept is discussed in the following section.

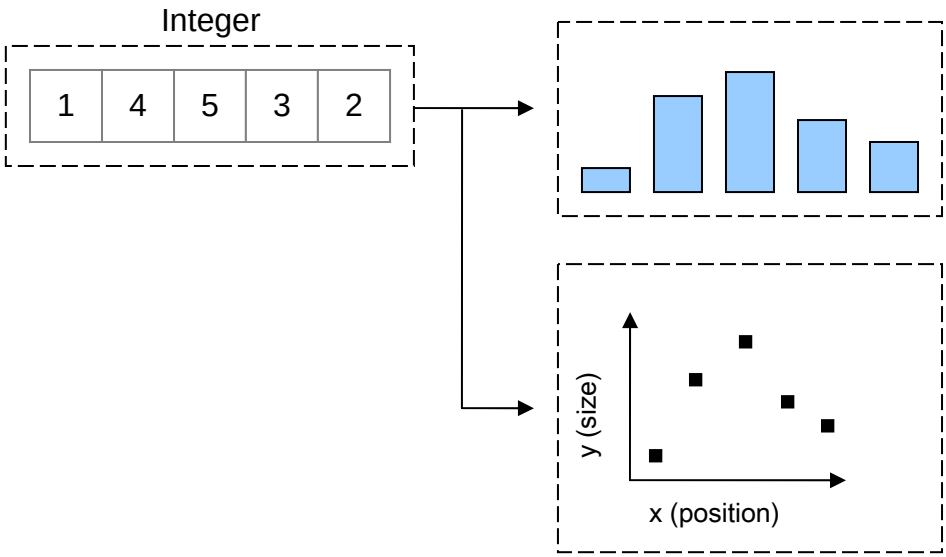


Figure 4.7: Integers are easier to represent in an intuitive visual form[10]

4.4.3 Operations cost

When algorithms operate on a data structure, there is an associated time interval required for the computer to work with its physical memory. The time taken to perform the operation is dependant on two factors. The first factor is the memory "weight" of the datatype, that is, the memory space required to store a single instance of the datatype. For example, a Character datatype of value `A` will take considerably less space in memory than a String datatype of value `Algorithm Animations`. Operations involving a Character type will thus use less time than one involving a String type. The second factor is the nature of the operation performed on the data structure, whether data is being compared or exchanged. Given an identical instance of a datatype, a compare operation is faster than a exchange operation.

[10] The visual styles are further discussed in Section 5.5.

An animation should be capable of reflecting the time-cost characteristics of the algorithm's operation on a data structure. However, it is not practical to try and capture the time-cost for each algorithm operation, since inconsistencies may result due to performance differences among different computers. Inaccuracy may result due to lag from additional operations performed by the driver algorithm during execution, such as the operations to capture interesting events.

Thus, a method is devised to provide a standardised way for simulating time-cost for algorithm operations within the framework. The framework's data structure is designed to store a *virtual memory size* of the data, which provides a theoretical size of the datatype used. Each operation which works with the data structure can thus be associated with a virtual time cost based on the data's memory size. Each data structure consists of two memory size information: the *total memory size* is the sum of each element's memory in a complex datatype[11], and the *compare memory size* is the size of the element field within the datatype which is used for relative comparisons among data. Total memory size and operation memory size will be identical if the data structure uses a single primitive datatype.

When working with complex datatypes, certain operations of an algorithm might work on a subset of data element fields, such as the use of a single primitive within a datatype containing multiple primitives. An example is illustrated in Figure 4.8, where each primitive type is given a theoretical memory size to contrast the memory used for two different operations. In the example, an array of datatype *Student* which stores the name, student number and grade result of students is declared. When the array is to be sorted based on student number, the algorithm reads the student number field to

[11] Examples of complex datatypes include Classes in C++ and Records in Delphi.

compare two *Student*s, but when two *Students* need to be exchanged, all data fields must be accessed and modified. Under such circumstances, exchange operations, which work with relatively larger amounts of memory than compare operations, will have a relatively higher time cost.

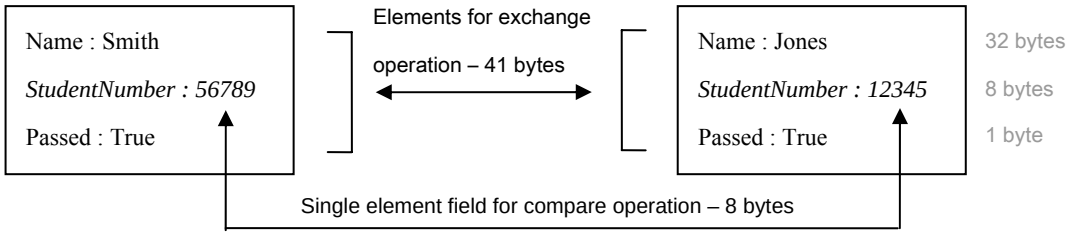


Figure 4.8: Effect of data size on operations

Figure 4.9 summarises the role of the data structure discussed. The data array allows algorithms to perform their operations by reading from and writing to its values. The event API can also directly extract information from the data array. Algorithm events which operate on the data array are assigned a time-cost based on the virtual time information stored in the data structure.

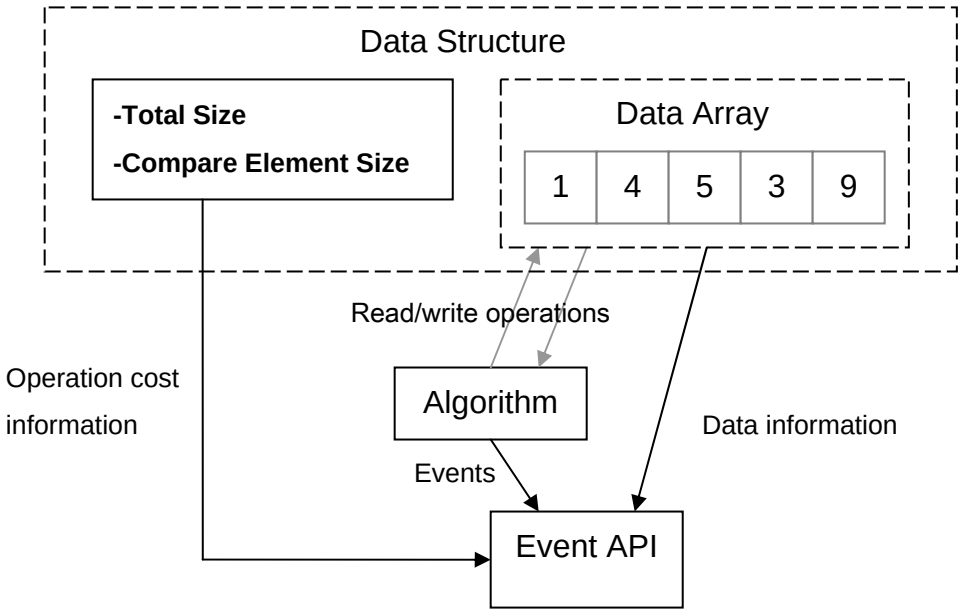


Figure 4.9: Functions of the Data Structure

4.5 Algorithm

The algorithm component is used to store a collection of algorithms to drive animations. Algorithm animations are derived by first observing the activities of an algorithm during its execution, and then visualising the activities. The algorithm which is observed provides information for the content of the animation, and is referred to as the *driver algorithm*. The purpose of the driver algorithm is to generate a list of execution traces called interesting events, a concept which is a fundamental part of the imperative paradigm (Section 3.3.1).

4.5.1 Driver Algorithm

In the framework, interesting events are generated by running algorithm procedures in conjunction with the data structure types specified within the framework. The driver algorithm is annotated with event calls to an Event API, which captures the activities of interest within the algorithm. The event calls consist of program code which are inserted into, and form part of the executable statements of the algorithm, though the calls do not change the semantical structure of the algorithm. The concepts are illustrated in Figure 4.10 (algorithm annotator is discussed in Section 4.5.2).

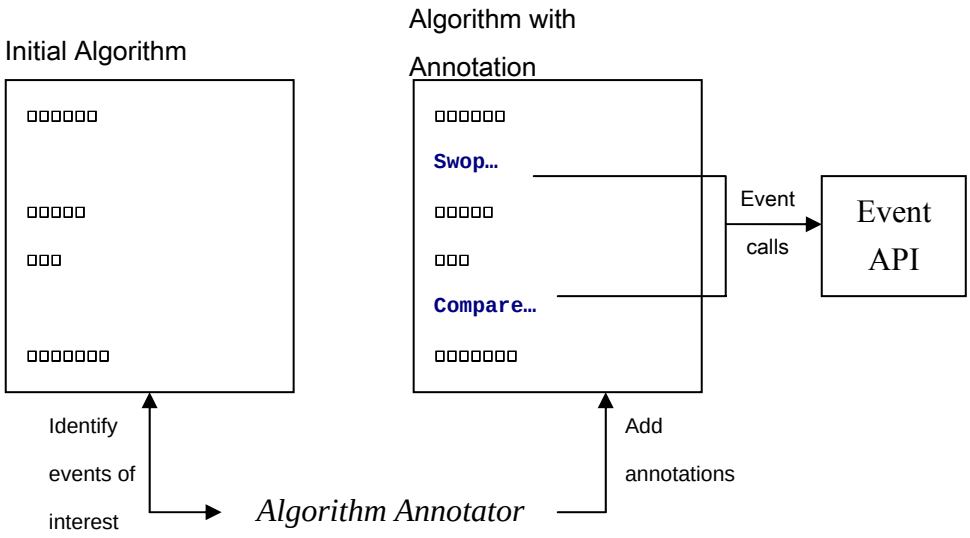


Figure 4.10: Annotation of a driver algorithm

The creation of a driver algorithm involves coding the actual algorithm, and then annotating the algorithm with event calls. First, an algorithm is written in the algorithm repository component. During implementation of the algorithm, declaration of data that will be used in the visualisation is restricted to data structures predefined in the framework. This ensures that visualisations can later access the data for display purposes. The implemented algorithm will initially be functionally identical to algorithms typically written in laboratory practical sessions, and can be run and tested through input created from the input generator. Validating the semantical correctness of the algorithms will ensure that the list of interesting events captured is an accurate account of the algorithm activities, in turn ensuring that the associated visualisation will be accurate.

Once the algorithm is implemented and tested, the phenomena of interest for the algorithm are identified. Appropriate event calls to the Event API are then inserted into the algorithm, which will result in the capture of an interesting event when relevant sections are executed. The event call specifies the type of interesting event

that has occurred, and provides input parameters to the call, which contain information relevant to the event, such as data values or output messages (code examples are provided in Section 5.3.1).

4.5.2 Algorithm Annotator

At this point, the question arises: "which user role is responsible for annotating an algorithm?" In the discussion of an algorithm animation model, Brown (1988a) defines a user role called *algorithmatician*, whose function it is to implement the algorithm, and annotate it with appropriate event calls. This approach is logical, since the person implementing the algorithm is expected to possess reasonable understanding of its concepts and operations, and thus most able to identify the events which may be interesting for visualisation. However, the visualisation designer, who is responsible for crafting the visualisation from the algorithm's event outputs, is also expected to have a good understanding of the original algorithm's operations in order to produce an animation suitable for the given scenario.

Based on the argument, and within the context of the proposed framework, the role of *algorithm annotator* can be taken up by either the algorithm programmer or the visualisation designer. Although the algorithm annotator is not directly involved in all of the processes for creating an animation from scratch, the algorithm concepts and final animation must still be understood in order to bridge the two mediums.

Events of interest for an animation vary from one algorithm to another, and it is the task of algorithm annotators to identify and evaluate the relevance of each type of event. With that said, the formula of identifying interesting events is not hard-set.

Thus, different algorithm annotators might identify different events of interest when provided with an identical algorithm. While there are certain events that are essential for accurately portraying an algorithm, there are events which can be considered optional.

For example, in a Bubblesort algorithm, events involving swopping items and comparing items must be captured, since these are the events that ultimately enable each item in a list to be placed in the correct order. In the same example, the capturing of concept events, such as noting that an additional item at the end of the array is sorted, may be considered optional (Figure 4.11). Thus, an algorithm can contain different annotations based on the preferences and perspectives of the annotator, depending on what optional events are seen as helpful in demonstrating the algorithm in the given circumstances (Section 2.5).

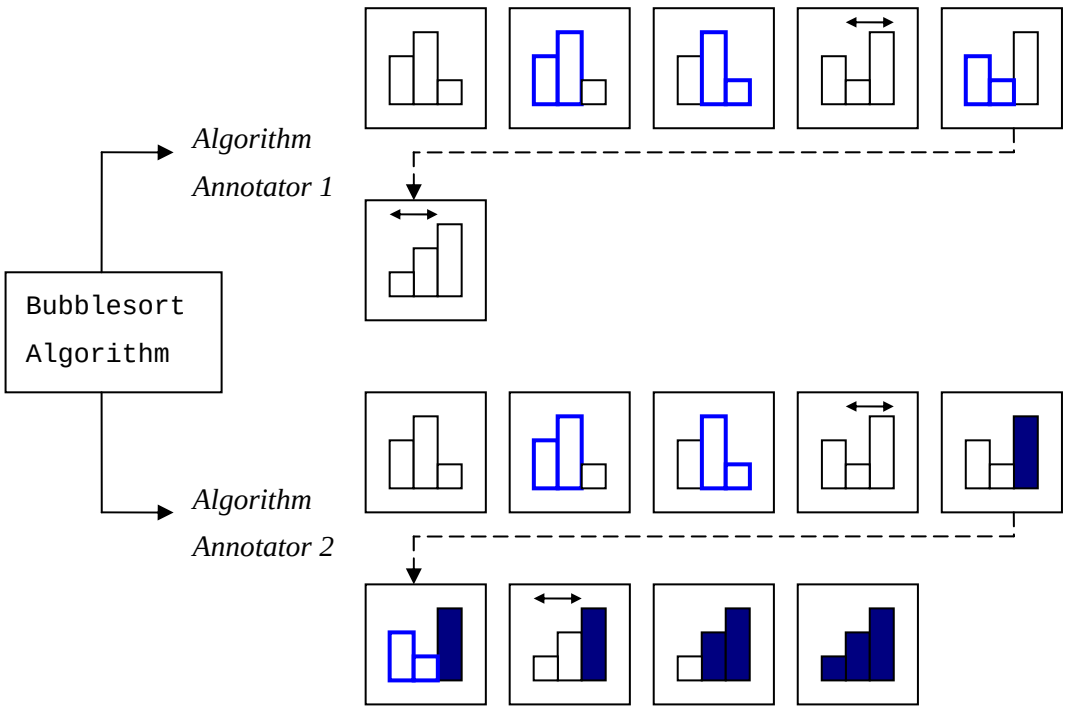


Figure 4.11: Different algorithm annotators may see an algorithm differently

4.6 Event API

An event script is a list of interesting events of a single execution of an algorithm on a given data structure, representing a recording of the algorithm operations which are pertinent to the algorithm's visualisation presentation (Section 3.3). An interesting event consists of a name to identify its event type, and additional parameters which are applicable to the event type. Event scripts are designed to offer a generic method of capturing the workings of an algorithm, whereby interesting events are captured without the need to consider how they are to be represented visually. Thus, algorithm annotators can annotate an algorithm with the explicit purpose of producing an animation, but need no detailed knowledge of the implementation of events into animations.

The Event API contains a pre-defined collection of interesting events. Additional event definitions can be added to the API if required, such as when new events are identified for existing algorithms, or if new types of algorithms are added to the framework. The events utilised will differ depending on the algorithm, but algorithms of similar functions or domain can make use of certain common event types. For example, a *swop* event can be used across all sorting algorithms, and may be taken to imply the same operation (although the event's visual representation may differ). Thus, the Event API offers a method to standardise on event types which are utilised among groups of algorithms.

The purpose of the event API is to respond to calls from an annotated algorithm's event markers, create instances of the interesting event with all relevant parameter

inputs, and append the interesting events to an event script (Figure 4.12). The event script then serves as the output of the framework's data layer.

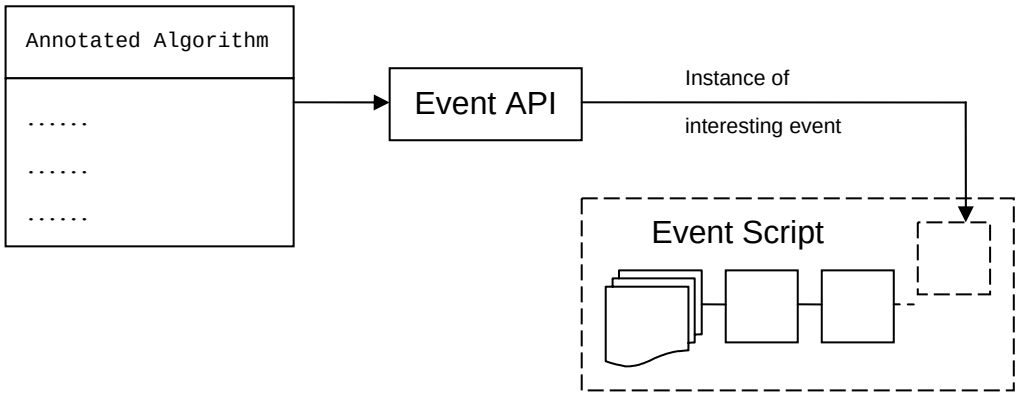


Figure 4.12: Function of the Event API

4.6.1 Event Classes

Four classes of events are specified by the framework. A *data structure definition event* stores information on a given data structure to be used by the algorithm. The information provided by the event allows the visualisation to examine the data structure, and assign it a visual representation. An instance of the event is created for every data structure that is used by the algorithm, except for data declarations which are not presented in the final visualisation, which need not be captured. Data structure definition events are invoked by the driver algorithm before the execution of the main algorithm procedures. This allows the visualisation to understand subsequent events which make use of the data structures.

Operational events communicate the crucial operations of an algorithm. The event type consists of operations which directly utilise the data list of an algorithm. Each event contains an attribute to identify its operation type, and a direct reference to the

data list and index that is used for the operation. The operations consist of two types: operations which change the state of the data structure, such as *write* operations, and operations which do not, such as *read* operations. An operation event is the only event class that is associated with a time-cost. Within the framework, algorithm operations are given a time-cost as a standardise method of portraying algorithm speed. The speed is derived according to the type of operation performed, and the virtual memory size of the data structure.

Conceptual events store information on events which represent abstract concepts that do not directly affect the algorithm's functional goals. In other words, the driver algorithm did not explicitly perform a related operation which is logically linked to the event. The purpose of conceptual events is to provide additional information which will aid in the visual demonstration and explanation of the algorithm. For example, in a Bubblesort algorithm, by marking sorted items with a different colour, the students can understand why the algorithm sorts faster after each iteration (Figure 4.11), since it is visible evidence that the algorithm no longer re-examines sorted items.

Message events are used to send textual information through to the final algorithm animation's peripheral views. A message event stores a textual message, which can be a combination of static text and information based on the context of an algorithm's execution. The event type is used as a method for the algorithm programmer to convey algorithm related information to the animation viewer, acting as a form of real-time narration of the algorithm.

4.6.2 Abstraction of Algorithm Operations

There are a number of common procedures which are utilised within algorithms. In certain cases, a single interesting event is used to represent a group of code statements within an algorithm. Algorithm animations aim to provide a high-level representation of algorithms (Section 2.3), thus certain operations may be simplified (or even omitted) in order to place focus on algorithm concepts rather than low-level details. For example, the event of swopping two items to place them in relative sequence is common among the sorting algorithms. In an exchange of two items, a third item is typically employed to temporarily store one of the item's values, whilst the items are copied over to their respective positions. As a result, three operations are used to shuffle the items. When generating an event script, the operations are typically summarised into a higher level concept of a *swop* event (Figure 4.13). This method is employed in algorithm animations to demonstrate algorithm concepts whilst reducing unnecessary details. However, as illustrated in Figure 4.13, a detailed representation is nevertheless possible.

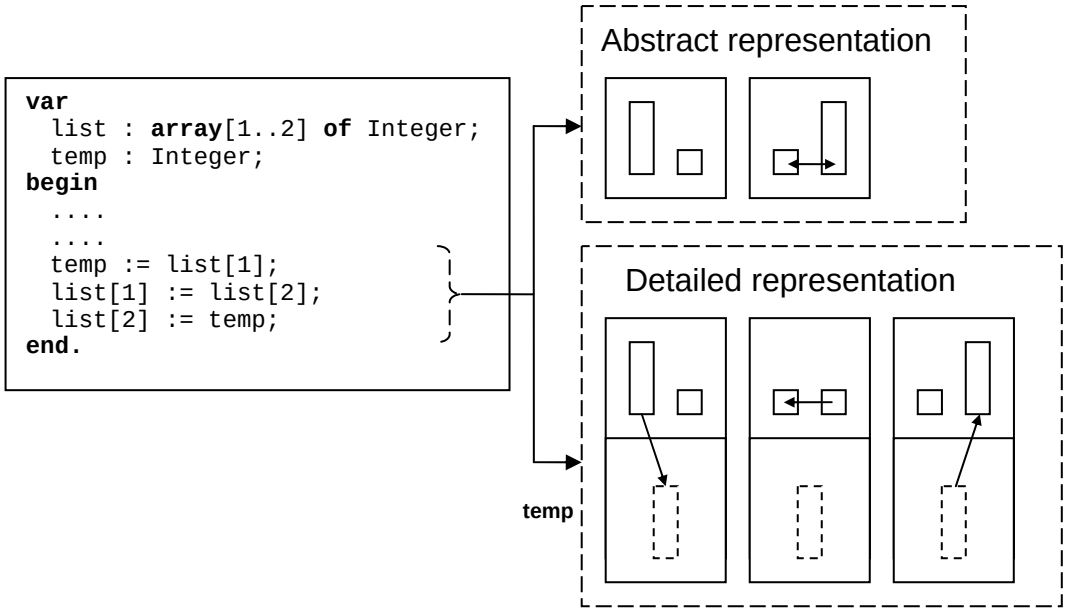


Figure 4.13: An algorithm may be presented in different levels of detail

4.7 Interpreter

The function of the interpreter is the conversion of an algorithm's events into its visual presentation. The implementation of interpreters is performed by the visualisation designer. In order to understand the purpose of the interpreter layer, one must look at the layer's input and output. The input consists of an event script from the data layer, which provides a detailed description of the algorithm's operation and data structure related events, without any specifications regarding the events' visual presentation. The output is a list of low-level graphical commands sent to the animation layer, which details the exact actions of the visualisation, including visual appearance, colour, visual object co-ordinates and action timing. However, no information is retained of the source algorithm's purpose. A process is thus needed to transfer one medium of communication into another.

The interpreter is used as an intermediate component to convert event scripts into graphical commands. This allows for an easier process of designing algorithms and animations. The driver algorithm only needs to be annotated with event callers to register interesting events, which are then collected into an event script and processed by the interpreter after the algorithm has completed execution. This approach eliminates the need to introduce into the driver algorithm additional code and data declarations which control the visualisation display properties. As a result, the algorithm programmer and the visualisation designer need not be integrated as a single role, since the coding responsible for the designing of the visualisation is not merged as part of the driver algorithm. Furthermore, when creating new methods of representing an existing algorithm, a new interpreter can be designed to interpret the same event script from the algorithm, rather than rewriting the driver algorithm in order to embed new visualisation instructions.

4.7.1 Component Structure

The basic premise of an interpreter's input interface is to accept interesting events which are relevant to a particular algorithm, and understand how the events are to be converted. Each event is taken and forwarded to an *event interpretation procedure* which handles the event. Within the procedure, event details are extracted and graphical commands are generated to represent the event. The details accessed include the event type, text messages, event time-cost, and data structure information (Section 4.4), which determines the parameters for the graphical commands. The visualisation designer is responsible for implementing the interpreter's procedures based on the type of events produced by the driver algorithm, and the designer's envisaged outcome of the visualisation. Each interpreter is typically customised to process the

events from one algorithm (some flexibility is nevertheless possible, see discussion in Section 4.7.2). Thus, an interpreter which handles a Quicksort algorithm will only expect events related to the Quicksort, and will not need to know how to handle events related to a Mergesort. In order for the interpreter to process events, a simple event router is used to identify and forward each event to the appropriate interpreter procedure.

A function of the interpreter is to control the properties of the visualisation and its graphical objects, handling the role in place of the driver algorithm. The interpreter component will be defined with a section to hold global properties of the defined visualisation, such as the type of graphical objects to use, and the colours to use for the objects. These properties are set by the visualisation designer. Furthermore, a controller section is required to manage the coordinates of all graphical objects created by the interpreter, with each object tracked by a unique ID based on data structure information. Any changes in object coordinates are kept up-to-date within the controller. This allows the event interpretation procedures to access object coordinates to generate graphical commands, and update the coordinates based on the events processed. The discussion in this section is illustrated in Figure 4.14.

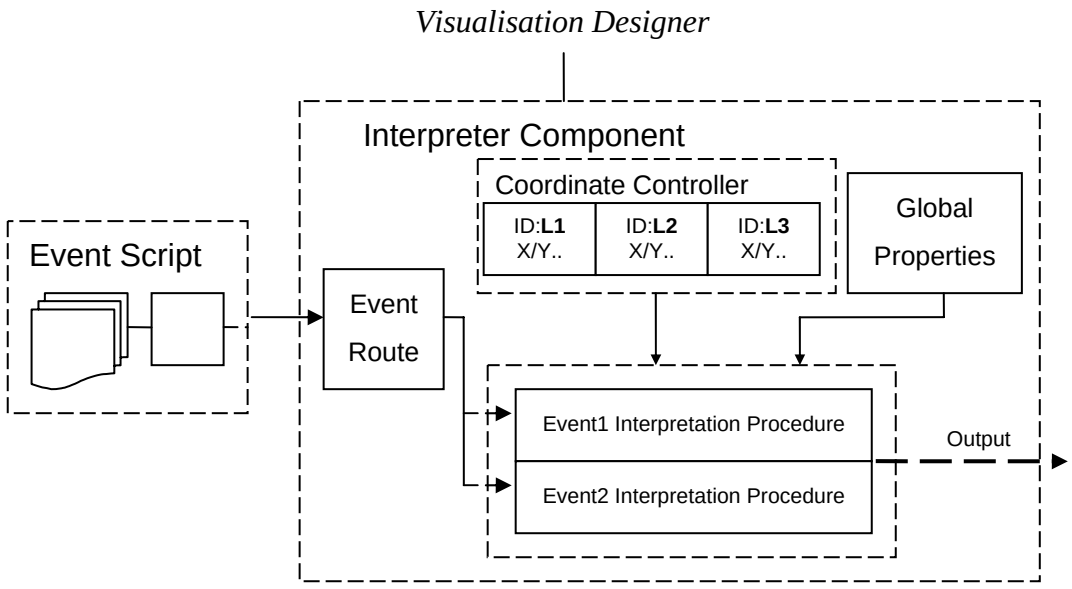


Figure 4.14: Interpreter structure and operations

4.7.2 Design of Interpreters for Related Algorithms

Algorithms which perform a similar function commonly work with a limited set of operations. In the case study of five different sorting algorithms, the common operations they employed can easily be summarised: namely *compare* and *swop* of two data items. This characteristic provides for a (circumstantially) convenient approach for reducing the implementation time of interpreters. Since functional equivalent algorithms typically perform identical operations, the visual representations are also similar. Thus, an interpreter designed to work with the event script of a particular algorithm can, in certain cases, be employed to interpret event scripts of other functionally similar algorithms.

4.8 Animation

The animation layer accepts graphical commands in the form of a defined scripting language, and generates an animated visualisation based on the commands. The function of the scripting language is to provide for a defined method of specifying graphical output to the end-user's display.

There are two sources of graphical commands, the animation component may either acquire input from the data layer (through the interpreter layer), or by taking the input directly from a text-based file. The animation layer is thus designed to be able to work independently of the data layer and the interpreter (discussed in Section 4.2.2).

4.8.1 Scripting Language

The scripting language of the proposed framework is designed to be simplistic and minimalist, and draws from the very basic common features of existing animation scripting languages. The proposed scripting language takes the basic feature set from extant languages, and proposing small changes to include support for the requirements of the framework. Various advance features of the extant languages[12], such as the layer ordering, domain-specific or advanced graphical objects, and graphic object grouping, are not incorporated into the current proposed language. Expanding the animator with new commands at a later stage is, however, possible. The main goal of the proposed scripting language is to include support for giving a time-cost to certain actions (see ActionTimed definition), thus allowing the "time-cost" for executing certain operations to be produced visually. A single time unit within the animation is

[12] See (Stasko 1997; Rößling 2000; Rodger 2002)

defined as 50 milliseconds. The proposed language is defined in this section and briefly discussed.

The **begin..end** block (Definition 4.6) allows multiple operations to be executed simultaneously. All commands within the block are run as a group. When individual commands within a command block finish at different times, the block command is considered done only when all individual commands have completed their operations.

```
commandBlock :
  begin command* end
```

Definition 4.6: Begin..end command

There are three main groups of commands within the scripting language (Definition 4.7), namely the **graphicObject**, **action** and **actionTimed** commands. Each of the commands are defined in the following discussions.

```
command :
  graphicObject | action | actionTimed

graphicObject :
  rectangleBuild

action :
  actionChange | textMessage

actionTimed :
  actionChangeTimed | moveRelative | delay
```

Definition 4.7: General command definitions

The **graphicObject** command is responsible for creating visual objects to represent data values. The command structure for creating and specifying the properties of a rectangle is provided in Definition 4.8.

rectangleBuild:
  rectangle *ObjectID x y width height fillcolour bordercolour*

Definition 4.8: Rectangle visual object command

Graphic objects contain properties which are either numbers (for coordinates or dimensions) or strings (for colours), specified in Definition 4.9.

ObjPropertyNumber :
  x | y | width | height
ObjPropertyColour :
  bordercolour | fillcolour

Definition 4.9: Visual properties command

Two actions are used to modify the properties of graphic objects. The **change** action (Definition 4.10) effects a change without associating a time-cost. The result of the action thus occurs immediately on-screen. The **changeTimed** action (Definition 4.11) takes an additional parameter in standard time units. The action thus occurs only after the given time has elapsed.

actionChange :
  change *ObjectID* [ObjPropertyNum *Number*] | [ObjPropertyColour *Colour*]

Definition 4.10: Change properties command

actionChangeTimed :
changetimed *TimeUnit ObjectID* [ObjPropertyNum *Number*] |
[ObjPropertyCol *Colour*]

Definition 4.11: Timed change properties command

The **progressmessage** command (Definition 4.12) is unique in that it does not directly affect the display of the animation. The message will be presented in a separate view attached to the animation, displaying messages during the running of the animation.

textMessage :
progressmessage *Message*

Definition 4.12: Message command

The **moveRelative** (Definition 4.13) is a timed command which is specifically used to change the coordinate of a graphic object. The command creates a linear interpolated movement of the graphical object, with the movement finishing based on the time parameter supplied. Smooth motion can thus be supported by the animation component. The command makes *x* and *y* axis movements simultaneously.

moveRelative :
moveRelative *TimeUnit ObjectID move-x move-y*

Definition 4.13: Animated movement command

The **delay** command (Definition 4.14) simply makes the animation time lapse for a set time without updating the properties of the graphic object on the visual display.

delay :
  delay *TimeUnit*

Definition 4.14: Time delay command

4.8.2 Animation Engine

The purpose of the animation engine is to interpret a set of graphical commands into an animated display. Each view of an algorithm is handled by a single instance of the animation engine. The engine is supported by a command parser, command interpreter, action repository, graphic object repository, and display renderer (Figure 4.15). The parser takes as input the graphical commands, from where the commands are tokenised and given to the command interpreter. The action library stores a repository of action classes. The graphic object library, similarly, stores a repository of graphic objects. The command interpreter examines the commands and parameters, and based on these, requests an instance of the action or graphic object, in which the command parameters are set. The action or command is then passed into the processing queue of the animation engine, which will render the scene to a view.

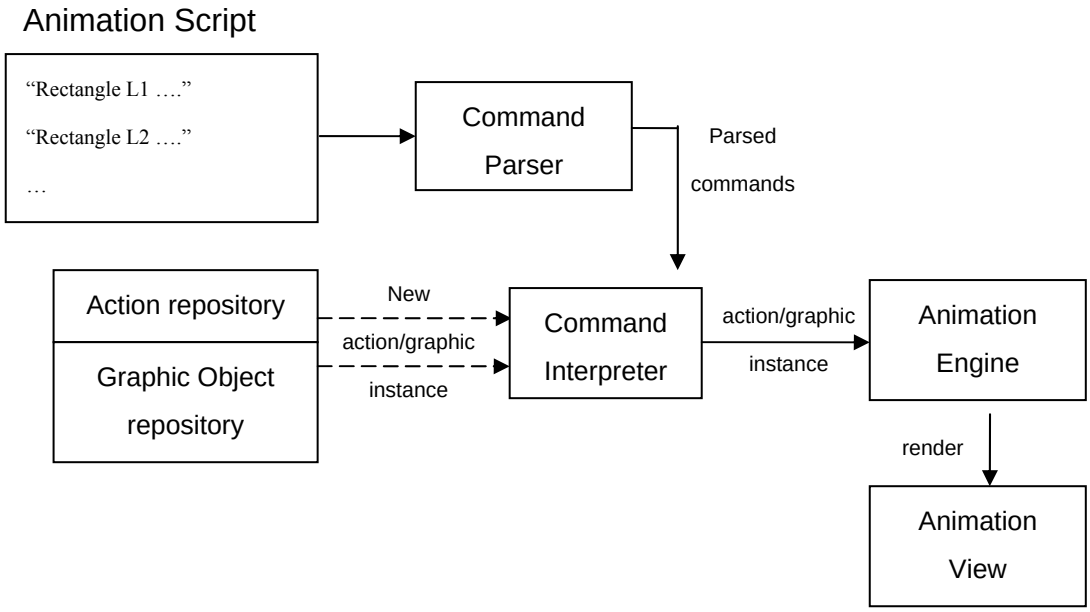


Figure 4.15: The components supporting the animation engine

The animation engine works based on an internal processing queue and graphic object controller. The processing queue stores all actions awaiting execution, each action is executed once during one clock tick. Actions which have already being executed (for un-timed actions) or have expired (for timed actions) are automatically removed from the processing queue. The scripting language is designed to allow a time-to-live for timed actions (**actionTimed** commands).

4.9 Interface

The components of the data layer and animation layer are connected to associated user interfaces, allowing specific inputs from and outputs to end-users (students and instructors). The purpose of the interfaces is to provide the end-users with a consistent method of interacting with the animations and its related options, independent of the type of algorithms being observed, or the style in which the algorithms are visualised. The data layer interface connects to the data layer, and the animation layer interface

connects to the animation layer. The implementation of the interfaces is discussed in Section 5.4.

Figure 4.16 summarises the inputs provided by the data layer interface to each of the components. The data layer interface enables end-users to provide parameters to the data generator component, provide time-cost information to the data structure component, select driver algorithms from an algorithm class repository (Section 5.2.3), and specify algorithm animation views by selecting an interpreter from a class repository.

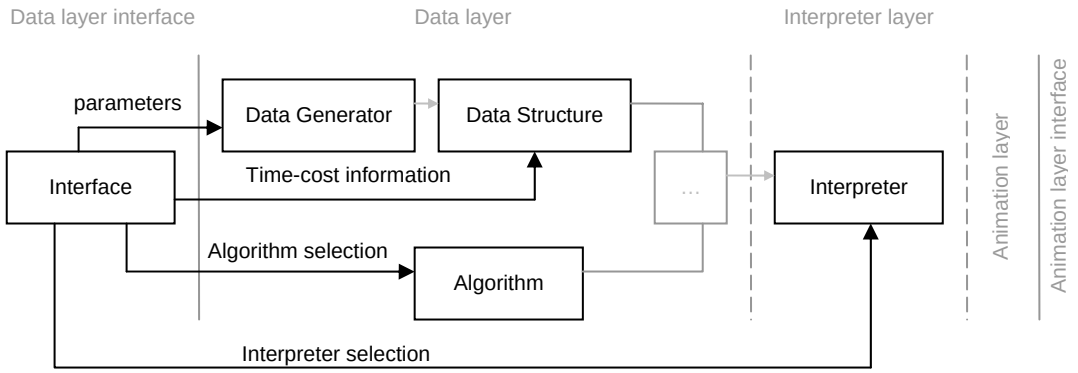


Figure 4.16: Data layer interface functions

Figure 4.17 shows the inputs provided by the animation layer interface to each of the animation components. The animation interface provides a unified control to direct the display of animation views. The interface can handle single animation view or multiple animation views running simultaneously. When multiple views are presented, all views will accept the same input from the interface. The interface sets the speed of the animation, and also contains options to play, pause and step through an animation.

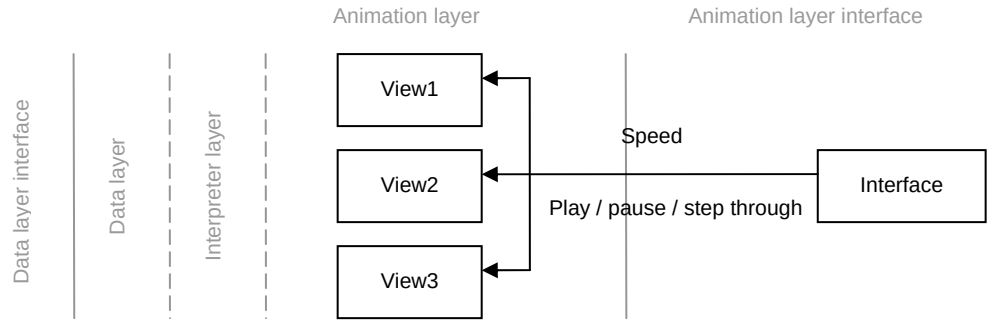


Figure 4.17: Animation layer interface functions

4.10 Conclusion

The algorithm animation framework offers a theoretical foundation for the construction of an algorithm animation system. The framework was created based specifically to support the functional and pedagogic requirements outlined in Chapter 3.

Chapter 4 discussed the structure of the proposed framework, which organised the framework components into functional layers. The framework structure was designed for extensibility and ease of implementation. The core of the framework consists of the data layer, interpreter layer, and animation layer. The processes within each layer are combined to allow generation of animations from algorithms. The data layer manages the execution of algorithms in order to capture interesting events. The events are converted into graphical commands through the interpreter layer. The graphical commands are then processed by the animation layer to produce an animated visualisation.

Each framework layer is made up of a component or group of components, each of which were discussed in detail. Within the discussions, the expected inputs and

outputs of each of the components were presented to illustrate their connections and interdependencies. Focus was placed on the design of the components and motivations for the designs. The next chapter discusses the implementation of a prototype based on the proposed framework, thus providing a practical perspective on the theories and concepts of the framework.

Chapter 5

Algorithm Animation Prototype

5.1 Introduction

The proposed algorithm animation framework, presented in Chapter 4, is designed to support a number of requirements which were identified and discussed in Chapter 3. The framework design covered the structure of the proposed framework. Discussions and motivations were presented on general design considerations and each of the framework's components. The final proposed framework is presented as a design concept to guide the implementation of an algorithm animation system.

This chapter, in turn, covers the implementation of a functional prototype system to demonstrate the effectiveness of the proposed framework (Section 5.2 and 5.3). The critical components will be examined in detail, whilst simple or auxiliary components will only be mentioned briefly. The prototype implementation will serve as an evaluation of the framework design. In addition, practical experience can be acquired by using the prototype to create sorting algorithm animations, which form the case study of the research (Section 5.5). A brief overview of the user interface to support end-user creation and viewing of algorithm animations is also provided (Section 5.4).

Issues observed during the implementation of the prototype and case study will highlight various aspects of the framework which may or may not be practical (Section 5.6).

5.2 Implementation Techniques

This section discusses general decisions which were taken during the system design phase to ensure effective and successful implementation of the prototype. In the initial phases of the project, literature reviews and studies into extant systems provided a theoretical foundation of concepts within an algorithm animation environment. An understanding of the practical application aspects of the theoretical concepts was an important factor in the design of the framework and the implementation of the prototype system. Section 5.2.1 discusses the methodology employed to address this requirement.

A number of component classes of the prototype system must allow for declaration of multiple instances of class objects, forming an important requirement in support of generating multiple algorithm animation scenario and views (illustrated in Section 4.2.2). Section 5.2.2 discusses how the object-orientated model is used to support the process of generating multiple scenarios. In order to support effective construction and output of class instances on demand, specialised repositories classes are constructed which maintain a list of clonable class objects. The classes are maintained by using a hash index system (Section 5.2.3).

5.2.1 Prototype Methodology

In order to effectively complement theoretical findings of algorithm animation systems with practical knowledge, the iterative prototyping methodology (Goldman and Narayanaswamy 1992; Kendall and Kendall 1996) was adopted for the implementation of the system. The methodology regards implementation of prototypes as a means to gain understanding of the system concepts and implementation requirements, and also to note trade-offs of different implementation designs. The iterative implementation process also meant that with each cycle, workable components can be reused whilst the impractical components may be redesigned or incrementally modified to evolve the system into its final form.

A number of simple algorithm animation demonstration prototypes were continuously developed based on observations and results of early phase studies, serving as test-of-concept systems. Each of the demo prototypes was designed as a stand-alone implementation, allowing course instructors to display algorithm animations which were, notably, generated and rendered in real-time (Figure 5.1). Little flexibility is offered to the users. Functionalities such as algorithm and data input choices, animation display speed, and display content settings, were not included.

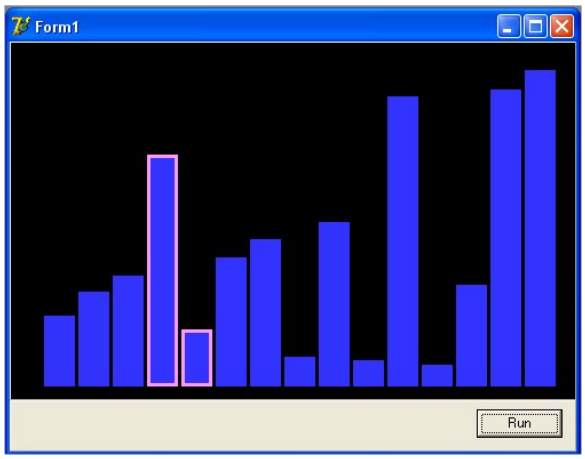


Figure 5.1: An early (feature free) prototype

5.2.2 Use of Object-Orientation

An important concept of the system is the support for parallel analysis of multiple scenarios, which requires the system to be able to support flexible associations between different combinations of data structure, algorithms and visual interpretations (Section 4.2.2 and Figure 4.3). In support of such requisites, an object-oriented approach is used to create the components, enabling multiple and different instances of each component to be generated, thus simplifying the process of generating unique cases which the end-users (students and instructors) would like to examine or demonstrate.

In addition to allowing for multiple instances of components, the object-orientated approach also allows for polymorphism (Cantù 2001). Certain class groups, such as algorithms and interpreters, are designed to cater for different scenarios, but share a number of common class interfaces or routines. This concept is highlighted in the component implementation discussions in Section 5.3, where inheritable and interface routines are marked with ***`virtual`*** or ***`virtual/abstract`*** directives, respectively.

These routines can then be inherited and extended while still maintaining an interface common to the parent class type. Flexibility can thus be gained by implementing with polymorphic design.

5.2.3 Class Repository

There are certain parent classes from which multiple instances are required, namely algorithm, interpreter, and graphic command actions and objects. These classes are provided with an abstract *clone* routine during implementation. During system initialisation, each parent class and its child classes are initialised and loaded into a predefined class repository. From the repository, an instance of any of its stored child classes may be requested by using a unique identification string.

An extract of a generic repository's interface is shown in Figure 5.2. The repository is constructed with a hash table, `FHashFactory`, which stores a key/class-object pair. The pairs are registered through the `RegisterPair` routine. Identification strings are used to search for and identify class objects. A generic instance of the class can then be constructed and sent as an output through the `GetClass` routine.

```
TFactory = class(TObject)
private
  FHashFactory : THashedStringList;
public
  ........
  procedure RegisterPair(KeyName : String; AAlgorithm : TSysObject);
  function GetClass(KeyName: String) : TSysObject;
end;
```

Figure 5.2: Extract example of a repository class interface

5.3 Discussions of Component Implementations

The implementation of the final prototype system includes the data generator, data structure definition, algorithm repository, event API, interpreter, animation and interface components (Figure 4.2). The focus of this section is on specific implementation techniques which support the goals of a component. Components which require straightforward implementations will offer little value to the discussion, and are thus not included. These include specifically the data generator and data structure definition components. Consequently, this section covers the algorithm and event API (Section 5.3.1), interpreter (Section 5.3.2), and animation (Section 5.3.3) components. The interface implementation is covered in Section 5.4. The framework's algorithm and event API components are implemented as a single unit in the prototype, with the motivation for this approach being addressed in Section 5.3.1.

The discussions are accompanied by relevant code extracts to illustrate examples from the implementation. The code extracts are presented using Pascal, created through Borland© Delphi™.

5.3.1 Algorithm and Event API

The algorithm discussion in Section 4.5 has highlighted the commonalities of algorithms which are designed for the same function, or are within the same domain. An algorithm parent class is used as a base class which different algorithm implementations may inherit from. Routines which are common to all algorithms are included as part of the base class and its inherited child classes. The most basic

routines consist of interfaces to execute the algorithm, and structures to store the event scripts generated from the execution.

In the system, the event API capturing routines are integrated as part of the parent class of the algorithm component, rather than being implemented as a separate unit. Using such a method, the event API routines can contain generic code to capture events. If modifications need to be made, the API can be inherited and rewritten as needed by child classes. Alternatively, an implementation of the event API separate from the algorithm implementation would mean that if particular algorithms need to capture events in a different way, a new event capture routine must be added to the event API class.

When different domains of algorithms are created, a new class catering for the domain is implemented and inherited from the base class. The new class will override parent routines or create additional interfaces based on new functionality requirements. New algorithm implementations within the domain may then inherit from the new class, gaining the routines and functionalities of the particular algorithm domain, instead of having to further modify and append the generic algorithm base class.

As part of the framework design, the interesting event approach is used to identify and capture events from within the driver algorithm (Sections 4.2.1, 4.5 and 4.6). An extract of the parent algorithm class is presented in Figure 5.3. The event API forms a part of the algorithm class (motivated above), with a routine for each of the possible event types to be captured been presented with a ***virtual*** directive. Support routines are used to direct the event calls, thus providing a more standardised interface for

algorithm annotators. The support routines are marked with ***overload*** directives to support different routine calls.

```
TAlgorithm = class(TSysObject)
  .....
protected
  FScriptList : TScriptList;
  .....

  procedure CaptureEventCompare(List1 : TDataList; Pos1 : Integer; List2 :
    TDataList; Pos2 : Integer);virtual;
  procedure CaptureEventExchange(List1 : TDataList; Pos1 : Integer; List2 :
    TDataList; Pos2 : Integer);virtual;
  procedure CaptureEventCopy(List1 : TDataList; Pos1 : Integer; List2 :
    TDataList; Pos2 : Integer);virtual;
  .....

  procedure CaptureEvent(AEventType : String; AList : TDataList);overload;
  procedure CaptureEvent(AEventType : String; AList : TDataList; Pos :
    Integer);overload;
  procedure CaptureEvent(AEventType : String; AList : TDataList; Pos1 :
    Integer;  BList : TDataList; Pos2 : Integer);overload;

public
  procedure Sort;virtual;
  procedure AssignDataList(AList : TDataList);virtual;abstract;

  function Clone(AScriptList : TScriptList) : TAlgorithm;virtual;abstract;
end;
```

Figure 5.3: Extract of the algorithm class interface

Figure 5.3 also shows a number of interfaces and routines common to all driver algorithm classes. The *Sort* routine, marked with a ***virtual*** directive, is responsible for activating the driver algorithm and performing initialisation actions, such as

clearing previous event scripts (`FScriptList` in figure). Routines marked with an additional **`abstract`** directive present a inheritable interface, which are overwritten and implemented by each child class. These common interfaces allow other components within the system to standardise routine calls to algorithm classes.

A driver algorithm may be separated into multiple routines, as is the case with Quicksort and Mergesort (which use various support routines), since the API event calls and data structures are accessible to all routines within algorithm classes. The only constraint is that the main execution routine must be linked to the `Sort` routine.

The actual implementations of the driver algorithms were, in fact, a fairly simple process, the theory of which is discussed in Section 4.5. The algorithm is created without any specific changes to indicate it as a driver algorithm, with the exception that data which need to be captured as part of an interesting event must use the system's integer-based data structure (Section 4.4). Simple console-based textual outputs can then be used to verify the correctness of the algorithm. Based on experience and literature (Mukherjea and Stasko 1993), preliminary testing throughout the implementation of new algorithms will minimise time spent debugging during the creation of visualisations. While it is true that algorithm animations can technically be employed as a debugging tool (Section 2.2), this approach will generally require relatively more time than by testing the functionality of the driver algorithm during its implementation.

Once an algorithm's function is verified, interesting event markers are then inserted into the code, using the approach introduced by BALSA (Brown and Sedgewick 1998). Figure 5.4 illustrates an implementation of a Bubblesort driver algorithm. The

addition of the markers does not generally hinder the readability of the algorithm, as the markers utilise a routine name which is easily distinguishable for the main driver algorithm code.

```
procedure TAlgorithmBubblesort.Sort;
var
  f : Integer;
  Sorted : Boolean;
  SortedNumber : Integer;
  temp : Integer;
begin
  inherited;
  CaptureEvent(eventLoadIntegerList, FList);

  SortedNumber := 0;
  repeat
    Sorted := True;

    for f := 0 to FList.Count - 2 - SortedNumber do
    begin
      CaptureEvent(eventMessage,
         'Comparing [' + IntToStr(f) + '] with [' + IntToStr(f+1) + ']');
      CaptureEvent(eventCompare, FList, f, FList, f+1);
      if FList.Items[f].Value > FList.Items[f+1].Value then
      begin
        CaptureEvent(eventExchange, FList, f, FList, f+1);
        temp := FList.Items[f].Value;
        FList.Items[f].Value := FList.Items[f+1].Value;
        FList.Items[f+1].Value := temp;
        Sorted := False;
      end;
    end;

    CaptureEvent(eventComplete, FList, FList.Count - 1 - SortedNumber);
    Inc(SortedNumber);
  until Sorted;
end;
```

Figure 5.4: Implementation of Bubblesort driver algorithm

Figure 5.4 highlights the routine calls to capture interesting events within the driver algorithm. Each of the event types identified in Section 4.6.1 are shown, namely the data structure definition event (`eventLoadIntegerList`), operational events (`eventCompare` & `eventExchange`), conceptual event (`eventComplete`) and message event (`eventMessage`). As specified in Section 4.6.1, data structure definition events and operational events are essential for a basic illustration of an algorithm. They are easily identified in a method similar to the declarative paradigm. In other words, any operations involving the data structure are marked as important. Conceptual events and message events require more in-depth knowledge of the algorithm to annotate, since it is used to superficially enhance or complement the animation to improve algorithm understanding, but do not directly represent any algorithm operations.

5.3.2 Interpreter

Each child interpreter class inherits from a base class which provides a common interface for accepting script events. An abstract interface is created for each type of interesting event which is usable by the driver algorithm (Figure 5.5). Thus, for each `CaptureEvent` routine in the algorithm base class, there is a corresponding script interpreting routine in the interpreter base class.

A new interpreter child class is created for each algorithm, or domain of algorithms (Section 4.7.2). The use of the abstract interfaces means that while only relevant routines need to be implemented, all interpreters are inherently capable of accepting any event type.

```
TProcessor = class(TSysObject)
  .....
protected
  FOutputScriptFileName : String;
  //universal properties
  FObjectColourDefault : String;
  FObjectColourHighlight : String;
  FObjectColourHighlightBorder : String;
  .....
  function ScriptCreateList(AScriptItem : TScriptDataStructure) :
    String;virtual;abstract;
  function ScriptCompare(AScriptItem : TScriptCompare) :
    String;virtual;abstract;
  function ScriptExchange(AScriptItem : TScriptExchange) :
    String;virtual;abstract;
  function ScriptCopy(AScriptItem : TScriptCopy) : String;virtual;abstract;
  .....
public
  FAnimationScriptFile : TextFile;
  .....
  function ScriptInterpret(AScriptItem : TScriptItem) :
    String;virtual;abstract;
end;
```

Figure 5.5: Extract of a interpreter class interface

Algorithm animation system research does not place much focus on low level algorithm-to-visualisation paradigm implementation details. That, in contrast with the amount of literature focusing on script-based animation systems (Stasko 1997; Rößling and Freisleben 2001; Akingbade, Finley, Jackson *et al*. 2003) and high-level paradigms (Brown and Sedgewick 1998; Roman 1998; Rößling and Freisleben 2000a; Rößling 2002), suggests that the intermediate interpretation process is open to some degree of flexibility. The implementation of routines for interpreting events thus made

use of iterative prototyping techniques, which allowed incremental development of the routines based on experience gained and limited documented resources.

```
function TProcessorDebug.ScriptExchange(
  AScriptItem: TScriptExchange): String;
var
  scriptStream : String;  pointList1, pointList2 : TPointList;
  objectID1, objectID2 : String;  offset1, offset2 : TPoint;
begin
  //obtain coordinates for converting to graphical commands
  pointList1 := GetPointList(AScriptItem.ListName1);
    .....

  AssignFile(FAnimationScriptFile, FOutputScriptFileName);
  Writeln(FAnimationScriptFile, 'begin');
  //move Object1 to where Object2 is
  offset1 :=
CalcRelativeMovement(pointList1.Items[AScriptItem.Pos1].PointPos,
            pointList2.Items[AScriptItem.Pos2].PointPos);
  scriptStream := 'moveRelative ' + IntToStr(AScriptItem.TimeToLive)
    + ' ' + objectID1 + ' ' + IntToStr(offset1.X) + ' ' + IntToStr(offset1.Y);
  Writeln(FAnimationScriptFile, scriptStream);
  //move Object2 to where Object1 is
    .....
  Writeln(FAnimationScriptFile, 'end');

  //update the object coordinates
  SwopID(pointList1, pointList2, AScriptItem.Pos1, AscriptItem.Pos2);
end;
```

Figure 5.6: Implementation of an interpreter routine for an Exchange event

A routine to interpret an event that exchanges two items is illustrated in Figure 5.6. The function first acquires the coordinates for the relevant graphical objects. Graphical commands are then created to plot the graphic objects towards their new

positions. Finally, the coordinate controller is updated with the positions of the now exchanged objects. Figure 5.7 shows an example of the output from the routine:

```
begin
  moveRelative 8 List70 8 0
  moveRelative 8 List71 -8 0
end
```

Figure 5.7: Graphical command example

The outputs from interpreters, in the form of graphical command scripts (Section 4.8.1), are placed in a standard ASCII text file. From experience, this approach has shown to ease the process of debugging. When problems are encountered during the implementation of an interpreter, the text file can be examined to find the possible sources of common errors, especially ones relating to command syntax and event interpretation. Command lines which are syntactically incorrect will result in parsing errors from the animation component. Interesting events, if interpreted in an incorrect or unsuitable manner, will often produce some unanticipated and unexplainable visual results.

5.3.3 Animator and Timer

The animator in the system is designed to accept graphical scripts stored in an ASCII-based text file, and render the graphics on the screen (Section 4.8). The *animation view* (Section 5.2.1) is thus functionally supported by the animator. The animator component renders animations based on the standard graphic language defined in Section 4.8.1.

The need to support parallel animation for comparative analysis (Section 3.6) means that support for multiple instances of the animator view is necessary. Most importantly, with more than one animator operating, the speed of each instance must be synchronised, in other words, each animator must run at an identical speed.

Each animator is designed to process operations based on a timer (Figure 5.8). Each time the timer ticks, it will send a signal to the animator to perform a single operation (discussed further below). Thus, the easiest approach to ensure that all animators run at the same pace is to let all animators work off a unified timer. To support this approach, a list of all currently active animators is maintained by an animator controller. Each time the unified timer ticks, the same signal is sent by the animator controller to each animator in turn, thus triggering all animators to process exactly a single pending operation. Once the operation is completed, each animator waits in a ready state until called again.

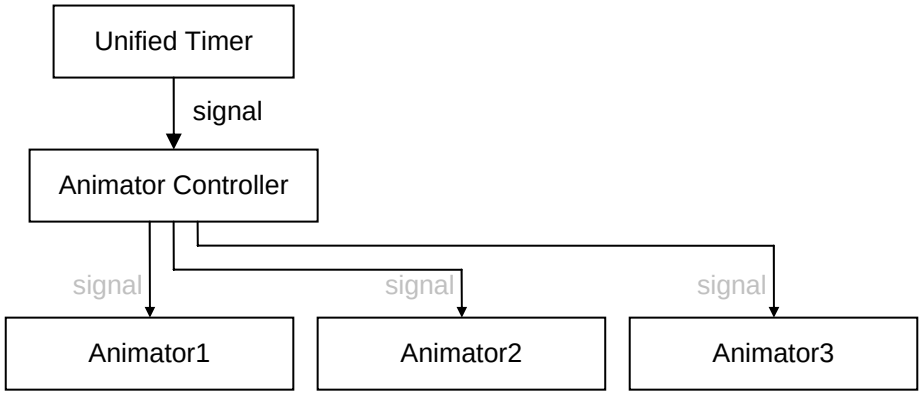


Figure 5.8: Multiple animators synchronised by the unified timer

An animator operation is divided into two main phases, an *update* phase and a *render* phase. The update phase consists of iterating through an internal processing queue (Section 4.8.2) consisting of commands to update the state of all graphical objects in

memory. An internal counter is then incremented to register the completion of a single operation. The render phase then iterates through the graphical objects, rendering each onto the animation view.

The functionality to control the speed of the animators is done by employing a standard time definition and timer multiplier. The standard time associated with a timer tick is 50 milliseconds (ms), thus, running the timer for one second (1000ms) would result in 20 ticks (and 20 operations). The time multiplier is used to artificially speed up or slow down the timer. To lengthen the time of an animation (in other words, to slow it down), the multiplier increases the standard time of the clock. This causes less time ticks to be registered per second, and thus reduces the frequency of calls to the animator.

Actual implementation showed that the abovementioned technique cannot, however, be used for increasing animation speed (that is, by increasing the ticks per second) for two reasons. Firstly, the implementation of a high-resolution timer[13] is beyond the scope of the prototype. Secondly, the refresh rate of the animator renderers will not be able to keep pace. In other words, even if a high-resolution timer is employed to register at 5ms intervals, the rendering speed of 200 frames-per-second is currently impractical.

An alternative approach was thus employed (Figure 5.9), whereby the animator operation is instead adjusted. Using this approach, when the multiplier is increased to

[13] Basic timers have limited tick interval register (Cantù 2001). For example, if the timer was set to run at ten-times speed of the animator's standard time (i.e. at 5ms), the timer may realistically only register intervals somewhat intermittently (i.e. between 12-35ms's). A multimedia timer linked directly to the OS may fair better. Regardless, it will not solve the overall problem, as explained in the main text.

speed up animations, the timer's resolution is maintained at 50ms, thus it will still only register 20 operations-per-second. However, the call to the animator is sent with an additional multiplier parameter. The animator then repeats the update phase of the operation the same number of times as the multiplier. For example, if the multiplier is set to 3×, the update phase will run 3 times when a single timer signal is received by the animator. Once the update phase is complete, the rendering phase is then run only once. Thus, regardless of the multiplier speed, the animation will run at an increased rate by firstly performing all update operation in the background before rendering the scene at the end. As a result, the renderer is only expected to draw at 20 frames-per-second, a reasonable speed for currently available PC-platforms.

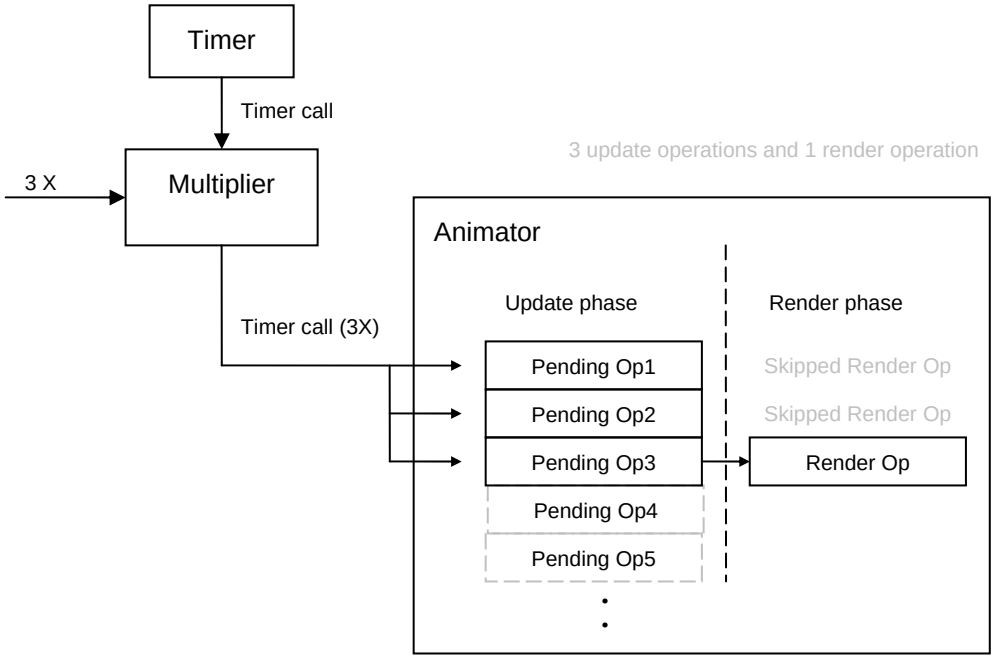


Figure 5.9: Animator processing a call with a 3x multiplier parameter

5.4 Interface Design

The role of the system interface is to allow users to access and make use of the features offered by the system. The implementation of the system interface follows the specifications of each of the data layer and animation layer interfaces presented in Section 4.9.

Each combination of data structure, algorithm and interpreter is defined as a *scenario*. Thus, when creating a scenario, the interface must enable end-users (Section 3.2.2) to make a selection of the algorithm, interpreter, and data list which make up the scenario. A unique scenario is generated for each unique case study to be examined. The parameters and settings for each scenario are controlled and setup through an associated *animation panel*. Once the scenario is setup, the system then visually represents the scenario by constructing a single *animation view*.

The design of the prototype centres on the concept of using a unified *algorithm animation desktop* (Figure 5.10), which allows the user to control the generation and display of animations. The desktop acts as a centralised placement area for animation panels, each of which presents a particular scenario. The desktop provides an integrated platform for comparative analyses, allowing any number of potential combinations of data lists and algorithms to be created and examined. These include comparisons of a data list using different algorithms, or different data lists using a single algorithm (Figure 4.3). When multiple scenarios are to be compared, an animation panel is created and setup for each scenario. The animation panel, once created, is placed within the unified desktop. Scenarios can then be individually selected for parallel display.

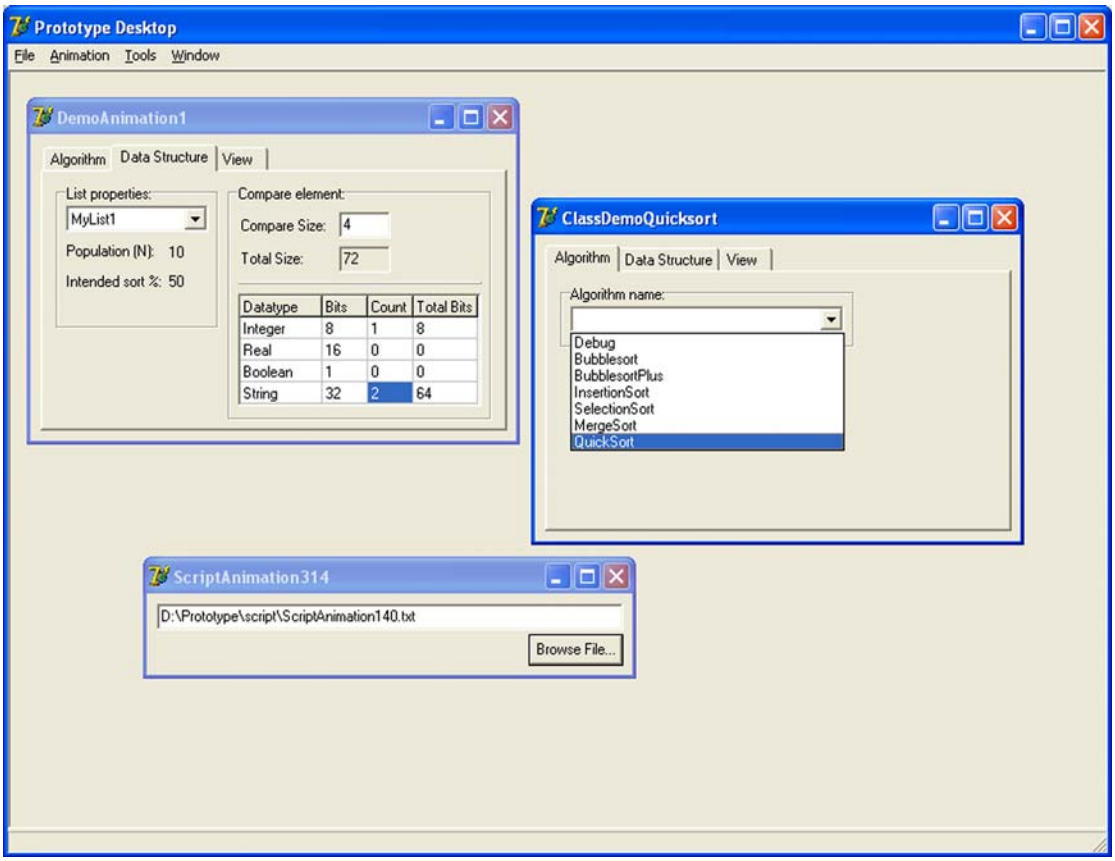


Figure 5.10: Unified algorithm animation desktop

When algorithm animations are created, the processes consist of setting up the scenario through the *data layer interface* (Section 5.4.1), and then selecting and controlling animation views from the *animation layer interface* (Section 5.4.2).

5.4.1 Data Layer Interface

The data list to be used by the algorithm is first generated, and an animation panel is then created to construct a scenario. This process is repeated for each animation the user would like to see in parallel. This section discusses the Graphical User Interface (GUI) designed to support the process.

Data Generator

The interface acts as a front-end for the data generator (Figure 5.11), where multiple data lists are created and maintained. Parameters are entered for the population size and the level of sortedness for each data list created.

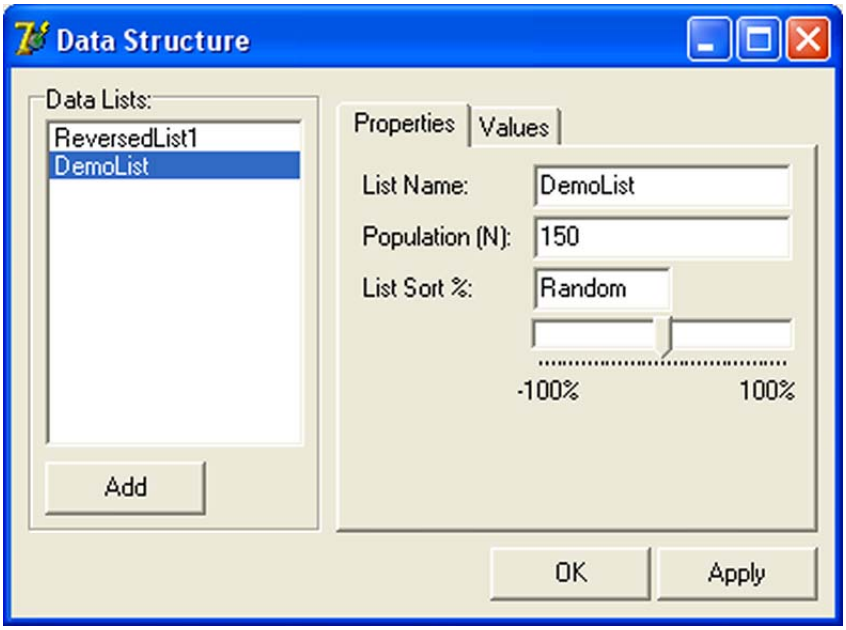


Figure 5.11: Data generator interface

The level of sortedness definition is based on the discussion in Section 4.3.3, where a measure of sortedness was adopted to express list order as a percentage level ranging from -100% to 100%. The concept is supported by using a slider control, allowing the end-user to adjust a pre-defined sorted listed in one percent increments. When the list is adjusted to 0%, a list of undefined ordering is generated based on the method discussed in Section 4.3.1.

Animation Panel

There are two variations of the animation panel, namely the *scenario-based* panel and the *script-based* panel. The scenario panel, as the name suggests, allows the user to create a customised scenario for examination. Within the scenario-based animation panel, selections are made on the combination of algorithm, data structure and interpreter (visual representation) of the scenario (Figure 5.12).

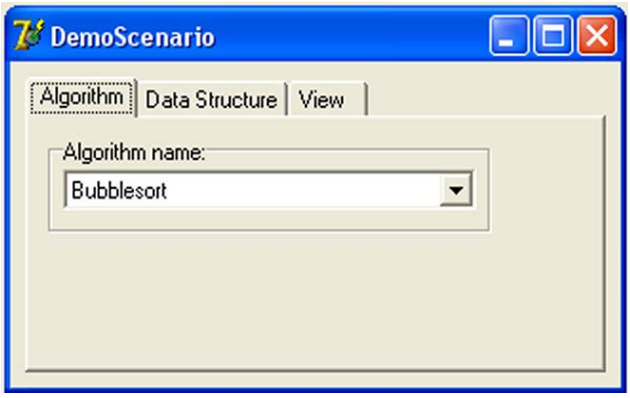


Figure 5.12: Scenario-based Animation Panel

Figure 5.13 shows the data structure interface of the animation panel, which is used to select the data list to use, and to modify the virtual element properties which affect the operational time-cost of the algorithm animation actions (Section 4.4.3). The *compare size* field is entered manually, whilst the *total size* is determined by the amount (*count*) of each virtual datatype used, multiplied by the *bits* size of the datatype.

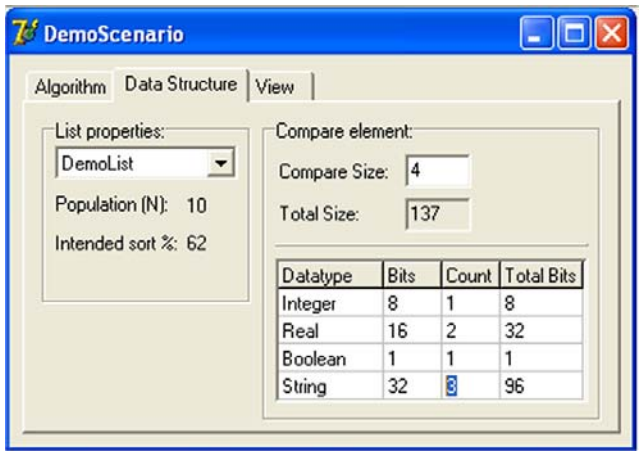


Figure 5.13: Interface for modifying the virtual element properties

The script panel allows the user to directly input a pre-generated animation script file for display. The panel requires a single parameter, which is the location of the animation script file (Figure 5.14).

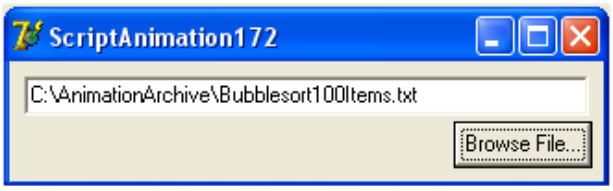


Figure 5.14: Script-based Animation Panel

The approach is designed to offer a flexible method for demonstrating algorithm animations. When the system is employed as a general purpose animation tool, or if students are asked to create basic animation demonstration, an animation script may be written manually without using a driver algorithm. The script can be written in any ASCII-based word processor, following the grammar presented in Section 4.8.1. Another use of the script panel is for demonstrating previously generated algorithm animations. Sometimes lecturers might want to demonstrate particular scenarios in a lecture or self-study laboratory environment. Under such circumstances, the animation script may be saved in an ASCII file in advance, and reloaded through the script panel

for the demonstration. Furthermore, algorithm animations which work on large data sets are time-consuming to generate. Thus, re-using existing scripts through the scrip panel presents an efficient alternative to recreating the animation through the scenario-based panel.

5.4.2 Animation Layer Interface

Once algorithm animation scenarios are setup with all the required properties, an animation selector is used to pick the animation to view. Multiple animations may be picked from the selector. Each animation is viewed through an animation view, which is controlled through the Play Control. The GUI used to control and view the animations is discussed in this section.

Animation Selector

The animation selector shows a list of all available animation panels on the desktop (Figure 5.15). Any number of animations can be selected for display, provided there is sufficient screen space. Although any combination of animations can be selected for parallel display, it is up to the user (and not the selector interface) to decide which animations have common relevance for side-by-side comparisons. For example, while it is possible to race a Bubblesort working on a large list with a Quicksort work on a small list, there is little meaningful knowledge to be gained from such a study.

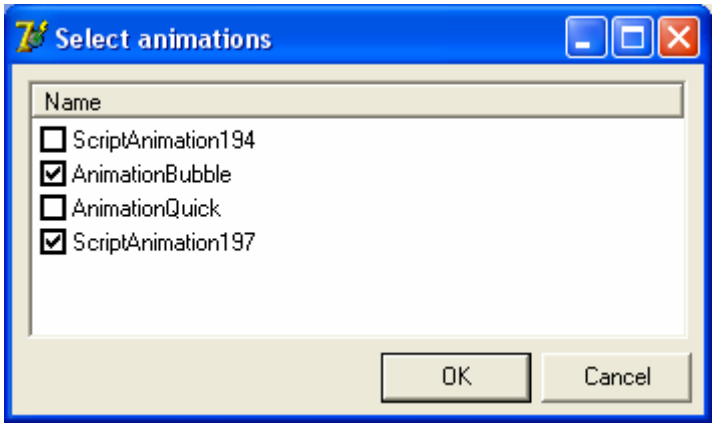


Figure 5.15: Animation Selector

Play Control

The unified play control (Figure 5.16) is used to directly manage the display of animation views, and act as the interface for the unified timer (Section 5.3.3). The control offers the ability to *play*, *pause* and *step through* the animation. A slide bar is used to control the speed of animations. The adjustment can vary from 1/5 through to 10 times the normal speed.

Figure 5.16: Play Control interface

The play control is presented as a floating toolbar which is only made visible to the end-user when there are initialised animation views. Figure 5.18 demonstrates the context in which the play control is presented.

Animation View

An animation view (Figure 5.17) is assigned to each animation panel which is selected for display. The view contains a canvas, which holds the actual visualisation.

The canvas is linked directly to the animation engine renderer. An information panel is provided on the right of the canvas. The information panel is used to display textual information concerning the animation being displayed. The information section of the panel displays messages generated from message events (Section 4.6.1). The operation section is designed to display an up-to-date count of the operations performed by the algorithm (further examined in Section 5.6).

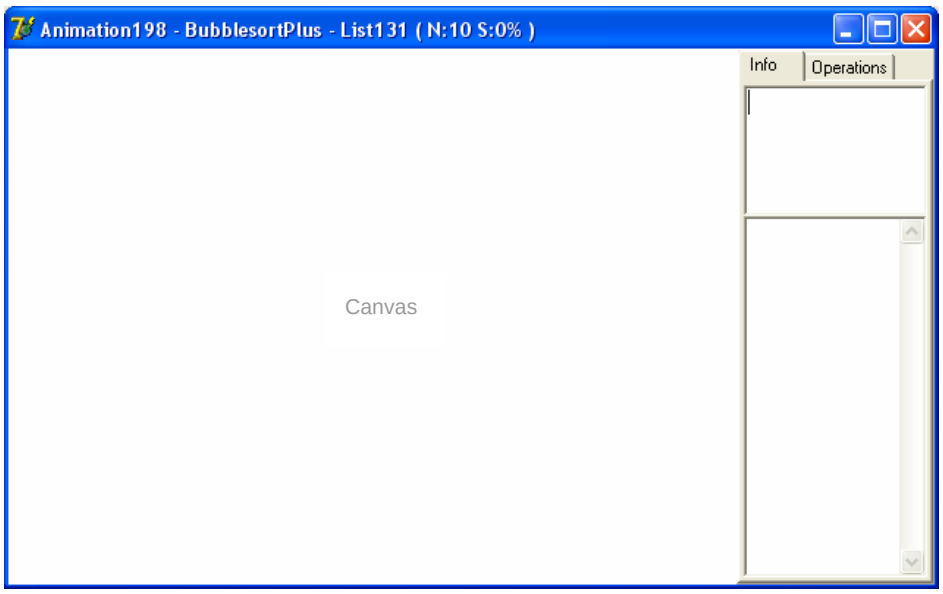


Figure 5.17: Animation view

A key feature of the unified play control and animation view is the support for running multiple algorithm animations in parallel, in effect simulating an algorithm race (Figure 5.18). A single play control handles the display settings of all the animation views, including display speed, start and pause. The *step through* functionality is disabled when multiple animations are shown.

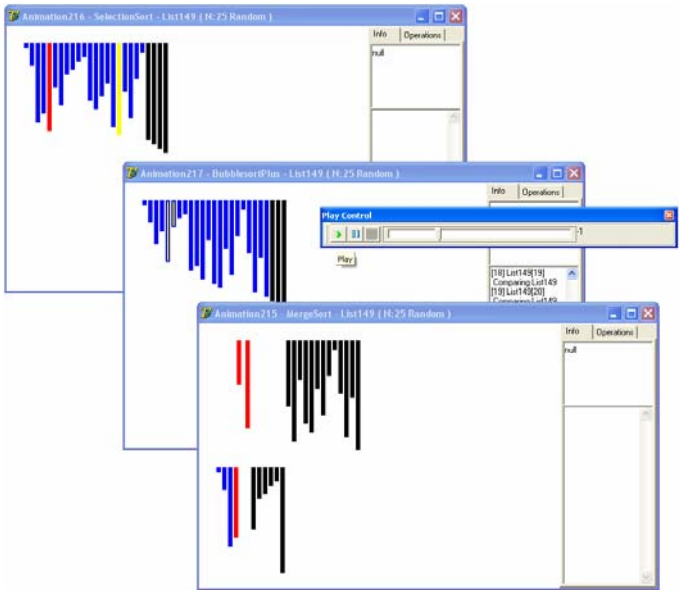


Figure 5.18: The play control managing an animation race

5.5 Implementation of Case Study

The implementation of sorting algorithms as a case study achieves two purposes. Firstly, a practical implementation of algorithm animations would serve to substantiate the concepts of the proposed framework, and to test the usefulness of the prototype system and its various components in constructing animations. Secondly, the availability of the sorting algorithm animations will provide immediate accessibility of the prototype system to algorithmic lecturers and students.

The visualisation design aspect of the case study algorithm animations is based on the two presentation styles initially created by the "Sorting Out Sorting" video (Section 3.5.1). The two styles consist of the rectangular blocks and grouped dots. These styles are referred to in the dissertation as the *rectangle manhattan* style, and the *dot cloud* style, respectively.

The usage of the abovementioned presentation styles are motivated by two reasons. Firstly, since the framework specification defined the use of the integer as the only datatype used by the data structure, this integer data type offers an inherently close mapping to the proposed presentation styles (Section 4.4). Secondly, the two styles offer a good way to demonstrate different aspects of algorithms. The rectangle manhattan (Figure 5.19a) can visualise small to medium size data sets to explain the operations of an algorithm. Visual details can also be added to enhance the animation. The dot cloud style (Figure 5.19b) is suitable for visualising larger data sets, where the step-by-step animated algorithm concept explanations are traded for discreet visual updates. That, in conjunction with the setup of the visualisation, where the data position is presented along the x-axis and the data size along the y-axis, allows the visualisation to present the bigger picture, showing trends and interesting algorithm characteristics.

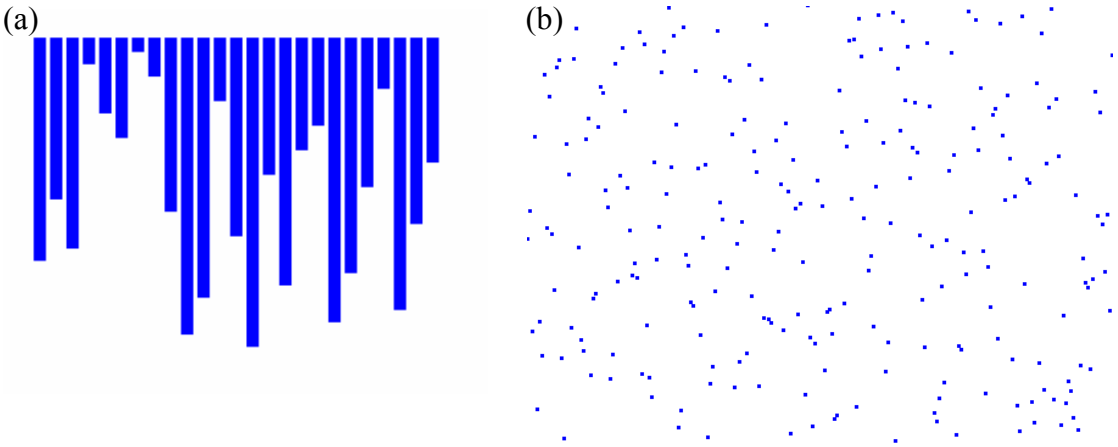


Figure 5.19: (a) Rectangle manhattan (b) Dot cloud

With complex algorithms, the algorithm annotator can capture a variety of conceptual events (Section 4.6.1). If the additional events are appropriately presented through the interpreter, various interesting properties or phenomena of the algorithm are exposed.

Figure 5.20 shows static illustrations of an animation for a Mergesort in action. The concept of a Mergesort is typically easy to describe, but extremely difficult and time-consuming to illustrate using static material.

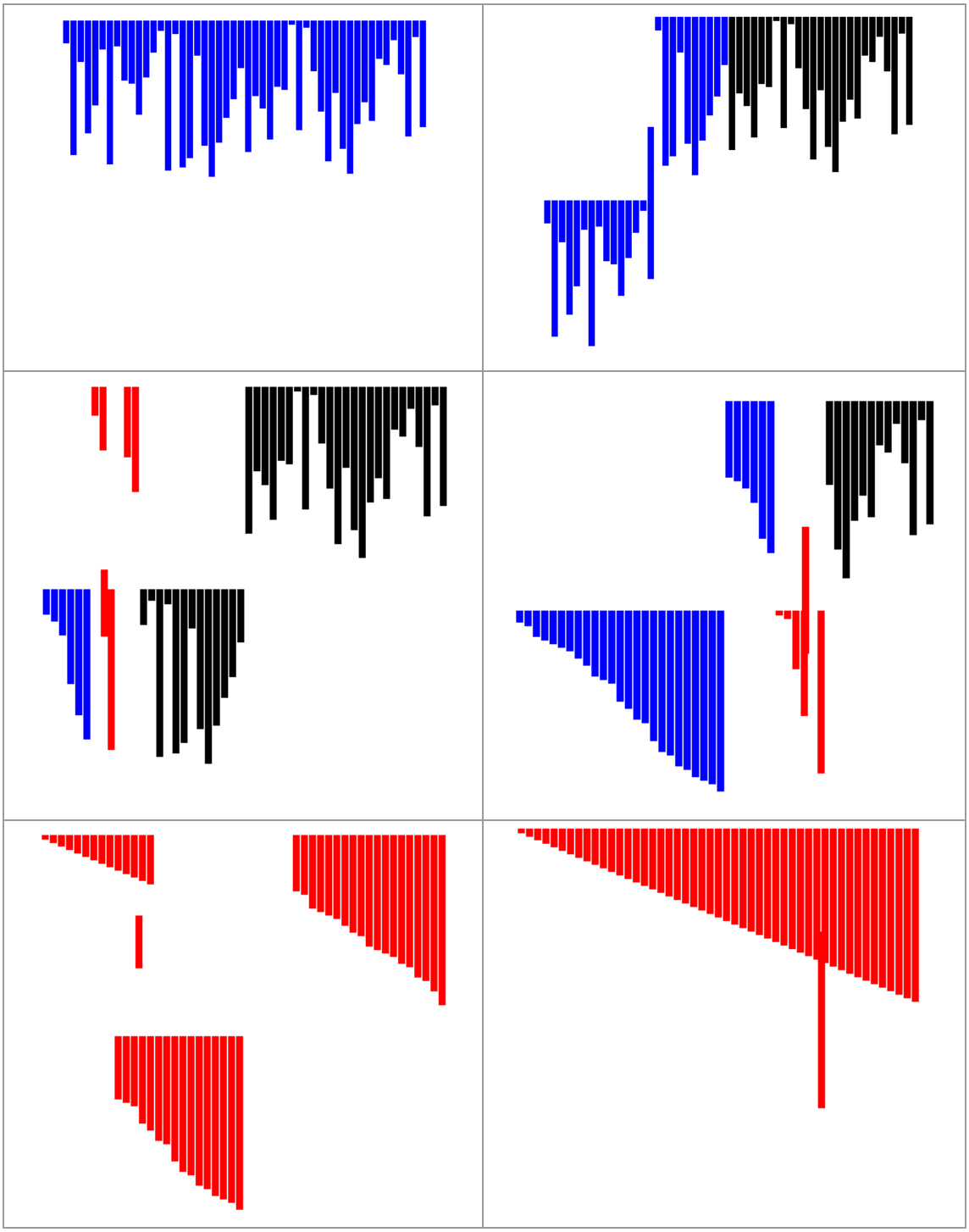


Figure 5.20: The Mergesort animation highlights some interesting details

In Figure 5.20, to highlight the characteristics of the Mergesort, the implemented animation scenario uses conceptual events to mark items as busy being processed (blue), merged (red) or ignored (black). The list is continuously subdivided into smaller lists, with the visualisation showing the exact details on how the Mergesort utilises a temporary list to perform list subdivisions. Once each sub-list is divided into its elementary form (*size* = 1), the sub-lists are systematically merged to form a final sorted list.

Figure 5.21 shows a series of illustrations for a Quicksort animation using a dot cloud style, highlighting the divide-and-conquer nature of the algorithm. The list is first chunked into boxes through iterations, with the sorting pivot (marked in red) forming the bottlenecks among the boxes. The sorting list is shown in blue, and the ignored list is shown in black. Each of the boxes is subdivided at a smaller scale until each box is in its elementary form (where *size* = 2), after which they are sorting back into the bigger picture. A clear advantage of using algorithm animations is that it can illustrate algorithm concepts or phenomena that are either too complex to visualise mentally, impractical to illustrate in a static format, or are not obvious to the viewer when represented in static formats.

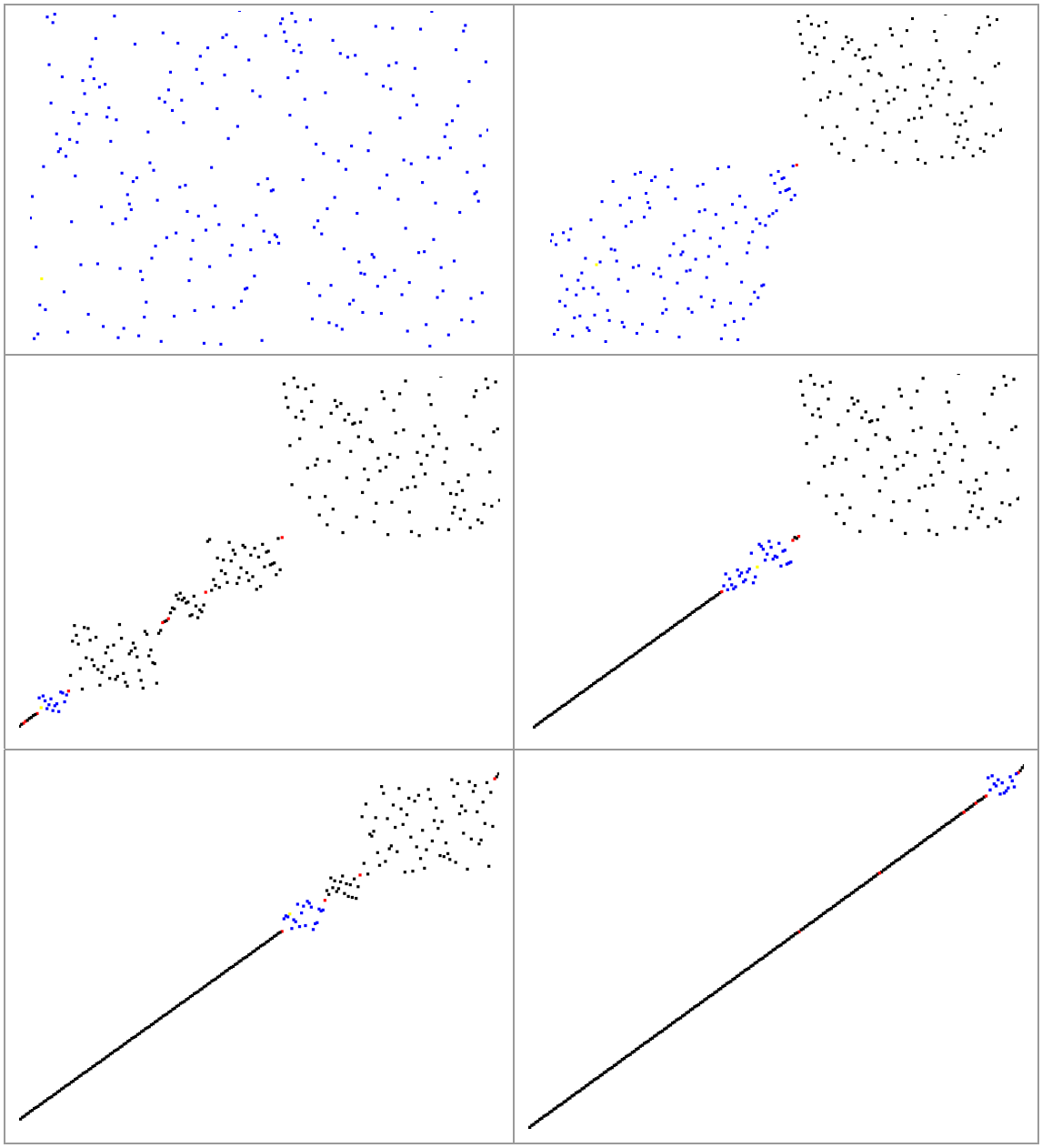


Figure 5.21: The iterative box pattern of the Quicksort

5.6 Implementation Observations

An extensive review of related research delivered few resources documenting the low-level implementations or concepts of interpreters for algorithm animations (Section 5.3.2). Even though the level of difficulty was estimated to be low, it was nevertheless an initial cause for concern. During the prototype implementation, the algorithm and data generator components were first written and tested. With the data

layer components in place, the construction of interpreters started. However, it was initially difficult to see how well the interpreter and the animation component fitted together, especially since the animation component was not implemented, and thus there was no visual output available with which to validate results. To overcome this problem, an extant script animation system, JAWAA (Section 3.5.6), was employed as a temporary animation component. The temporary use of JAWAA was an efficient solution, since the prototype's interpreter was designed to store the animation commands in an ASCII file (Section 5.3.2), which was the primary input method of JAWAA.

Interpreters were initially implemented based on JAWAA's scripting language. The actual implementation was flexible. The methods used to convert generic events into an animation were usually results of iterative prototyping and testing. After the implementation of the interpreters, the prototype's animation component was then constructed. The existing interpreters were then modified to suit the script language of the framework, an easy process due to the similarities of the commands (Section 4.8.1).

The separation of the framework into independent layers (Section 4.2.2), coupled with the use of object instances (Section 5.2.2), resulted in a simpler process of implementation and testing of components. The streaming of events and scripts to an animation view is an independent process of each combination of data, algorithm and view (defined as a scenario to the end-user). The functions of each layer are completed before passing the results through to the next layer for processing (Figure 5.22). Thus, while the prototype seem to run a number of processes, executing layer functions and controlling multiple views, the processes are all done in discreet steps.

As a result, complexities of implementing processes in parallel, such as using application threads, are avoided.

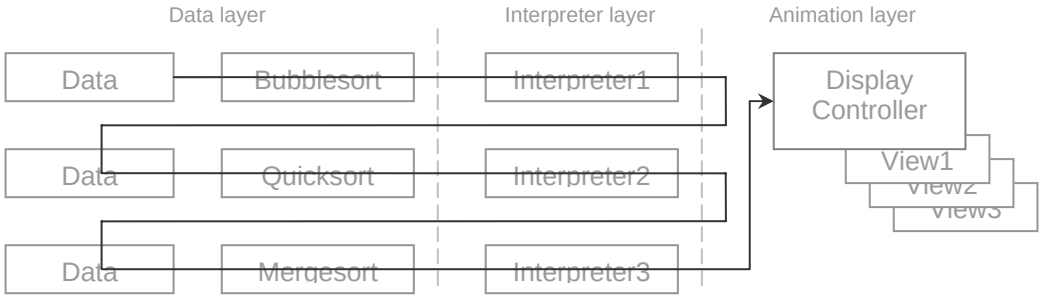


Figure 5.22: Generating animations without using parallel processing

The structuring of the framework, however, did have its inherent limitations. The interpreter took full responsibility for converting event scripts to animation commands, a process which strips the interesting events of its algorithmic origins and properties. Two problems were thus observed.

Firstly, an interesting event, such as a *single* compare operation, is often represented in a rectangle manhattan view by having the examined rectangular blocks being highlighted, and then un-highlighted, which uses *two* graphical operations. This disparity only becomes noticeable when the user tries to run through an animation step-by-step. The user must click the *step* button once to highlight, and again to un-highlight, thus giving the impression that the operation actually consist of two steps, or that the operation is relatively time-consuming (which it is usually not).

A more notable problem is concerned with the information shown to complement the animation (Section 5.4.5). A complementary requirement was to show efficient information relating to each type of operations, such as the number of exchanges and

comparisons performed by an algorithm. This can be used to highlight the relative efficiency of the algorithm. However, once the interpretation of events is done, the graphical scripts no longer contain event-related information. As a result, the animation view is unable to identify or keep count of the algorithm operations performed, and thus cannot display such information.

5.7 Conclusion

Successful implementation of the prototype from the proposed framework resulted. The algorithm animation framework thus proved to be an effective design for a system for generating algorithm animations. The layered structure of the framework, in conjunction with the prototype methodology, supported modularised implementation of the prototype. Furthermore, the prototype based on the framework design was capable of supporting the pedagogic requirements identified in Chapter 3. The user interface of the prototype, while simple, was capable of supporting the interaction requirements of the framework design.

The implementation of the algorithm animation case studies provided an interesting challenge. While the concepts and exact operation of each of the sorting algorithms have already been defined (Appendix A), there is much that can still be learnt of the characteristics of an algorithm from a well planned animation. Even with the very limited set of animation commands offered by the framework, the visualisation designer still has ample opportunity to create informative animations. The structure of the framework was successful in supporting generation and display of multiple algorithm animations due to the modular concept presented in Section 4.2.2.

The limitation of the framework is that, due to the layer design and the use of the interpreter, some algorithm related information may be lost while being processed into animations. This meant that some supplementary information is no longer available for presentation with the algorithm animation.

Chapter 6

Conclusions and Recommendations

6.1 Introduction

The challenges of teaching abstract algorithm concepts to introductory algorithmic students were highlighted together with the potential benefits of employing algorithm animations to increase the students' algorithm comprehension (Chapter 1). The objective of the dissertation was thus the design and implementation of an extensible framework which provides for the generation and display of algorithm animations in an algorithm course environment. The proposed framework and implemented prototype will integrate into the NMMU CS&IS department's goal of increasing student throughput through research and utilisation of technological support tools (Section 1.3.3).

The pedagogic potential of algorithm animations formed the primary motivation for the initial study. Literature reviews showed algorithm animation as an established field of study with well-defined theoretical and practical background (Chapter 2). In support of the dissertation's goals, studies were performed to gain an understanding of algorithm animation, including current and past research, extant systems, and

available pedagogic applications. The results of the study guided the design and implementation of the algorithm animation framework and its associated prototype.

This chapter provides a brief summary of the research achievements (Section 6.2), contributions (Section 6.3), implications (Section 6.4) and limitations (Section 6.5). These are followed by recommendations for future research (Section 6.6).

6.2 Research Achievements

Chapter 1 presented a context into the utilisation of algorithm animation to support the teaching of algorithm courses. The goals of the research were the design of an algorithm animation framework, and the evaluation of the effectiveness of the framework through a prototype implementation. The achievement of these goals is evident in two areas, namely theoretical and practical. A number of research questions were posed to guide the investigation towards achieving these goals (Table 6.1). Based on these findings, questions 1 through 6 and 8 contribute to the theoretical achievement, while the others contribute to the practical achievement.

	Research questions	**Relevant chapter/section(s)**
1.	What is software visualisation?	Section 2.2
2.	What is algorithm animation?	Section 2.3
3.	What elements are used to form an algorithm animation?	Section 2.3
4.	How are algorithm animations used in teaching and learning algorithms?	Section 2.4
5.	What are the issues to be considered in the design and specification of an algorithm animation framework?	Sections 3.2 and 3.3
6.	What are the criteria for the design of effective algorithm animation systems?	Section 3.4
7.	How do extant algorithm animation systems match the criteria?	Section 3.5
8.	What does an algorithm animation framework look like?	Chapter 4
9.	What are the implementation issues faced by developers of algorithm animation systems?	Sections 5.2, 5.3 and 5.6
10.	How effective is the proposed framework?	Chapter 5
11.	How does the algorithm animation prototype developed match the identified measurement criteria?	Chapter 5
12.	What are the limitation and contribution of the framework and prototype?	Chapter 6

Table 6.1: Research questions of the dissertation

This section provides an overview of the theoretical (Section 6.2.1) and practical (Section 6.2.2) achievements resulting from answering each of the research questions. Specific and relevant sections of the dissertation are summarised to show how the questions were addressed.

6.2.1 Theoretical Achievements

Software visualisation involves the use of graphical techniques to improve the presentation and appearance of programs, with the aim of facilitating understanding of the programs (Section 2.2). Algorithm animation is classified as a form of software

visualisation which visualises the working of algorithms through a high level of abstraction (Section 2.3). In other words, the content of algorithm animations is strategically chosen to focus on issues of relevance to a particular topic, whilst unimportant or less relevant concepts are hidden or shown only superficially (Section 2.5).

Algorithm animation involves the utilisation of primarily visual elements to represent the data structures of an algorithm, and to display the dynamic operations of the algorithm in execution. The three visual elements of algorithm animations were identified as visual metaphors, animation and colour (Section 2.3). Algorithm animations may be used in lecture demonstrations, or be made accessible to students in a laboratory environment or over the internet (Section 2.4). A review of software visualisation and algorithm animation as a field of research, the communication techniques employed by algorithm animations, and the educational value of algorithm animations formed the focus of the foundation to the study (Chapter 2).

Having established an understanding of the concepts and uses of algorithm animation, a study is conducted into the various elements of an algorithm animation system (Chapter 3). A number of issues were considered to support the designing of the framework. Identifying the types of users within an algorithm animation system environment and the general components of algorithm animation systems provided an understanding of the functional requirements of a system (Section 3.2). Another issue considered was the paradigms used to connect algorithms to visualisations, which affect both the design and operational characteristics of an algorithm animation system. The two algorithm-to-visualisation paradigms identified were the imperative and declarative paradigm (Section 3.3).

An important research objective was the establishment of a list of requirements which will effectively complement the pedagogic objective of the framework. A number of preliminary requirements were identified and divided into two sections (Section 3.4). The first section categorised requirements into an interaction level taxonomy proposed by Naps *et al* (2003), and the second section consisted of complementary requirements. The preliminary requirements were then reviewed, from which two requirements were removed due to project scope restrictions (Section 1.4.2). A list of nine requirements was proposed as the criteria for effective algorithm animation systems (Section 3.6).

With the theoretical foundations and requirements established, the specification and design of an algorithm animation framework were proposed (Chapter 4). The framework is designed to support the list of requirements previously established, whilst also utilising the algorithm animation user, component and visualisation paradigm concepts. The structure of the framework was divided into independent layers based on the functionalities of component groups.

6.2.2 Practical Achievements

The list of requirements (Section 3.4) was used as an instrument for evaluating seven extant algorithm animation systems (Section 3.5). The evaluation showed that no extant system was able to address all of the requirements identified.

The implementation of a prototype based on the proposed framework acted as an evaluation of the effectiveness of the framework design. The evaluation (Chapter 5), focused on determining the effectiveness of the framework through the

implementation of an algorithm animation system, and producing a number of sorting algorithm animations using the implemented system.

The prototype implementation was performed using the iterative prototyping methodology (Section 5.2.1). Problems were encountered in implementation where certain informative data cannot be presented along with the algorithm animation (Section 5.6). Two algorithm animation design styles were successfully applied to the case study sorting algorithms using the prototype, namely the rectangle manhattan and dot cloud style (Section 5.5). The results of the prototype and successful algorithm animation case study implementation are indicative of the proposed framework to be an effective design. Furthermore, the prototype based on the framework was capable of supporting all the requirements identified in Chapter 3. The objectives of the framework design have thus been achieved successfully.

6.3 Research Contributions

This section outlines the research contributions, which represent the outputs and deliverables of the current investigation. Section 6.3.1 discusses the theoretical contributions, and Section 6.3.2 discusses the practical contributions.

6.3.1 Theoretical Contributions

Many studies have established a variety of compiled requirements intended for or based on algorithm animation systems (Gurka and Citrin 1996; Hansen, Narayanan and Hegarty 2002; Saraiya 2002; Saraiya, Shaffer, McCrickard and North 2004). The specification of a list of requirements was thus needed to support the design of the

proposed algorithm animation framework. However, literature studies, supported by the extant system analysis, showed that there is currently no unified and commonly accepted requirements framework for evaluating the effectiveness of algorithm animation systems in an algorithm course environment. The derived list of requirements (Table 6.2) is a theoretical contribution towards identifying instructionally effective features of algorithm animation systems.

Requirements for Algorithm Animations	
R1	Allow speed control of algorithm animation
R2	Allow rewinding of the animation
R3	Accept user input data for the algorithm
R4	Provide questions to predict algorithm behaviour
R5	Allow stepping control of algorithm animation
R6	Support construction of animation by students
R7	Support for smooth motion
R8	Include capabilities for comparative algorithm analysis
R9	Provide multiple views of an algorithm
R10	Provide additional instructional material
R11	General purpose framework

Table 6.2: Identified Requirements

The algorithm animation framework, illustrated in Figure 6.1, forms the primary theoretical contribution of the dissertation. The framework made use of knowledge gained from literature reviews and system analysis. Various design concepts were also proposed (Chapter 4). The framework consists of five layers, with the core layers placed within two user interface layers (Section 4.2.2). The framework core consists of the data layer, interpreter layer, and animation layer. The data layer produces interesting events by executing a driver algorithm with a generated data structure. The interpreter layer converts the interesting events into a predefined animation command script, which is used by the animation layer to render the algorithm animation. The

design allows layer outputs to be combined to produce different algorithm scenarios for animation.

User interaction with the data and interpreter layers are provided through the data layer interface, and the animation layer through the animation layer interface (Section 4.9). The layering design maps the user types with defined components or component groups, thereby clarifying the functions of user types within the framework. The algorithm programmer interacts with the data layer, the visualisation designer with the interpreter layer, the visualisation tool developer with the animation layer, and the students and instructors with the data and animation layer interfaces.

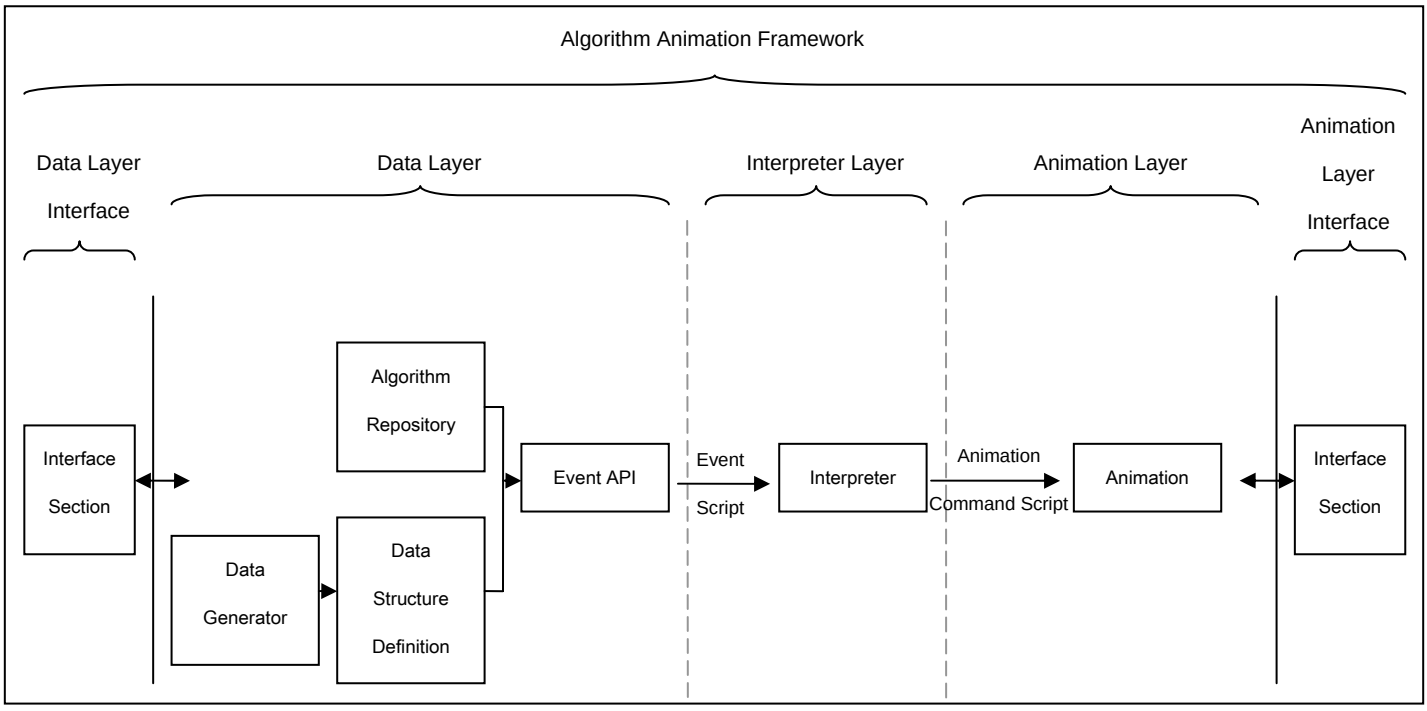


Figure 6.1: Framework structure

6.3.2 Practical Contributions

The list of requirements (Table 6.2) is used as an instrument for evaluating seven extant algorithm animation systems (Section 3.5). The evaluation, summarised in Table 6.3, forms a practical contribution of the current investigation.

	Sorting Out Sorting	BALSA-II	GAIGS	JCAT	JSAMBA	JAWAA	ANIMAL+ JHAVE
R1: Allow speed control of algorithm animation		✓		✓	✓	✓	✓
R2: Allow rewinding of the animation			✓				✓
R3: Accept user input data for the algorithm		(✓)	✓				✓
R4: Provide questions to predict algorithm behaviour							✓
R5: Allow stepping control of algorithm animation		✓	✓	✓	✓		✓
R6: Support construction of animation by students					✓	✓	✓
R7: Support for smooth motion	(✓)			✓	✓	✓	✓
R8: Include capabilities for comparative algorithm analysis	(✓)	✓					
R9: Provide multiple views of an algorithm	(✓)	✓	✓	✓			
R10: Provide additional instructional material	(✓)		(✓)	✓	(✓)	(✓)	✓
R11: General purpose framework					✓	✓	✓

Table 6.3: Criteria met by extant systems
✓ Support for feature (✓) Partial support for feature

The effectiveness of the framework was evaluated through the implementation of a prototype system (Figure 6.2). The algorithm animation system prototype formed the primary practical contribution of the research. An additional contribution is the animation of the sorting algorithm case studies, which illustrated the concepts of the algorithms using visual metaphor, animation and colour techniques.

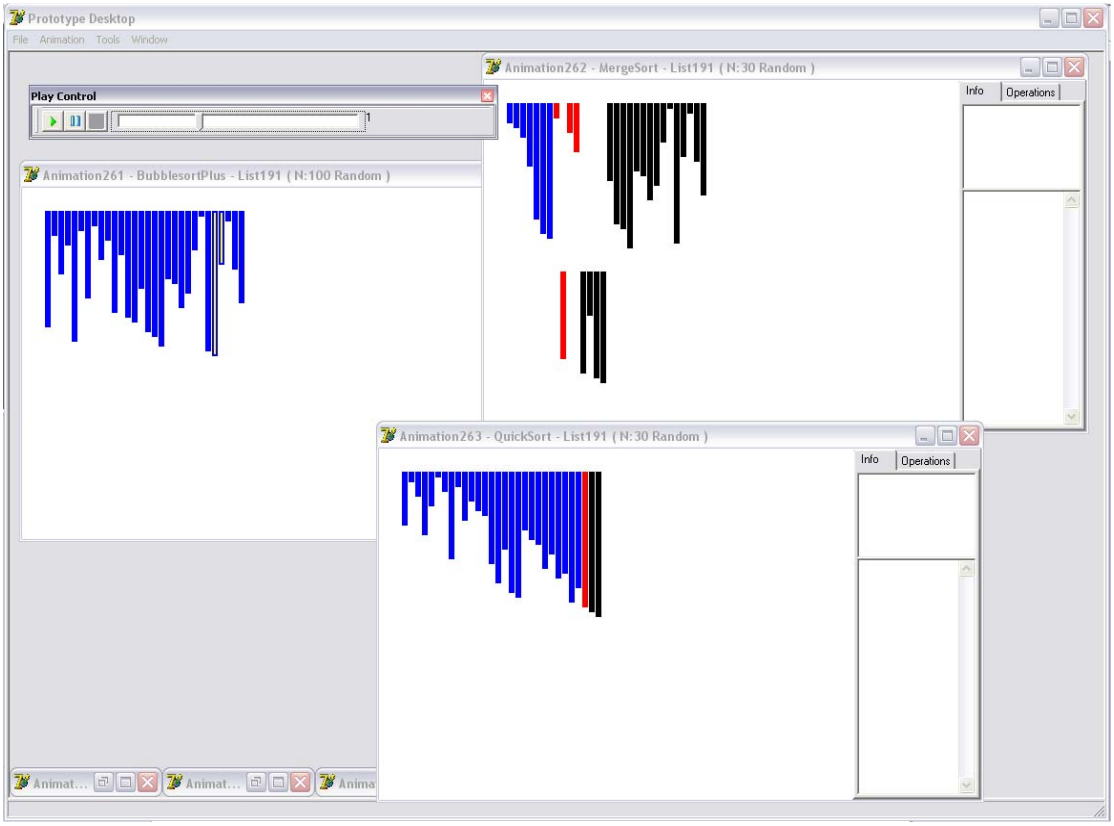


Figure 6.2: Prototype screenshot

6.4 Implications of Research

This section will discuss how the contributions of the research can be applied to future research and algorithm learning environments.

There is currently an absence of a unified and common instrument for evaluating the pedagogic effectiveness of algorithm animation systems. This highlights the value of the proposed list of requirements. The list of requirements can be used as a foundation for the creation of a unified algorithm animation system evaluation instrument. Tertiary educational institutions can also utilise the proposed requirements as a preliminary method for evaluating the suitability of algorithm animation systems in particular course environments, with the aim of integrating the systems to complement the institution's existing teaching strategies.

A gap currently exists within the community for a standard and widely accepted guideline for the design of algorithm animation systems. The proposed framework is presented as an effective model for addressing this gap. An independent layer approach was used to design the framework. As a result, iterative modification or improvement of various framework components can be done without affecting other layer components. The extensible design of the framework allows it to be further expanded to integrate concepts in support of current or future studies. The framework can thus be seen as a flexible model for supporting further research within the community.

An evaluation was performed on seven extant algorithm animation systems based on the proposed list of requirements. The results of the evaluation provided a general understanding of the characteristics of each extant system, and may aid instructors in making informed decisions on the choice of algorithm animation systems to utilise in algorithm courses. The evaluation results can be used to support discussions of future system evaluations. Evaluations of additional systems will complement the knowledge on the extant systems, rather than act as stand-alone evaluations.

The implemented prototype is a working system which may be integrated into an algorithm course environment. The prototype may be utilised in an interactive lecturing environment to demonstrate algorithms, with support for live generation and display of algorithm animations. This will allow instructors to provide immediate visualised feedback to the students' queries. Pre-generated animations can also be saved and replayed in lectures to highlight specific scenarios of interest. The prototype development focused on the animation of sorting algorithms in particular.

The immediate outcome of the implementation is two-fold. Firstly, the theoretical design of the framework can be tested in practice to evaluate its ability to match the proposed requirements. Secondly, the prototype and the sorting animations can be rapidly integrated as a pilot into existing algorithm curricula, complementing the technological support related research within the NMMU. The prototype is well suited for integration into laboratory environments. Students can utilise the interactive features of the prototype to complement their studies of algorithms. Furthermore, practical assignments can be created to encourage the use of algorithm animations by students.

6.5 Limitations of Research

Chapter 3 produced preliminary requirements for an algorithm animation framework, upon which the final list of requirements was based on. The two requirements which were not included were the support for rewinding of animations, and provision of questions to predict algorithm behaviour.

The support for animation rewinding was not considered due to scope limitations of the current research, discussed in Section 3.6.2. The provision of questions as a form of interactive learning may increase the comprehension of material. However, the scope of the project did not include features for interactive questions (Section 1.4.2).

A limitation of the framework found during the implementation was discussed in Section 5.6. The limitation was due to the structure of the framework, which separated the concepts of the algorithm from that of the animation by utilising an intermediate interpreter process. As a result, certain informative data relating to the original algorithm is no longer accessible to the animation display. It is also worth mentioning that the final design of the framework would not have been suitable for integrating interactive questions with the algorithm animations. Placing interactive questions in relevant sections of the algorithm animation would require some understanding of the original algorithm's operational context, which as mentioned, is lost on the conversion to the animation phase due to the independence of layers.

6.6 Recommendations for Future Research

The conclusion of the current investigation offers a number of possibilities for future research projects. These can be separated into projects that focus on theoretical contributions, and projects which extend the practical contributions of the system.

Projects which provide theoretical contributions to the field of algorithm animations include the following:

- Empirical studies can be conducted to evaluate the pedagogic effectiveness of the prototype. The studies can focus on the effectiveness of the prototype as a whole, or concentrate on a particular group of features.
- A critical analysis can be conducted to determine the relevance of the list of requirements for evaluating system effectiveness. This may result in specific features being added or removed. The critical analysis may be supported by the abovementioned empirical studies.
- Implementation of additional case studies can be done to evaluate the feasibility of the framework. The case studies can include different sorting algorithms, or algorithms of other domains.
- An investigation can be performed into incorporating declarative paradigm concepts into the framework, and how these concepts will affect the characteristics of the framework.

A comprehensive implementation of the algorithm animation system based on the proposed framework can offer a number of practical contributions:

- The system will provide a relatively reliable platform for integration into NMMU curricula, and to conduct empirical studies.
- The implementation can incorporate the feature for rewinding animations.
- During implementation, the design of the interface can incorporate established usability design methodologies. This may lead to further research using usability evaluations and eye-tracking technologies.
- Administrative features may be included to support enabling and disabling of system features, which will aid empirical studies into the effectiveness of particular features and requirements. Administrative features may also include

the monitoring of student usage on particular system features, or the scenarios which are examined by students.

The abovementioned research recommendations, which extend from the current research, can be used to derive ways to further improve the pedagogic potential of algorithm animation.

6.7 Summary

Algorithm animation is a technological support tool which supports the comprehension of abstract algorithm data and concepts. The creation of an algorithm animation framework to support the research strategy for increasing student throughput within the NMMU CS&IS formed the basis of the current research.

The current research has successfully made substantial theoretical and practical contributions towards the research direction of the NMMU. These research contributions are:

- List of requirements for algorithm animation systems.
- A comparative study of extant algorithm animation systems using the list of requirements as an evaluation instrument.
- An algorithm animation framework model to support the implementation of algorithm animation systems.
- A prototype system based on the proposed framework.
- Sorting algorithm animations implemented in the prototype system.

The research has satisfied the proposed goals and objectives by addressing each of the identified research questions. The dissertation successfully reported on the demonstration of the proposed algorithm animation framework as an effective design model. Future research will thus focus on determining the pedagogic effectiveness of the prototype developed in an algorithm course environment, and the effectiveness of the framework in supporting animations of algorithms in other programming domains.

REFERENCES

AKINGBADE, A., FINLEY, T., JACKSON, D., PATEL, P. and RODGER, S.H. (2003): JAWAA: Easy Web-Based Animation from CS 0 to Advanced CS Courses. *Proc. Technical Symposium on Computer Science Education*, Reno, Navada, USA. **35**:162 - 166, ACM Press

ALESSI, S.M. and TROLLIP, S.R. (2001): *Multimedia for Learning: Methods and Development*. 3rd Edn, Allyn and Bacon.

ANDERSON, J.M. and NAPS, T.L. (2000): A Context For The Assessment Of Algorithm Visualization Systems As Pedagogical Tools. *Proc. First Program Visualization Workshop*, Helsinki, Finland.

ARIK, S. (2005): Algorithm Animation as a Narrative. CS-2005-03. Efi Arazi School of Computer Science, The interdisciplinary center.

BAECKER, R. (1998): Sorting out Sorting: A case study of software visualization for teaching computer science. In *Software Visualization: Programming as a Multimedia Experience*. 369 - 381. STASKO, J., DOMINGUE, J., BROWN, M.H. and PRICE, B.A. (eds). The MIT Press.

BAECKER, R. and PRICE, B. (1998): The Early History of Software Visualization. In *Software Visualization: Programming as a Multimedia Experience*. 29 - 34. STASKO, J., DOMINGUE, J., BROWN, M.H. and PRICE, B.A. (eds). The MIT Press.

BAECKER, R.M. (1981): Sorting out Sorting - 30 minute colour sound film, Dynamic Graphics Project, University of Toronto.

BAILEY, D.A. (1999): *Java Structures: Data structures in Java for the principled programmer*. McGraw Hill.

BAKER, J.E., CRUZ, I.F., LIOTTA, G. and TAMASSIA, R. (1996): Algorithm Animation Over the World Wide Web. *Proc. AVI '96*, Gubbio, Italy. ACM Press

BALDWIN, D. and SCRAGG, G. (2004): Instructor Notes - Tactics for Teaching Algorithms and Data Structures: The Science of Computing. In *Algorithms and Data Structures: The Science of Computing*. Charles River Media.

BALL, T. and EICK, S.G. (1996): Software Visualization in the Large. *IEEE Computer* **29**(4):33-43

BALOIAN, N. and LUTHER, W. (2001): Visualization for the Mind's Eye. *Proc. International Dagstuhl Seminar of Software Visualization*, Schloss Dagstuhl, Germany. 354 - 367, DIEHL, S., EADES, P. and STASKO, J. (eds).

BARBU, A., DROMOWICZ, M., GAO, X., KOESTER, M. and WOLF, C. (2001): Bubblesort Animation - Softwareergonomische Aspekte bei der Gestaltung von WWW-basierter Lernsoftware am Beispiel vorlesungsbegleitender Materialien zu "Sortieren". http://olli.informatik.uni-oldenburg.de/fpsort/

BARTRAM, L. (1997): Perceptual and interpretative properties of motion for information visualization. *Proc. Workshop on New paradigms in information visualization and manipulation*, Las Vegas, Nevada, United States. 3 - 7,

BARTRAM, L.R. (2001): Enhancing Information Visualization with Motion. Ph.D thesis. School of Computing, Simon Fraser University.

BAZIK, J., TAMASSIA, R., REISS, S.P. and VAN DAM, A. (1998): Software Visualization in Teaching at Brown University. In *Software Visualization: Programming as a Multimedia Experience*. 383 - 398. STASKO, J., DOMINGUE, J., BROWN, M.H. and PRICE, B.A. (eds). The MIT Press.

BENTLEY, J.L. and KERNIGHAN, B.W. (1991): A System for Algorithm Animation - Tutorial and User Manual. Computing Science Technical Report 132. AT&T Bell Laboratories. Murray Hill, New Jersey 07974.

BRODAL, G.S., FAGERBERG, R. and MORUZ, G. (2005): On the Adaptiveness of Quicksort. *Proc. 7th Workshop on Algorithm Engineering and Experiments*.

BROWN, D. (2001): B#: A visual programming tool. Honours thesis. Computer Science & Information Systems, University of Port Elizabeth.

BROWN, M.H. (1988a): Algorithm Animation. PhD thesis. Brown University.

BROWN, M.H. (1988b): Exploring algorithms using BALSA-II. *IEEE Computer* **21**(5):14-36.May 1988.

BROWN, M.H. (1988c): Perspectives on algorithm animation. *Proc. SIGCHI conference on Human factors in computing systems*. 33 - 38, ACM Press

BROWN, M.H. (1998): A Taxonomy of algorithm animation displays. In *Software Visualization: Programming as a Multimedia Experience*. 35. STASKO, J., DOMINGUE, J., BROWN, M.H. and PRICE, B.A. (eds). The MIT Press.

BROWN, M.H. and HERSHBERGER, J. (1991): Color and Sound in Algorithm Animation. DEC Systems Research Center.

BROWN, M.H. and HERSHBERGER, J. (1998a): Fundamental Techniques for Algorithm Animation Displays. In *Software Visualization: Programming as a Multimedia Experience*. 81 - 101. STASKO, J., DOMINGUE, J., BROWN, M.H. and PRICE, B.A. (eds). The MIT Press.

BROWN, M.H. and HERSHBERGER, J. (1998b): Program Auralization. In *Software Visualization: Programming as a Multimedia Experience*. 138 - 143. STASKO, J., DOMINGUE, J., BROWN, M.H. and PRICE, B.A. (eds). The MIT Press.

BROWN, M.H. and RAISAMO, R. (1997): JCAT: Collaborative active textbooks using Java. *Computer Networks and ISDN Systems*

BROWN, M.H. and SEDGEWICK, R. (1984): A System for Algorithm Animation. *SIGGRAPH - Computer Graphics* **18**(3)

BROWN, M.H. and SEDGEWICK, R. (1998): Interesting Events. In *Software Visualization: Programming as a Multimedia Experience*. 154-171. STASKO, J., DOMINGUE, J., BROWN, M.H. and PRICE, B.A. (eds). The MIT Press.

BYRNE, M.D., CATRAMBONE, R. and STASKO, J.T. (1996): Do Algorithm Animation Aid Learning? Technical GIT-GVU-96-18. Georgia Institute of Technology.

BYRNE, M.D., CATRAMBONE, R. and STASKO, J.T. (1999): Evaluating animations as student aids in learning computer algorithms. *Computers & Education* **33**(4):253 - 278

CALITZ, A.P. (1997): The Development of an Evaluation of a Strategy for the Selection of Computer Science Students at the University of Port Elizabeth. Ph.D thesis. Computer Science & Information Systems, University of Port Elizabeth.

CANTÙ, M. (2001): From Automation to COM+: Interfaces, Variants, and Dispatch Interfaces: Testing the Speed Difference. In *Mastering Delphi 6*. 863-865. Sybex.

CARRANO, F.M. and PRICHARD, J.J. (2002): *Data Abstraction and Problem Solving with C++*. 3 Edn, Addison Wesley.

CHEN, J. and CARLSSON, S. (1991): On partitions and presortedness of sequences. *Proc. second annual ACM-SIAM symposium on Discrete algorithms*, San Francisco, California, United States. 62 - 71, Society for Industrial and Applied Mathematics

CHIKOFSKY, E.J. and RUBENSTEIN, B.L. (1988): CASE: Reliability Engineering for Information Systems. *Proc. 14th international conference on Software engineering*. 11 - 16, IEEE Computer Society Press

CHRISTIANS, D. (2003): A Simple IDE for Delphi. Honours thesis. Computer Science & Information Systems, University of Port Elizabeth.

CILLIERS, C. (2005): A Comparison of Programming Notations for a Tertiary Level Introductory Programming Course. Ph.D thesis. Computer Science & Information Systems, University of Port Elizabeth.

COLOMBO, B.A., DEMETRESCU, C., FINOCCHI, I. and LAURA, L. (2003): A System for Building Animated Presentations over the Web. *Proc. AICCSA'03*

Workshop on Practice and Experience with Java Programming in Education, Tunis.

COSTELLOE, E. (2004): Teaching Programming The State of the Art. *CRITE Technical Report*

CRESCENZI, P., DEMETRESCU, C., FINOCCHI, I. and PETRESCHI, R. (2000): Reversible execution and visualization of programs with LEONARDO. *Journal of Visual Languages and Computing* **11**(2):125 - 150

DANN, W., COOPER, S. and PAUSCH, R. (2001): Using visualization to teach novices recursion. *Proc. Annual Joint Conference Integrating Technology into Computer Science Education*, Canterbury, United Kingdom. 109 - 112, ACM Press

DE JAGER, S. (2004): SimplifIDE. Honours thesis. Computer Science & Information Systems, University of Port Elizabeth.

DEMETRESCU, C., FINOCCHI, I. and STASKO, J. (2001): Specifying Algorithm Visualizations: Interesting Events or State Mapping? *Proc. International Dagstuhl Seminar of Software Visualization*, Schloss Dagstuhl, Germany. 16-30, DIEHL, S., EADES, P. and STASKO, J. (eds).

DERSHEM, H.L. and BRUMMUND, P. (1998): Tools for Web-based sorting animation. *Proc. Twenty-ninth SIGCSE technical symposium on Computer science education*, Atlanta, Georgia, United States. 222 - 226,

DIEHL, S., GÖRG, C. and KERREN, A. (2002): Animating Algorithms Live and Post Mortem. *Software Visualization, LNCS State-of-the-Art Survey* **2269**:46-57

DÖLLNER, J., HINRICHS, K. and SPIEGEL, H. (1997): An interactive environment for visualizing and animating algorithms. *Proc. Proceedings of the thirteenth annual symposium on Computational geometry*, Nice, France. 409 - 411, ACM Press

DUSKIS, S. (undated): JSAMBA -- Java version of the SAMBA Animation Program -. Last updated:

EICK, S.G. (1998): Maintenance of Large Systems. In *Software Visualization: Programming as a Multimedia Experience*. 35. STASKO, J., DOMINGUE, J., BROWN, M.H. and PRICE, B.A. (eds). The MIT Press.

FALTIN, N. (2001): Structure and Constraints in Interactive Exploratory Algorithm Learning, Springer-Verlag. **2269**:213 - 226.

FLEISCHER, R. and KUČERA, L. (2002): Algorithm animation for teaching. *Software Visualization, State-of-the-Art Survey*:113 - 128

GAMIELDIEN, R. (2003): Activity Logger. Honours thesis. Computer Science & Information Systems, University of Port Elizabeth.

GARNER, S. (2003): Learning Resources and Tools to Aid Novices Learn Programming. *Proc. Informing Science 2003*, Pori, Finland.

GIANNOTTI, E.I. (1987): Algorithm Animator: A Tool for Programming Learning. *Proc. SIGCSE technical symposium on Computer science education*, St. Louis, Missouri, United States. 308 - 314, ACM Press

GLOOR, P.A. (1998): User Interface Issues For Algorithm Animation. In *Software Visualization: Programming as a Multimedia Experience*. 145 - 152. STASKO, J., DOMINGUE, J., BROWN, M.H. and PRICE, B.A. (eds). The MIT Press.

GOLDMAN, N. and NARAYANASWAMY, K. (1992): Software evolution through iterative prototyping. *Proc. 14th international conference on Software engineering*. ACM Press

GREYLING, J. (2000): The compilation and validation of a computerised selection test battery for Computer Science and Information System students. Ph.D thesis. Computer Science & Information Systems, University of Port Elizabeth.

GRISSOM, S., MCNALLY, M.F. and NAPS, T. (2003): Algorithm visualization in CS education: comparing levels of student engagement. *Proc. 2003 ACM symposium on Software visualization*, San Diego, California. Software Visualization, 87 - 94, ACM Press

GURKA, J.S. and CITRIN, W. (1996): Testing Effectiveness of Algorithm Animation. *Proc. 1996 IEEE Symposium on Visual Languages*. 182, IEEE Computer Society

HAAJANEN, J., PESONIUS, M., SUTINEN, E., TARHIO, J., TERASVIRTA, T. and VANNINEN, P. (1997): Animation of user algorithms on the Web. *Proc. IEEE Symposium on Visual Languages (VL '97)*. 360-367, IEEE Computer Society

HAMILTON-TAYLOR, A.G. and KRAEMER, E. (2002): SKA: Supporting Algorithm and Data Structure Discussion. *ACM SIGCSE Bulletin* **34**(1).March 2002.

HANSEN, S.R., NARAYANAN, N.H. and HEGARTY, M. (2002): Designing Educationally Effective Algorithm Visualizations. *Journal of Visual Languages and Computing* **13**(2):291-317

HANSEN, S.R., NARAYANAN, N.H. and SCHRIMPSHER, D. (2000): Helping learners visualize and comprehend algorithms. *Interactive Multimedia Electronic Journal of Computer-Enhanced Learning* **2**(1).May 2000.

HENNING, J. (2004): Visual Code Reorganization Tool. Honours thesis. Computer Science and Information Systems, University of Port Elizabeth.

HÜBSCHER-YOUNGER, T. and NARAYANAN, N.H. (2002): Influence of Authority on Convergence in Collaborative Learning. *Proc. Computer Support for Collaborative Learning Conference*, Boulder, Colorado USA.

HÜBSCHER-YOUNGER, T. and NARAYANAN, N.H. (2003): Dancing Hamsters and Marble Statues: Characterizing Student Visualizations of Algorithms. *ACM Symposium on Software Visualization*

HUNDHAUSEN, C. (1993): The search for an empirical and theoretical foundation for algorithm visualization - Unpublished technical report. Department of Computer & Information Science, University of Oregon, Eugene.

HUNDHAUSEN, C.D. (1997): A Meta-Study of Software Visualization Effectiveness - Unpublished comprehensive exam paper, Department of Computer and Information Science, University of Oregon, Eugene.

HUNDHAUSEN, C.D. (2002): Integrating algorithm visualization technology into an undergraduate algorithms course: ethnographic studies of a social constructivist approach. *Computers & Education* **39**(3):237 - 260.November 2002.

HUNDHAUSEN, C.D., DOUGLAS, S.A. and STASKO, J.T. (2002): A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages and Computing* **13**(3):259 - 290

HWANG, H.K., YANG, B.Y. and YEH, Y.N. (2000): Presorting algorithms: an average-case point of view. *Theoretical Computer Science* **242**(1-2):29 - 40

IEEE and ACM (2001): Computing Curricula 2001 - Computer Science. The Joint Task Force on Computing Curricula, IEEE Computer Society, Association for Computing Machinery. http://www.sigcse.org/cc2001/

JARC, D.J., FELDMAN, M.B. and HELLER, R.S. (2000): Accessing the Benefits of Interactive Prediction Using. Web-based Algorithm Animation Courseware. *Proc. 31st SIGCSE technical symposium on Computer science education*, Austin, Texas. **32(1)**:377-381,

JEFFERY, C.L. (1998): A Menagerie of Program Visualization Techniques. In *Software Visualization: Programming as a Multimedia Experience*. 73 - 79. STASKO, J., DOMINGUE, J., BROWN, M.H. and PRICE, B.A. (eds). The MIT Press.

KANN, C., LINDEMAN, R.W. and HELLER, R. (1997): Integrating algorithm animation into a learning environment. *Computers & Education* **28**(4):223 - 228.May 1997.

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

KENDALL, K.E. and KENDALL, J.E. (1996): Prototyping. In *System Analysis and Design*. 199 - 233. Prentice Hall.

KNUTH, D.E. (1973): *The Art of Computer Programming*. Reading, Mass., Sorting and Searching. Vol. 3. Addison-Wesley.

LATTU, M., MEISALO, V. and TARHIO, J. (2003): A visualisation tool as a demonstration aid. *Computers & Education* **41**(2):133 - 148.September 2003.

LAWRENCE, A.W., BADRE, A.N. and STASKO, J.T. (1994): Empirically Evaluating the Use of Animations to Teach Algorithms. *Proc. IEEE Symposium on Visual Languages*, St. Louis, MO. 48-54,

LAXER, C. (2001): Treating computer science as science: An experiment with sorting. *ACM SIGCSE Bulletin* **33**(3):189.September 2001.

LIEBERMAN, H. and FRY, C. (1998): ZStep95: A Reversible, Animated Source Code Stepper. In *Software Visualization: Programming as a Multimedia Experience*. 277 - 292. STASKO, J., DOMINGUE, J., BROWN, M.H. and PRICE, B.A. (eds). The MIT Press.

LISTER, R. and LEANEY, J. (2003): Introductory Programming, Criterion-Referencing, and Bloom. *Proc. 34th SIGCSE Technical Symposium on Computer Science Education*. 143 - 147,

MAMTANI, B. (2004): Web-based Presentation System for Programming Logic Information Flow. Honours thesis. Computer Science and Information Systems, University of Port Elizabeth.

MARCUS, A., FENG, L. and MALETIC, J.I. (2003): 3D representations for software visualization. *Proc. ACM symposium on Software visualization*, San Diego, California. 27 - ff, ACM Press

MCCAULEY, R. (1998): Warning! Instructional Animation Tools Abound on the Web. *SIGCSE Bulletin* **30**(4)

MUKHERJEA, S. and STASKO, J.T. (1993): Applying algorithm animation techniques for program tracing, debugging, and understanding. *Proc. 15th International Conference on Software Engineering*, Baltimore, Maryland, United States. 456 - 465,

MYERS, B.A. (1990): Taxonomies of Visual Programming and Program Visualization. *Journal of Visual Languages and Computing* **1**(1):97 - 123

NAJORK, M. (2001): Web-based Algorithm Animation. *Proc. DAC*, Las Vegas, Nevada, USA. ACM Press.June 18-22, 2001.

NAPS, T.L. and BRESSLER, E. (1998): A multi-windowed environment for simultaneous visualization of related algorithms on the World Wide Web. *Proc. SIGCSE technical symposium on Computer science education*, New York, NY, USA. 277 - 281, ACM Press

NAPS, T.L., EAGAN, J.R. and NORTON, L.L. (2000): JHAVE - An environment to Actively Engage Students in Web-based Algorithm Visualizations. *Proc. Technical Symposium on Computer Science Education*, Austin, TX, USA. 109 - 113, ACM Press

NAPS, T.L., FLEISHER, R., MCNALLY, M., RODGER, S., VELAZQUEZ-ITURBIDE, J.A., RÖßLING , G., ALMTSTRUM, V., DANN, W., HUNDHAUSEN, C., KORHONEN, A. and MALMI, L. (2003): Exploring the role of visualization and engagement in computer science education. *Annual Joint Conference Integrating Technology into Computer Science Education*:131 - 152

NAPS, T.L., RÖßLING , G., ANDERSON, J., COOPER, S., DANN, W., FLEISHER, R., KOLDEHOFE, B., KORHONEN, A., KUITTINEN, M., LESKA, C., MALMI, L., MCNALLY, M., RANTAKOKKO, J., and ROSS, R.J. (2003): Evaluating the Educational Impact of Visualization. *Proc. ITiCSE 2003*, Thessaloniki, Greece. ACM Press

NAPS, T.L. and SWANDER, B. (1994): An Object-Oriented Approach to Algorithm Visualization - Easy, Extensible, and Dynamic. *Proc. ACM SIGCSE Technical Symposium*. 46 - 50,

NASSI, I. and SHNEIDERMAN, B. (1973): Flowchart techniques for structured programming. *ACM SIGPLAN Notices* **8**(8):12 - 26

PALLIER, C. (2002): Shuffle: a program to randomize lists with optional sequential constraints. http://www.pallier.org/papers/ Last updated: 02-Sept-2005).

PETRE, M., BLACKWELL, A. and GREEN, T. (1998): Cognitive Questions in Software Visualization. In *Software Visualization: Programming as a Multimedia Experience*. 453 - 480. STASKO, J., DOMINGUE, J., BROWN, M.H. and PRICE, B.A. (eds). The MIT Press.

PIERSON, W.C. and RODGER, S.H. (1998): Web-based animation of data structures using JAWAA. *Proc. twenty-ninth SIGCSE technical symposium on Computer science education*, Georgia, United States. 267 - 271, ACM Press

PRICE, B., BAECKER, R. and SMALL, I. (1993): A principled taxonomy of software visualization. *Journal of Visual Languages and Computing* **4**(3):211 - 266

PRICE, B., BAECKER, R. and SMALL, I. (1998): An Introduction to Software Visualization. In *Software Visualization: Programming as a Multimedia Experience*. 4 - 5. STASKO, J., DOMINGUE, J., BROWN, M.H. and PRICE, B.A. (eds). The MIT Press.

RAMSHAW, L. (1997): Java-Based Collaborative Active Textbooks - Compaq Computer Corporation. http://research.compaq.com/SRC/JCAT/ Last updated:

RASALA, R., PROULX, V.K. and FELL, H.J. (1994): From animation to analysis in introductory computer science. *Proc. twenty-fifth SIGCSE symposium on Computer science education*, Phoenix, Arizona, United States. 61 - 65, ACM Press

RODGER, S. (2002): JAWAA 2.0 The JAWAA Homepage - Java and Web based Algorithm Animation. http://www.cs.duke.edu/csed/jawaa2/

ROMAN, G.-C. (1998): Declarative Visualization. In *Software Visualization: Programming as a Multimedia Experience*. 173-186. STASKO, J., DOMINGUE, J., BROWN, M.H. and PRICE, B.A. (eds). The MIT Press.

ROMAN, G.-C. and COX, K.C. (1992): Program visualization: the art of mapping programs to pictures. *Proc. 14th international conference on Software engineering*, Melbourne, Australia. 412 - 420, ACM Press

ROMAN, G.-C. and COX, K.C. (1993): A Taxonomy of Program Visualization Systems. *IEEE Computer* **26**(12):11 - 24

RÖßLING, G. (2000): ANIMALSCRIPT - The Reference.

RÖßLING, G. (2002): Key Decisions in Adopting Algorithm Animation for Teaching. *Proc. IFIP TC3/WG3.1 & 3.2 Open Conference on Informatics and the digital society: Social, Ethical and Cognitive Issues on Informatics and ICT*. 149 - 156, ACM Press

RÖßLING, G. and FREISLEBEN, B. (2000a): Approaches for Generating Animations In Lectures. *Proc. AACE 11th International Society for Information Technology and Teacher Education (SITE 2000) Conference*, San Diego, California. 809-814, Association for the Advancement of Computers in Education (AACE)

RÖßLING, G. and FREISLEBEN, B. (2000b): Experiences in using animations in introductory computer science lectures. *Proc. Technical Symposium on Computer Science Education*, Austin, Texas, United States. 134 - 138, ACM Press

RÖßLING, G. and FREISLEBEN, B. (2001): AnimalScript: an extensible scripting language for algorithm animation. *Proc. SIGCSE technical symposium on Computer Science Education*, Charlotte, North Carolina, United States. 70 - 74, ACM Press

RÖßLING, G. and FREISLEBEN, B. (2002): ANIMAL: A system for supporting multiple roles in algorithm animation. *Journal of Visual Languages and Computing* **13**:341--354

RÖßLING, G. and NAPS, T.L. (2002): A Testbed for Pedagogical Requirements in Algorithm Visualizations. *Proc. Annual Joint Conference Integrating Technology into Computer Science Education*, Aarhus, Denmark. ACM Press

RÖßLING, G., SCHÜLER, M. and FREISLEBEN, B. (2000): The ANIMAL algorithm animation tool. *Proc. Annual Joint Conference Integrating Technology into Computer Science Education*, Helsinki, Finland. 37 - 40, ACM Press

SARAIYA, P. (2002): Effective Features of Algorithm Visualizations. Masters thesis. Virginia Polytechnic Institute and State University. Blacksburg.

SARAIYA, P., SHAFFER, C.A., MCCRICKARD, D.S. and NORTH, C. (2004): Effective Features of Algorithm Visualizations. *Technical Symposium on Computer Science Education - Proceedings of the 35th SIGCSE technical symposium on Computer science education*:382 - 386.March 2004.

SCHÜLER, M. and RÖßLING, G. (2001): ANIMAL - A NEW INTERACTIVE MODELLER FOR ANIMATIONS IN LECTURES (Ver 2.0).

SILICONGRAPHICS (1999): JAL Algorithm Animation. http://reality.sgi.com/austern/java/demo.html

SONNIER, D.L. and HUTTON, S.L. (2004): Enhancing visual aids through the use of animation. *Proc. Mid-South College Computing Conference*. Mid-South College Computing Conference

STASKO, J. (1998a): Building Software Visualizations through Direct Manipulation and Demonstration. In *Software Visualization: Programming as a Multimedia Experience*. 186-203. STASKO, J., DOMINGUE, J., BROWN, M.H. and PRICE, B.A. (eds). The MIT Press.

STASKO, J. (1998b): Smooth, Continuous Animation for Portraying Algorithms and Processes. In *Software Visualization: Programming as a Multimedia Experience*. 104 - 118. STASKO, J., DOMINGUE, J., BROWN, M.H. and PRICE, B.A. (eds). The MIT Press.

STASKO, J., BADRE, A. and LEWIS, C. (1993): Do algorithm animations assist learning? an empirical study and analysis. *Proc. SIGCHI conference on Human factors in computing systems*, Amsterdam, The Netherlands. 61 - 66, ACM Press

STASKO, J.T. (1997): SAMBA Animation Designer's Package. Georgia Insitute of Technology.

STASKO, J.T. and LAWRENCE, A.W. (1998): Empirically Assessing Algorithm Animations as Learning Aids. In *Software Visualization: Programming as a Multimedia Experience*. STASKO, J., DOMINGUE, J., BROWN, M.H. and PRICE, B.A. (eds). The MIT Press.

STASKO, J.T. and PATTERSON, C. (1993): Understanding and Characterizing Program Visualization Systems. Technical Report GIT-GVU-91-17 Georgia Institute of Technology.

STEPHENS, R. (1998): *"Ready-to-Run Visual Basic Algorithms"*. 2nd Edn, Wiley Computer Publishing.

STERN, L., SØNDERGAARD, H. and NAISH, L. (1999): A strategy for managing content complexity in algorithm animation. *Proc. Annual Joint Conference Integrating Technology into Computer Science Education*, Cracow, Poland. 127 - 130, ACM Press

SUMNER, W.N. and BANU, D. (2003): JSAVE: Simple and Automated Algorithm Visualization Using the Java Collection Framework. Tenth Annual Consortium for Computing Sciences in Colleges Hope College. Holland.

SYRJAKOW, M., BERDUX, J. and SZCZERBICKA, H. (2000): Interactive Web-based animations for teaching and learning. *Proc. 32nd conference on Winter simulation*, Orlando, Florida. 1651 - 1659, Society for Computer Simulation International

THOMAS, J. (2002): B# Version 2: A visual programming tool. Honours thesis. Computer Science & Information Systems, University of Port Elizabeth. Port Elizabeth.

TUDOREANU, M.-E. (2002): Economy of Interaction in Program Visualization: Designing Effective Visualization Tools for Reducing User's Cognitive Effort. Ph.D thesis. Department of Computer Science, Washington University.

TUDOREANU, M.E. (2003): Designing effective program visualization tools for reducing user's cognitive effort. *Proc. 2003 ACM symposium on Software visualization*, San Diego, California. 105 - ff, ACM Press

VAN TONDER, M. (2003): A Java Development Tool. Honours thesis. Computer Science & Information Systems, University of Port Elizabeth. Port Elizabeth.

VICKERS, P. and ALTY, J. (2003): Siren Songs and Swan Songs - Debugging with music. *Communications of the ACM* **46**(7)

WIGGINS, M. (1998): An overview of program visualization tools and systems. *Proc. ACM Southeast Regional Conference - 36th annual Southeast regional conference*. 194 - 200, ACM Press

WILHELM, R., MÜLDNER, T. and SEIDEL, R. (2001): Algorithm Explanation: Visualizing Abstract States and Invariants. *Proc. International Dagstuhl Seminar of Software Visualization*, Schloss Dagstuhl, Germany. 381 - 394, DIEHL, S., EADES, P. and STASKO, J. (eds).

WILSON, J., AIKEN, R. and KATZ, I. (1996): Review of animation systems for algorithm understanding. *Proc. 1st conference on Integrating technology into computer science education*. 75 - 77, ACM Press

WILSON, J., KATZ, I.R., INGARGIOLA, G., AIKEN, R. and HOSKIN, N. (1995): Students' use of animations for algorithm understanding. *Proc. Human Factors in Computing Systems*. 238 - 239, ACM Press

YEH, C.L. (2003): Implementing interactive tracing and debugging tools: B# (Ver3). Honours thesis. Computer Science and Information Systems, University of Port Elizabeth.

APPENDIX A - Sorting Algorithms for the Case Study

Sorting algorithms are one type of algorithm examined in the introductory and intermediate algorithm curricula. There are two classes of common sorting algorithms taught, namely the O(N^2) quadratic sorting algorithms - bubble sort, selection sort, insertion sort and shellsort - and the O(N *log* N) sorting algorithms - quicksort, mergesort and heapsort (IEEE and ACM 2001). The sorting algorithms taught in the NMMU Computer Science introductory and intermediate curricula are Bubblesort, Insertion Sort, Selection Sort, Mergesort and Quicksort. This section explains and illustrates each of the sorting algorithms taught at the NMMU.

The discussion on these algorithms highlights the unique operational characteristics of each algorithm, and the issues dealing with complexity, comprehension, implementation and relative efficiency. These sorting algorithms form the problem domain for the evaluation of the framework proposed. Examples of the sorting algorithms, implemented in Delphi, are included as part of the discussions. The sorting algorithms illustrated in this section arrange items in ascending order[14].

The complexity of an algorithm has a direct correlation with its relative efficiency. Algorithm complexity is represented using the Big-O notation, with *O* representing the complexity of the algorithm and a value *N* representing the population size of the dataset (Bailey 1999).

[14]This dissertation will treat a *sorted list* or *fully ordered list* as a list sorted in ascending order unless mentioned otherwise.

The Bubblesort is the simplest algorithm to comprehend, and it is typically used to introduce students to sorting algorithms. The selection sort and insertion sort are more efficient than the Bubblesort, but more complex to understand. The Mergesort and the Quicksort are relatively efficient sorting algorithms which are generally the most difficult to understand due to the concept of recursion employed. The code diagrams utilises the data structure presented in Figure A.1.

```
const
  MaxElements = 10;

type
  TIntList = record
    List : array[1..MaxElements] of Integer;
    Count : Integer;
  end;
```

Figure A.1.: Data Structure Definition

A.1 Bubblesort

The Bubblesort is generally the first sorting algorithm presented to introductory algorithm students due to its simple concept and ease of implementation. The Bubblesort functions by comparing each item in the list with the next item, and exchanging the items if they are out of order. The algorithm will continue to iterate through the list until all items are in the correct order (Figure A.2 and A.3).

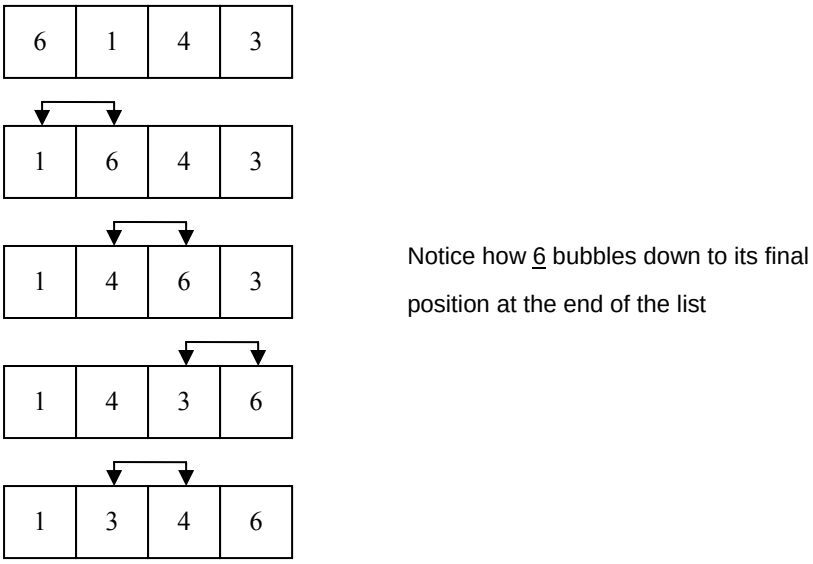


Figure A.2: Concept of the Bubblesort

Larger values "bubble" to the end of the list and smaller items towards the beginning of the list, hence the name of the algorithm. With the number of comparisons (*N-1*) and exchanges (potentially *N-1*) performed, the Bubblesort is regarded as the most inefficient sorting algorithm in common use, with a complexity of $O(N^2)$.

```
procedure BubbleSort(var L : TIntList);
var
  f : Integer;
  Sorted : Boolean;
  Temp, SortCount : Integer;
begin
  SortCount := 0;
  repeat
    Sorted := True;

    for f := 1 to L.Count – 1 - SortCount do
    begin
      if L.List[f] > L.List[f+1] then
      begin
        temp := L.List[f];
        L.List[f] := L.List[f+1];
        L.List[f+1] := temp;
        Sorted := False;
      end;
    end;
    SortCount := SortCount + 1;
  until Sorted;

end;
```

Figure A.3: Implementation of Bubblesort

A.2 Selection Sort

The selection sort logically divides the list into two parts, the sorted part and the unsorted part, at any point in time. The selection sort works on the concept of selecting the largest item from the available remaining unsorted items. The largest item found is then exchanged with the item in the next position to be filled, where it becomes a part of the sorted list. The process repeats until there is only one item to select (Figure A.4 and A.5).

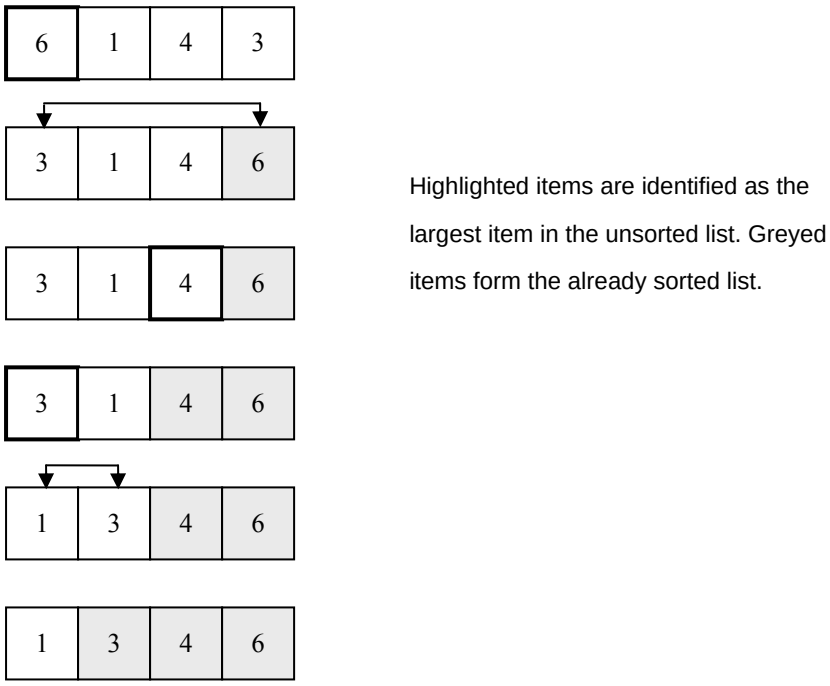


Figure A.4: Concept of the Selection sort

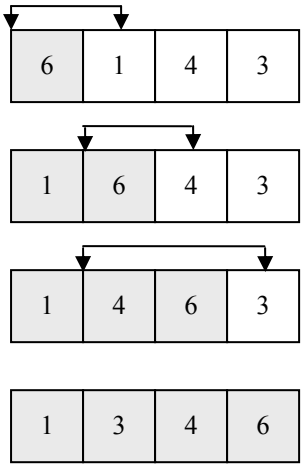
The selection sort performs significantly better than the Bubblesort since only one exchange is performed per pass through the list. The complexity of the selection sort is $O(N^2)$. The selection sort will produce the same performance regardless of the ordering of the initial list.

```
procedure Swap(var L :TIntList; pos1, pos2 : Integer);
var
  temp : Integer;
begin
  temp := L.List[pos1];
  L.List[pos1] := L.List[pos2];
  L.List[pos2] := temp;
end;


procedure SelectionSort(var L : TIntList);
var
  index : Integer;
  max : Integer;
  numUnsorted : Integer;
begin
  numUnsorted := L.Count;

  while numUnsorted > 1 do
  begin
    max := 1;

    for index := 2 to numUnsorted do
    begin
      if L.List[max] < L.List[index] then
        max := index;
    end;

    Swap(L, max, numUnsorted);
    dec(numUnsorted);

  end;

end;
```

Figure A.5: Implementation of Selection Sort

A.3 Insertion Sort

The insertion sort logically breaks up a list into two parts, the already sorted part (place in the beginning of the list), and the unsorted part. The first item from the unsorted list is taken and inserted in order into the sorted list, increasing the sorted list size by one and decreasing the unsorted list size by one. This iteration takes place until the unsorted list size is zero (Figure A.6).



Greyed items form the already sorted list. The arrow shows where each unsorted item is inserted into the sorted list.

Figure A.6: Concept of the Insertion Sort

The insertion sort is a $O(N^2)$ complexity sort. The time cost of the algorithm is dominated by the operation of inserting items into the sorted list. Due to this characteristic, insertion sorts are ideal for sorting a nearly ordered list.

```
procedure InsertionSort(var L : TIntList);
var
  numSorted : Integer;
  index : Integer;
  temp : Integer;
begin
  numSorted := 2;

  while numSorted <= L.Count do
  begin
    temp := L.List[numSorted];
    for index := numSorted downto 2 do
    begin
      if temp < L.List[index-1] then
        L.List[index] := L.List[index-1]
      else
        break;
    end;

    L.List[index] := temp;
    inc(numSorted);
  end;

end;
```

Figure A.7: Implementation of Insertion Sort

A.4 Mergesort

The Mergesort takes the list to be sorted, splits the list into two equal halves, sorts each halve, and then merges the sorted halves into one list again. The Mergesort divides and sorts the list recursively, and then merges the list together to form the final sorted list (Figure A.8, A.9 and A.10).

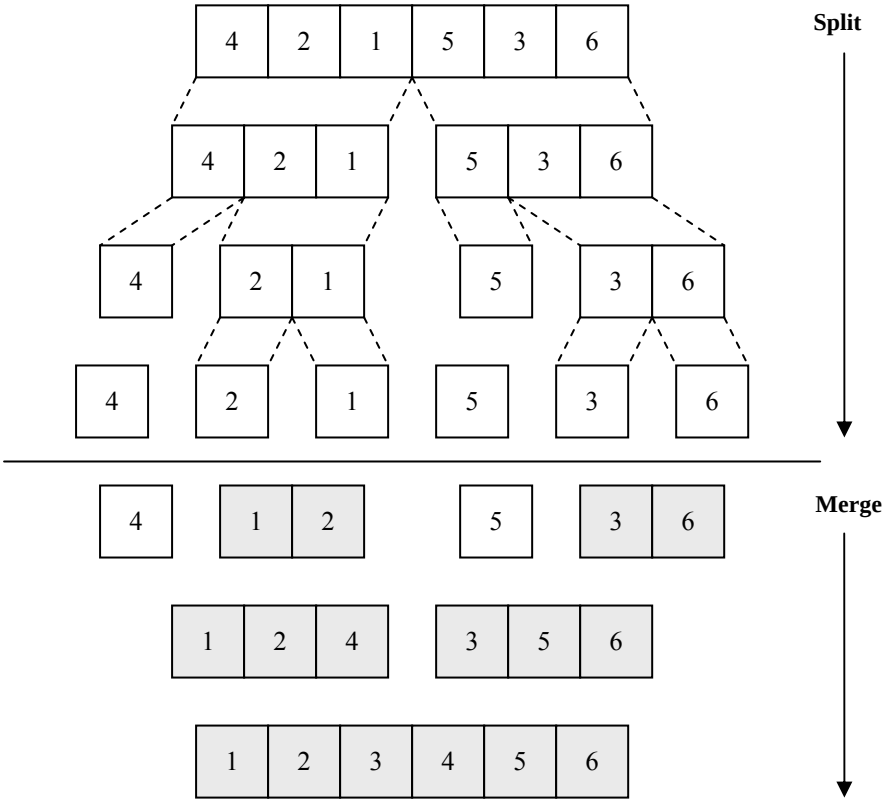


Figure A.8: Concept of the Mergesort

The Mergesort's complexity is $O(N \log N)$. A disadvantage of the Mergesort is its need to make use of a temporary list during the sorting process, thus doubling the memory used. Due to the concept of merging and recursion used in the algorithm, it is a difficult concept to explain and form a mental model of.

```
procedure Merge(var Data :TIntList;var Temp : TIntList; Low, Middle, High :
Integer);
var
  ri, ti, di : Integer;
begin
  ri := low;
  ti := low;
  di := middle;

  while (ti < middle) and (di <= high) do
  begin
    if Data.List[di] < Temp.List[ti] then
    begin
      Data.List[ri] := Data.List[di];
      inc(ri);
      inc(di);
    end
    else
    begin
      Data.List[ri] := Temp.List[ti];
      inc(ri);
      inc(ti);
    end;
  end;

  while (ti < middle) do
  begin
    Data.List[ri] := Temp.List[ti];
    inc(ri);
    inc(ti);
  end;

  Data.Count := High - Low + 1;
  ShowList(Data);
end;
```

Figure A.9: Implementation of MergeSort supporting merge routine

```
procedure MergeSortRecursive(var Data :TIntList;var Temp : TIntList; Low, High :
Integer);
var
  n, middle, i : Integer;
begin
  n := High - Low + 1;
  middle := Low + n div 2;

  if n < 2 then
    Exit;

  for i := low to (middle-1) do
  begin
     Temp.List[i] := Data.List[i];
  end;

  MergeSortRecursive(Temp, Data, Low, Middle-1);
  MergeSortRecursive(Data, Temp, Middle, High);

  Merge(Data, Temp, Low, Middle, High);

end;
```

Figure A.10: Implementation of MergeSort main routine

A.5 Quicksort

The Quicksort typically consists of four steps (Figure A.11, A.12 and A.13):

1. If there is one or zero items in the list to be sorted, return immediately.
2. Pick an item in the list to serve as a "pivot" point (Usually the left-most element in the list is used).
3. Split the list into two parts - one with elements larger than the pivot and the other with elements smaller than the pivot. This is done by having two markers move towards each other, swopping out of order items until the markers meet.
4. Recursively repeat the algorithm for both halves of the original list/sublist.

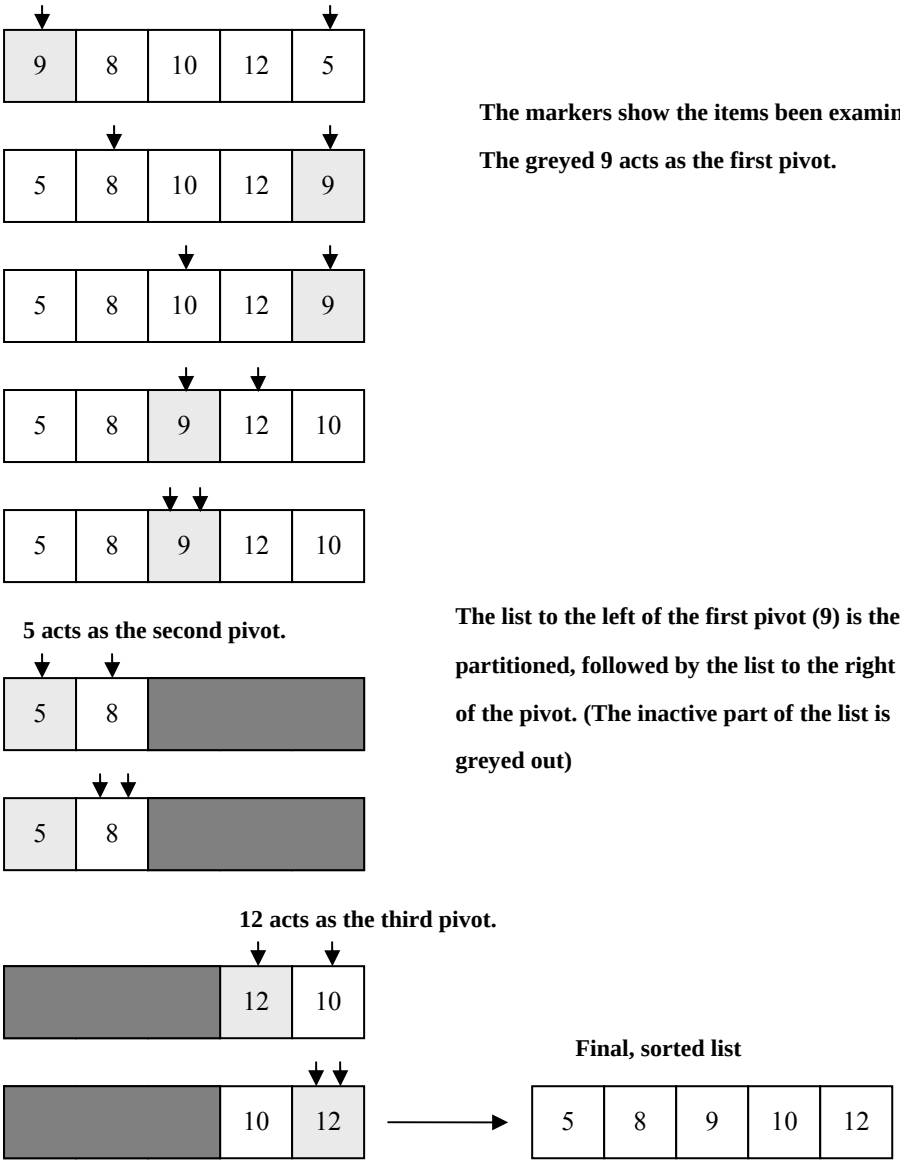


Figure A.11: Concept of the Quicksort

The Quicksort is the fastest commonly used general scenario sorting algorithm, with a complexity of $O(N \log N)$. However, the extensive use of recursion makes the Quicksort a difficult algorithm to comprehend and implement.

```
function partition(var L : TIntList; left, right : Integer) : Integer;
begin
  while true do
  begin

    while (left<right) and (L.List[left] < L.List[right]) do
      Dec(right);

    if (left<right) then
    begin
      Swap (L, left, right);
      inc(left);
    end
    else
    begin
      result := left;
      Exit;
    end;

    while (left<right) and (L.List[left] < L.List[right]) do
      inc(left);

    if (left<right) then
    begin
      Swap (L, left, right);
      dec(right);
    end
    else
    begin
      result := right;
      Exit;
    end;

  end;
end;
```

Figure A.12: Implementation of Quicksort support routine

```
procedure QuickSort(var L : TIntList; Left,Right : Integer);
var
  p : Integer;
begin

  if (left<right) then
  begin
    P := partition(L, left, right);
    quicksort(L, left, p-1);
    quicksort(L, p+1, right);
  end;

end;
```

Figure A.13: Implementation of Quicksort main routine

The discussion illustrates the basic operations of each sorting algorithm. Fundamentally, all sorting algorithms continuously swop items until the list is sorted. The methods of item selection employed by each algorithm are what differentiate

them in terms of performance and technique. These characteristics of algorithms thus influence the design of their associated animations.