Guido Rößling,
J. Ángel Velázquez-Iturbide (eds.)

Proceedings of the Fifth Program Visualization Workshop

Number 2008-04

Serie de Informes Técnicos DLSI1-URJC
ISSN 1988-8094
Departamento de Lenguajes y Sistemas Informáticos I
Universidad Rey Juan Carlos

Foreword

These are the proceedings of the *Fifth Program Visualization Workshop (PVW 2008)*, organized by the Universidad Rey Juan Carlos, Mostoles, Spain, help on July 3-4, 2008.

The Program Visualization Workshop series has been organized in Europe every second (even) year since 2000. Previous workshops have been organized in Porvoo, Finland (2000), at Hornstrup Centret, Denmark (2002), at the University of Warwick, UK (2004), and at the University of Florence (2006). The workshops have been organized in cooperation with ACM SIGCSE and in conjunction with the ITiCSE conference, to further promote participation in both conferences.

The aim of this workshop is to bring together researchers who design and construct visualizations or animations, as well as visualization or animation systems, for computer science, especially - but not exclusively - for programs, data structures, and algorithms. Above all, the workshop attracts educators who create, use, or evaluate visualizations and animations in their teaching. Due to the limited number of participants and the fact that most participants are working actively in the field of software visualization, the workshop is a great opportunity to exchange ideas and experiences, as well as announce novel systems.

The workshop in 2008 contained 18 papers coming from a total of 33 authors. The author list includes both “veterans” and “newcomers”, illustrating that the field is still alive and kicking.

This copy of the proceedings contains a copy of the papers submitted before the workshop, and therefore includes changes recommended by the program committee during the review phase. Each submission was reviewed by at least two program committee members.

I want to thank the program committee members for their critical and encouraging comments on this year’s submissions. The program committee consisted of:

- Guido Rößling (chair) (Darmstadt University of Technology, Germany)
- Mordechai Ben-Ari (Weizmann Institute of Science, Israel)
- Pierluigi “Pilu” Crescenzi (University of Florence, Italy)
- Camil Demetrescu (University of Rome “La Sapienza”, Italy)
- Ari Korhonen (Helsinki University of Technology, Finland)
- Lauri Malmi (Helsinki University of Technology, Finland)
- Thomas L. Naps (University of Wisconsin Oshkosh, USA)
- Rockford J. Ross (Montana State University, USA)
- Jaime Urquiza-Fuentes (Universidad Rey Juan Carlos, Spain)
- Ángel Velázquez-Iturbide (Universidad Rey Juan Carlos, Spain)

This workshop has been organized by the Departamento de Lenguajes y Sistemas Informáticos I of the Rey Juan Carlos University and sponsored by the Vicerrectorado de Extensión Universitaria of the Rey Juan Carlos University. The local arrangements were organized by Jaime Urquiza (co-chair), Ángel Velázquez (co-chair), Francisco Almeida, Raquel Hijón, Carlos Lázaro, Ascensión Lovillo, Antonio Pérez, Manuel Rubio, and Liliana Santacruz. Without their support and effort, the workshop would not have been possible – thank you!

Darmstadt, Germany / Mostoles, Spain, June 2008

Guido Rößling and Ángel Velázquez-Iturbide

Contents

Adaptive Hypermedia and Visualization
Pilar Rodríguez **1**

Visual Interactive Analysis: Insights into Software Comprehension
Roberto Therón **2**

Integrating Multiple Approaches for Interacting with Dynamic Data Structure Visualizations
James H. Cross II, T. Dean Hendrix, Larry A. Barowski **3**

Rationale behind the design of the EduVisor software visualization component
Jan Moons, Carlos De Backer **11**

Visualization of Procedural Abstraction
Stefan Schaeckeler, Weijia Shang, Ruth Davis **19**

First Steps Towards a Visualization-Based Computer Science Hypertextbook as a Moodle Module
Guido Rößling, Teena Vellaramkalayil **29**

Towards Seamless Merging of Hypertext and Algorithm Animation
Ville Karavirta **37**

Integrating test generation functionality into the Teaching Machine environment
Michael Bruce-Lockhart, Pierluigi Crescenzi, Theodore Norvell **45**

Kick-Start Activation to Novice Programming — A Visualization-Based Approach
Essi Lahtinen, Tuukka Ahoniemi **53**

Experiences on Using TRAKLA2 to Teach Spatial Data Algorithms
Jussi Nikander, Juha Helminen, Ari Korhonen **59**

Using Graphviz as a Low-cost Option to Facilitate the Understanding of Unix Process System Calls
Miguel Riesco, Marian Diaz Fondon, Dario Alvarez **67**

Dynamic Evaluation Tree for Presenting Expression Evaluations Visually
Essi Lahtinen, Tuukka Ahoniemi **73**

Work in Progress: Automatic Generation of Algorithm Animations for Lecture Slides
Otto Seppälä, Ville Karavirta **79**

PathFinder: A Visualization eMathTeacher for Actively Learning Dijkstra's algorithm
M. Gloria Sánchez-Torrubia, Carmen Torres-Blanc, Miguel A. López-Martínez **85**

Animation and Interactive Programming: A Practical Approach
Phillip Benachour **91**

Animalipse - An Eclipse Plugin for AnimalScript
Guido Rößling, Peter Schroeder **97**

A Java API for Creating (not only) AnimalScript
Guido Rößling, Stephan Mehlhase, Jens Pfau **105**

A Design of Automatic Visualizations for Divide-and-Conquer Algorithms
J. Ángel Velázquez-Iturbide, Antonio Pérez-Carrasco, Jaime Urquiza-Fuentes **113**

A First Set of Design Patterns for Algorithm Animation
Guido Rößling **121**

Pedagogical Effectiveness of Engagement Levels - A Survey of Successful Experiences
Jaime Urquiza-Fuentes, J. Ángel Velázquez-Iturbide **129**

Adaptive Hypermedia and Visualization

Pilar Rodríguez
Universidad Autónoma de Madrid

Pilar.Rodriguez@ii.uam.es

Abstract

Adapted, or customized systems, are usual in most application areas. Those systems are able to take some variables into account and, depending of their values, behave differently. Customized systems can be very sophisticated but, once their foreseen parameters are taken into account, their interfaces are consequently fixed, and no change takes place at runtime. By no change we mean that there is no need of making new information visible: from that point of view, user interactions will not affect system behavior once the application is running.

Beside adapted systems, adaptive systems are becoming quite useful in some areas. Adaptive systems differ from customized ones in several senses. Mainly, they aim to adapt themselves to user interactions at runtime. When talking about hypermedia systems, adaptation means not only adapting contents and/or navigation issues on the fly, but also adapting interfaces to the changing parameters. And always keeping in mind not to disturb users while interacting with the system

Consequently, visualization issues play a key role from adaptive system perspective. It is clear that, in adaptive systems, data visualization can be much more complex and subtle than in many other applications. In this work, main visualization issues for adaptive systems are addressed, making use of existing applications for illustrating the depicted facts.

Visual Interactive Analysis: Insights into Software Comprehension

Roberto Therón
Universidad de Salamanca

theron@usal.es

Abstract

Program visualization has traditionally been seen as a discipline for producing animated views of program executions. Beyond that, software comprehension is a matter of making sense of a particular kind of data that is complex, dynamic, often not evident, and evolving.

The goal of visual analytics is to facilitate the analytical reasoning process through the creation of software that maximizes human capacity to perceive, understand, and reason about complex and dynamic data and situations. Very impressive results have been achieved in program visualization. The main effort has been done with regard to finding suitable visual representations and retrieving the needed data, however, lesser attention has been paid to the use of interaction theory. Here is where visual analytics can be very helpful. The application of visual representations and interactions must necessarily be adapted to fit the needs of the task at hand, i.e., the comprehension of software in terms of debugging, evaluating and improving program performance, evaluating and reducing resource utilization, evaluation of algorithms in the context of complete programs and real data, understanding program behavior and teaching.

In this talk, I will explain how interactive visualization can be used in order to facilitate analytical insight and provide ways for better explore, analyze, and understand problems at the different stages of the software lifecycle. I will show several examples of successful experiences in other fields of approaching a problem from the visual analytics perspective that can be very inspiring for the program visualization community.

Integrating Multiple Approaches for Interacting with Dynamic Data Structure Visualizations

James H. Cross II, T. Dean Hendrix, and Larry A. Barowski
Department of Computer Science and Software Engineering
Auburn University
Auburn, Alabama 36849-5437 USA

{crossjh, hendrtd, barowla}@auburn.edu

Abstract

jGRASP 1.8.7 has integrated three approaches for interacting with its dynamic viewers for data structures: the debugger, the workbench, and a new text-based interactions tab that allows individual Java statements to be executed and expressions to be evaluated. While each of these approaches is distinct and can be used independently of the others, they can also be used together to provide a complementary set of interactions with the dynamic viewers. In order to integrate these approaches, the jGRASP visual debugger, workbench, and viewers had to be significantly redesigned. During this process, the structure identifier, which provides for the identification and rendering of common data structures, was also greatly improved by examining the examples from 20 data structure textbooks. The overall result of this integration effort is a highly flexible approach for user interaction with the dynamic data structure visualizations generated by a robust structure identifier.

1 Introduction

Visualizations of data structures have been used in limited ways for many years. To overcome a major obstacle to widespread use, namely lack of easy access and use, the jGRASP[1] IDE provides a structure identifier that attempts to identify and render traditional abstract visualizations for common data structures such as stacks, queues, linked lists, binary trees, heaps, and hash tables (Cross et al., 2007). These are dynamic visualizations in that they are generated while running the users program in debug mode. This technique helps bridge the gap between implementation and the conceptual view of data structures, since the visualizations can be based on the users own code.

jGRASP 1.8.7[2] provides numerous ways for the user to interact with the data structure visualizations: (1) the debugger, (2) the workbench, and (3) a new text-based interactions tab. Each of these is briefly described below and then more fully with examples in the sections that follow.

The most common way to open a viewer on a data structure object is via the debugger. The user simply sets a breakpoint on a statement near the creation of the data structure, and runs the program in debug mode. After the program stops at the breakpoint, the user single steps as necessary until the object is created, and then opens a viewer on the object by dragging it from the debug tab. As the viewer is opened, the structure identifier determines the particular type of data structure and then renders it appropriately. When the user steps through the executing program, the viewer is updated to show the effects.

In addition to interacting with the data structure visualizations via the debugger, jGRASP allows the user to interact with viewers via the workbench. Objects can be created and their methods invoked via menus and buttons on the UML class diagram and/or the source code edit windows. When the user invokes a method on the object (e.g., to insert a node into the data structure), the visualization is updated to show the effect of the method.

The third approach for interacting with the viewers is a text-based interactions tab, which is essentially a Java interpreter. That is, when the user enters a Java source statement and

[1]The jGRASP research project is funded, in part, by a grant from the National Science Foundation.
[2]jGRASP 1.8.7 is currently in pre-release and scheduled for full release in late summer 2008.

presses the ENTER key, the statement is immediately executed. If an expression is entered, it is evaluated and its value is displayed. This means that while at a breakpoint, the user can interact with any of the variables in the debug tab or any variables on the workbench. In addition, if the user creates an instance of a class via the interactions tab (e.g., LinkedList list = new LinkedList();), list is shown on the workbench. A viewer can be opened in the usual way by dragging the reference from the debug tab or workbench. Once opened, the viewer is updated as appropriate when statements or expressions are entered in the interactions tab. In the following sections, we discuss related work, and then we provide an extended linked list example to illustrate the integration of the debugger, workbench, and text-based interactions with the viewers for data structures. This is followed by examples of other common data structures generated by jGRASP. A brief summary of the integration effort is presented, including future integration tasks, and then we close with concluding remarks.

2 Related Work

Both the method and degree of user interaction with software visualizations have been shown to be primary contributors to effectiveness. Research indicates that passive modes of interaction with visualizations, e.g., only watching an animation of an algorithms behavior, are not as effective as more active engagement strategies, e.g., having the user manipulate elements of the visualization or respond to prompts (Stasko and Lawrence, 1998; Lauer, 2006). The context in which the visualization appears has also been shown to play a vital role in effectiveness (Hansen et al., 2002). These issues are now widely seen as fundamental to the advancement of software visualization research in education (Naps, 2000; Naps and Roessling, 2005).

We hold the pragmatic view that much of the learning and indeed much of the opportunity for learning occurs while a student is actively working on a source code implementation of an algorithm or data structure. Thus, we provide software visualizations that are directly coupled to real source code in the context of a full-featured IDE. This is similar to “scenario three” of user interaction described by Naps and Roessling (2005).

The three basic software visualization interaction techniques that jGRASP offers are based directly on common tools and idioms available in various IDEs: the debugger, object workbench, and interactions pane. The debugging process and the use of a debugger are central to almost all courses in which programming is involved and is supported by almost all IDEs. In addition, the use of a debugger as a pedagogical tool and as an important application area for software visualization has long been recognized (Baecker et al., 1997; Cross et al., 2002). The object workbench paradigm has been made popular by BlueJ (www.bluej.org), a popular IDE for early programming courses. In BlueJ, students use menus and dialogs to create object instances for the workbench and to invoke methods on those instances, without the need for a running program. In jGRASP, this notion of workbench has been extended to allow the user to open type-specific viewers on objects or primitives, as well as a viewer that identifies data structures on the fly. DrJava (www.drjava.org) has demonstrated the utility of an interactions tab for experimenting with Java source statements and expressions without the necessity of compiling and running a complete program. This is especially useful in a classroom setting. In jGRASP, we have extended the basic notion of the interactions tab in several ways. Most importantly, it is fully integrated with the debugger, workbench, and viewers.

The approach we have taken for data structure viewers in jGRASP is to automatically generate the visualization from the users executing source code and then to dynamically update it as the user steps through the source code in either debug mode or workbench mode, or as statements are executed in the interactions tab. This general approach is somewhat similar to the method used in Jeliot (Kannusmaki et al., 2004). However, jGRASP differs significantly from Jeliot in its target audience. Whereas Jeliot focuses on beginning concepts such as expression evaluation and assignment of variables, jGRASP includes visualizations for more complex structures such as linked lists and trees. In this respect, jGRASP is similar to DDD (Zeller, 2001). The data structure visualization in DDD shows each object with its fields and shows field pointers and reference edges. In jGRASP, each category of data structure (e.g., linked list vs. binary tree) has its own set of views and subviews which are intended to be similar to those found in textbooks. Although we are planning to add a general linked structure view, we began with the more intuitive "textbook" views to provide the best opportunity for improving the comprehensibility of data structures.

We have specifically avoided basing the visualizations in jGRASP on a scripting language, which is a common approach for algorithm visualization systems such as JHAVE (Naps, 2005). We also decided against modifying the users source code as is required by systems such as LJV (Hamer, 2004). Our philosophy is that for visualizations to have the most impact on program understanding they must be generated as needed from the users actual program during routine development and the user should be allowed to interact with the visualizations through commonly-used features of the IDE.

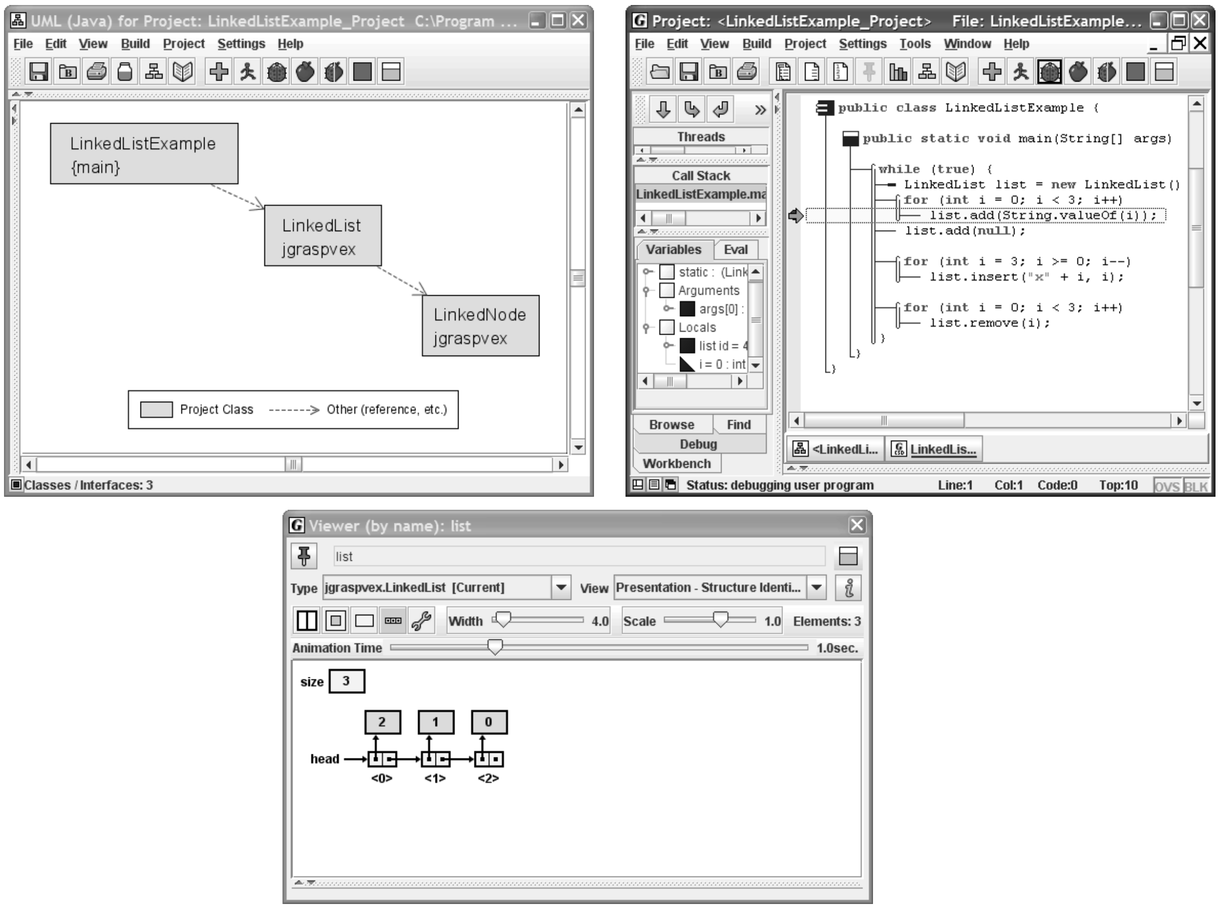
3 An Example – Integrated User Interaction

To see a meaningful visualization of a data structure in a viewer requires that an instance be created and its methods be invoked. The integrated approach in jGRASP allows this to be done by (1) using the debugger in the traditional way, (2) using the workbench menus and dialogs from the UML window or the edit window, and (3) entering source code into the interactions tab for direct execution. In each of these approaches, the viewer is opened on an instance by dragging it from the debug or workbench tab. Below, we will begin by using the debugger approach and then we will show how the user can interact with the visualization using both workbench and text-based approaches.

3.1 Using the Debugger

Consider the program LinkedListExample.java, which is provided with the jGRASP distribution. The UML class diagram in Figure 1(a) shows the dependencies among LinkedListExample, LinkedList, and LinkedNode. The LinkedList class is intended to be representative of a "textbook" example or of what a student or instructor may write. Figure 1(b) shows LinkedListExample.java stopped at a breakpoint where the list.add method is about to be invoked. Prior to stopping at the breakpoint, list was assigned to an instance of LinkedList, and thus it appears in the Variables tab of the Debug window. To open a viewer on list, the user simply drags it from the Debug window. During the process of opening the viewer, the Data Structure Identifier mechanism in jGRASP determines, in this case, that the object is a linked list structure and opens the appropriate viewer. As the user steps through the program, the nodes appear in the viewer. Figure 1(c) shows the visualization of list after three elements have been added.

The debugger approach is perhaps the most natural way for students to interact with the viewers. Since it involves stepping through a program, the visualization reinforces or clarifies the effect of each step by providing live feedback at a conceptual level. For example, Figure 2(a) shows the program after stepping into the insert method. The statement about to be executed will complete the linking of node into the data structure. Figure 2(b) shows the synchronized view of list at this point in the program. The visualization clearly indicates that prev.next is pointing to the last node in the structure (labeled "⟨3⟩"). As soon as the statement "prev.next = node;" is executed, the new node with value "x3" will move up into the structure in an animated fashion. The visualization in the viewer allows the student to make the connection between the implementation of the insert method and the concept of inserting an element in a linked list.



```
public class LinkedListExample {

   public static void main(String[] args)

      while (true) {
         LinkedList list = new LinkedList()
         for (int i = 0; i < 3; i++)
            list.add(String.valueOf(i));
         list.add(null);

         for (int i = 3; i >= 0; i--)
            list.insert("x" + i, i);

         for (int i = 0; i < 3; i++)
            list.remove(i);
      }
   }
}
```

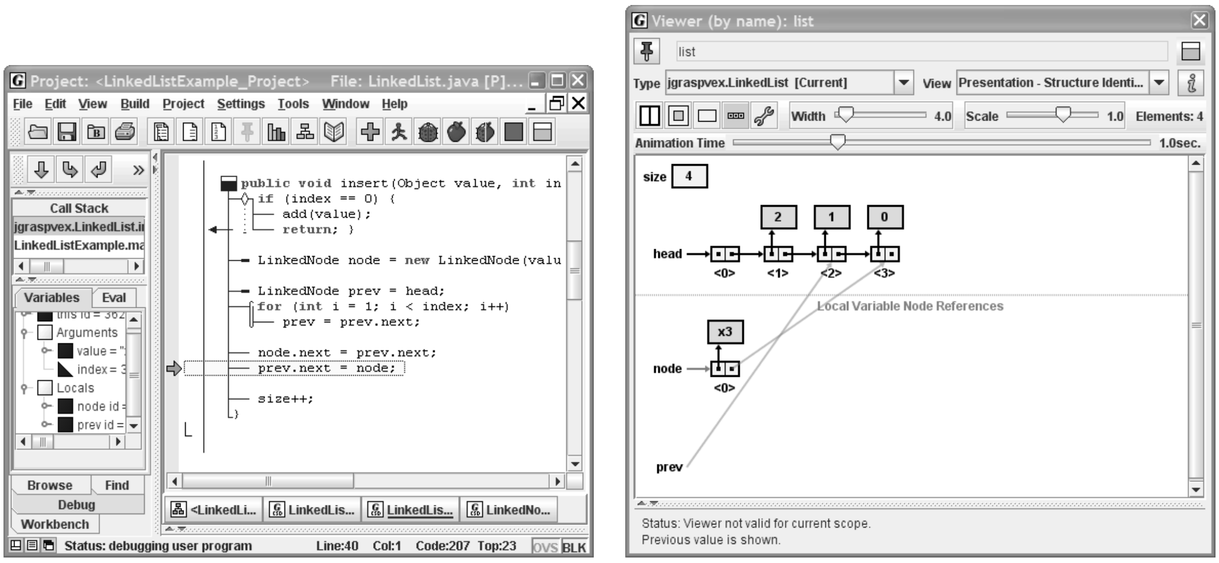
Figure 1: (a) UML diagram for LinkedListExample (b) Method main at breakpoint (c) View of list after three elements added

3.2 Using the Workbench

In the workbench approach for interacting with data structure visualizations, the user creates one or more instances of the class using menus or buttons on the UML class diagram or source code editing windows. From the UML diagram, this can be done by selecting Create New Instance on the right-click menu for the class, which pops up a Create instance dialog for class LinkedList as shown in Figure 3 below. When the instance is created, it appears on Workbench tab, similar to the way list appeared in the Debug tab in Figure 1 above. As before, a viewer is opened on the instance by dragging it from the workbench. The Invoke Method dialog for the instance is popped up by either right-clicking on the instance and selecting Invoke Method or by clicking the Invoke Method button on the viewer (upper right corner). Figure 3 shows the Invoke Method dialog for jgraspvex-LikedList-1 with the add method selected. When a method is invoked, the viewer on the instance is updated to show the new state. Thus, the user can interact with the data structure visualizations in the viewer by invoking a sequence to add, insert, and/or remove methods. The operations just described could have been performed alone or in the context of a running program. In the latter case, the operations could also have been performed on an instance of LinkedList created by the program itself.

3.3 Using Text-Based Interactions

The new Interactions tab in jGRASP 1.8.7 provides a Java interpreter that allows the user to enter expressions and statements and have them immediately evaluated/executed. To use



Project: <LinkedListExample_Project>   File: LinkedList.java [P]...

```
public void insert(Object value, int in
   if (index == 0) {
      add(value);
      return; }

   LinkedNode node = new LinkedNode(valu

   LinkedNode prev = head;
   for (int i = 1; i < index; i++)
      prev = prev.next;

   node.next = prev.next;
   prev.next = node;

   size++;
}
```

Status: debugging user program   Line:40   Col:1   Code:207   Top:23

Viewer (by name): list

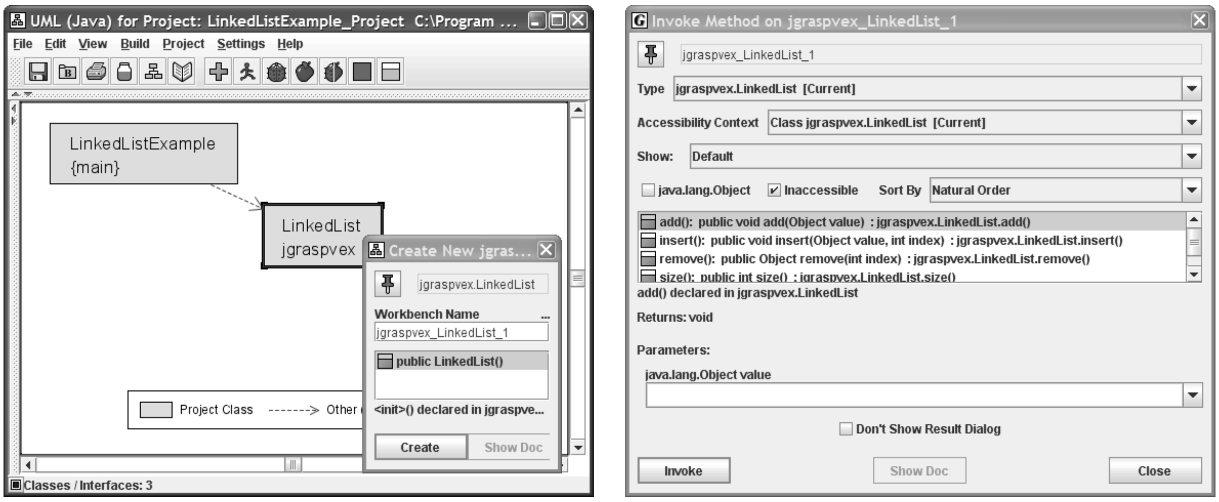
Type jgraspvex.LinkedList [Current]   View Presentation - Structure Identi...

Width 4.0   Scale 1.0   Elements: 4

Animation Time 1.0sec.

Status: Viewer not valid for current scope.
Previous value is shown.

Figure 2: (a) Stepping in list insert method (b) View of list showing details of insert



UML (Java) for Project: LinkedListExample_Project C:\Program ...

Create New jgras...

jgraspvex.LinkedList

Workbench Name: jgraspvex_LinkedList_1

public LinkedList()

<init>() declared in jgraspve...

Classes / Interfaces: 3

Invoke Method on jgraspvex_LinkedList_1

jgraspvex_LinkedList_1

Type: jgraspvex.LinkedList [Current]

Accessibility Context: Class jgraspvex.LinkedList [Current]

Show: Default

☐ java.lang.Object   ☑ Inaccessible   Sort By: Natural Order

add(): public void add(Object value) : jgraspvex.LinkedList.add()
insert(): public void insert(Object value, int index) : jgraspvex.LinkedList.insert()
remove(): public Object remove(int index) : jgraspvex.LinkedList.remove()
size(): public int size() : jgraspvex.LinkedList.size()

add() declared in jgraspvex.LinkedList

Returns: void

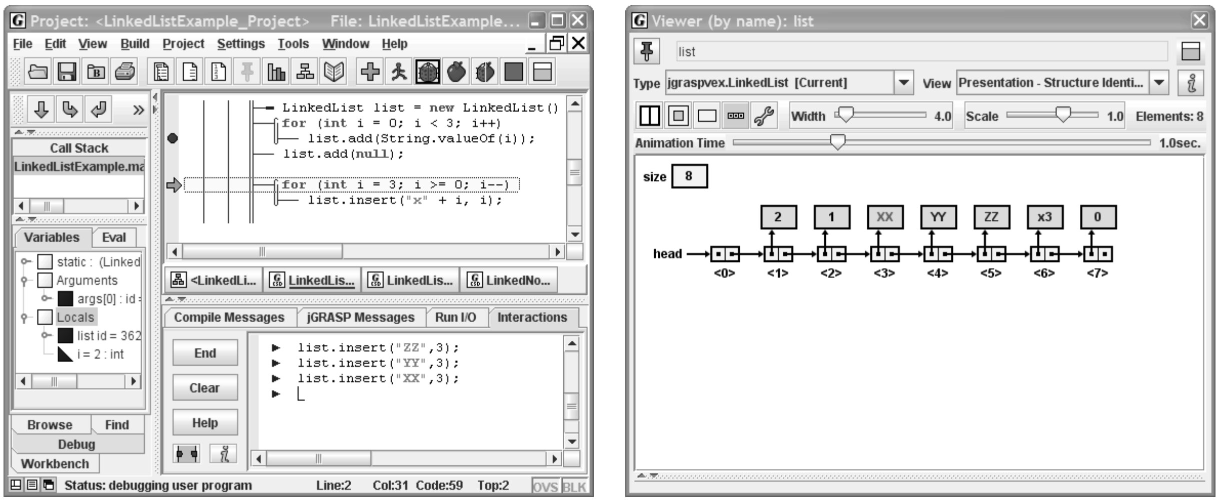
Parameters:

java.lang.Object value

☐ Don't Show Result Dialog

Figure 3: (a) Creating instance from UML diagram (b) Invoke method dialog

this feature in a stand-alone fashion with respect to data structure visualizations, the user could enter the code to create an instance and then enter statements that invoke methods on the instance. The advantage of this approach is that it allows the user to enter actual Java statements and execute them without having to enter and run an entire program, though the interactions tab can also be used to interact with elements in a running program. The integration with the workbench means that when a variable is declared and initialized (e.g., LinkedList myList = new LinkedList();), it appears on the workbench. This allows the user to open a viewer on the variable myList. Once the viewer is opened, any effects of the statements entered in the Interactions tab involving the methods invoked on myList will be seen in the viewer. To illustrate the degree of integration of the text-based interactions, consider our original example that we were running in debug mode. Figure 4(a) shows LinkedListExample stopped at the second for loop. Three list.insert method calls are shown in the Interactions tab. As these statements were entered, “ZZ”, “YY”, and “XX” respectively were inserted at location 3. As each node was entered, the existing nodes were moved to the right with the final result shown in Figure 4(b).



```
LinkedList list = new LinkedList()
for (int i = 0; i < 3; i++)
   list.add(String.valueOf(i));
list.add(null);

for (int i = 3; i >= 0; i--)
   list.insert("x" + i, i);
```

```
▶  list.insert("ZZ",3);
▶  list.insert("YY",3);
▶  list.insert("XX",3);
▶  
```

Status: debugging user program Line:2 Col:31 Code:59 Top:2

Viewer (by name): list

Type jgraspvex.LinkedList [Current] View Presentation - Structure Identi... Width 4.0 Scale 1.0 Elements: 8 Animation Time 1.0sec.

Project: <LinkedListExample_Project> File: LinkedListExample...
File Edit View Build Project Settings Tools Window Help
Call Stack
LinkedListExample.ma
Variables Eval
static : (Linked
Arguments
args[0] : id
Locals
list id = 362
i = 2 : int
Browse Find
Debug
Workbench
<LinkedLi... LinkedLis... LinkedLis... LinkedNo...
Compile Messages jGRASP Messages Run I/O Interactions
End
Clear
Help
size 8
2 1 XX YY ZZ x3 0
head
<0> <1> <2> <3> <4> <5> <6> <7>

Figure 4: (a) Interactions (b) Viewer for list after interactions

4 Examples of Other Data Structures

The jGRASP data structure identifier mechanism, which provides for the identification and rendering of common data structures, has been greatly improved by examining the examples from 20 data structure textbooks. The detailed results of the testing are described in (Montgomery et al., 2008). Example visualizations now recognized by the mechanism are shown in Figure 5. These were generated from the example source code provided with jGRASP and include a queue, doubly linked list, binary tree, binary heap, red-black tree, and hash table.

5 Summary of Integration

The debugger, workbench, and interactions in jGRASP are flexible and well-integrated. The workbench and interactions may be used alone, together, or in conjunction with a program that is being debugged. Any object visible from the debugger, workbench, or in a viewer, including any sub-component object, can be placed on the workbench. Any object or primitive value visible from the debugger, workbench, or in a viewer, including any sub-component, can be displayed separately in a viewer. Objects created in interactions appear on the workbench. Non-void return values from workbench-initiated method invocations are displayed in viewers. Items on the workbench and in viewers are named, and those names may be referenced by source code in interactions, or in expressions passed as parameters to workbench object creation and method invocation dialogs. Java expressions (not just objects) can be displayed in a viewer by dragging them from the "Eval" tab in the debugger or by using an interface that directly launches a viewer for an expression.

More integration is planned for the future. We intend to make it possible to echo workbench object creation and method invocation actions to the interactions tab as source code text. Also, special syntax will be recognized by interactions so that viewers for values and expressions can be launched directly from the interactions tab without leaving the text-based interface.

6 Conclusion

The data structure visualizations provided by jGRASP are intended to support teaching and learning activities in courses that include data structures. In U.S. computer science programs, this would typically be CS2 or CS3. The highly visual debugger in jGRASP, which provides a natural interface for the data structure visualizations, has also been a major strength for CS1 students as they learn the basics of object-oriented programming. Those students who

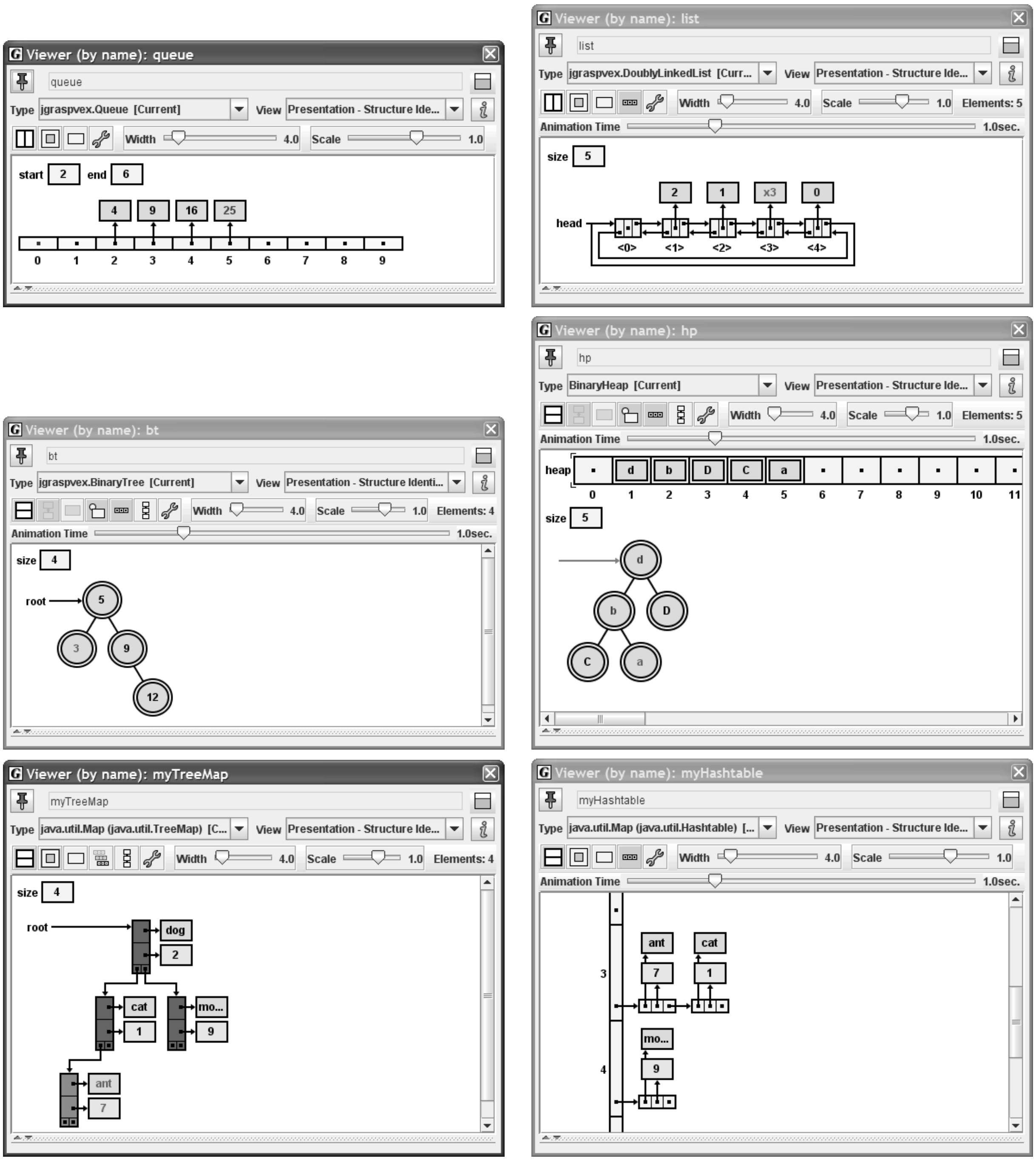

Viewer (by name): queue
Type jgraspvex.Queue [Current]
View Presentation - Structure Ide...
start 2 end 6
Viewer (by name): list
Type jgraspvex.DoublyLinkedList [Curr...
size 5
head
Viewer (by name): bt
Type jgraspvex.BinaryTree [Current]
size 4
root
Viewer (by name): hp
Type BinaryHeap [Current]
heap
size 5
Viewer (by name): myTreeMap
Type java.util.Map (java.util.TreeMap) [C...
size 4
root
Viewer (by name): myHashtable
Type java.util.Map (java.util.Hashtable) [...
Width 4.0
Scale 1.0
Animation Time 1.0sec.

Figure 5: Example visualizations automatically generated by jGRASP

use BlueJ in their CS1 course are likely to be comfortable with the workbench approach for developing their programs, and those who use DrJava are familiar with the text-based interactions approach. By integrating these three approaches, students in CS2 and CS3 are allowed to interact with their data structure visualizations in the manner with which they are most comfortable. Furthermore, the integration allows the students to mix and match the operational aspects of each approach in a seamless manner.

The process of integrating the approaches required that the jGRASP visual debugger, workbench, and viewers be significantly redesigned. The redesign of the viewers included significant improvements to the data structure identifier mechanism which generates the visualizations. Example programs from 20 data structure textbooks were used to tune the data structure identifier mechanism. The overall effect of the redesign should be a highly flexible approach for user interaction with a rich set of automatically generated dynamic data structure visualizations.

References

R. Baecker, C. DiGiano, and A. Marcus. Software visualization for debugging. *Communications of the ACM*, 40(4):44–54, 1997.

J. H. Cross, T. D. Hendrix, and L. A. Barowski. Using the debugger as an integral part of teaching cs 1. In *Proceedings of Frontiers in Education 2002*, November 2002.

J. H. Cross, T. D.Hendrix, J. Jain, and L. A. Barowski. Dynamic object viewers for data structures. In *Proceedings of the SIGCSE 2007 Technical Symposium*, pages 4–8, March 2007.

J. Hamer. A lightweight visualizer for java. In *Proceedings of Third Progam Visualization Workshop*, pages 55–61, July 2004.

S. R. Hansen, N. H. Narayanan, and M. Hegarty. Designing educationally effective algorithm visualizations: Embedding analogies and animations in hypermedia. *Journal of Visual Languages and Computing*, 13(2):291–317, 2002.

O. Kannusmaki, A. Moreno, N. Myller, and E. Sutinen. What a novice wants: students using program visualization in distance programming course. In *Proceedings of Third Progam Visualization Workshop*, pages 126–133, July 2004.

T. Lauer. Learner interaction with algorithm visualizations: Viewing vs. changing vs. constructing. In *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education (ITiCSE 2006)*, pages 202–206, June 2006.

L. N. Montgomery, J. H. Cross, T. D. Hendrix, and L. A. Barowski. Testing the jgrasp structure identifier with data structure examples from textbooks. In *Proceedings of the 46th ACM Southeast Conference*, pages 198–203, March 2008.

T. Naps. Instructional interaction with algorithm visualizations. *Journal of Computing Sciences in Colleges*, 16(1):7–8, 2000.

T. Naps. Jhave: supporting algorithm visualization. *IEEE Computer Graphics and Applications*, Sep/Oct:49–55, 2005.

T. Naps and G. Roessling. Development of xml-based tools to support user interaction with algorithm visualization. *ACM SIGCSE Bulletin*, 37(4):123–138, 2005.

J. Stasko and A. Lawrence. Empirically assessing algorithm animations as learning aids. In *Software Visualization: Programming as a Multimedia Experience*, pages 419–438, 1998.

A. Zeller. Visual debugging with ddd. *Dr. Dobb's Journal*, July, 2001.

Rationale behind the design of the EduVisor software visualization component

Jan Moons, Carlos De Backer
Universiteit Antwerpen
Faculty of Applied Economics
Department of Management Information Systems
Prinsstraat 13, 2000 Antwerpen

jan.moons@ua.ac.be, carlos.debacker@ua.ac.be

Abstract

The EduVisor software visualization component is a new pedagogical tool specifically developed to address some wide-spread problems in teaching object-oriented technology to novice programmers. The visualization tool is integrated in a world-class IDE, and shows the students the structure of their own creations at runtime. EduVisor is based on a solid grounding in literature and over 25 years of combined experience in teaching a CS1 course. With this component we have set the goal of helping our students progress faster through the most difficult initial stages of programming.

1 Introduction

Over the past decades software design has often been described as a *wicked* or difficult problem (Jeffries et al., 1981; Kim and Lerch, 1997; Kölling, 1999; Budgen, 2003). Dalbey and Linn (1986) note that the average student does not make much progress in an introductory programming course. More recently, there are many reports corroborating this position. For instance, in the infamous McCracken Report (McCracken et al., 2001) the authors noted that the average score on a programming test was only 22.89 out of 110 points for a sample of 216 students. As difficult as it is for students to acquire programming and software design skills, just as difficult is it for teachers to teach those skills.

This paper is concerned with a novel visualization tool that can be used as a teaching aid in CS1 courses. The tool is called EDUcational VISual Object Runtime or EduVisor, and seeks to incorporate a lot of the acquired knowledge from previous visualization projects. The goal of EduVisor is threefold. First, we want to improve students' comprehension of the concepts introduced during the CS1 course. Second, we want them to be able to debug their programs faster. Third, we want to increase the enthusiasm of students by visualizing (and thus reducing the abstraction level) their own efforts at the push of a button. The design of the tool is based on several decades of combined CS1 teaching experience and on a thorough grounding in relevant literature.

In section 2 we describe the driving forces behind the design of EduVisor. Section 3 describes the most important runtime issues one encounters during a CS1 course, which will be used as input to the design of EduVisor. Section 4 shows a small sample of the graphical representation used in the EduVisor component based on a simple use case. Section 5 provides an overview of the resulting properties of the component. Section 6 discusses the similarities and differences between EduVisor and related work. Finally, in section 7 we present our conclusions and provide an outlook on the future development of the EduVisor component.

2 Rationale of the EduVisor software visualization component

As so many educational institutions, the University of Antwerp has migrated from Pascal to C, later to C++ and finally to Java over the past two decades as the language of choice in our CS1 course. The switch to Java was made seven years ago. During our course we have noticed the same basic errors appear again and again, causing students to loose valuable time and generating frustration and disappointment.

On the highest level, these errors can be divided in compile-time errors and runtime errors. The code editor can help with some of the compile-time errors (although the compiler messages are very cryptic to novice programmers), but does nothing to aid in understanding runtime behavior. Thus, over the past five years, we have designed a visual language to illustrate the runtime behavior of a program. The language is, as we tend to say, as simple as possible and as complicated as necessary. We use this visual language when explaining programs at the whiteboard, and students' comprehension of these specific programs has improved markedly. However, when it is time to start programming their own exercises, the same errors tend happen all over again.

This is caused by several issues. First, the nature of their programming efforts is very much trial and error - which is actually a well known fact (Ben-Ari, 1998). Second, the students do not go through the effort of drawing out their solutions in the way we do at the whiteboard. This is not *that* surprising - creating the visual representations for a running program takes quite some time. Encouraging however is that, when we force them to draw their programs on a sheet of paper, most of the time they are able to pinpoint the problems themselves.

Therefore we concluded that an automated software component based on our language could help students in recognizing and correcting their problems sooner. We did an extensive review of visualization components that address some of these issues, but none were found to be completely satisfactory. Section 6 talks in more detail about these closest alternatives. EduVisor was thus conceived and designed to our specifications. With this new component we have set three interrelated goals:

1. **Improve students' comprehension of basic programming constructs**: The abstract nature of programming languages makes understanding the concepts very hard for beginners. We, along with many other researchers (e.g. Hundhausen et al. (2002); Milne and Rowe (2002); Naps et al. (2002)), believe this difficulty can be reduced to some extent by using engaging visualization techniques.

2. **Speed up the debugging process of runtime problems**: debugging runtime errors is difficult even for experienced programmers. The standard debuggers that come with the major IDE's are very powerful, but also very difficult to operate - too difficult for novice programmers.

3. **Increase their enthusiasm about object-oriented programming**: The visual representation will provide an important incentive to students. As stated by Ross (1991), *it is a tacitly known fact that programmers like to see their creations in action. All artisans are intrigued by what they create, and they like to observe their work from all angles [. . .].*

3 CS1 runtime issues

After describing our reasons for developing EduVisor we take a look at the specific problems we would like to address. Table 1 presents a listing which is loosely based on the list of Garner et al. (2005), but restructured and rephrased to fit our purpose in two ways. First, the list is rephrased to present the causes rather than the symptoms of programming difficulties. Second, we only include runtime problems in the list, because this is the focus of our visualization tool. The next section details a use-case based on one of these problems and specifies how EduVisor will address it using visualization.

4 A simple EduVisor GUI use-case

This example details a problem we have witnessed recently with one of our students during our first lesson on objects. The goal was to write a program consisting of two classes, a `Bank` class and an `Account` class. The following code presents the main method located in the `Bank`

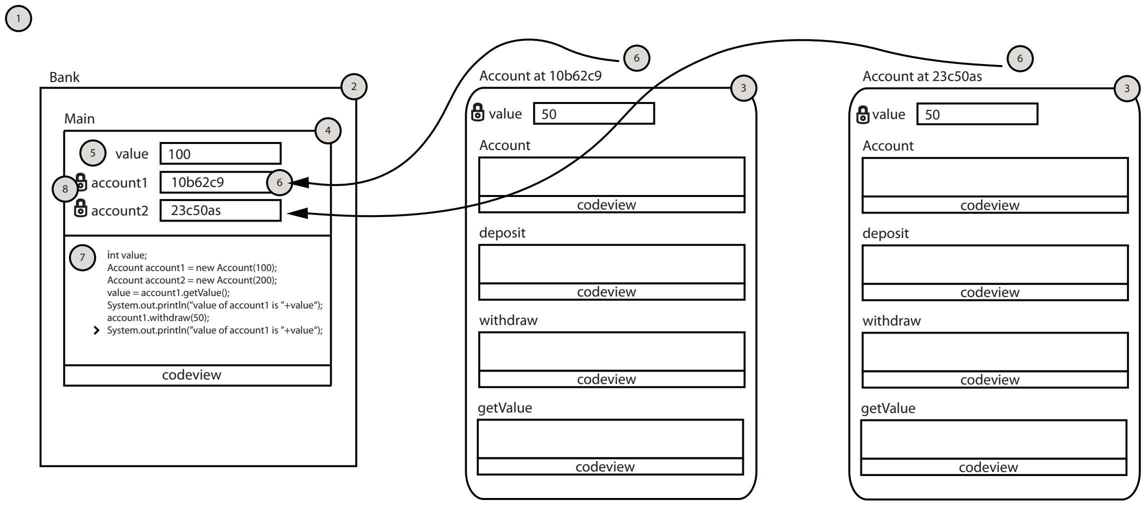
Error	description
A. Failing to understand program design	1. Failing to identify the correct classes. 2. Failing to identify the correct methods. 3. Failing to construct the correct algorithms. 4. Failing provide the necessary variables.
B. Failing to understand the nature of objects	1. Failing to understand that objects are persistent structures in memory holding their own state. 2. Failing to understand that a method can instantiate multiple objects of the same kind. 3. Failing to understand the difference between static and non-static structures. 4. Failing to understand that objects can only be queried for their state through a reference. 5. Failing to understand the nature of references (e.g. returning a reference when the calling method already holds that reference).
C. Failing to understand message passing	1. Failing to understand that methods have to be actively called. 2. Failing to understand the parameter passing mechanism. 3. Failing to understand that return variables have to be caught.
D. Failing to understand variables	1. Failing to understand variable scoping. 2. Failing to keep track of the values of variables in a running program. 3. Failing to understand the necessity and operation of a control variable inside a loop.

Table 1: A list of common causes of runtime errors encountered during a CS1 course, loosely based on Garner et al. (2005)

class, containing the problem. The code in italics was **not** present in the student's solution - i.e. the `getValue` method did not get called after calling the `withdraw` method.

```
public static void main(String[] args){
        int value;
        Account account1 = new Account(100);
        Account account2 = new Account(200);
        value = account1.getValue();
        System.out.println(``value of account1 is ''+value );
        account1.withdraw(50);
        //value = account1.getValue();
        System.out.println(``value of account1 is ''+value );
}
```

The student thought he was using the variable of the Account instance because he was referring to the object just before. This is a typical B4 problem - *Failing to understand that objects can only be queried for their state through a reference.* EduVisor will help the student trace this error through dynamic visualization of the program runtime. Figure 1 shows a snapshot of the proposed visualization style. The following list details some of the visual features of EduVisor.



```
int value;
Account account1 = new Account(100);
Account account2 = new Account(200);
value = account1.getValue();
System.out.println("value of account1 is "+value");
account1.withdraw(50);
> System.out.println("value of account1 is "+value");
```

Figure 1: EduVisor visualization snapshot of this use-case

1. All information is presented on one single canvas.

2. Every class holding static information is represented as a rectangle with the name of the class positioned above the rectangle.

3. Every object is represented as a rounded rectangle with the name of the originating class and the hash-code of the object positioned above the rounded rectangle.

4. Every method (and every scoped block within a method) has its own area to hold the local variables. The variables are dispensed when the method execution is complete.

5. Every variable is represented as a named rectangle that can hold a value, either of primitive or of reference type.

6. At object instantiation, a new object gets drawn on the canvas including member variables and method areas. The object lives as long as there are references pointing to the object. At instantiation, the memory address is transported to the variable holding the address.

7. Every method has a code area which can be uncollapsed. The code area shows the method implementation.

8. Every variable and method with reduced visibility relative to an active method (local variables as well as private member variables) is adorned with a lock symbol. The symbols are dynamically updated synchronously with the current active method.

5 EduVisor solutions to CS1 problems

Because we can not elaborate on the architectural details of our solution in this restricted space, we will not detail the libraries and code representations used by EduVisor. Rather, in this paper we want to describe the ways in which EduVisor will help in solving the categories of CS1 problems we have defined in section 2. The following list contains references to the number of the problem (cf. table 1) that an EduVisor feature addresses. It should be noted that the effectiveness of the software component has not yet been tested with students. The primary reason for this section is to detail how we *expect* EduVisor to help, and any claims presented in this section have yet to be confirmed.

1. **Failing to understand program design:** the single canvas approach allows the novice programmer to see the entire structure of the program at any time during the execution. All static and dynamic structures as well as all available variables can be seen without having to switch representations. Panning and zooming capabilities help with understanding more complex structures. This unified visual presentation will help the students to see e.g. which classes (A1), methods (A2) and variables (A4) are part of their program and help them understand the deficiencies in their design. The step-wise nature of the visualization will help them understand their algorithms (A3) better.

2. **Failing to understand the nature of objects:** every single object is explicitly represented on the canvas using rounded rectangles (B2). Every object contains only non-static member variables, explaining to the students the difference in runtime behavior between static and non-static structures (B3). The values of these variables are *always* visible, which will help the student in understanding the persistent and autonomous nature of an object (B1). Reference variables are represented in a different color than regular variables, and the value of the reference variable is the hash-code of the object. By clicking on the reference variable the corresponding object is highlighted, which will help in understanding the nature of references (B4 and B5).

3. **Failing to understand message passing:** Active objects are highlighted on the diagram. This way students see that an object is only active when a method of that object is called (C1). In addition, the values of the variables that are passed as parameters to a method are animated from the calling method to the called method, which helps in understanding the variable passing mechanism (C2). Return variables are also animated. Those return values that are not stored in a variable disappear, explaining the need to store return values (C3).

4. **Failing to understand variables:** All variables are always visible on the canvas and presented in their own scope (class, method or block) and adorned with modifier symbols that are dynamically adjusted to reflect the variables visible to an active method. This helps in understanding scoping (D1). In addition to the variables themselves the values of these variables are also visible, helping students keep track of program state (D2) and helping with understanding control variables in loop and selection structures (D3).

6 EduVisor contrasted with related work

Over the past three decades many studies have focused on improving and refining teaching methods for CS1 courses, which has resulted in an extensive pedagogical toolbox that can be used by computer science teachers. Some of the tools teachers have at their disposal are specialized IDE's such as JGrasp (Hendrix et al., 2004), BlueJ (Kölling, 1999; Kölling et al., 2003) and ProfessorJ (Gray and Flatt, 2003), programming micro-worlds such as Alice (Cooper et al., 2003) and ObjectKarel (Xinogalos et al., 2006) and advanced visualization environments such as JEliot3 (Moreno et al., 2004) and JIVE (Gestwicki and Jayaraman, 2005). For reasons of conciseness, in this paper we limit ourselves to only the last category.

The tools we discuss in detail are JEliot3[1] and JIVE[2]. Both have great merit and had considerable influence on the design of EduVisor. JEliot3 is a tool based on over ten years of development, starting with JEliot, later JEliot2000 and finally JEliot3. JIVE has a long history itself, starting as a stand-alone tool and recently reborn as an Eclipse plug-in. We also discuss the program state visualization tool by Seppälä (2004), which states similar goals as EduVisor.

JEliot uses several simultaneous representations to present the visualization. The canvas is divided in a memory stack, a constants area and an object heap. In addition, JEliot presents the data as it is processed by the virtual machine, i.e. using a method stack. Our emphasis is on *understanding the program architecture*, i.e. type A problems, not the VM. In addition, due to their particular implementation it is not possible to view all values on the method stack with one look at the canvas. This makes it difficult to *keep track of the values of variables in a running program* (D2). In addition, JEliot's canvas is based directly on the Java AWT classes and proprietary development. This implies certain restrictions, such as the complete absence of select, zoom and pan tools. These features are very important, as described by (Shneiderman, 1996). His visual mantra of *overview first, zoom and filter, and then detail on demand* is often mentioned as one of the cornerstones of good visualization tools. EduVisor is much more ambitious in this regard, thanks to it's use of an advanced open source visualization library, the Netbeans Visual Library[3].

JIVE has multiple representations of the same runtime behavior. We are presented with an object diagram and with a sequence diagram. However, it is not possible to see the values of variables contained in objects and methods nor the values of the parameters passed to methods and the return values of methods. EduVisor, on the other hand, uses the single canvas approach and shows dynamic behavior directly on this single canvas. This includes all values of reference and primitive variables in the program at any time. JIVE uses the Eclipse Graphical Editing Framework[4] to provide the representation. Jive should thus have zoom and pan features. However, in the most recent version zooming features are available through menu buttons and no easy panning or selection features exist.

The program state visualization tool mentions some of the same goals as Eduvisor. In Seppälä (2004), the authors state that *[their] notation attempts to show most of the runtime state of the program in a single diagram. Essentially, this means displaying all relevant instances, all references to them and some of the contents of the runtime stack together.* However, in their paper the presentation of program state seems to be quite different from the EduVisor presentation. The diagrams do not show the values of the variables, which is crucial in our system. For instance, the diagrams show references between objects as arrows between these objects, but there is no mention of the reference variables holding the objects. In addition, the diagrams do not show objects as environments of execution, i.e. the methods are not represented in the objects. We have found no further mention of this tool in literature.

One of our demands for a visualization tool was easy integration in a widely used IDE. We chose Sun's Netbeans as our platform, an thus EduVisor runs on the same platforms as Netbeans. Jeliot does not provide integration with a widely used IDE. Jive, on the other hand, is integrated with Eclipse - another widely used java IDE. It is not clear whether the program state visualization tool by Seppälä provides IDE integration, but according to the screenshots, it does not.

7 Outlook and conclusion

With EduVisor we have devised a visualization component that can be integrated easily in a world-class IDE such as Netbeans. The code is currently in alpha status but is being

[1]JEliot3 is available online at `http://cs.joensuu.fi/~jeliot/`

[2]JIVE is available online at `http://www.cse.buffalo.edu/jive/`

[3]The Netbeans Visual Library is available online at `http://graph.netbeans.org/`

[4]GEF is available online at `http://www.eclipse.org/gef/`

further developed as part of the PhD project of the first author. The final intent is to include additional ITS (Intelligent Tutoring System - see Wei et al. (2005)) functions such as pop quizzes and course material through XML based code-injection into the intermediate visualization code. Once the code reaches beta in the course of this year, it will be released on a public server. Our first goal now is to further develop this code base, starting with the visualization features and working our way up to the code infusion. Next we will perform experiments to research important features such as the one-canvas philosophy, the animation features and the utility of the additional pedagogical features afforded by the ITS functions.

The main goal of this paper was to present the design philosophy of our EduVisor visualization component. Based on literature and experience we have created a list of common causes of CS1 runtime problems. This list is currently being validated during course sessions and the intermediate results indicate that the list indeed represents the most common issues. The list also serves as input to the design of EduVisor. Finally, we have presented the solutions EduVisor offers to these common problems and contrasted our work with that of similar environments.

References

Mordechai Ben-Ari. Constructivism in computer science education. *SIGCSE Bulletin*, 30(1): 257–261, 1998. ISSN 0097-8418. doi: http://doi.acm.org/10.1145/274790.274308.

David Budgen. *Software Design.* Addison Wesley, second edition, May 2003. rapid.

Stephen Cooper, Wanda Dann, and Randy Pausch. Teaching objects-first in introductory computer science. In *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 191–195, New York, NY, USA, 2003. ACM. ISBN 158113648X. doi: 10.1145/611892.611966. URL `http://portal.acm.org/citation.cfm?id=611966`.

John Dalbey and Marcia C. Linn. Cognitive consequences of programming: Augmentations to basic instruction. *Journal of Educational Computing Research*, 2:75–93, 1986.

Sandy Garner, Patricia Haden, and Anthony Robins. My program is correct but it doesn't run: a preliminary investigation of novice programmers' problems. In *ACE '05: Proceedings of the 7th Australasian conference on Computing education*, pages 173–180, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc. ISBN 1-920682-24-4.

Paul Gestwicki and Bharat Jayaraman. Methodology and architecture of jive. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 95–104, New York, NY, USA, 2005. ACM. ISBN 1-59593-073-6. doi: http://doi.acm.org/10.1145/1056018.1056032.

Kathryn E. Gray and Matthew Flatt. Professorj: a gradual introduction to java through language levels. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 170–177, New York, NY, USA, 2003. ACM. ISBN 1-58113-751-6. doi: http://doi.acm.org/10.1145/949344.949394.

T. Dean Hendrix, II James H. Cross, and Larry A. Barowski. An extensible framework for providing dynamic data structure visualizations in a lightweight ide. *SIGCSE Bulletin*, 36 (1):387–391, 2004. ISSN 0097-8418. doi: http://doi.acm.org/10.1145/1028174.971433.

C. D. Hundhausen, S. A. Douglas, and J. T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages & Computing*, 13:259–290, 2002.

R. Jeffries, A. Turner, P. Polson, and M. Atwood. *The processes involved in designing software. Cognitive Skills and Their Acquisition.*, pages 225–283. Erlbaum, Hillsdale, N.J., 1981.

J. Kim and F.J. Lerch. Why is programming (sometimes) so difficult? programming as scientific discovery in multiple problem spaces. *Information Systems Research*, 8(1):25–50, 1997.

M. Kölling. Teaching object orientation with the blue environment. *Journal of Object-Oriented Programming*, 12(2):14–23, 1999.

M. Kölling, B. Quig, A. Patterson, and J. Rosenberg. The bluej system and its pedagogy. *Computer Science Education*, 13(4), December 2003.

Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. *SIGCSE Bulletin*, 33(4):125–180, 2001. ISSN 0097-8418. doi: http://doi.acm.org/10.1145/572139.572181.

Iain Milne and Glenn Rowe. Difficulties in learning and teaching programming - views of students and tutors. *Education and Information Technologies*, 7(1):55–66, 2002. ISSN 1360-2357. doi: http://dx.doi.org/10.1023/A:1015362608943.

Andrés Moreno, Niko Myller, Erkki Sutinen, and Mordechai Ben-Ari. Visualizing programs with jeliot 3. In *AVI '04: Proceedings of the working conference on Advanced visual interfaces*, pages 373–376, New York, NY, USA, 2004. ACM. ISBN 1-58113-867-9. doi: http://doi.acm.org/10.1145/989863.989928.

Thomas L. Naps, Guido Rössling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger, and J. Ángel Velázquez-Iturbide. Exploring the role of visualization and engagement in computer science education. In *ITiCSE-WGR '02: Working group reports from ITiCSE on Innovation and technology in computer science education*, pages 131–152, New York, NY, USA, 2002. ACM. doi: http://doi.acm.org/10.1145/960568.782998.

Rockford J. Ross. Experience with the dynamod program animator. In *SIGCSE '91: Proceedings of the twenty-second SIGCSE technical symposium on Computer science education*, pages 35–42, New York, NY, USA, 1991. ACM. ISBN 0-89791-377-9. doi: http://doi.acm.org/10.1145/107004.107013.

Otto Seppälä. Program state visualization tool for teaching cs1. In *Program Visualization Workshop*, pages 62–67, Warwick, UK, 2004. The University of Warwick.

Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *VL '96: Proceedings of the 1996 IEEE Symposium on Visual Languages*, page 336, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7508-X.

Fang Wei, Sally H. Moritz, Shahida M. Parvez, and Glenn D. Blank. A student model for object-oriented design and programming. *J. Comput. Small Coll.*, 20(5):260–273, 2005. ISSN 1937-4771.

Stelios Xinogalos, Maya Satratzemi, and Vassilios Dagdilelis. An introduction to object-oriented programming with a didactic microworld: objectkarel. *Comput. Educ.*, 47(2): 148–171, 2006. ISSN 0360-1315. doi: http://dx.doi.org/10.1016/j.compedu.2004.09.005.

Visualization of Procedural Abstraction

Stefan Schaeckeler, Weijia Shang, Ruth Davis
Department of Computer Engineering, Santa Clara University, Santa Clara, CA 95053

sschaeck@engr.scu.edu

Abstract

Visualizing impacts of an optimization pass helps to reason about, and to gain insight into, the inner workings of the optimization pass. In this paper, we visualize the impacts of two procedural abstraction passes. For this, we modified two procedural abstraction post pass optimizers to visualize the difference in machine code before and after optimization by drawing abstracted fragments in the original program. We explain how the generated visualizations aid in better understanding the optimization passes.

1 Introduction

Visualizations are often used in mechanical engineering, chemistry, physics, and medicine Diehl (2007), but are occasionally used in computer science to aid program understanding as well (see for example the ACM Symposia on Software Visualization (SOFTVIS), the IEEE Workshops on Visualizing Software for Understanding and Analysis (VSSOFT), or the Program Visualization Workshops (PVW)). For program understanding, program executions generate often very large traces. It is a challenging task to represent these masses of data in a digestable form and a lot of research is conducted for appropriate visualization techniques. Visualizations for understanding optimization passes are not always so complex. We found for procedural abstraction natural visual representations, that are simple yet powerful enough to *completely* understand in its entirety. We believe visualizations are a great aid for compiler writers to understand their optimization passes in greater depth and we hope the gained insight might help them to improve the optimization passes.

In this paper, we visualize procedural abstraction. All we see from running a size optimization pass such as procedural abstraction is one number only—the reduction of the program. To make its inner workings visible, we generate from the internal data-structures of our optimization passes several visualizations. After a brief review in the next section on procedural abstraction, we introduce these visualizations in sections 3 and 4 and explain how they help in better understanding procedural abstraction. Section 5 discusses future work and section 6 concludes the paper.

2 Background on Procedural Abstraction

Optimizing compilers traditionally target execution speed, but may also target code size as this becomes increasingly important for embedded systems. A common technique for compacting code is procedural abstraction. In its standard form, hereafter called *traditional procedural abstraction*, equivalent code fragments are identified, abstracted in a new procedure, and eventually replaced by procedure calls. This saves all but one occurrence of the fragments and adds a small overhead of one procedure call per fragment and one return instruction per abstracted procedure. Abstracted procedures are minimalistic functions without function prologues or epilogues.

Example 1 (Traditional Procedural Abstraction) *In the original code of Fig. 1a, either two code fragments à four instructions (Fig. 1b) or three code fragments à three instructions (Fig. 1c) can be abstracted. Whatever abstraction is more beneficial in terms of code size can then be chosen.*

The challenge of procedural abstraction is to efficiently find fragments for abstraction. Fraser et al. (1984) and Cooper and McIntosh (1999) use suffix trees to identify fragments for

```
a. load r1, $5200   load r1, $5200   load r1, $5300
   add r1, r2       add r1, r2       add r1, r2
   rot r1, $2       rot r1, $2       rot r1, $2
   mul r1, r1       mul r1, r1       mul r1, r1


b. call f           call f           load r1, $5300   f: load r1, $5200
                                     add r1, r2          add r1, r2
                                     rot r1, $2          rot r1, $2
                                     mul r1, r1          mul r1, r1
                                                         ret


c. load r1, $5200   load r1, $5200   load r1, $5300   f: add r1, r2
   call f           call f           call f              rot r1, $2
                                                         mul r1, r1
                                                         ret
```

Figure 1: Example of Procedural Abstraction

abstraction in O(n∗log(n)) time. The details do not concern us in this paper and we assume fragments for abstraction are already identified.

3 Visualization of Procedural Abstraction

We implemented in (Schaeckeler and Shang, 2008) a traditional procedural abstraction post-pass optimizer for Intel's 32-bit architecture (IA32). This optimization pass could reduce code sizes of seven programs from the MediaBench embedded systems benchmark suite on average by 2.502%. We use in this paper the mpeg encoder `mpeg2enc` optimized with earlier versions of our compactors as a running example. It has with 13, 599 instructions and 49, 927 bytes the right size for visualization on paper. We found in this program 333 abstracted fragments which could be abstracted in 66 procedures. This results in a compression ratio of 98.840%.

Programs are usually visualized either graph or pixel based. For procedural abstraction, we worked out several pixel based visualizations, that are not only a natural choice, but avoid also known shortcomings of graph based visualizations like scalability, layout and mapping problems.

We visualize instructions in the original program as what we call a *program map*. For program maps, there can be two levels of abstraction in which pixels represent either whole instructions or individual bytes of instructions, in ascending order from left to right, starting in the upper left corner and wrapping around at the end of each line. As the main purpose of program maps is to identify fragments, it is convenient to introduce a new term and call all pixels representing an individual fragment a *string*.

Pixels are quadratic and have length and area. Color may be used to emphasize pixels and strings. In the byte representation, the lengths and areas of pixels and strings are proportional to the sizes of instructions and fragments, and their quantities can be easily estimated from the visualization. If this is not necessary, then the more compact instruction representation may be sufficient and can be used instead.

Procedural abstraction has a flat view on the code, because abstracted are fragments, i.e. sequences of instructions, which are in turn bytes. Hence, the two levels of abstraction are enough to capture its essence.

We generated program maps for the first time in (Schaeckeler and Shang, 2008). We used for each abstracted instruction the same color and it was not possible to distinguish adjunct abstractions from single abstractions. In Fig. 3, we refine the program map by using light gray for the last pixel of an otherwise gray string. This reveals two times adjunct fragments which we haven't seen so before.

In Fig. 3, we see a lot of fragments consisting of one instruction, only. Table 1 gives a detailed breakdown. More than 50% of all fragments are individual instructions, while the remaining fragments consist of two to seven instructions. Fig. 4 gives the program map over bytes and light gray—here of the last five pixels—is used to mark ends of strings, again. It can be seen that there are a lot of short fragments. Table 2 gives a detailed breakdown. More than 50% of all fragments are with six or seven bytes pretty short. The remaining fragments extend gradually up to 20 bytes, and then there are two additional fragments of 27 bytes.

Table 1: Number of Fragments and Procedures over their Lengths in Instructions

sequence length [instr.]	0	1	2	3	4	5	6	7	≥ 8
number of fragments	0	182	43	23	34	39	10	2	0
number of procedures	0	18	12	7	12	11	5	1	0

Table 2: Number of Fragments and Procedures over their Lengths in Bytes

sequence length [bytes]	≤ 5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	≥ 27
number of fragments	0	47	144	8	10	7	39	22	9	17	10	8	4	2	2	2	0	0	0	0	0	2	0
number of procedures	0	4	15	2	1	2	7	9	4	8	4	4	2	1	1	1	0	0	0	0	0	1	0

As short fragments imply small net gains, we further investigate the net gains of fragments. A light gray pixel in Fig. 3 can be interpreted to represent the function call instruction responsible for the overhead per fragment, but this is a distorted view as instruction lengths range on IA32 from one to 17 bytes. A function call instruction `call <32-bit address>`, for instance, is five bytes in length, one byte for the opcode and four bytes for the address field. A byte representation is necessary for capturing the function call overhead. Because the light gray pixels of strings in Fig. 4 have exactly the size of a function call overhead, this figure can be used to analyze the overhead. Net gains of abstracted fragments are then represented by the remaining gray pixels. For each abstracted procedure, there is also an overhead of one byte for the return instruction `ret`. The accumulated area for all 66 return instructions occupies 27.5% of a line and is given at the bottom of Fig. 4 as a black string.

The areas of all 333 abstracted fragments, e.g. of all gray and light gray pixels, compromise 4.627% of the whole program map, i.e. 4.627% or 2,310 bytes of code is abstractable. As the overhead is five bytes per abstracted fragment and one byte per abstracted procedure, this results in an accumulated overhead of 1,731 bytes or 3.461% of the program size and what remains is a net gain of merely 579 bytes or 1.160%.

That the overhead is almost three times the net gain is quite disappointing. This observation motivated us to investigate alternative computer architectures with different function call / return overheads. If the function call / return overhead were less, then there will be not only less overhead for abstraction, but also further abstractable fragments will emerge, because more fragments have then a non-negative benefit, i.e. are larger than the function call instruction. Table 3 gives statistics for function call and return instructions of varying sizes. The upper limit is for no function call / return overhead and would result in a reduction of 20.548%. The corresponding program maps of Fig. 5 and Fig. 6 show the high redundancies of instructions. Two interesting cases that can be implemented in hardware, are:

call instr. size = 5 bytes; return instr. size = 0 bytes: Encoding the length of the abstracted procedure in a function call instruction can reduce the program size by 1.318%. This has also other interesting consequences. Abstracted procedures can overlap (see Liao et al. (1999)) or don't need to be abstracted in new procedures (see Lau et al. (2003)). This might lead to further reductions.

call instr. size = 3 bytes; return instr. size = 1 byte: A new function call instruction with a relative address mode can reach all procedures in programs $\leq 65,536$ bytes with an address field size of two bytes. This can reduce the program size by 3.603%. If this addressing mode is then also used for regular functions, then a further reduction can be expected.

Table 3: Statistics over the Lengths of Call and Return Instructions

size of call [bytes]	0	1	2	3	4	5
size of ret [bytes]	0	0	0	0	0	0
procedures [#]	654	600	370	226	138	79
fragments [#]	4026	3470	1610	969	651	376
overhead [bytes]	0	3470	3220	2907	2604	1880
net gain [bytes]	10259	6230	3339	2025	1186	658
compression [%]	20.548	12.478	6.688	4.056	2.375	1.318

size of call [bytes]	0	1	2	3	4	5
size of ret [bytes]	1	1	1	1	1	1
procedures [#]	653	491	309	180	121	66
fragments [#]	4024	3231	1458	852	609	333
overhead [bytes]	643	3722	3225	2736	2557	1731
net gain [bytes]	9605	5630	2969	1799	1048	579
compression [%]	19.238	11.276	5.947	3.603	2.099	1.160

Not all fragments can be used for abstraction. Fragments must be single entry–single exit regions and can extend in our implementation up to single basic blocks. Furthermore, fragments should not include function calls or stack accesses[1] as calls to abstracted procedures modify the stack by pushing the return address on the stack and then wrong stack slots might be accessed. See (Schaeckeler and Shang, 2008) for implementation details. Keeping fragments within basic blocks but ignoring the latter two constrains results in a saving of 3,822 bytes or 7.655%. Fig. 7 and Fig. 8 give the corresponding program maps. Non-white colors indicate abstracted instructions: gray is used for the regularly abstractable instructions, e.g. for the instructions of Fig. 3 and Fig. 4, black for instructions accessing the stack and light gray for the remaining instructions. Apparently, not being able to abstract stack accesses results in a 6.6 times lower net gain. Fig. 7 and Fig. 8 suggest that this huge reduction is due to black pixels, i.e. directly due to stack access instructions, but also due to light gray pixels, i.e. indirectly due to stack access instructions, which, when part of a fragment, can reduce the abstractable part below the size of the call instruction or influence the combinations of fragments for abstraction and leave some fragments unabstracted.

As mentioned in the previous paragraph, fragments lie within single basic blocks. The program map[2] in Fig. 9 shows how fragments fill out basic blocks. Abstracted fragments are

[1]Our current implementation is very conservative in the sense that every instruction accessing the stack- or frame pointer register is regarded as a stack access.

[2]The program map over instructions is enough as a program map over bytes doesn't give us in this case any additional information.

represented as gray pixels and basic block boundaries as black pixels. For this, we replace each jump and branch instruction with a black pixel and insert at each jump or branch target a black pixel. This distorts the program map somewhat. It can be seen that 36.949% abstracted fragments are whole basic blocks while 63.051% are not.

To reduce the cost of finding fragments, Debray et al. (2000) limit the search for fragments in their compactor to whole basic blocks, only. We learned from our visualization that this would drastically reduce the efficiency of *our* compactor.

4 Visualization of Procedural Abstraction Variants

A variant of procedural abstraction, hereafter called *tail merging procedural abstraction*, merges tails of fragments as shown for Fig. 2a in Fig. 2b. Fragments of different sizes are replaced by procedure calls *into* the procedure.

```
a. load r1, $5200    load r1, $5200    load r1, $5300
   add r1, r2        add r1, r2        add r1, r2
   rot r1, $2        rot r1, $2        rot r1, $2
   mul r1, r1        mul r1, r1        mul r1, r1


b. call f1           call f1           load r1, $5300    f1: r1, $5200
                                       call f2           f2: add r1, r2
                                                             rot r1, $2
                                                             mul r1, r1
                                                             ret
```

Figure 2: Example of Tail Merging Procedural Abstraction

Earlier work on tail merging procedural abstraction by Liao et al. (1999) and Gyimóthy et al. (2005) did not provide any comparison with traditional procedural abstraction and it remained unclear whether there is a visible improvement for real programs. This lack of comparison data motivated us to write not only a post pass optimzer for traditional procedural abstraction, but also one for tail merging procedural abstraction. We have shown in (Schaeckeler and Shang, 2008) that traditional procedural abstraction and tail merging procedural abstraction could reduce the code size of seven MediaBench programs on average by 2.502% and 2.716%, respectively.

To understand from where the improvements were coming, we generated in Fig 10a a program map for traditional procedural abstraction and in Fig 10b a program map for tail merging procedural abstraction. As they are too similiar, it was necessary to generate in addition the *difference map* of Fig 10c. Gray pixels represent instructions that could be abstracted with both procedural abstractions. Black pixels represent instructions that could be abstracted with tail merging procedural abstraction only, and light gray pixels represent instructions that could be abstracted with traditional procedural abstraction only. The black pixels in Fig. 10c indicate the higher code size reduction of tail merging procedural abstraction.

As before, the program maps of Fig. 10a and 10b have been directly generated from our optimization passes from their internal data-structures. These program maps have been the input for a script to generate the difference map of Fig. 10c.

When we familiarized us with traditional and tail merging procedural abstraction, we expected fragments to extend, e.g. finding in Fig. 10c black sub-strings left-adjunct to gray sub-strings. We found twelve such extended fragments, but to our surprise, we found also 21 black strings in isolation, i.e. new fragments emerged and joined other fragments for abstraction. Visualization gave us a deeper understanding of tail merging procedural abstraction.

5 Future Work

We intend to write an interactive program map, e.g. a java applet which lets the user interactively explore abstractions in a program. It will be able to not only display the program maps discussed so far, but also allow to show references between abstractions of a procedure, e.g. clicking on a fragment will highlight all other fragments of the same procedure.

Interactively removing and re-adding fragments will show the current compression ratio and, if sufficient profiling information is available, show the estimated run-time of the program.

We hope that such an interactive map will not only remain a toy but that it will give us also playful insight into interactions between abstractions, run-time, and compression ratio.

6 Conclusion

We hope this paper may inspire other compiler writers to visualize optimization passes to help them to reason about, and to understand, the inner workings. Visualizations can be also used in compiler classes to make optimizations less abstract and to give students a better understanding of where and how often optimizations are applied in the code.

References

Keith D. Cooper and Nathaniel McIntosh. Enhanced code compression for embedded RISC processors. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 139–149, New York, NY, USA, 1999. ACM Press. ISBN 1-58113-094-5.

Saumya K. Debray, William Evans, Robert Muth, and Bjorn de Sutter. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.*, 22(2):378–415, 2000. ISSN 0164-0925.

Stephan Diehl. *Software Visualization.* Springer Berlin, 2007.

Christopher W. Fraser, Eugene W. Myers, and Alan L. Wendt. Analyzing and compressing assembly code. In *SIGPLAN '84: Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, pages 117–121, New York, NY, USA, 1984. ACM Press. ISBN 0-89791-139-3.

Tibor Gyimóthy, Rudolf Ferenc, Gábor Lehotai, Ákos Kiss, and Attila Bicsak. US patent nr. 7,293,264: Method and a device for abstracting instruction sequences with tail merging, 2005.

Jeremy Lau, Stefan Schoenmackers, Timothy Sherwood, and Brad Calder. Reducing code size with echo instructions. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 84–94, New York, NY, USA, 2003. ACM. ISBN 1-58113-676-5.

Stan Liao, Srinivas Devadas, and Kurt Keutzer. A text-compression-based method for code size minimization in embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, 4(1): 12–38, 1999. ISSN 1084-4309.

Stefan Schaeckeler and Weijia Shang. Code compaction with reverse prefix trees. In *CASES '08 Proceedings of the 2008 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, New York, NY, USA, 2008. ACM Press. Submitted, under review.

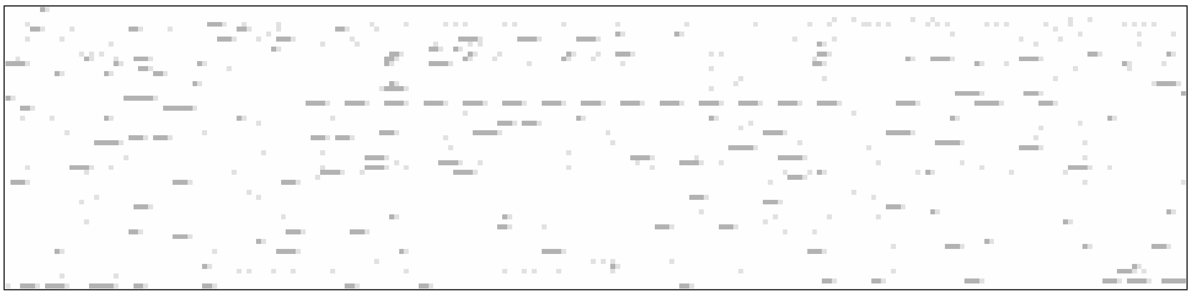


Figure 3: Visualization of Abstracted Fragments [instructions]: Fragments include a light gray End Marker.

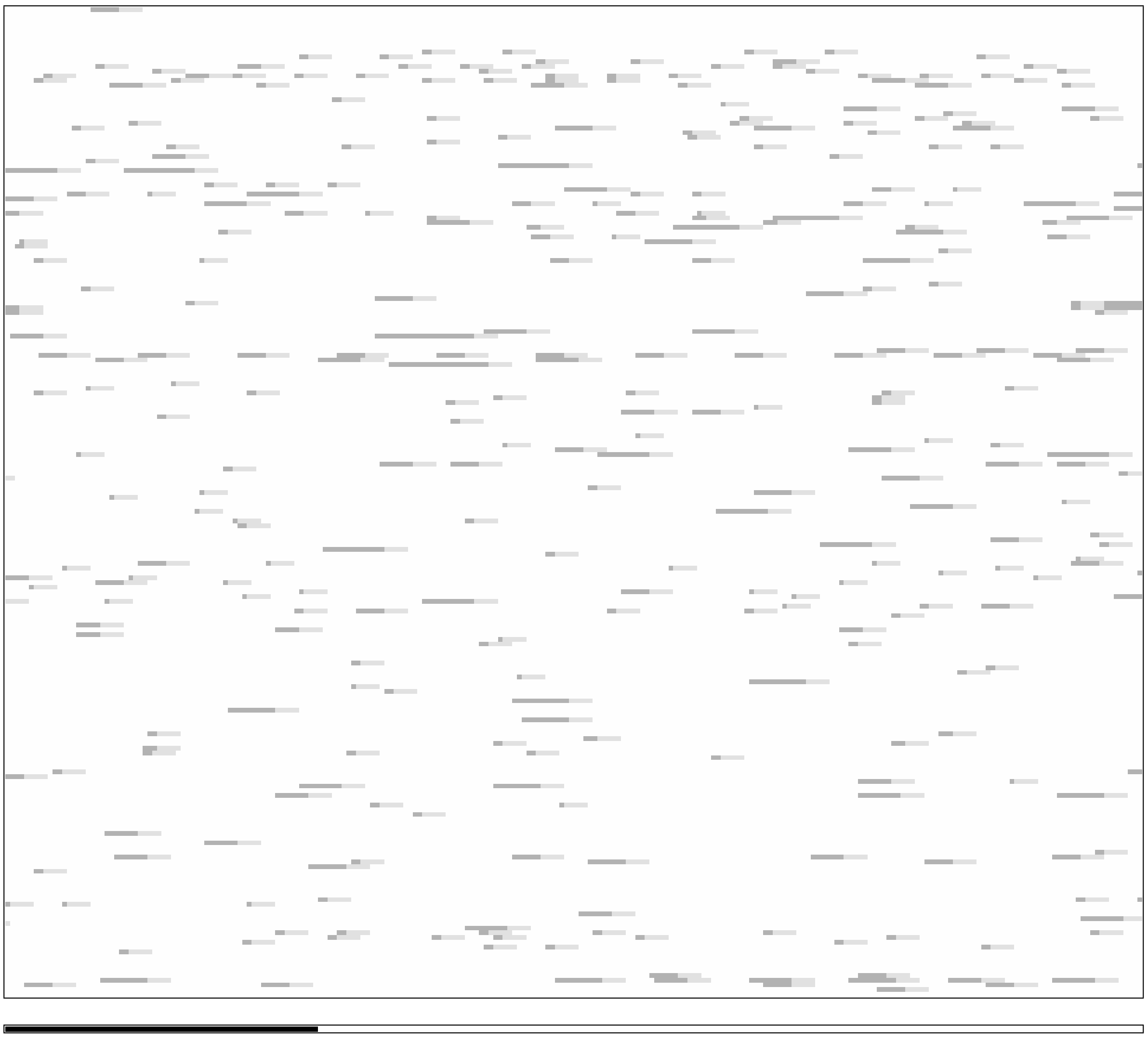


Figure 4: Visualization of Abstracted Fragments [bytes]: Fragments include a five Pixel long light gray End Marker of the Size of the Function Call Overhead.

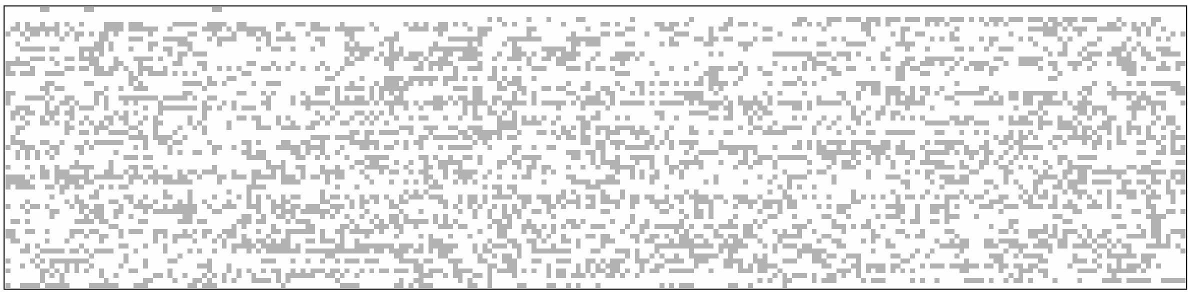


Figure 5: Visualization of Abstracted Fragments [instructions]: Fragments for no Function Call / Return Overhead.

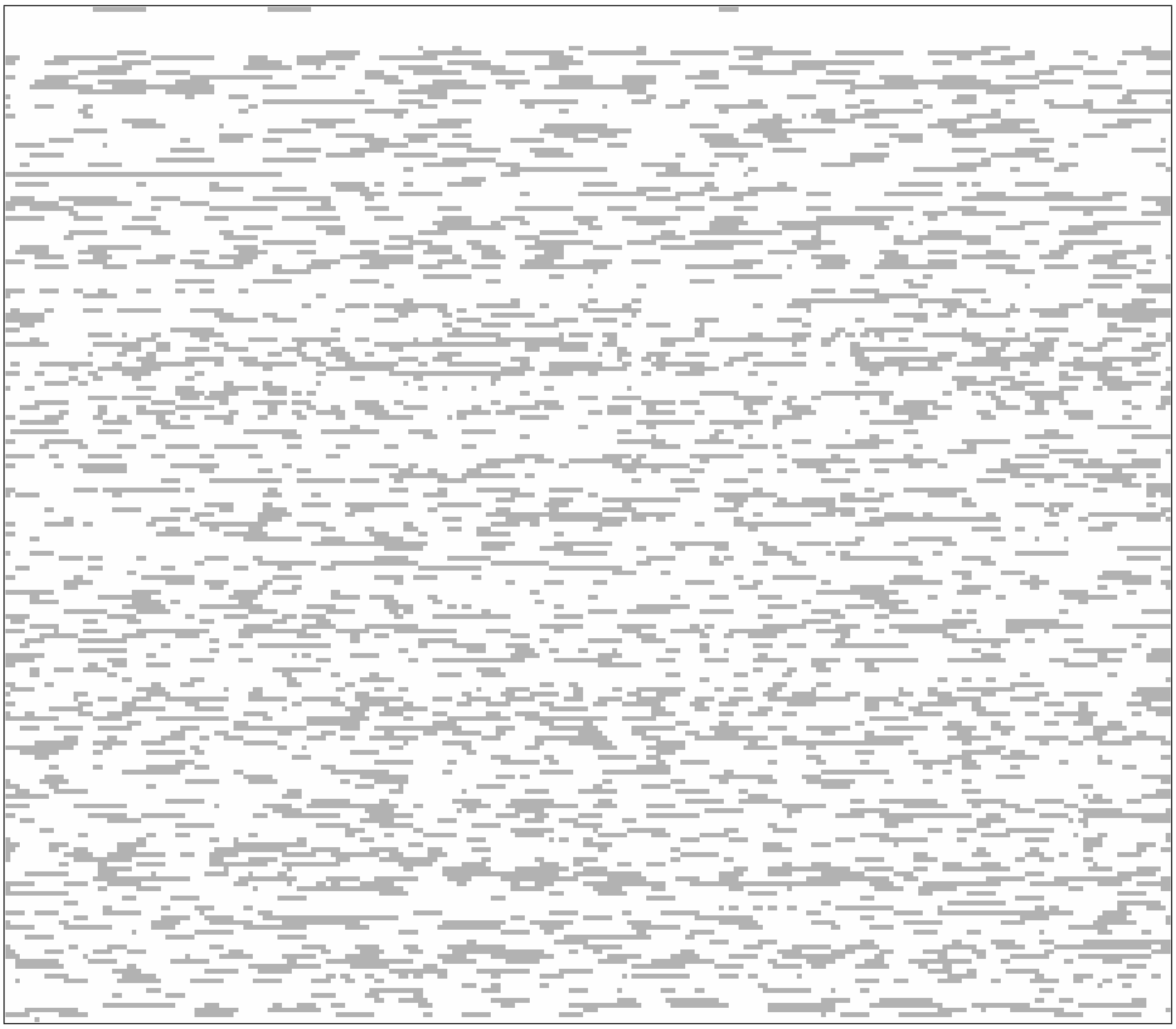


Figure 6: Visualization of Abstracted Fragments [bytes]: Fragments for no Function Call / Function Overhead.

Figure 7: Visualization of Abstracted Fragments [instructions]: Fragments including Function Calls and Stack Accesses in black.

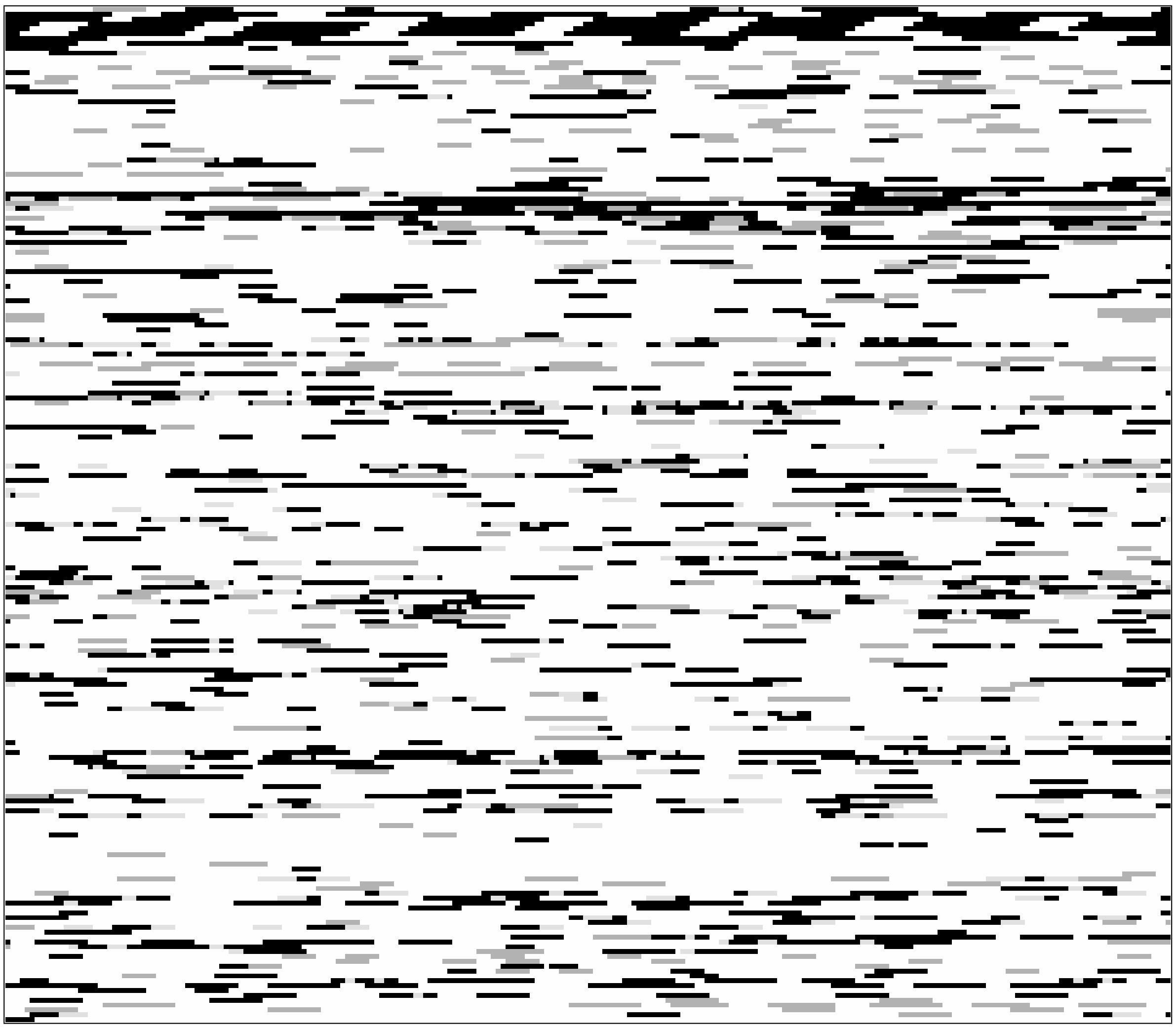


Figure 8: Visualization of Abstracted Fragments [bytes]: Fragments including Function Calls and Stack Accesses in black.

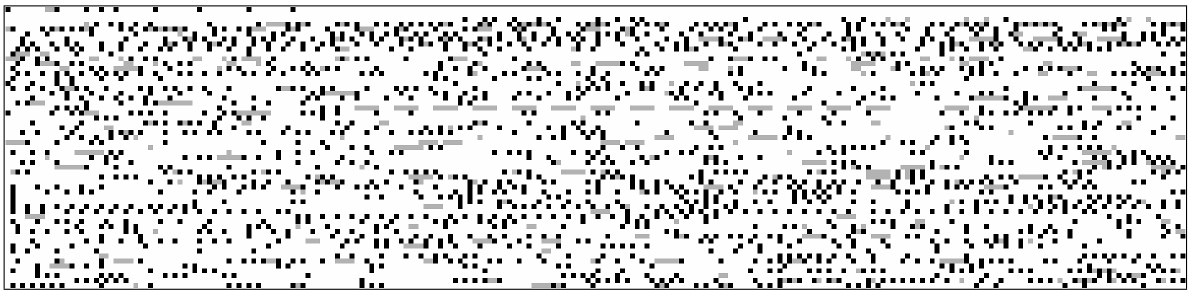
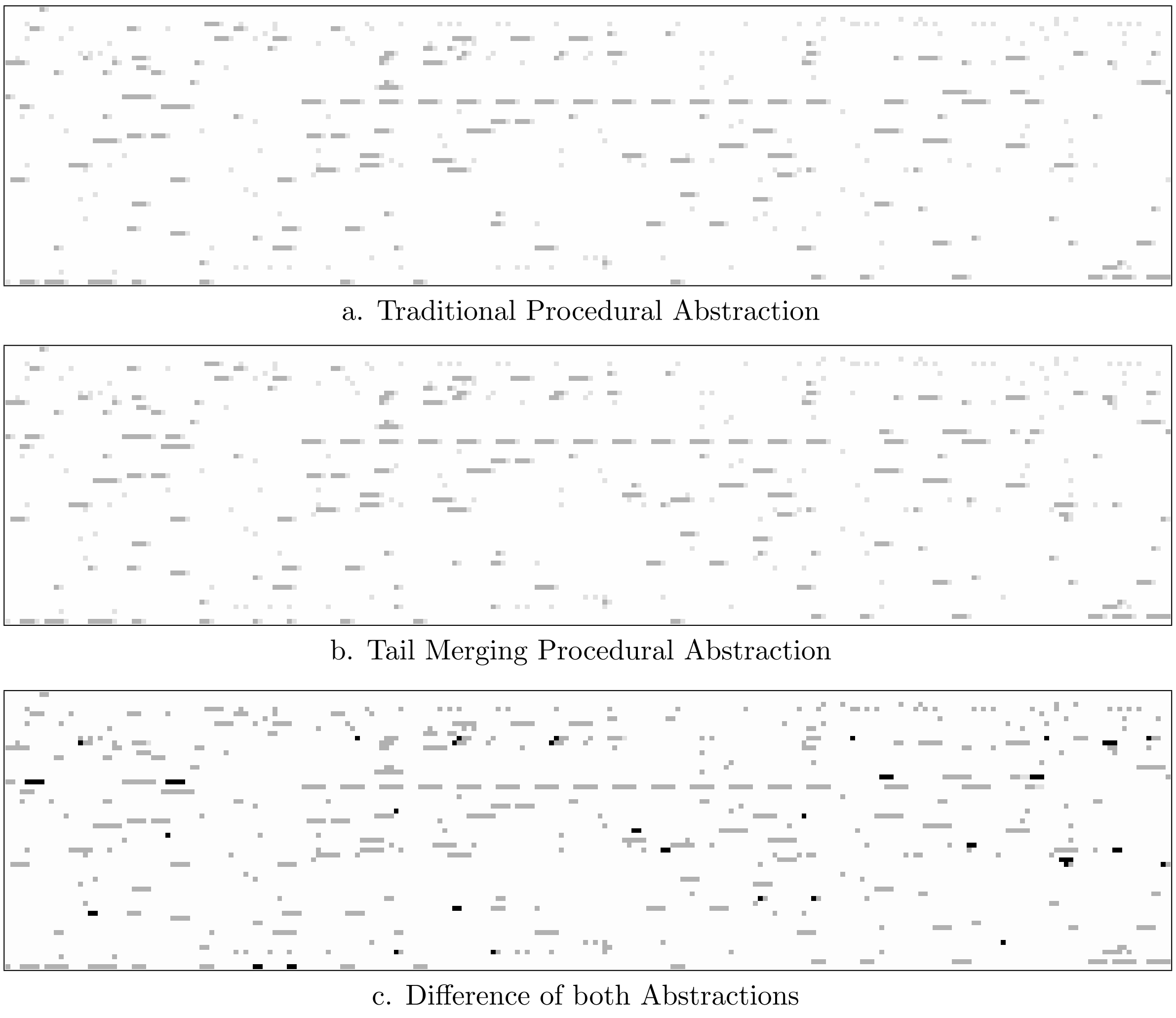


Figure 9: Visualization of Abstracted Fragments [instructions]: Fragments within Basic Blocks.



a. Traditional Procedural Abstraction

b. Tail Merging Procedural Abstraction

c. Difference of both Abstractions

Figure 10: Visualization of Traditional and Tail Merging Procedural Abstracted Fragments.

First Steps Towards a Visualization-Based Computer Science Hypertextbook as a Moodle Module

Guido Rößling, Teena Vellaramkalayil
CS Department, TU Darmstadt
Hochschulstr. 10
64289 Darmstadt, Germany

`roessling@acm.org`

Abstract

Hypertextbooks for Computer Science contents present an interesting approach to better support learners and integrate algorithm animations into the learning materials. We have developed a prototype for integrating a selection of the functionality of such a hypertextbook into the established Moodle LMS. This paper describes the goals and realization of this module together with an example.

1 Introduction

Algorithm visualization (AV) has a long tradition in visually presenting dynamic contents - typically, algorithms and data structures. The discipline and its associated tools promise easier learning and better motivation for learners. However, while surveys usually show interest in AV use, the adoption of AV by educators is lower than the proponents and developers of such systems would hope and expect. In the following, we will use the term AV whenever we refer to algorithm or program visualization or animation.

In a survey performed by the ITiCSE 2002 Working Group on ‘Improving the Educational Impact of Algorithm Visualization”, the main reasons why educators do not use AV materials in their lectures can be reduced to two aspects: the *time* required to do so and the *lack of integration* with existing teaching materials (Naps et al., 2003).

Since that report, several approaches have addressed the *time* aspect, for example by providing tools or generators for quickly producing content that fits the educator’s or learner’s expectations, and allow the user to specify the input values (Rößling and Ackermann, 2006; Naps, 2005). However, the integration of AV into the learning materials still needs to be addressed.

A 2006 ITiCSE Working Group therefore proposed a combination of hypertext-based textual materials with image, video, and AV content, as well as aspects from a learning management system (Rößling et al., 2006). This combination was called a *Visualization-based Computer Science Hypertextbook (VizCoSH)* to illustrate the main aspect form the Working Group’s point of view: the seamless integration of AV materials into the learning materials used for a course.

In this paper, we present our first approach of implementing a VizCoSH. As stated by previous authors of related hypertextbooks, the effort required to create a full-fledged hypertextbook is intense. The well-known theory hypertextbook *Snapshots of the Theory of Computing* (Ross, 2006; Boroni et al., 2002), while far from finished, already represents the work of about twelve years. Therefore, it seems unlikely that a full-fledged VizCoSH - including the features “borrowed” from course or learning management systems described in the Working Group Report (Rößling et al., 2006) - could already exist if it were built from scratch since 2006.

Our first prototype for a VizCoSH is based on the popular and established Moodle learning content management system (Cole and Foster, 2007). Section 2 presents the goals for the development of the modue. Section 3 describes the approach taken for meeting these goals, followed by a short demo of the resulting content pages in Moodle in Section 4. Section 5 presents a brief evaluation of the module and concludes the paper.

2 Goals for implementing a VizCoSH

The report that presented the concept of a VizCoSH set many ambitious goals for a full-fledged VizCoSH. For example, these concerned navigation, adaptation of the contents and learning paths to the user, and the integration of tracking and testing facilities to better determine the user's understanding. While most of the goals are already implemented in "some" systems, their combination - especially with the seamless integration of animations envisioned for a VizCoSH - has not been managed so far.

For the purpose of this research, we had to scale down the expectation towards a full VizCoSH to a manageable amount. Essentially, we expected that our VizCoSH prototype should offer the following features:

- Adaptation of layout (such as fonts and color settings) and language to the user's needs,
- Addition of arbitrary elements, such as text blocks, images, or hyperlinks at any position in the contents,
- Support for asking multiple-choice quizzes and performing knowledge tests,
- Support for different user roles, at least distinguishing between *teacher* and *student*,
- Logging the user's activities, in order to be able to track individual progress,
- Basic communication features, such as chats, forums and votings,
- Structured textual elements organized similarly to a text book (otherwise, the resource could not be called a hypertextbook),
- Enabling the printing of the learning materials with about the same comfort as for a "regular" text book,
- Seamless integration of AV content at (almost) any position in the contents,
- Support for fixed AV content as well as for "random" or user-generated AV content.

Many of these goals are already addressed by a variety of software. For example, AV systems such as Animal already provide the last two items in the list (Rößling and Ackermann, 2006). For most of the communication- and layout-based goals, there is a whole set of software that is geared to provide these aspects: learning content management systems including the popular *Moodle* system (Cole and Foster, 2007). We therefore decided to base our implementation on Moodle, which already offers the first six of the 10 required features.

3 Realizing a VizCoSH prototype as a Moodle module

Moodle is a highly extensible system, making it (comparatively) easy to provide additional features. The large international developer community can help in locating bugs and fixing them. However, the popularity of Moodle and the large number of developers also means that the number of offered modules or plugins for download is very large - currently, the web page lists more than 320 such elements.

The first six aspects - adaption of the visual layouts and language, management of arbitrary elements, quizzes and tests, user roles, logging, and communication features - are already integrated into Moodle and do not require further work. For the text structure similar to a book including useful printing facilities, we found a fitting "activity module" called *Book* (Škoda, 2007). This module provides the "significant structure" required by a VizCoSH, ensuring that the creation of meaningful text-based learning materials with AV content additions are possible.

The *Book* module allows printing the current "chapter" or the full book. Some limitations exist; for example, the author of the module has decided not to support sub-chapters or deeper levels of structure. Additionally, the module is not interactive, so that forums, chats etc. cannot be integrated into the content, but can be linked from anywhere in the page.

We have extended the Book module to include support for AV content and renamed it to *vizcosh*. Teachers can maintain a list of supported AV content files inside the module. New AV content can be added by providing the following information about the content: title, description, author (by default, the user currently logged in), and topic(s) covered. Additionally, the animation file has to be uploaded, optionally together with an image to be used as a thumbnail. Finally, the user has to select the animation format.

The *vizcosh* module currently supports the following formats:

- JAWAA (Akingbade et al., 2003),
- GAIGS (Naps and Rößling, 2006),
- JHAVÉ with a local file as a parameter (Naps, 2005),
- JHAVÉ with a specific input generator (Naps, 2005),
- the generators offered by ANIMAL, where the content author can either select the generator front-end, preselect an algorithm category, or specify a specific generator (Rößling and Ackermann, 2006),
- and the internal and ANIMALSCRIPT-based formats supported by ANIMAL (Rößling and Freisleben, 2002).

Each animation format description also contains a template for starting the content with an appropriate JNLP file. The JNLP file uses a set of placeholders to substitute the actual file name etc. when it is started inside Moodle, as shown in Table 1. Note that depending on the underlying system, not all of these placeholders may be used.

Variable	**Use**
`JNLP-PATH`	Describes the base path for all relative paths used in the JNLP file, as specified by the *codebase* attribute of the JNLP specification.
`JNLP-FILENAME`	Defines the name (relative to `JNLP-PATH`) for the JNLP file, used for the *href* attribute of the JNLP root element.
`JAR-PATH`	Defines the location of the JAR file(s) for the *jar* element of the *resources* JNLP element; describes where the JAR file(s) for the AV system are to be found.
`DATA-TYPE`	Describes the type of the file, needed if the chosen system can handle more than one type of file.
`DATA-PATHFILENAME`	Specifies the path and name for a file attribute to be passed in to the chosen AV system.

Table 1: Variables used for JNLP templates

An example JNLP template for the ANIMAL AV system is shown in Listing 1.

Listing 1: Example JNLP specification for ANIMAL

```
<?xml version="1.0" encoding="utf-8"?>
<!-- JNLP Specification for Animal 2.3.14 distribution -->
<jnlp spec="1.0+"
    codebase="<JNLP-PATH>"
```

```
5        href="<JNLP-FILENAME>">
6      <information>
7        <title>Animal Algorithm Animation, version 2.3.14</title>
8        <vendor>Animal Developer Team / Dr. Guido Roessling</vendor>
9        <homepage href="http://www.animal.ahrgr.de" />
10       <description>Animal Algorithm Animation, v. 2.3.14</description>
11       <description kind="short">An extensible algorithm animation tool
12         used for Computer Science Education purposes</description>
13       <icon href="Animal.gif"/>
14       <icon kind="splash" href="Animal.gif"/>
15       <offline-allowed/>
16     </information>
17     <security>
18       <all-permissions/>
19     </security>
20     <resources>
21       <j2se href="http://java.sun.com/products/autodl/j2se" version="1.5+"/>
22       <jar href="<JAR-PATH>/Animal-2.3.14.jar"/>
23     </resources>
24     <application-desc>
25       <argument><DATA-TYPE></argument>
26       <argument><DATA-PATHFILENAME></argument>
27     </application-desc>
28   </jnlp>
```

Lines 4 and 5 show that the *JNLP-PATH* and *JNLP-FILENAME* describe the location of the JNLP specification file. Lines 6 to 16 provide metadata about the AV system (here, the Animal AV system), such as the title, vendor, homepage, description and icon. In line 18, all permissions are requested to allow users to save animation files to their local disk.

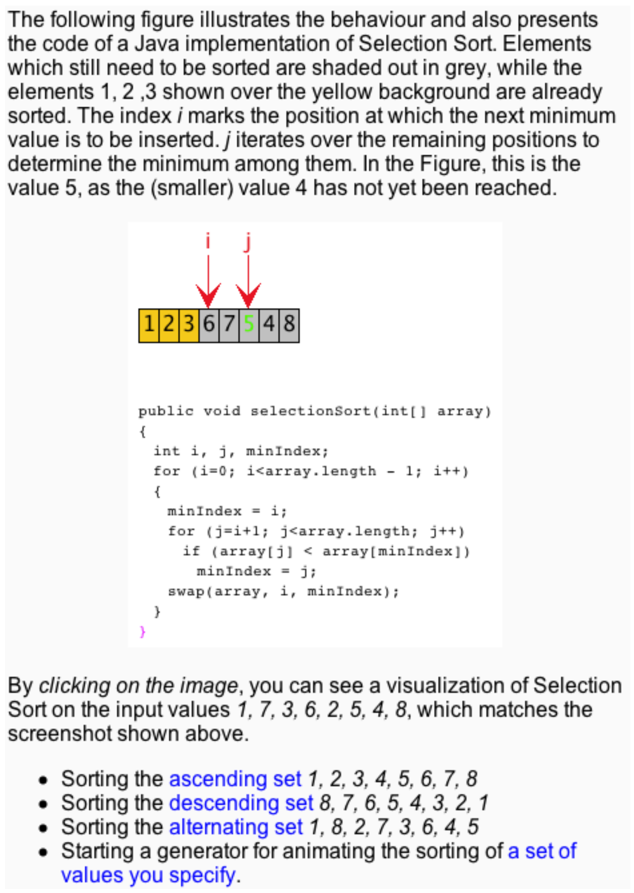
The JAR file for running the AV content is defined in line 22. Finally, lines 25 and 26 describe the run-time parameters for an animation, here the format and name of the animation file to be loaded.

Using the “Add Algorithm Visualization” button next to the standard editor, the user can easily add AV content at the end of the text. To do so, he or she simply chooses one of the existing elements from the AV content list, or creates a new entry. If the link appears in the wrong place, it can easily be cut and pasted to the right target position.

4 Example Output

Figure 1 shows an excerpt of a VizCoSH page created using the Moodle module. On this page, the image in the center is a link to the concrete visualization of the underlying sorting algorithm (here, Selection Sort). When the user clicks on this image, the AV system is started and shows the content indicated by the thumbnail. Additionally, a set of links to different alternative AV content are placed in the item list below the image, including the possibility for the user to adapt the content to his own preferences.

To keep the Figure readable, we present only a segment without the navigation elements placed above, below, and to the left of the page contents. The page also contains a paragraph describing the algorithm (Selection Sort) above Figure 1, as well as a paragraph about the complexity of the algorithm. Additionally, a number of exercises are also put on the page. In a future version of the VizCoSH, the user shall also be able to submit a solution to these exercise tasks for (semi-)automatic grading. However, this part of the module does not exist yet. We mention it to illustrate why a VizCoSH can be so much more than a “simple” text book. A “complete” VizCoSH can incorporate automatically evaluated tests that may also have a effect on how further content is presented to the user - or even which content will be visible.



The following figure illustrates the behaviour and also presents the code of a Java implementation of Selection Sort. Elements which still need to be sorted are shaded out in grey, while the elements 1, 2 ,3 shown over the yellow background are already sorted. The index *i* marks the position at which the next minimum value is to be inserted. *j* iterates over the remaining positions to determine the minimum among them. In the Figure, this is the value 5, as the (smaller) value 4 has not yet been reached.

```
public void selectionSort(int[] array)
{
  int i, j, minIndex;
  for (i=0; i<array.length - 1; i++)
  {
    minIndex = i;
    for (j=i+1; j<array.length; j++)
      if (array[j] < array[minIndex])
        minIndex = j;
    swap(array, i, minIndex);
  }
}
```

By *clicking on the image*, you can see a visualization of Selection Sort on the input values *1, 7, 3, 6, 2, 5, 4, 8*, which matches the screenshot shown above.

- Sorting the ascending set *1, 2, 3, 4, 5, 6, 7, 8*
- Sorting the descending set *8, 7, 6, 5, 4, 3, 2, 1*
- Sorting the alternating set *1, 8, 2, 7, 3, 6, 4, 5*
- Starting a generator for animating the sorting of a set of values you specify.

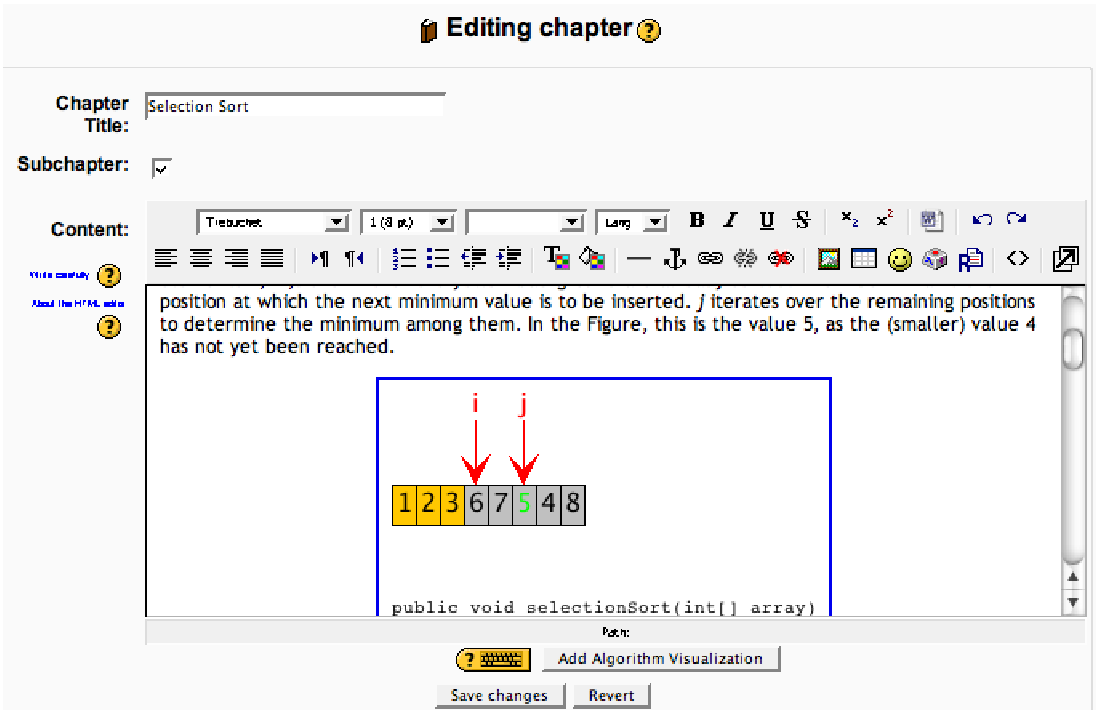
Figure 1: VizCoSH example from the *vizcosh* Moodle module

Figure 2 shows an example of the (modified) content editor provided by the *vizcosh* module. The WYSIWYG editor is already provided by Moodle. Our addition to this editor is the button *Add Algorithm Visualization* shown next to the keyboard icon. When the user clicks on this button, he is led to the collection of all registered animations. By selecting one of these entries, the associated thumbnail or text is inserted into the text editor - the AV content is now ready to be run as soon as the changes have been saved.

If the user wants to add an algorithm visualization to the VizCoSH, he clicks on a “+” icon above the list of known AV content. He is then led to the page shown in Figure 3. Here, the user can enter the title and description of the AV content. The “Author” field is automatically set to the name of the user currently logged in to Moodle. The “Format” list displays a set of predefined formats. Each format has a proper JNLP file that is automatically adapted to run the new animation, based on the animation file uploaded by the user and the JNLP features. Finally, the user can decide to use the default thumbnail , download a new one, or instead display only a text for the hyperlink. By pressing the “Create” button, the file is uploaded and placed into the proper directory, and a new JNLP file will be created that fits the AV content.

5 Evaluation and Future Work

Using the *vizcosh* Moodle module presented in this paper, it is easy to incorporate animations of any of the supported types listed in Section 3. The average time effort for adding a new


Editing chapter

Chapter Title: Selection Sort

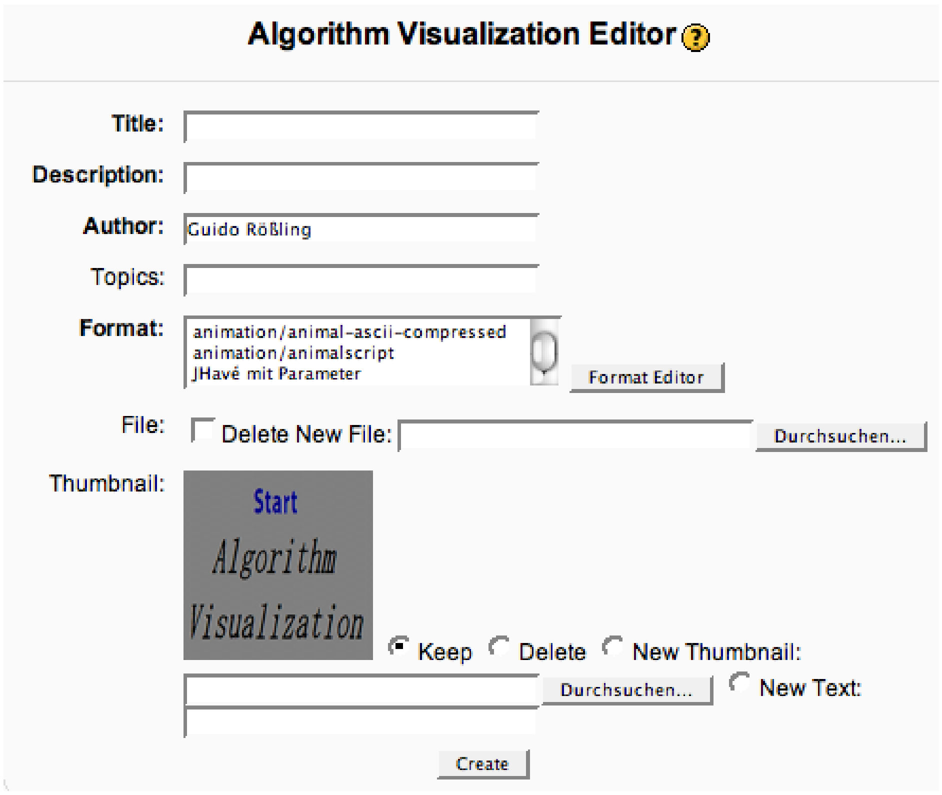
Subchapter: ✓

Content:

position at which the next minimum value is to be inserted. *j* iterates over the remaining positions to determine the minimum among them. In the Figure, this is the value 5, as the (smaller) value 4 has not yet been reached.

```
public void selectionSort(int[] array)
```

Figure 2: VizCoSH example from the *vizcosh* Moodle module


Algorithm Visualization Editor

Title:	
Description:	
Author:	Guido Rößling
Topics:	
Format:	animation/animal-ascii-compressed animation/animalscript JHavé mit Parameter
File:	Delete New File:
Thumbnail:	[✓ Keep] Delete New Thumbnail: New Text:

Figure 3: VizCoSH example from the *vizcosh* Moodle module

visualization to a given module page is less than two minutes, as it only requires creating a new animation entry and selecting the proper animation format.

At the moment, our module only supports contents that are based on a scripting notation: JAWAA, GAIGS, JHAVÉ and ANIMAL. This does not mean that the approach is in any way restricted to scripting input - it would be just as easy to support animation systems that use a stored file of some type. However, as we are most familiar with the listed systems, there is currently no example for the support of other systems. There are real "technical" reasons for this lack of support. The main task to be done is to provide a working JNLP template for a new system, similar to the one shown in Listing 1. The author can create a new format template with a few mouse clicks by opening the "Format Editor" using the button next to the Format list shown in Figure 3. However, the basic JNLP file has to be edited manually to adapt it to the target format. This especially concerns the *application-desc* element of the JNLP specification, which states the main class of the JAR file(s) and the invocation arguments. The module itself does not have to be changed if a new format is introduced.

The ability to print the current chapter including the thumbnails is also helpful. Here, the underlying *Book* module simply renders the page content(s) without the navigation elements and redisplays them as a Web page in a new browser window. Of course, the dynamic visualization elements are reduced to static images in this approach.

Our prototype can only represent the first step towards implementing a VizCoSH. Still, we expect that it will be easy to use for others once we publish the module, and may make adoption of AV easier. Future work for the module includes a seamless incorporation of interactive features, such as Moodle's forums and chat abilities with the page content and the visualizations. For example, users should be able to link to a given visualization easily in all other areas of Moodle. User tracking for performance in built-in knowledge tests would be another important addition.

We are interested in cooperating with researchers and teachers who want to use the module. This especially concerns the support for other algorithm animation or visualization systems that can be run using Java Webstart. Several persons may not use Moodle, but some other platform, for example due to a university-enforced policy. Most of the module is not specific to Moodle and should be easy to carry over to other learning content management systems or content management systems, such as *Drupal, Plone* or *Typo3.* The main Moodle-specific aspect is the connection to the user management and database, which has to rewritten for each target platform.

We are also looking for partners who could contribute content, and allow us to take one step further towards the vision of a "community-shared" VizCoSH, as anticipated by the ITiCSE 2006 Working Group (Rößling et al., 2006).

References

Ayonike Akingbade, Thomas Finley, Diana Jackson, Pretesh Patel, and Susan H. Rodger. JAWAA: Easy Web-Based Animation from CS 0 to Advanced CS Courses. In *Proceedings of the* 34th *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2003), Reno, Nevada*, pages 162–166. ACM Press, New York, 2003.

Christopher Boroni, Frances Goosey, Michael Grinder, and Rockford Ross. Active Learning Hypertextbooks for the Web. *Journal of Visual Languages and Computing*, 13(2):341–354, 2002.

Jason Cole and Helen Foster. *Using Moodle: Teaching with the Popular Open Source Course Management System.* O'Reilly, 2007. ISBN 978-0596529185.

Thomas Naps. JHAVÉ – Addressing the Need to Support Algorithm Visualization with Tools for Active Engagement. *IEEE Computer Graphics and Applications*, 25(6):49–55, December 2005.

Thomas L. Naps and Guido Rößling. JHAVÉ - more Visualizers (and Visualizations) Needed. In Guido Rößling, editor, *Proceedings of the Fourth Program Visualization Workshop, Florence, Italy*, pages 112–117, June 2006.

Thomas L. Naps, Guido Rößling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger, and J. Ángel Velázquez-Iturbide. Exploring the Role of Visualization and Engagement in Computer Science Education. *ACM SIGCSE Bulletin*, 35(2):131–152, June 2003.

Rockford J. Ross. Snapshots of the theory of computing. Available online at `http://www.cs.montana.edu/webworks/projects/snapshots/`, 2006.

Guido Rößling and Tobias Ackermann. A Framework for Generating AV Content on-the-fly. In Guido Rößling, editor, *Proceedings of the Fourth Program Visualization Workshop, Florence, Italy*, pages 106–111, June 2006.

Guido Rößling and Bernd Freisleben. ANIMAL: A System for Supporting Multiple Roles in Algorithm Animation. *Journal of Visual Languages and Computing*, 13(2):341–354, 2002.

Guido Rößling, Thomas Naps, Mark S. Hall, Ville Karavirta, Andreas Kerren, Charles Leska, Andrés Moreno, Rainer Oechsle, Susan H. Rodger, Jaime Urquiza-Fuentes, and J. Ángel Velázquez-Iturbide. Merging Interactive Visualizations with Hypertextbooks and Course Management. *SIGCSE Bulletin inroads*, 38(4):166–181, December 2006.

Petr Škoda. *book* module for Moodle. `http://docs.moodle.org/en/Book_module`, 2007.

Towards Seamless Merging of Hypertext and Algorithm Animation

Ville Karavirta
Helsinki University of Technology
Department of Computer Science and Engineering

vkaravir@cs.hut.fi

Abstract

The integration of algorithm animations into hypertext is seen as an important topic today. This paper will present a prototype algorithm animation viewer implemented purely using HTML and JavaScript. The viewer is capable of viewing animations in Xaal (eXtensible Algorithm Animation Language). This solution is extremely suitable to be used in hypertext learning material.

1 Introduction

Online learning material that students use by themselves is one of the typical usages of algorithm animation (AA). Thus, the integration of algorithm animations into hypertext is seen as an important topic today. The hope is that by making it easier to use AA in hypertext, algorithm animations will become more widely adopted in teaching. This could then solve the problem AA has been fighting for years: teachers believe that AA is beneficial but they do not use it in their teaching (Naps et al., 2003b). In 2006, a working group titled *Merging interactive visualizations with hypertextbooks and course management* convened at ITiCSE 2006 to consider how algorithm visualizations could and should be merged with hypertext. In the working group report, the features seen important were included seamless visualization integration, increasing student engagement, providing a richer learning environment, integration of CMS features, and aesthetics. (Rößling et al., 2006)

The technologies for building interactive web applications have evolved fast in the past few years. This has made it possible to implement complex applications online. As a result, we have seen a surge of many typical desktop applications being implemented as rich internet applications (or, RIAs). These applications include mail clients, office software, and even photo-editing software. The main benefits of online applications are the ease of which they can be taken into use and the ease of maintenance from the developers perspective.

When the goal is to seamlessly merge algorithm visualizations with hypertext, a natural future direction for algorithm animation systems is to implement them as RIAs. This paper introduces one such solution. We will present a prototype algorithm animation viewer implemented purely using HTML and JavaScript. The viewer is capable of viewing animations in Xaal (eXtensible Algorithm Animation Language), a language designed to allow easy transformation of AAs between various formats (Karavirta, 2007). This solution is extremely suitable to be used in hypertext learning material. In the end, our goal is to create an interactive system integrated with hypertext that supports several current algorithm animation description languages.

The paper is organized as follows. First, Section 2 introduces related research. Section 3 discusses the requirements for algorithm animation systems. Section 4 in turn describes the implementation of the new AA viewer. Finally, Section 5 discusses the suitability of this new approach and Section 6 provides conclusions and some possible future directions.

2 Related Work

Algorithm animation has been an active area of research. From the perspective of this paper, the most important research directions are merging AA and hypertext and developing a common, XML-based algorithm animation language.

Early work on algorithm visualization in hypertextbooks has been done by Ross and Grinder (2002). In Ross's hypertextbooks, the inclusion of visualizations is done using Java applets. This is currently a common way used also in, for example, TRAKLA2 (Malmi et al., 2004), WinHIPE (Pareja-Flores et al., 2007), and a number of topic-specific visualizations. Another popular method at the moment is Java WebStart where users launch Java applications from the web. This approach is used, for example, in Animal (Rößling and Freisleben, 2002), Jeliot (Moreno et al., 2004), and JHAVÉ (Naps, 2005).

In ITiCSE 2006, a working group considered how algorithm visualizations should be merged with hypertext. The group considered visualization based hypertextbooks an important factor in promoting algorithm animation adoption in teaching. In the working group report, the features of such hypertextbooks the working group considered important were seamless visualization integration, increasing student engagement, providing a richer learning environment, integration of CMS features, and aesthetics. (Rößling et al., 2006)

Another related research topic is the integration of algorithm animation systems. One possible way to achieve this integration has been taken in the JHAVÉ algorithm visualization environment. JHAVÉ allows AA system developers to add their systems as visualization plugins into JHAVÉ. This way a common user interface for the end-user (typically, a student) is achieved. Another approach to integration, focusing on the teacher's point of view, is developing a common format for the algorithm animation systems. In AV scope, this problem has been discussed in another ITiCSE working group. The group's report gives examples of a common AA language and suggestions on an architecture to implement it (Naps et al., 2005).

One implementation of the ideas of the group is Xaal (eXtensible Algorithm Animation Language) (Karavirta, 2007). Xaal supports describing animations on different levels of abstraction: using graphical primitives and transformations on them, or using data structures and operations on them. The goal of Xaal and the tools supporting it has been to allow easy transformation of AAs between various formats/systems. The import and export features of visualization systems is a significant research problem even in the wider scope of Software Visualization (Diehl, 2007).

From the purely technical point of view, several rich internet application (or, RIA) technologies have been introduced lately. These technologies allow creating complex applications that run in web browsers. In this work, we will focus on JavaScript. However, we will introduce some alternatives in Section 4. On the field of JavaScript, a multitude of libraries aiding in web development have been developed, and new ones are popping up constantly. Some of the most well-known libraries include Dojo, mootools, Prototype, Scriptaculous, jQuery, and Google Web Toolkit, just to mention a few.

When discussing algorithm animation in the context of education [1], one cannot ignore the importance of user engagement. It has been shown that when users interact with algorithm animations, it has a positive impact on their learning (Hundhausen et al., 2002). The levels of engagement were specified by Naps et al. (2003a) by introducing a taxonomy of engagement. The levels are *viewing*, *responding*, *changing*, *constructing*, and *presenting*. *Viewing* is passive watching of an animation where a student only controls the visualization execution. In *responding*, the student is engaged by asking questions about the visualization. *Changing* requires the student to modify the visualization, for example, by changing the input data. In *constructing*, the student is required to construct his/her own algorithm animation. At the highest level, *presenting*, the student presents a visualization for an audience.

3 Requirements for an Algorithm Animation System

Over the years, a lot of research on the requirements of algorithm animation systems has been carried out. Rößling and Naps (2002b) introduced pedagogical requirements for algorithm

[1]In fact, the most common application area for algorithm animation lies in context of education (Diehl, 2007).

visualizations (AV). In another article, Rößling and Naps (2002a) provide more guidelines for AV systems. The following summarizes the requirements of these two articles. We have numbered the requirements to help referring to them later in the article.

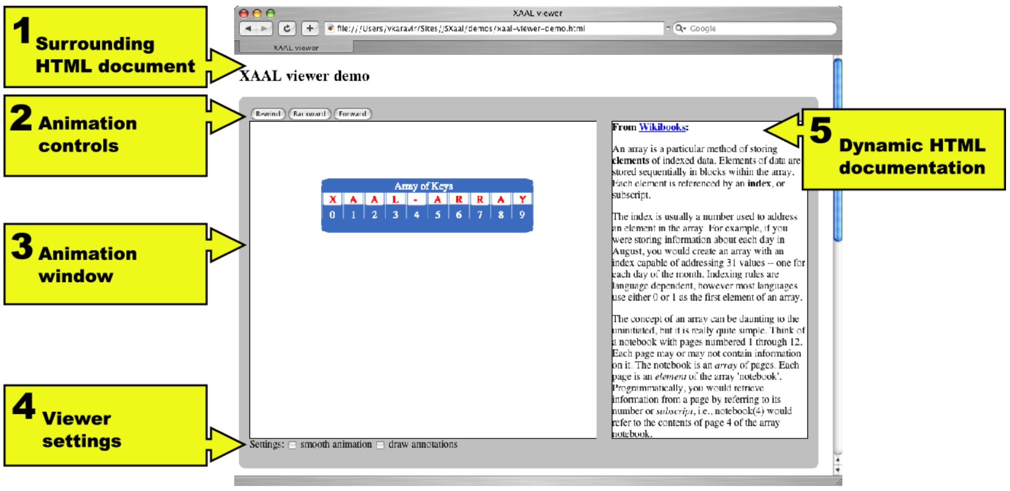
- The system's platform should be chosen to allow the widest possible target audience. [R1]
- The system should support visualization rewinding so that users can return to the place where they lost track of the content. [R2]
- Learners should be able to adapt the display to their current environment. This includes the choice of display background color to account for diverse lighting situations, transition speed and display magnification. [R3]
- The AA system should preferably be a general-purpose system instead of a topic-specific system due to the chance for reuse and better integration into a given course. The main benefit of general-purpose systems is the ability to offer a common interface to a large number of animations. [R4]
- The user should be offered the choice between smooth visualization transitions and discrete steps. A break between consecutive steps, or at least a pause button, should also be provided. [R5]
- The visualizations should include documentation that accompany the visualization. There are several types of documentation: static documentation, dynamic documentation aware of the algorithm's state, and pedagogically dynamic documentation that is aware of the algorithm's state as well as the expertise of the student. [R6]
- Asking questions about the algorithms behavior in following states should be supported. The questions should incorporate feedback. [R7]
- The system should be integrated with a database for course management facilities. The database can then be used for example to store the points received by answering questions. [R8]
- The system should allow users to provide custom input to the algorithm. [R9]
- The visualization should offer a structural view of the algorithm's main parts that can also be used to jump to associated visualization steps. [R10]

The articles also include two more requirements. One system requirement is that for the visualization author, the system should make it possible to group questions based on the question topic area and to assign a certain number of points to each question and inform the learners on their progress. Another requirement is to include reusable visualization modules, thus aiding in the authoring of AV. Since both of these are requirements for the author, we do not consider it relevant in this case when building an algorithm animation viewer.

The requirements above in a way include engagement levels viewing (R2), responding (R7), and changing (R9). However, we feel that AA system requirements should include support even for the level constructing (we will add this as requirement R11).

4 Implementation

Based on the requirements for an algorithm animation viewer, we implemented such a system using only HTML and JavaScript. As the whole viewer will be running inside the user's browser without any additional plugins, every computer equipped with a modern browser can use the viewer (requirement R1). The implemented prototype can be easily incorporated



From Wikibooks:

An array is a particular method of storing **elements** of indexed data. Elements of data are stored sequentially in blocks within the array. Each element is referenced by an **index**, or subscript.

The index is usually a number used to address an element in the array. For example, if you were storing information about each day in August, you would create an array with an index capable of addressing 31 values -- one for each day of the month. Indexing rules are language dependent, however most languages use either 0 or 1 as the first element of an array.

The concept of an array can be daunting to the uninitiated, but it is really quite simple. Think of a notebook with pages numbered 1 through 12. Each page may or may not contain information on it. The notebook is an *array* of pages. Each page is an *element* of the array 'notebook'. Programmatically, you would retrieve information from a page by referring to its number or *subscript*, i.e., notebook(4) would refer to the contents of page 4 of the array notebook.

Figure 1: XAAL viewer in browser showing an animation and related documentation.

into web material using XAAL animations as the source data. The following describes the implemented features and, for the interested reader, the technologies used.

Figure 1 shows the animation viewer in the Safari browser. In the figure, arrow 1 points to the surrounding HTML document. This document can contain any HTML. Arrow 2 points to the animation controls. Here, we have the controls to rewind and move backwards and forwards in the animation (fulfilling requirement R2). Arrow 3 points to the actual animation window where the contents of the animation are visualized. Arrow 4, in turn, indicates the settings panel for the animation viewer. Finally, arrow 5 points to the HTML documentation that is included in the XAAL document and shown next to the visualization. The appearance of the viewer can be easily changed using a different CSS stylesheet. Through CSS, one can also change the positions and sizes of the various parts of the viewer (partially fulfilling R3). To completely fulfill the requirement, features like speed and magnification changing should also be supported.

Listing 1 gives an example on how to add a XAAL animation into an HTML document. As can be seen, it is quite simple and requires only a few lines of code. Multiple animations can be embedded on the same page by repeating lines 1 and 5 of Listing 1 with different data.

```
1 <div id="animation" class="jsxaal"></div>
2 ...
3 <script type="text/javascript">
4   Event.observe(window, 'load', function() {
5     new JSXaalViewer("slides.xml", "animation", { showNarrative: true });
6   });
7 </script>
```

Listing 1: Example of including a XAAL animation in hypertext.

In addition to the code in Listing 1, the JavaScript files for the viewer need to be loaded, which adds another couple of lines. Listing 2 shows these requirements. The first five lines of these are libraries used by the viewer (see Section 4.1 for details about the libraries), while the last line is the actual XAAL viewer.

```
1 <script src="lib/prototype/prototype.js" type="text/javascript"></script>
2 <script src="lib/pgf/pgf-core-min.js" type="text/javascript"></script>
3 <script src="lib/pgf/pgf-renderer-min.js" type="text/javascript"></script>
4 <script src="lib/scriptaculous/scriptaculous.js" type="text/javascript"></script>
5 <script src="lib/scriptaculous/effects.js" type="text/javascript"></script>
6 <script src="dist/jsxaal-core-min.js" type="text/javascript"></script>
```

Listing 2: Example of loading the libraries required to add a XAAL animation in hypertext.

The graphical primitives of XAAL make it possible to use the viewer not only for animation of data structures and algorithms, but everything that can be visualized using graphical primitives. Thus, requirement R4 is met.

As mentioned in the previous section, smooth animation should be optional for the end-user. Thus, our viewer includes the option for smooth animation to be toggled on or off. This can be done using one of the viewer settings (arrow 4 in Figure 1). This was requirement R5.

Since the whole animation viewer is based on HTML, integration with hypertext is simple and natural. Static documentation can be provided outside the viewer. As Listing 1 showed, including XAAL animations in hypertext requires only a couple of lines of HTML and JavaScript. There is also another method of including hypertext when using XAAL animations; each step in the animation is allowed to include a description that can be arbitrary XHTML. Documentation added this way is shown in area labeled 5 in Figure 1. This documentation is dynamic in nature, as it is different for every state of the animation. The only unsupported type of documentation mentioned in the requirements (R6) is dynamic documentation based on user's experience.

Requiring users to respond to questions during the animation was another requirement for an AA viewer (R7). Currently, the viewer supports some typical question types such as true/false questions and multiple choice questions. Since XAAL has no way to specify questions, we have implemented support for the question specification of the GaigsXML algorithm animation language (Naps et al., 2006). For the final version of this paper we will implement questions where the user can provide the answer by selecting graphical objects in the visualization. To connect to a database to submit the students' responses, AJAX (or, Asynchronous JavaScript And XML) calls can be made. Thus, requirement R8 is easy to implement on the client-side. However, it requires some interface on the server side which we haven't considered.

Allowing users to specify their own input was requirement R9. Again, since we are working with HTML and JavaScript, allowing users to specify their own input data for the viewer is extremely simple in cases where the animation is using data structures. This is because any scripts included in the HTML document can interact with the animation viewer. For example, Listing 3 gives an example how to add a textfield to an HTML page that adds the user-given data into the binary search tree in the animation.

```
<input id="insValue" type="text"></input>
  <button id="insButton">Insert</button>
...
<script type="text/javascript">
  Event.observe($('insButton'), 'click', function() {
    var bst = viewer.dsStore.get("bst");
    bst.insert($('insValue').value);
    bst.draw(viewer.renderer);
</script>
```

Listing 3: Example of allowing user input for algorithms.

4.1 Underlying Technologies

The implementation of the viewer is based on some JavaScript libraries. The lowest level of these libraries is Prototype[2], which offers, for example, Ajax support as well as advanced features for dynamically manipulating the client-side HTML. The visualizations are drawn using Prototype Graphic Framework (PGF)[3], a Prototype-based framework that allows drawing arbitrary data on various browsers. PGF supports multiple rendering technologies for different browsers: Scalable Vector Graphics (SVG), HTML Canvas element, and Vector Markup Language (VML). These different renderers can be used through one programming interface. The

[2]http://www.prototypejs.org
[3]http://prototype-graphic.xilinus.com/

animation features in our viewer use Scriptaculous[4], a Prototype-based animation framework. The animation is achieved by extending Scriptaculous's effects to modify graphical objects drawn using PGF.

When discussing web applications, the size of the files and the loading time are essential. Table 4.1 shows the load times and file sizes of the different components used to implement the XAAL viewer. The total size is slightly over 400 kilobytes. This size can be reduced by minimizing and compressing the files.

Table 1: Load times and file sizes of the different components needed for the viewer.

Component	**Load time (in ms)**	**Size (in kb)**
Prototype	107	124.1
Scriptaculous	177	124.7
Prototype Graphic Framework	72	88.7
XAAL viewer	48	64.7
Total	404	402.2

5 Discussion

The viewer supports the XAAL specification only partially. At the moment of writing, all the graphical primitives are supported. Of the data structures, only tree is currently implemented. However, for the final version of this paper, we hope to have a more complete implementation of the XAAL specification.

Some of the requirements are not implemented at this point. The support for constructing level of engagement should be added (R11). This could be done by, for example, adding visual algorithm simulation capabilities like in TRAKLA2 (Malmi et al., 2004) to the viewer.

Some user interface components should be added as well. These include the options to change the speed and magnification of the visualization (R3) as well as visualizing the algorithm's structure (R10). All these could be implemented with reasonable effort.

Connecting to a database to course management facilities (R8) can be done using AJAX calls. This, however, requires server-side support as well. An interesting new technology that could make implementing this quite simple, is Aptana Jaxer[5]. Jaxer is an AJAX server that allows running the same JavaScript code and manipulating the same DOM both on server-side and client-side.

There are still some problems unsolved. First, platform specific problems do arise, although the JavaScript libraries make writing browser independent code a lot easier. Currently, the implementation has been used in Firefox, Safari, Opera, and SeaMonkey but does not work in Internet Explorer. However, there should not be any major impediments in fixing this in IE. In addition, using JavaScript libraries other than Prototype in the HTML document can cause problems. However, this is not usual in current learning environments. Another problem is that due to the nature of JavaScript, all the source code is available to the student. Thus, any client-side assessment results cannot be trusted if such a system is to be used in evaluating students.

5.1 Alternative RIA Technologies

When building rich internet applications, JavaScript is not the only choice. In fact, there is an increasing number of promising technologies available. The discussion of all of these is not possible in the scope of this paper. However, the following mentions some of the most potential candidates.

[4]`http://script.aculo.us`

[5]http://www.aptana.com/jaxer

Adobe's Flash and Flex provide technology for building cross-platform RIAs. The tools for developing applications are quite sophisticated and powerful. However, the tools are commercial software products developed by Adobe. Another rising technology is Microsoft Silverlight, which uses a lot of the same technologies as the .NET framework making it suitable for developers familiar with .NET. However, Silverlight is not cross-platform compatible. Finally, we mention JavaFX, a family of products from Sun Microsystems based on Java technology. However, this technology is not ready for production use at the moment. On a positive side, Sun plans on releasing parts of the JavaFX family as open source.

So, why did we choose the JavaScript road? First, by using JavasScript we do not depend on software provided by any corporation but are using open source libraries. Second, JavaScript works on all platforms without any plugins, whereas, for example, Silverlight is not available on Linux at the time of writing. In addition, our approach can use any server side components. Finally and most importantly, for the JavaScript approach, the technology is mature, widely used, and supported by an ever-growing number of useful libraries.

6 Conclusions and Future Research

In this article, we have introduced a solution for using algorithm animations in hypertext online material. Our solution is a pure HTML and JavaScript implementation of an algorithm animation viewer that fulfills most of the requirements for an AV system. In the future, we hope to be able to use this system in actual material used by learners. At this point, we are not aware of any similar systems being implemented and we see this as an important step towards the seamless merging of AV and hypertext called for by the ITiCSE 2006 working group.

There are naturally many more possible features that could be implemented. Implementing the rest of the requirements and to support the complete Xaal specification are high on our wish list. However, the nature of HTML and JavaScript offers some unusual possibilities. The following is a list of the most interesting future development ideas.

- In the current version, there is already support for drawing annotations on the animation. In the future, we could store these annotations and then later show them for the same student, or even share the annotations between students.
- We could support opening animations in different formats directly in the browser.
- Due to the nature of JavaScript, replacing functions on the fly is trivial. This would allow creation of automatically assessed exercises where the student is required to code some algorithm using the data structures in the viewed animation.
- The current solution requires internet access if used in evaluation. However, with cutting edge technologies like Google Gears or Dojo.Offline, offline usage of animations where assessment/results are submitted when the student is online, could be developed.

References

S. Diehl. *Software visualization: Visualizing the Structure, Behaviour, and Evolution of Software.* Springer New York, 2007.

Christopher D. Hundhausen, Sarah A. Douglas, and John T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing*, 13(3): 259–290, June 2002.

Ville Karavirta. *Facilitating Algorithm Animation Creation and Adoption in Education.* Licentiate's thesis, Helsinki University of Technology, December 2007. Available online at `http://www.cs.hut.fi/Research/SVG/publications/karavirta-lis.pdf`.

Lauri Malmi, Ville Karavirta, Ari Korhonen, Jussi Nikander, Otto Seppälä, and Panu Silvasti. Visual algorithm simulation exercise system with automatic assessment: TRAKLA2. *Informatics in Education*, 3(2):267–288, 2004.

Andres Moreno, Niko Myller, Erkki Sutinen, and Mordechai Ben-Ari. Visualizing programs with Jeliot 3. In *Proceedings of the International Working Conference on Advanced Visual Interfaces*, pages 373 – 376, Gallipoli (Lecce), Italy, May 2004.

Thomas Naps, Myles McNally, and Scott Grissom. Realizing XML driven algorithm visualization. In *Proceedings of the Fourth Program Visualization Workshop*, pages 129–135, 2006.

Thomas L. Naps. JHAVÉ: Supporting Algorithm Visualization. *Computer Graphics and Applications, IEEE*, 25(5):49–55, 2005.

Thomas L. Naps, Guido Rößling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodgers, and J. Ángel Velázquez-Iturbide. Exploring the role of visualization and engagement in computer science education. *SIGCSE Bulletin*, 35(2):131–152, June 2003a.

Thomas L. Naps, Guido Rößling, Jay Anderson, Stephen Cooper, Wanda Dann, Rudolf Fleischer, Boris Koldehofe, Ari Korhonen, Marja Kuittinen, Charles Leska, Lauri Malmi, Myles McNally, Jarmo Rantakokko, and Rockford J. Ross. Evaluating the educational impact of visualization. *SIGCSE Bulletin*, 35(4):124–136, December 2003b.

Thomas L. Naps, Guido Rößling, Peter Brusilovsky, John English, Duane Jarc, Ville Karavirta, Charles Leska, Myles McNally, Andrés Moreno, Rockford J. Ross, and Jaime Urquiza-Fuentes. Development of XML-based tools to support user interaction with algorithm visualization. *SIGCSE Bulletin*, 37(4):123–138, December 2005. doi: http://doi.acm.org/10.1145/1113847.1113891.

Cristóbal Pareja-Flores, Jamie Urquiza-Fuentes, and J. Ángel Velázquez-Iturbide. WinHIPE: an ide for functional programming based on rewriting and visualization. *ACM SIGPLAN Notices*, 42(3):14–23, 2007. doi: http://doi.acm.org/10.1145/1273039.1273042.

Rockford J. Ross and Michael T. Grinder. Hypertextbooks: Animated, active learning, comprehensive teaching and learning resource for the web. In Stephan Diehl, editor, *Software Visualization: International Seminar*, pages 269–283, Dagstuhl, Germany, 2002. Springer.

Guido Rößling and Bernd Freisleben. ANIMAL: A system for supporting multiple roles in algorithm animation. *Journal of Visual Languages and Computing*, 13(3):341–354, 2002.

Guido Rößling and Thomas L. Naps. Towards intelligent tutoring in algorithm visualization. In *Second International Program Visualization Workshop*, pages 125–130, Aarhus, Denmark, 2002a.

Guido Rößling and Thomas L. Naps. A testbed for pedagogical requirements in algorithm visualizations. In *Proceedings of the 7th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education, ITiCSE'02*, pages 96–100, Aarhus, Denmark, 2002b. ACM Press, New York.

Guido Rößling, Thomas Naps, Mark S. Hall, Ville Karavirta, Andreas Kerren, Charles Leska, Andrés Moreno, Rainer Oechsle, Susan H. Rodger, Jaime Urquiza-Fuentes, and J. Ángel Velázquez-Iturbide. Merging interactive visualizations with hypertextbooks and course management. *SIGCSE Bulletin*, 38(4):166–181, 2006.

Integrating test generation functionality into the Teaching Machine environment

Michael Bruce-Lockhart, Theodore Norvell
Computer Engineering Research Labs
Faculty of Engineering and Applied Science
Memorial University of Newfoundland
St. John's, NF, Canada A1B 3X5

Pierluigi Crescenzi
Dipartimento di Sistemi e Informatica
Università degli Studi di Firenze
Viale Morgagni 65, 50134 Firenze, Italy

mpbl@engr.mun.ca, theo@mun.ca, piluc@dsi.unifi.it

Abstract

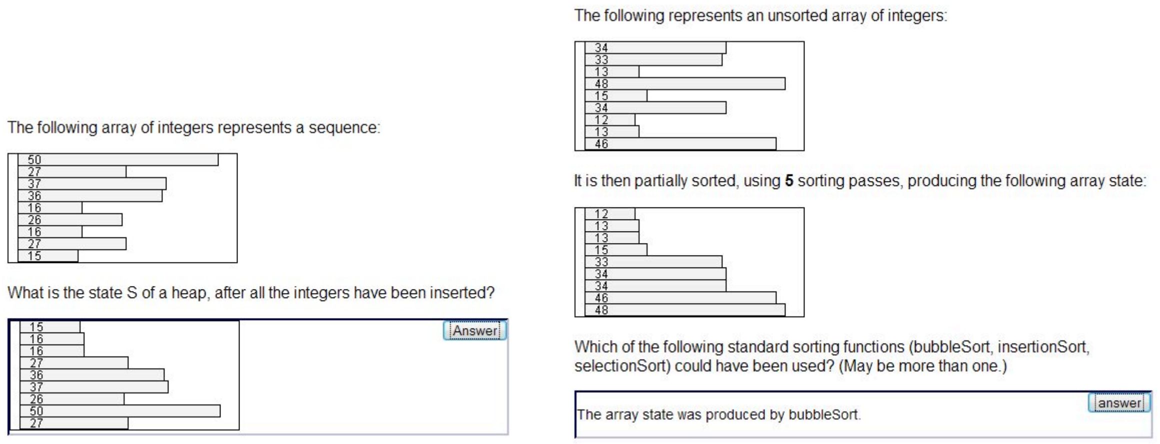
In this paper we introduce an extension of the Teaching Machine project, called Quiz Generator, that allows instructors to produce assessment quizzes for courses in algorithms and data structures quite easily. This extension makes use of visualization techniques and is based on a new feature of the Teaching Machine that allows third-party visualizers to be added as plugins. In essence Quiz Generator builds on new scripting capabilities for both the Teaching Machine and the WebWriter++ authoring system. Using these new capabilities, several question types have already been produced.

1 Introduction

Allowing students to test their knowledge in an autonomous and automatic way is certainly one of the most important topics within computer science education and distance learning. Indeed, many systems have been proposed in the literature for automatic assessment of exercises on programming (e.g. Vihtonen and Ageenko (2002); Ala-Mutka (2005)), algorithm and data structures (e.g. Malmi et al. (2004); Laakso et al. (2005)), and object-oriented design (e.g. Higgins et al. (2002)). Two of the most important features that these systems should exhibit are, from the teacher point of view, ease of use and, from the student point of view, the possibility of replicating the same kind of test with different data. In this paper, we focus our attention on the automatic generation of assessment quizzes in the field of algorithm and data structures and on the use of visualization techniques in generating quizzes (Cooper, 2007).

In order to keep the level of difficulty encountered by the teacher while generating a new kind of test reasonably low, we decided to avoid tests based on the manipulation of a data structure, such as the ones described in (Krebs et al., 2005): nonetheless, we think that the coverage of test types proposed in the following section is quite wide.

Our basic approach is to add test generation functionality to the existing Teaching Machine (Bruce-Lockhart and Norvell, 2007; Bruce-Lockhart et al., 2007) and WebWriter++ (Bruce-Lockhart and Norvell, 2006) environment. The Teaching Machine is a tool for visualizing how Java or C++ code runs on a computer. It contains compilers for both languages and an interpreted run-time environment that provides a high-level object model for the state of its virtual machine. This object model is accessed by a number of standard visualizers that present the state graphically to the user. Recently we extended the Teachine Machine to allow third-party visualizer to be added as plugins. The Teaching Machine is written in Java and may be run as an applet or as an application. WebWriter++ is a small authoring system written in JavaScript whose purpose is to allow authors of pedagogical web pages to focus on content rather than technology. It interfaces with the Teaching Machine as well as providing a number of other automated facilities such as displaying colour stained code in a visual container with buttons to run it or edit it or run a video about it.


The following array of integers represents a sequence:

What is the state S of a heap, after all the integers have been inserted?

The following represents an unsorted array of integers:

It is then partially sorted, using **5** sorting passes, producing the following array state:

Which of the following standard sorting functions (bubbleSort, insertionSort, selectionSort) could have been used? (May be more than one.)

The array state was produced by bubbleSort.

Figure 1: The first and second test types: predicting a data structure state (left) and determining the data structure or the algorithm (right)

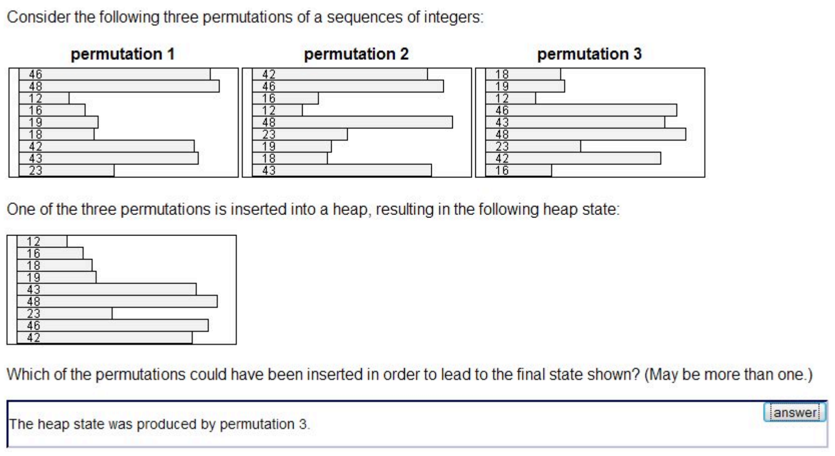
1.1 Test types

Our system features the following kinds of tests:

- Given an input, determine the state S of a data structure after the input elaboration and/or the algorithm execution: for example, given a sequence of integers, the student is asked to derive the state S of a heap, after all the integers have been inserted (see left part of Figure 1). In this case, a visualizer plugin is used during the assessment phase in order to show the correct answer to the student.

- Given an input and the state S of a data structure, determine which kind of data structure and/or which algorithm has been used in order to produce the state S after the (partial) input elaboration: for example, given a sequence of integers and given a partially sorted array, the student is asked to determine which sorting algorithm has been applied in order to produce the partially sorted array starting from the initial sequence of integers (see right part of Figure 1). In this case, a visualizer plugin is used in order to produce the visualization of the state of the data structure.

- Given a set of different inputs and the state S of a data structure, determine which input has been elaborated by the data structure and/or the algorithm in order to reach

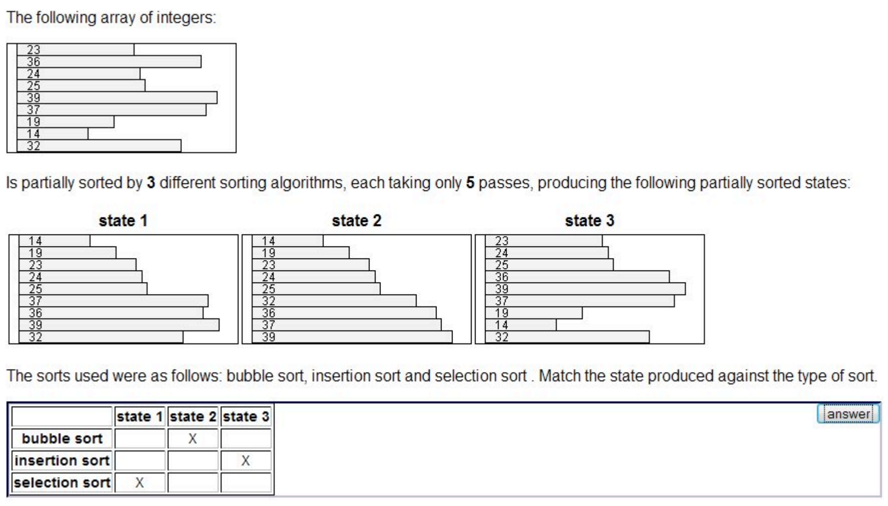

Consider the following three permutations of a sequences of integers:

One of the three permutations is inserted into a heap, resulting in the following heap state:

Which of the permutations could have been inserted in order to lead to the final state shown? (May be more than one.)

The heap state was produced by permutation 3.

Figure 2: The third test type: determining the input


The following array of integers:

Is partially sorted by **3** different sorting algorithms, each taking only **5** passes, producing the following partially sorted states:

The sorts used were as follows: bubble sort, insertion sort and selection sort . Match the state produced against the type of sort.

	state 1	state 2	state 3
bubble sort		X	
insertion sort			X
selection sort	X		

Figure 3: The fourth test type: determining the input

the state S: for example, given a set of sequences of integers and given a heap H, the student is asked to determine which sequence produces the heap H after all the integers of the sequence have been inserted (see Figure 2). Again, a visualizer plugin is used to produce the visualization of the state of the data structure.

- Given an input and a set of states of different data structures, determine which kind of data structures and/or algorithms have been used in order to elaborate the input: for example, given a sequence of integers and given three partially sorted arrays, the student is asked to determine which sorting algorithm (among a specified set) produces each of the three arrays (see Figure 3). In this case, the visualizer plugin is used several times in order to produce the visualizations of the states of the data structures.

Clearly, creating new test types depends quite heavily on the availabilty of specialized visualizers as well as the development of a means to capture their outputs at specific points.

1.2 Structure of the paper

In the next section, we briefly describe the Teaching Machine and WebWriter++ extension, which has been produced in order to develop the *Quiz Generator* framework: this extension mainly consists of a new plugin architecture and an enhanced scripting capability. In Sections 3 and 3.1, we describe a test example and how the new features of the Teaching Machine and the WebWriter++ tools allow the system to visualize and assess the test. We conclude in Section 5 by listing some research questions concerning the possibility of using Quiz Generator as a testing tool, and not only as a self-assessment tool.

2 System software architecture

The Quiz Generator project is an extension of the Teaching Machine project. As such it extends the two primary tools of this latter project — the Teaching Machine, which is a programming animation tool written in Java, and WebWriter++, which is a JavaScript library for authoring interactive web pages for learning programming.

2.1 Visualization plugins

In particular, Quiz Generator leverages a rewrite of the Teaching Machine carried out in 2006-2007 which permitted the incorporation of third party visualizer plugins for the Teaching Machine. This allows instructors to develop their own visualizers without touching, or even recompiling, the Teaching Machine core. While plugins are not confined to visualizers, we believed visualizers would be an important need. The objective is to allow experienced developers to create new visualizers in a matter of between one and three days of programming. Indeed, it was the availability of this capability that got us thinking about developing a quiz generator capability in the first place.

An important goal for our system is to allow the instructor to produce a test quite easily. The kinds of tests proposed require a number of different visualizer plugins which, even at only a day or two apiece, can require significant time to develop. Nevertheless, such development cannot be charged against test development as it would be unreasonable to present students with a visualization on a quiz that they had not seen in the course. Thus, for the purposes of this exercise, we assume that appropriate visualization plugins already exist and have been used in the course.

2.2 Scripting the Teaching Machine

Easy production also means a teacher should be able to produce a test without modifying the implementation of a data structure and/or of an algorithm. For example, if we refer to the first test type example and if we assume that the teacher has already programmed a Java or C++ class implementing a heap, then the test can be deployed without modifying this code by simply inserting a few scripting commands in the Java or C++ code, as comments, and by inserting a few JavaScript commands within the test web page.

The communication between the host web-page, the Teaching Machine, and the subject (Java or C++) code goes as follows.

1. A JavaScript command within the web page initiate the compilation and execution of the Java or C++ code within the Teaching Machine.

2. As the Java or C++ code executes in the Teaching Machine, scripts embedded as comments in the code command the Teaching Machine to produce images representing the state of one or more data structures.

3. Once the Teaching Machine has finished executing, JavaScript commands within the web page collect the images and send them to applets (called portholes) embedded within the question text.

This approach builds on earlier work with interactive learning pages which utilizes the connection between the WebWriter++ authoring tool and the Teaching Machine.

What was needed for the Quiz Generator project was a richer set of embedded scripting controls for the Teaching Machine than we had had in WebWriter++. For example, our learning web pages can currently display a code fragment for discussion, then allow a student to launch the example in the Teaching Machine to run it for herself, to edit it, or to possibly watch a video about it. Creating quizzes is more demanding.

3 A sample quiz

Here we expand in more detail the example given for the second type of test described in Section 1.1 (see right part of Figure 1). Ideally, a student would be presented with a visualization of an unsorted array, randomly populated according to parameters laid out by the instructor. A second snapshot of the array is presented after a partial sort, together with a list of algorithms. The student is told how many sorting passes were done and asked to check all algorithms that could have created the second snapshot.

Again, it is assumed that both the appropriate visualization plugins and an implementation of the sorting code and data structure already exist and have been used in the course.

External scripting commands		
Command	*Effect*	*Status*
`run(filename)`	Loads filename into the Teaching Machine and waits at 1st line	Pre-existing
`autoRun(filename)`	Loads filename into Teaching Machine and runs it invisibly	Built
`insertPorthole(name)`	Create a container in the quiz for a snapshot	Built
`putSnaps()`	Load all snapshots from the Teaching Machine into portholes	Built
`addCLArg(arg)`	Add a command line argument for the program to be run in Teaching Machine	Built

Internal scripting commands		
Command	*Effect*	*Status*
`relay(id, call)`	Relay function call to plugin id	Built
`snapshot(id, name)`	Take a snapshot of plugin id for porthole name	Built
`stopAuto()`	Stop execution at this point	Built
`breakPoint(id)`	Create backup point id here	Planned
`backup(id)`	Backup Teaching Machine to point id	Planned
`makeRef(id)`	Use the data structure in plugin id as a reference for comparison	Built
`compare(id)`	Compare data structure in plugin id to the reference data structure	Built
`returnResults`	Ship snapshot results back to quiz script	Built

Table 1: Scripting commands

To create the quiz, the instructor first instruments the code with testing parameters, for example:

1. The size of the array (or a range of sizes, from which one size would be randomly picked).
2. The value range desired for random population of the array.
3. The sorting algorithm to be used.
4. The number of sorting passes (or, again, a permissable range).

The code (or really code sets, since different pieces of code are required for different topics) and the visualizations would form a resource base for creating actual quizzes. The quizzes themselves are created in HTML (or XHTML) using QuizWriter++, an extension to WebWriter++.

3.1 Scripting from inside and outside

Let us first consider the following simpler quiz question: given an unsorted array A and a snapshot of A after a specific sorting algorithm has been partially applied, the student is asked to determine how many sorting steps have been executed. In terms of controlling the Teaching Machine, this question is quite limited. We need to:

1. Load the appropriate code into the Teaching Machine.
2. Pass it some parameters, such as the size of the array, the selection of the bubble sort implementation and the number of sorting steps.

3. Start up the Teaching Machine to run invisibly (so the student cannot inspect it).
4. Specify the visualizer and the data whose pictures we want.
5. Have the Teaching Machine stop after it has executed the requisite number of sorting steps.
6. Recover the two snapshots (before and after) from the visualizer.

Nevertheless, it requires far more detailed control of the Teaching Machine than we have ever exercised before, for example the requirement to run it invisibly and pass it parameters. Such control was done previously by simple scripting from WebWriter++ generated pages. Once the Teaching Machine applet was loaded, JavaScript calls could readily invoke Teaching Machine applet functions. QuizWriter++ simply extends this capability by adding new functionality both to the Teaching Machine and to the scripts, for example creating an autorun mode in the Teaching Machine (previously it had always been run manually, like a debugger), and allowing the passing of arguments from a script.

That alone is not enough, however. We found it was also convenient to control the Teaching Machine from within the running code, that is, to allow the example running to issue commands directly to the Teaching Machine (such as when to halt or when to drop a snapshot). In essence, it was necessary to develop a second scripting capability. To distinguish between them we call scripting from the JavaScript on the quiz page external scripting and scripting from with the running code internal scripting. Table 1 shows a number of potential scripting calls as well as their current status.

The quiz questions of Figures 1-3 were produced by using the capabilities of Table 1 that are already built. For example, to engage fully the question posed at the beginning of Section 3 would require something like the following:

1. Load the appropriate code into the Teaching Machine.
2. Pass it some parameters, such as the size of the array, the selection of the bubble sort implementation and the number of sorting steps.
3. Start up the Teaching Machine to run invisibly (so the student cannot inspect it).
4. Specify the visualizer and the data whose pictures we want.
5. Have the Teaching Machine stop after it has executed the requisite number of sorting steps and take a snapshot.
6. Back the Teaching Machine up and rerun it on every other sorting algorithm specified.
7. For each algorithm, have the visualizer compare the state of the array after the reference algorithm to the state of the array after the current sort.
8. Recover the two snapshots (before and after) from the visualizer.
9. Recover data specifying algorithms that produced equivalent sorts.

The scripting calls in Table 1 were arrived at by examining just such quiz scenarios.

4 Related work

This paper fits into the third level (that is, the responding level) of the learner engagement taxonomy presented in Naps et al. (2002). As stated in the introduction, it tries to avoid some of the impediments listed in Naps et al. (2003) and faced by instructors, while adopting visualization techniques, by making as easy as possible the development of new quizzes and by integrating them within a unified framework, such as the one provided by WebWriter++ and the Teaching Machine (By the way, Naps et al. (2002) and Naps et al. (2003) provide a good background for the research and development described in this paper, as well as test settings for evaluation). Other papers deal with the development of interactive prediction facilities such as Jarc et al. (2000) and Naps et al. (2000), where web-based tools are presented and evaluated, and Rößling and Häußge (2004), where a tool-independent approach is described.

5 A work in progress

By constructing and examining quiz scenarios we are currently refining what capabilities we need in order to be able to achieve the kinds of quizzes laid out in Section 1.1. Nevertheless, the existing capabilities already span almost the entire space of controls needed, in the sense that they require almost all the structural extensions to the Teaching Machine that are needed. The additional scripting mostly requires the addition of new functions rather than fundamental changes to the Teaching Machine structure.

5.1 Research questions

Indeed, we are quite excited to have come this far. In the early days of scripting development it was by no means always certain that we would be able to achieve all our objectives. Now that the design space is largely spanned we can focus on the development of the extra functionality required. Once that is done, it will allow us to concentrate on the research questions that are at the core to the whole endeavour of automated testing:

1. Given a space of possible questions an instructor might want to ask in data structures and algorithms, can we build a quiz generator that does a reasonable job of spanning that space? That is, can an instructor use it to examine most of the issues he might want?

2. Even if we are successful in 1, can we produce enough variations in questions for the tool to be useful over a large number of uses? That is, can we produce enough different quizzes?

3. If we are succesful in 2, can we produce a set of quizzes that are reasonably equivalent? That is, would students taking different quizzes from each other perceive that they had been treated fairly?

The last question, of course, moves beyond the realm of self-testing into the more vexing issue of testing for credit. That brings up a whole set of important issues such as quiz security and the proper gathering of quiz data. Nevertheless, until these three primary questions can be answered positively, there is no point in embarking upon these other issues. We are very hopeful that our current approach will be sufficiently successful to require these other issues to be tackled in the future.

References

K. Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2):83–102, 2005.

M. Bruce-Lockhart and T. S. Norvell. Interactive embedded examples: a demonstration. *SIGCSE Bulletin*, 38(3):357–357, 2006.

Michael P. Bruce-Lockhart and Theodore S. Norvell. Developing mental models of computer programming interactively via the web. In *Frontiers in Education Conference - Global Engineering: Knowledge without Borders, Opportunities without Passports*, pages 3–8, 2007.

Michael P. Bruce-Lockhart, Theodore S. Norvell, and Yiannis Cotronis. Program and algorithm visualization in engineering and physics. *Electronic Notes in Theoretical Computer Science*, 178:111–119, 2007.

M.L. Cooper. Algorithm visualization: The state of the field. Master thesis at Virginia Polytechnic Institute and State University, 2007.

C. Higgins, P. Symeonidis, and A. Tsintsifas. The marking system for coursemaster. In *Proc. 7th Annual Conference on Innovation and Technology in Computer Science Education*, pages 46–50, 2002.

D. J. Jarc, M. B. Feldman, and R. S. Heller. Assessing the benefits of interactive prediction using web-based algorithm animation courseware. *SIGCSE Bull.*, 32(1):377–381, 2000.

M. Krebs, T. Lauer, T. Ottmann, and S. Trahasch. Student-built algorithm visualizations for assessment: flexible generation, feedback and grading. In *Proc. 10th Annual Conference on Innovation and Technology in Computer Science Education*, pages 281–285, 2005.

M. Laakso, T. Salakoski, L. Grandell, X. Qiu, A. Korhonen, and L. Malmi. Multi-perspective study of novice learners adopting the visual algorithm simulation exercise system TRAKLA2. *Informatics in Education*, 4:49–68, 2005.

L. Malmi, V. Karavirta, A. Korhonen, J. Nikander, O. Seppälä, and Panu Sistali. Visual algorithm simulation exercise system with automatic assessment: TRAKLA2. *Informatics in Education*, 3:267–288, 2004.

T. Naps, S. Cooper, B. Koldehofe, C. Leska, G. Rößling, W. Dann, A. Korhonen, L. Malmi, J. Rantakokko, R. J. Ross, J. Anderson, R. Fleischer, M. Kuittinen, and M. McNally. Evaluating the educational impact of visualization. In *Working Group Reports from 8th Annual Conference on Innovation and Technology in Computer Science Education*, pages 124–136, 2003.

T. L. Naps, J. R. Eagan, and L. L. Norton. JHAVÉ—an environment to actively engage students in web-based algorithm visualizations. *SIGCSE Bull.*, 32(1):109–113, 2000.

T. L. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, and J. Á. Velázquez-Iturbide. Exploring the role of visualization and engagement in computer science education. In *Working Group Reports from 7th Annual Conference on Innovation and Technology in Computer Science Education*, pages 131–152, 2002.

G. Rößling and G. Häußge. Towards tool-independent interaction support. In *Proc. 3rd International Program Visualization Workshop*, pages 110–117, 2004.

E. Vihtonen and E. Ageenko. VIOPE-computer supported environment for learning programming languages. In *Proc. Int. Symposium on Technologies of Information and Communication in Education for Engineering and Industry*, pages 371–372, 2002.

Kick-Start Activation to Novice Programming — A Visualization-Based Approach

Essi Lahtinen, Tuukka Ahoniemi
Tampere University of Technology, Department of Software Systems

essi.lahtinen@tut.fi, tuukka.ahoniemi@tut.fi

Abstract

In the beginning of learning programming students have misconceptions of what programming is. We have used a kick-start activation in the beginning of an introductory programming course (CS1) to set the record straight. A kick-start activation means introducing the deep structure of programming before the surface structure by making the students solve a certain type of problem in the first lecture. The problem is related to a realistic computer program, simple enough for everyone to understand and allow students to participate in debugging. A visualization-based approach helps making the example more concrete for students.

In this article we present the concept kick-start activation and one concrete example. To support the example, we have also developed a visualization using the visualization tool JHAVÉ. We got positive feedback on the example and suggest further development of kick-start activations in order to make the beginning of learning programming more motivating for students.

1 Introduction

Students who enroll to introductory programming courses (CS1) have plenty of misconceptions about the nature of programming and some students do not know what programming is at all. The course typically starts with the teacher trying to correct the misconceptions by emphasizing that programming is more problem-solving and thinking than typing program code. The concept of the algorithm is introduced, as well as some tools for implementing algorithms and designing programs, such as pseudocode or flow charts.

A classical first example of an algorithm is a recipe in a cook-book. A recipe is a relatively unambiguous, detailed set of instructions. If you follow the instructions carefully you will have a food portion as the result. However, there are problems with this example. Firstly, it is not at all related to computers. Thus students might feel that the teacher is stating the obvious or even explaining nonsense when he/she is talking about cooking and algorithms. Secondly, even though comparing cooking recipes and algorithms gives a clear idea on what an algorithm is, it does not really help to understand what a programmer does. The underlying idea of programming is not delivered to the students. Thirdly, the methaphor also does not help in explaining the programming process for the student.

We introduce a different way to start the course: *kick-start activation*. In this approach, we get into the deep structure of programming before the surface structure is even introduced. Our target audience is especially the students who do not know anything about programming before the kick-start activation. In this article we first present the idea of a kick-start activation in Section 2. Then we introduce our example and explain how we use it in Section 3. Section 4 presents the visualization and feedback. Finally, discussion and conclusion are included in Section 5.

2 Criteria for a Kick-Start Activation

In our opinion, to make the opening of the course interesting for students, one needs to get directly into the real problems, i.e., a problem that requires an algorithmic solution. In the case of programming this means skipping the surface structure, such as the syntax of the programming language, and starting from the deep structure of programming, i.e., a problem that the students solve themself. We call this kind of an introduction *kick-start activation* because it is a fast-forward jump-in approach and it engages students in the example since they solve the problem.

Our *main criterion* for the example presented in the kick-start activation is that it has to be based on *a real computer program.* The benefits of a real programming example are that 1) in addition to introducing the concepts of algorithms, pseudo code and flow charts one can also introduce problem solving and the phases of programming, the idea of testing algorithms and programs, and show what the work of a programmer is like. 2) It helps to explain the difference of human thinking and the way the computer works. 3) The execution of the algorithm can be explained and demonstrated with a computer. 4) One can also show an implementation in a programming language to give an example. Students can identify the control structures of the pseudo code from the program code. 5) It can be concretized by a program visualization that the students can run.

Our *second criterion* is that the kick-start activation needs to be simple enough so it can be understood by everyone. Firstly, we decided that it has to be an example that *relates to everyday life.* Secondly, we chose not to use a real programming language nor any terms, pictures, or other details that relate to computers. For example, we did not want `[]`-operators in the algorithm or memory addresses in the pictures. These would just add extra details that are irrelevant at this stage. Instead of using a programming language it is easier to fade out the surface structure of programming by using a natural-language-like pseudo code presentation and flow charts. To concretize the pseudo code and flow chart we developed a visualization that illustrates how the algorithm would be run by a computer if the computer could understand it.

The *third criterion* for a kick-start activation was to make *students take part* in the example. As programming is much more thinking and problem-solving than using the programming language syntax, there are numerous programming related activities that students can try already in the beginning of the course. For instance testing an algorithm is a task that can be given to a student. One practical way of doing this is developing a buggy version of an algorithm that the students can debug.

3 Our Example: Hyphenating Finnish Words

The topic of our kick-start activation was the hyphenation rules of the Finnish language. Word processors have spell checking and automatic hyphenation, i.e., computer programs are hyphenating Finnish words. In addition, every student knows how to spell[1] so the topic is general enough.

The exact rules for hyphenating Finnish are not common knowledge in Finland even if it is easy to hyphenate Finnish for everyone who knows how to speak the language. Fortunately the rules are simple enough to be explained to students in a few sentences. Still, it is non-trivial to build a hyphenation algorithm. The algorithm requires a loop structure to go through the letters of the hyphenated word and a couple of if-statements to choose which hyphenation rule to apply.

For example, the first of three hyphenation rules called *the consonant rule* states the following: *if there is a vowel followed by one or more consonants, a hyphen is placed directly before the last consonant.* The window on the right hand side in Figure 1 presents the algorithm based on the rules. The consonant rule can be identified in the marked area of the figure.

A word is a data structure that can be understood even without knowing the data type `string`. A word can also be drawn like a line of alphabet building blocks (See the window on the left hand side in Figure 1). Introducing the computer memory or other similar details for the student is unnecessary. Drawing the data structure as a line of building blocks actually

[1]In this situation actually: *in Finland* every student knows how to spell *Finnish.*

allows us to visualize the addition of a hyphen: a picture animation where a block with the character '-' slides and slips in between the blocks of the word.

On the lecture, our intention was to highlight that designing and testing the algorithm with pen and paper is a big part of programming. To describe this clearly we used a three step example: 1) First we quickly designed a hyphenation algorithm. Though it seemed to be correct the hasty design had on purpose produced a buggy solution. 2) Then the students tested the algorithm and hopefully found the error. After this we discussed how important it is to understand the problem before you start designing the algorithm. 3) Finally, we explained the hyphenation rules deeper for the students and designed a new algorithm properly. The final result was a correctly working algorithm. The example included two algorithms. We call these *the premature algorithm* (produced in step 1) and *the mature algorithm* (produced in step 3).

The purpose of the testing phase was to activate the students. They were actually performing a programming related task even if they thought they did not know any programming yet. The idea is that the students can use the visualization to run and test the algorithm. The testing could of course be done using only pen and paper, but the visualization is handy in it. We gave a link to the visualization to the students for later use so that they could revise the lecture using the visualization.

4 The Visualization

There are many program visualization tools available for presenting basic programming structures for novice programmers. These visualization tools work on program code level, so they assume that the student already understands some programming language and thus are not suitable for our target audience. There is also a visualization tool called RAPTOR (Giordano and Carlisle, 2006) where the students can construct flow charts and the tool will visualize them for the student. The RAPTOR flow charts are also close to the program code level, e.g., the tool shows the content of variables and arrays.

We needed a completely syntax-free common purpose visualization tool where we can write the algorithm in a few Finnish sentences and draw the building blocks exactly according to our needs. Thus, the existing program visualization tools did not suit our purposes. However, in the field of algorithm visualizations there was one tool flexible enough: JHAVÉ (Naps et al., 2000) and its Gaigs support class package. With a bit of imagination we were able to use this algorithm visualization tool slightly unorthodoxically and produce the hyphenation visualization.

The info screen of JHAVÉ's execution window is normally used for showing algorithm specific instructions written in HTML. The tool allows the use of images as a part of the HTML page with the <image> tag. This feature let us implement the flow chart animation with a set of fixed images. The images were then presented in the correct order by showing a particular image in each state of the program. With the possibility of using HTML and images in JHAVÉ, one could design many sorts of examples as the technical implementation is limited solely to the creation of the images.

Using JHAVÉ, we implemented two different presentations of *the hyphenation algorithm visualization*: a pseudo code view and a flow chart. Both of these presentations also contain a window with the alphabet building block picture of the hyphenated word. Screen shots can be seen in Figure 1 and Figure 2. There were two different algorithms that we visualized: the premature and mature. Since there are two different presentations of both the algorithms we actually had four different visualizations.

4.1 Student Engagement

According to research on the field of visualizations, student engagement is vital for learning when a student uses visualization (Stasko and Hundhausen, 2004). Naps et al. (2003) present

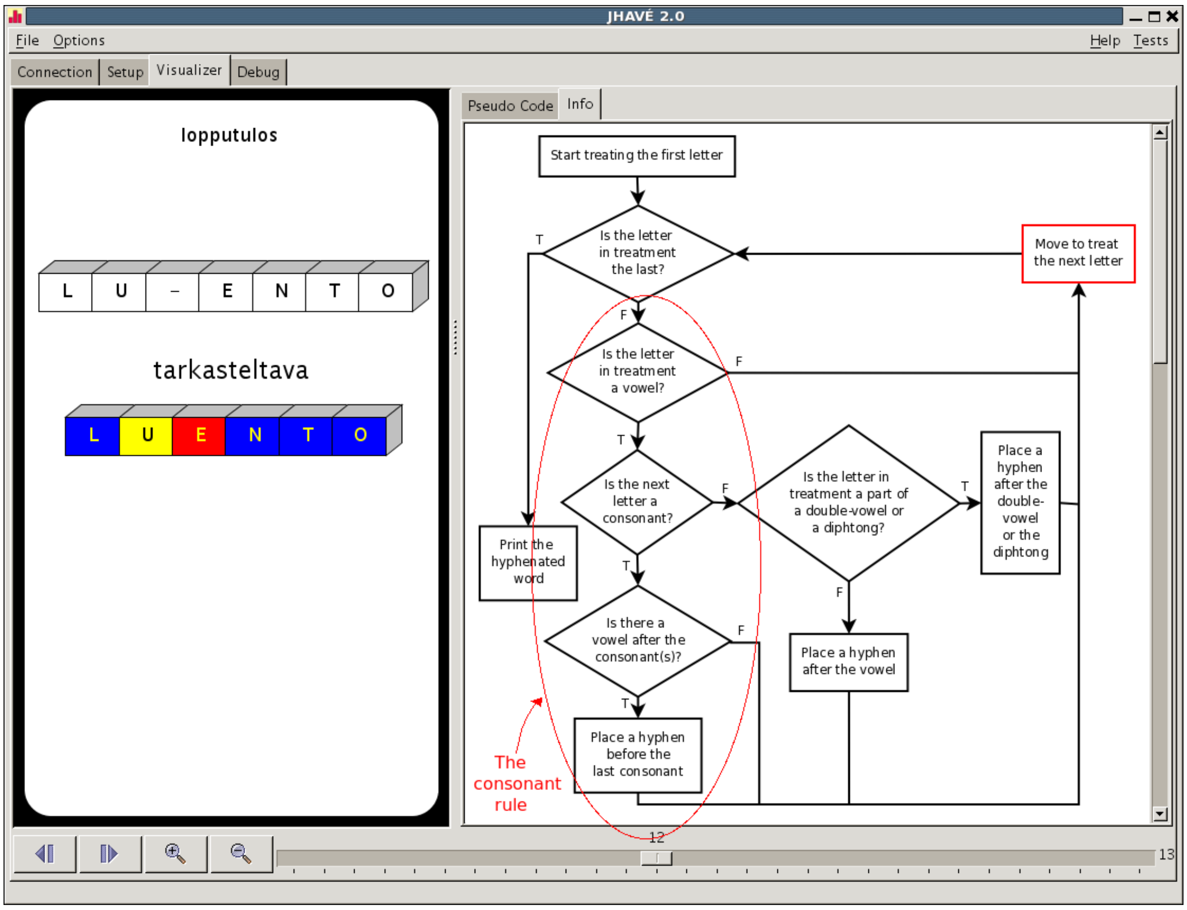

JHAVÉ 2.0
File Options
Help Tests
Connection Setup Visualizer Debug
Pseudo Code Info
lopputulos
L U – E N T O
tarkasteltava
L U E N T O
Start treating the first letter
Is the letter in treatment the last?
Move to treat the next letter
Is the letter in treatment a vowel?
Is the next letter a consonant?
Is the letter in treatment a part of a double-vowel or a diphtong?
Place a hyphen after the double-vowel or the diphtong
Print the hyphenated word
Is there a vowel after the consonant(s)?
Place a hyphen after the vowel
Place a hyphen before the last consonant
The consonant rule
T
F
12
13

Figure 1: The flow chart version of the visualization.

a Visualization Engagement Taxonomy that describes six levels of learner engagement with visualization technology. On top of the lowest level of existing engagement—*Viewing*—are the more active levels: *Responding* and *Changing* an existing visualization and *Constructing* and *Presenting* ones own visualization.

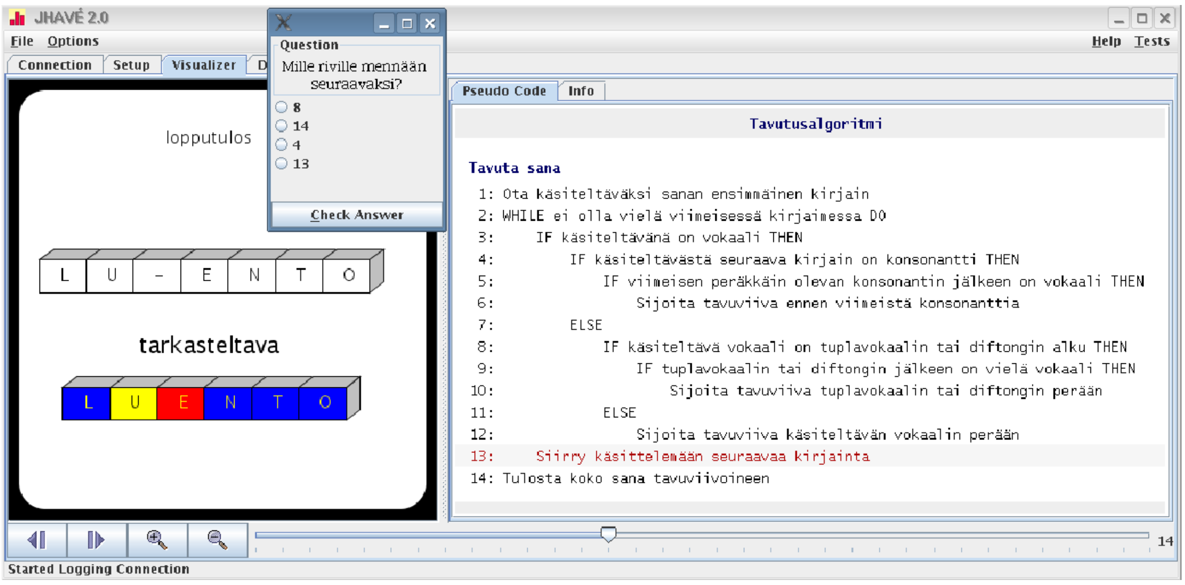
As the algorithm is given fixed in the hyphenation algorithm visualization, the student engagement is enhanced by allowing the student to provide his/her own input word for the algorithm. This corresponds to the level *Change* of the Visualization Engagement Taxonomy (Naps et al., 2003). To attain the level *Response* also, the flow of the program is interrupted with pop-up questions querying about the next behavior of the program.

4.2 Student Feedback

We evaluated the visualization with a quantitative survey after the lecture where we used it. We handed in a questionnaire on paper for the students. We received altogether 113 responses. 71 of the respondents (63%) had no programming experience before the course.

The feedback was generally positive since 53% of the respondents said that the visualization looked nice (agree or totally agree), 86% thought that is was useful for learning (agree or totally agree), and only 5% thought that it disturbed the lecture (agree or totally agree).

We performed a crosstabulation and a χ^2-test for some of the variables and found out that the students with no earlier programming experience thought that the visualization was more useful for learning than the students who had programmed before coming to the course. This difference is statistically significant ($p < 0,05$). The reason is also obvious: the students with earlier programming experience already had an understanding on how algorithms and flow charts work so they do not need the visualization for understanding the hyphenation



Question
Mille riville mennään seuraavaksi?

- 8
- 14
- 4
- 13

Check Answer

Tavutusalgoritmi

```
Tavuta sana
 1: Ota käsiteltäväksi sanan ensimmäinen kirjain
 2: WHILE ei olla vielä viimeisessä kirjaimessa DO
 3:     IF käsiteltävänä on vokaali THEN
 4:         IF käsiteltävästä seuraava kirjain on konsonantti THEN
 5:             IF viimeisen peräkkäin olevan konsonantin jälkeen on vokaali THEN
 6:                 Sijoita tavuviiva ennen viimeistä konsonanttia
 7:         ELSE
 8:             IF käsiteltävä vokaali on tuplavokaalin tai diftongin alku THEN
 9:                 IF tuplavokaalin tai diftongin jälkeen on vielä vokaali THEN
10:                     Sijoita tavuviiva tuplavokaalin tai diftongin perään
11:             ELSE
12:                 Sijoita tavuviiva käsiteltävän vokaalin perään
13:     Siirry käsittelemään seuraavaa kirjainta
14: Tulosta koko sana tavuviivoineen
```

Figure 2: The pseudo code version of the visualization with an activating pop-up question.

algorithm. This result shows that we managed to help the students who were the target audience of the visualization.

After all, the most important feedback was that our students were listening to the hyphenation example intensively on the lecture. Two teachers tried the example and both of them could sense a notable difference in the lecture situation compared to the cook-book example.

5 Discussion and Conclusions

The kick-start activation received positive feedback both from the students and the teachers who used it. We think that our approach was successful because the criteria were designed carefully and there was a visualization tool that aided both presenting the example and understanding it. This example could be used as a source of ideas for other topics to build kick-start activations of.

There are not many program visualization tools available for our target audience—the students who do not know anything about programming yet. In addition to our visualization we have found a system called SICAS (Mendes et al., 2005) that could probably also be used for presenting a kick-start activation. It is based on similar principles and allows students to construct their own flow charts and visualize them. However, currently it is not used the same way we used our visualization.

The conceptual framework of programming knowledge developed by McGill and Volet (1997) suggests that in addition to syntactic and conceptual knowledge a programmer also needs strategic knowledge of programming. Reports on the state of field show that visualizations are often used for only presenting programming concepts (Shaffer et al., 2007). The scope of our visualization is more in the strategic knowledge since it focuses on the programming phases: testing and design.

In the development of the visualization we also emphasized student engagement in the levels of the Visualization Engagement Taxonomy (Naps et al., 2003). The visualization is most activating when the student is guided to use it in the three step lesson we described in Section 3. This requires either a teacher to explain the hyphenation problem and the need for debugging the first version of the algorithm or the student to read this from the material by himself. The idea of connecting a visualization to a certain study material is similar to the one presented in an ITiCSE working group report about hypertextbooks (Rössling et al., 2006). We think that the visualization of the mature version of the algorithm could also be used without the debugging phase just for presenting the concepts algorithm, pseudo code, and flow chart. This way the example would be less challenging and the activation of the student would be left only to the pop-up questions.

The best possibility for activating students would be to make them correct the bug or build a completely new correct algorithm after finding the bug from the premature version of the algorithm. This can, however, be very challenging for a novice student so we did not try it. It would be an interesting future work idea to build a visualization tool where the student could build the correct algorithm by modifying the flow chart. Another idea for future work is that we could implement different kinds of premature algorithms. There could be easier and more difficult bugs for the debugging task.

6 Acknowledgments

Special thanks to Prof. Thomas Naps (University of Wisconsin, Oshkosh) for his enormously useful guidance with visualizations in common and help with the JHAVÉ visualization tool. Nokia Foundation has partly funded this work.

References

John C. Giordano and Martin Carlisle. Toward a more effective visualization tool to teach novice programmers. In *SIGITE '06: Proceedings of the 7th conference on Information technology education*, pages 115–122, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-521-5. doi: http://doi.acm.org/10.1145/1168812.1168841.

T. McGill and S. Volet. A conceptual framework for analyzing students' knowledge of programming. *Journal on research on Computing in Education*, 29(3):276–297, 1997.

António José Mendes, Anabela Gomes, Micaela Esteves, Maria José Marcelino, Crescencio Bravo, and Miguel Angel Redondo. Using simulation and collaboration in cs1 and cs2. *SIGCSE Bull.*, 37(3):193–197, 2005. ISSN 0097-8418. doi: http://doi.acm.org/10.1145/1151954.1067499.

Thomas L. Naps, James R. Eagan, and Laura L. Norton. JHAVé - An environment to actively engage students in web-based algorithm visualizations. *ACM SIGCSE Bulletin , Proceedings of the thirty-first SIGCSE technical symposium on Computer science education SIGCSE '00*, 32(1):109–113, 2000.

T.L. Naps, G. Rössling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, and J.A. Velazquez-Iturbide. Exploring the role of visualization and engagement in computer science education. *SIGCSE Bulletin*, 35(2):131–152, June 2003.

Guido Rössling, Thomas Naps, Mark S. Hall, Ville Karavirta, Andreas Kerren, Charles Leska, Andrés Moreno, Rainer Oechsle, Susan H. Rodger, Jaime Urquiza-Fuentes, and J. Ángel Velázquez-Iturbide. Merging interactive visualizations with hypertextbooks and course management. In *ITiCSE-WGR '06: Working group reports on ITiCSE on Innovation and technology in computer science education*, pages 166–181, New York, NY, USA, 2006. ACM. ISBN 1-59593-603-3. doi: http://doi.acm.org/10.1145/1189215.1189184.

Clifford A. Shaffer, Matthew Cooper, and Stephen H. Edwards. Algorithm visualization: a report on the state of the field. In *SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education*, pages 150–154, New York, NY, USA, 2007. ACM. ISBN 1-59593-361-1. doi: http://doi.acm.org/10.1145/1227310.1227366.

John T. Stasko and Christopher D. Hundhausen. Algorithm Visualization. In *Computer Science Education Research*, pages 199–228, The Netherlands, Lisse, 2004. Taylor and Francis.

Experiences on Using TRAKLA2 to Teach Spatial Data Algorithms

Jussi Nikander, Juha Helminen, Ari Korhonen
Helsinki University of Technology
Department of Computer Science and Engineering

`{jtn, jhhelmi2, archie}@cs.hut.fi`

Abstract

This paper reports on the results of a two year project in which visual algorithm simulation exercises were developed for a spatial data algorithms course. The success of the project is studied from several point of views, i.e., from developer's, teachers's, and student's perspective. The amount of work, learning outcomes, and feasibility of the system has been estimated based on the data gathered during the project. The results are encouraging, which motivates to extend the concept also for other courses in the future.

1 Introduction

Spatial data algoritms (SDA) are algorithms that work on location data, such as geographic data. These algorithms are an integral parth of geoinformatics, a branch of science where information technology is applied to cartography and geosciences. Since geoinformatics often uses a lot of illustrations, such as maps and other diagrams, the students are well acquainted with visualization. Thus, software visualization is a natural tool for teaching SDA and its applications to geoscientists.

Tracing exercises are a teaching method employed by many instructors in which the students trace an algorithm by keeping track of the changes in data structures while stepping through the algorithm. In *visual algorithm simulation exercises*, this procedure is supported by a graphical learning environment that provides visualizations, which eliminates the burden of drawing the same data structure over and over again. TRAKLA2 is a learning environment that utilizes visual algorithm simulation to deliver tracing exercises to students. The system can automatically assess the solutions and give feedback on the correctness of the simulation.

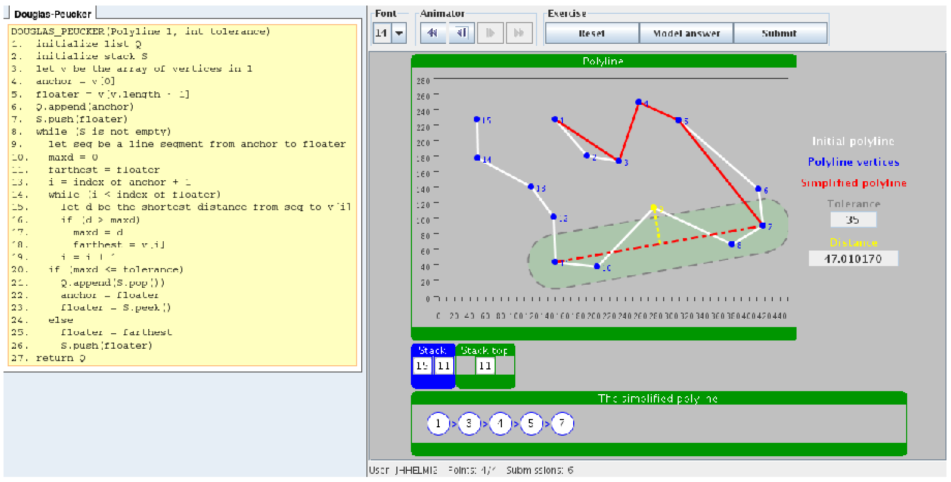
Previous studies (Korhonen et al., 2002; Laakso et al., 2005) have shown that there are no differences in the learning results between a test group that solves visual algorithm simulation exercises on the web and a control group that solves tracing exercises in a classroom as long as the assignments are the same. This is an encouraging result that motivated us to apply visual algorithm simulation exercises for spatial data algorithms. The challenge was to extend the TRAKLA2 system to cover a new area of algorithmics. The main research question was whether applying the visual algorithm simulation concept to new application areas is worth the effort spent on extending the framework. This question has three separate aspects: the developer's, teacher's and student's point of views. From the developer's perspective, we are trying to find out under what circumstances this kind of project pays off. In particular, how much time and effort does it take to extend the system to cover a new application area? And, what kind of challenges we expect to encounter during implementation and in the design of new visualizations? From the teacher's point of view, we are interested in the learning results: the level of learner engagement (i.e., how much work they did) and the overall performance in the final examination (i.e., the correlation between the exercises and the achieved learning results). Finally, from the student's point of view, by interviewing them, we seek to find out how this new technology affects the learning process.

After the implementation (i.e., extending the application framework and implementing the exercises) the use of the system requires only minimal effort. Thus, this research aims not to answer whether somebody should repeat our experiment, but to answer whether the concept of visual algorithm simulation exercises is mature enough to be applied to other disciplines than data structures and algorithms. The main challenge is that it requires the instructor to be *proactive* rather than *reactive*, that is, the workload is much higher before the course than in traditional teaching in which the work (grading the exercises) is done during or after the course. This might explain the slow adoption of such systems in every day teaching.

In this paper, we report the results on our experiences from a two year project in which we implemented spatial TRAKLA2 exercises. Overall, taking into account that this was partially a research project and partially a course development project, it was a success. However, from a single instructor's point of view, two years appears to be too short a time to reap the benefits of automatic assessment, unless the system is used on a large course with hundreds of students. Furthermore, student feedback suggests that they want more visual algorithm simulation exercises on the course.

2 Spatial Extension to TRAKLA2

TRAKLA2 is a framework for automatically assessed visual algorithm simulation exercises. The system is built on the concept of visual algorithm simulation. The user can construct animations of algorithm execution via GUI actions such as dragging and dropping data items. In the exercises, students can freely step backward and forward in these animations. They can also reset the exercise and get new randomly generated input. A model answer is also available as an animation. The exercises are deployed as Java applets within a web environment. Figure 1 shows a screenshot of the TRAKLA2 environment.



```
DOUGLAS_PEUCKER(Polyline l, int tolerance)
  initialize list Q
  initialize stack S
  let v be the array of vertices in l
  anchor = v[0]
  floater = v[v.length - 1]
  Q.append(anchor)
  S.push(floater)
  while (S is not empty)
    let seg be a line segment from anchor to floater
    maxd = 0
    farthest = floater
    i = index of anchor + 1
    while (i < index of floater)
      let d be the shortest distance from seg to v[i]
      if (d > maxd)
        maxd = d
        farthest = v[i]
      i = i + 1
    if (maxd <= tolerance)
      Q.append(S.pop())
      anchor = floater
      floater = S.peek()
    else
      floater = farthest
      S.push(floater)
return Q
```

Tolerance 35
Distance 47.010170

User: JHHELMI2 Points: 4/4 Submissions: 6

Figure 1: TRAKLA2 exercise for the Douglas-Peucker line simplification.

The SDA extension to TRAKLA2 consists of three major components: spatial primitives, spatial data structures and spatial data visualization. In addition, implementations of geometric functions and algorithms for processing spatial data and generating random input for the algorithms are required. For now, the extension is limited to 2-dimensional data and exercises.

The primitives on which the spatial data structures and algorithms operate are geometric entities, such as points, lines, and polygons. This multidimensional data can be stored in the form of key values derived from their geometric properties. In this respect, a spatial data element is an ordered list of values, a tuple, which in the spatial context represents a specific geometric entity. For example, a polygon can be stored as an ordered list of the coordinates of its vertices.

In previously existing TRAKLA2 exercises, the data is 1-dimensional, single-letter characters and integers. Their relationships are simple, and obvious without any additional visualization. The SDA extension introduces two new visualizations for use with spatial data.

First, the *tuple representation* of a spatial primitive can be visualized in simple and exact tabular form. Despite being precise, this data-intensive approach does a poor job of conveying the geometric nature of and the relationships between the data elements. Thus, a different visualization is needed to illustrate the spatial attributes. This is the *area visualization*, which is fundamentally a 2-dimensional coordinate plane, onto which the geometric entities of the spatial primitives are drawn (Nikander and Helminen, 2007).

Furthermore, spatial data is represented at three different levels of visualization. First, at the exact data item level, the data is shown as tuples of values based on their geometric properties. Second, at the data structure level, we have canonical visualizations specific to the data structures, where spatial data items are labeled with unique identifiers or shown as tuples. At the highest level of abstraction, the conceptual relationships and spatial attributes are illustrated with the area visualization showing the data items and the structure with possibly additional visual cues to represent algorithm constructs, such as a sweep line or an in-circle test. Figure 2 depicts two visualizations of the same R-tree containing polygons. The area visualization shows the areas covered by the polygons and tree nodes. The tree visualization shows how the tree is organized.

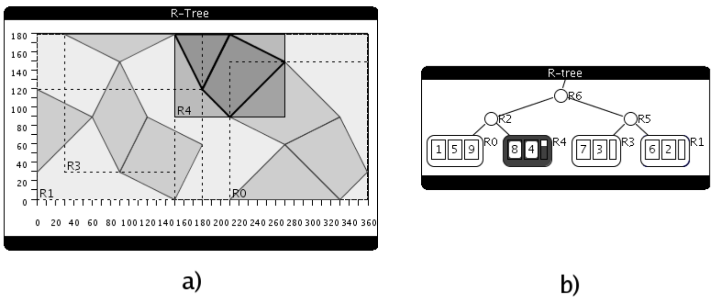


Figure 2: R-tree a) drawn as an area and b) drawn as a tree.

A particular challenge with the SDA exercises is the generation of spatial data for use as input to the algorithms. For the visualization to contribute to learning, the generated data sets must be clear and visually pleasing. There has to be enough spacing between the geometric entities to distinguish and select them. Also, labels must not overlap too much. In addition, each exercise has very specific constraints for the data, which makes it hard to create a generic data source. Therefore, most exercises have their own mechanism for generating data such as simple polygons.

Currently, the SDA extension comprises 12 visual algorithm simulation exercises. All of the exercises can be found on our web site[1]. The exercises fall into two categories: *tracing exercises* and *open tracing exercises* (Korhonen and Malmi, 2004). In tracing exercises, an algorithm and its input is given, and the student's task is to work out the output and construct an animation of the algorithm's progress by simulating it step-by-step. The simulation is done by emulating the algorithm's operations by dragging, dropping and selecting data items, as well as invoking operations via other GUI components, such as buttons. The user-created animation is then compared against a correct sequence of states created by an actual execution of the algorithm, and graded based on the number of matching steps. The goal is to give a conceptual understanding of the algorithm.

In open tracing exercises, the algorithm is not strictly specified and therefore the exercises are more exploratory in nature. The student is given a goal, such as creating the Delaunay triangulation (Okabe et al., 2000) of a point set, and the means to achieve it, *e.g.*, adding edges between vertices and to carry out the in-circle test. The student can then interactively explore the structures by making modifications to them and observing the changes. Finally,

[1]http://www.cs.hut.fi/Research/TRAKLA2/exercises/index.shtml

the correctness is assessed by comparing the final state with the expected outcome. The overall goal is not to teach some specific algorithm but a concept, such as the min-max-angle criterion related to Delaunay triangulations.

3 Results

The spatial TRAKLA2 exercises were first adopted in the spring 2007 spatial data algorithms course and were used again on the spring 2008 course. The course is aimed at third year geoinformatics students. In addition to spatial exercises, some other TRAKLA2 exercises were also included on the course. These exercises covered data structures important for the understanding of spatial algorithms. The details of the course are shown in Table 3. The table tells how many students started the course, how many participated in the final examination, how many TRAKLA2 exercises there were on the course (and how many of those were spatial exercises), the total number of TRAKLA2 submissions, and average score gained (compared to maximum).

Table 1: Basic course data for the Spatial Data Algorithm course

year	**# students**	**# in exam**	**# exer. (SDA)**	**# subs**	**avg. score**
2007	16	9	15 (9 SDA)	723	67%
2008	20	16	16 (10 SDA)	1036	83%

Before TRAKLA2 was introduced, the spatial data algorithms course consisted of combined lecture and studio sessions, a programming project and an exam. In the studio sessions, the students worked in groups and studied spatial algorithms on a conceptual level. In the programming project, the students implemented one of the algorithms discussed on the course. The exam was held after the last lecture. TRAKLA2 was added without reducing any other requirements on the course.

In the course, each student got personalized input for all TRAKLA2 exercises. The exercises were divided into rounds with 1–3 exercises in each. In order to pass the course, the students needed to gain at least 50% of the points from each round. Students were not penalized for returning exercises late. The TRAKLA2 exercises did not affect the students' final course grade.

3.1 Developer's Point of View

The project for creating the TRAKLA2 spatial extension was started in February 2006. The first exercises were introduced in January 2007. At that time, 9 spatial exercises were used in the course. In the project, a total of 12 spatial data exercises have been implemented, and 11 of them have been used in practice. The one untested exercise was finished so late that it could not be added to the spring 2008 course. Several people participated in the project, but most of the time there were two people working on it. A crude approximation of the amount of work put into the project is 10 person–months.

The implementation work itself can be divided into two separate tasks: extending the exercise framework and implementing the exercises. Most of the work in the project went into the design and implementation of the exercises. Less than 20% of the total effort consisted of extending the framework. Based on our experience, the implementation of a spatial exercise typically was more time–consuming than basic data structure or algorithm exercise.

3.2 Teacher's Point of View

Data on the students' learning results was collected from TRAKLA2 exercises and the course exam. TRAKLA2 kept record of each student's final points and number of submissions to each exercise. The data was collected the same way in both years.

Statistical analysis was used to see if the students' TRAKLA2 performance were a good indicator of their exam results. The analysis was made both between TRAKLA2 results and exam results as a whole, as well as between TRAKLA2 results and a single exam question that covered R–trees (Guttman, 1984). In both years, there were two TRAKLA2 exercises and an exam question about R–trees. Summary of the results can be found in Table 3.2.

The results in Table 3.2 are divided into three categories. First, Course info indicates the number of students who passed the TRAKLA2 exercises and participated in the first final exam. The second category contains characteristics of linear regression analysis between TRAKLA2 results and exam results as whole, and the third category regression results for the R–tree exercises. For the linear regression ρ (correlation), adjusted R^2, and its statistical significance are reported.

Table 2: Learning results on the spatial data algorithms course

Course info		**Whole exam**			**R-trees**		
Year	N	ρ	adj. R^2	p	ρ	adj. R^2	p
2007	9	0,83	0,65	0,005	0,85	0,69	0,003
2008	16	0,48	0,18	0,058	0,60	0,31	0,015

There was a significant change in the TRAKLA2 results between the years 2007 and 2008. The change, however, can be explained by the modifications made to TRAKLA2 exercises between the two courses. One exercise was removed, and two new ones were added. The exercise removed after 2007 course was one of the hardest exercises (average score 54%), while the exercises added to 2008 course were among the easiest (average score 98%). The exam results were similar in both years.

As can be seen in Table 3.2, in both years, there was a strong correlation between students' performance in TRAKLA2 and in the course exam, especially for the R–tree exercises. All results were statistically significant ($p < 0,02$) except the correlation between TRAKLA2 results and exam results in 2008, which was almost significant ($p = 0,058$). Similar results have been observed in data structures and algorithms courses (Korhonen et al., 2002).

3.3 Student's Point of View

In spring 2008, we carried out interviews to learn about student experiences with the spatial exercises. We used the interview guide approach (Patton, 2002), where the interviewer has an outline of topics to be covered, but may vary the wording and order of the questions to some extent. We interviewed a total of 4 students (two males, and two females) with two different nationalities, thus two of the interviews were in Finnish and two in English. The age (22 to 28 years) and background of the interviewees varied as they had had their education in different countries and universities/high schools.

Two main paths of questioning were explored: what was the student's subjective opinion of the system and what did they think about it compared to other teaching methods and learning materials? For each question, we also had a set of follow-up questions that expanded on the subject to help us in getting more informative responses.

All interviewees found the system to be beneficial and also thought it was an important learning tool that should continue to be utilized on the course. Furthermore, they felt that compared with the lectures, they had learned more details about the algorithms from the TRAKLA2 exercises, but lectures were still considered to be the most important learning method. Compared with reading an article, they felt that it was faster to grasp the idea from TRAKLA2 exercises, yet before attempting the exercises, students thought they should have some basic knowledge of the algorithm first.

The main benefits mentioned were the visual appearance and interactivity. Students felt that they were able to better make sense of an algorithm's principles by observing animations of them. Indeed, all said that the model answer animation was very helpful. However, it was unclear from the responses whether they truly thought that it aided in learning the algorithm or that it simply helped them to succesfully solve the exercise. Similarly, the students found that the simulation aspect of the system, which allowed them to actually practice the algorithm, makes it easier to memorize the algorithm's principles. One student pointed out that by observing and manipulating a visualization, you can actually see how the algorithm progresses, unlike in a programming exercise in which you need to implement the algorithm, but typically cannot observe its execution very well.

The issues with the system were related to the automatic feedback and exercise-specific simulation interfaces. When an incorrect solution is submitted, the system replies with the number of correct steps from the beginning of the animation. All felt that while you may this way find the first error, searching for it by stepping through the long model answer animation is cumbersome. In addition, students thought it was unfair that the system does not give any points for the correct execution of the algorithm after the first mistake, and that you cannot continue to solve the exercise from where you made the mistake, leaving you to trace the algorithm over and over from the beginning. Moreover, they complained about the mappings between algorithm operations and simulation interface actions such as pushing buttons and dragging data items. According to the students, this exercise-specific behaviour was not documented well enough and as one student phrased it, it took some effort to learn how to get the things move in the way they are supposed to. The pseudocode included in every TRAKLA2 exercise was considered either completely useless or very useful depending on the interviewee's familiarity with programming. A student having strong programming skills found the pseudocode useful, while less skilled students did not pay attention to it.

4 Discussion

4.1 Developer's Point of View

In this project, the design and implementation of spatial data exercises was found to take significantly more time and effort than most of the data structure exercises done prior to this experiment. The most important factor was that spatial algorithms were mostly unfamiliar to the development team in the beginning of the project. Thus, for each exercise, the algorithm in question needed to be studied in order to comprehend it on a level required to implement, visualize and teach it. This is a time–consuming task. In addition, the use of more complex visualizations increased the amount of required effort. For example, the use of area visualizations is not as straight–forward as using basic data structure level visualizations. The area view is flexible, but specifying how to visualize data using it is more time–consuming than when using canonical data structure views.

Two years and 10 person–months for creating just 12 exercises may seem to be a lot effort for quite a little gain, at least from a single instructor's point of view. However, once the exercises have been implemented, using them on a course requires only a very small, constant amount of effort, regardless of the size of the course or the number of exercises used on it. This proactive approach is in direct opposition to the traditional reactive approach of manually assessed classroom exercises. In manual assessment, most effort goes into assessing the students' answers, and it is proportional to the number of exercises and the size of the course. In addition, the comparison is not straight-forward as this number would be even greater if resubmissions could be allowed (that is typically not the case, due to the fact that it increases logistical problems and work load too much). Furthermore, manual assessment needs to be done on each iteration of the course. Thus, the longer the automated exercises are in use, the more benefit they offer. Eventually, automated exercises will require less overall effort than traditional classroom exercises, since the job needs to be done only once.

The time it takes for this to happen depends on the number of exercises and students on the course. In large courses (many students), this time limit is reached very soon (in a couple of years). For example, in a data stuctures and algorithms course, some 500 students make some 50.000 submissions with 40 exercises (approximately 2.5 resubmissions/exercise). Thus, it is easy to see that the system pays off very soon in this case (it is hard to find enough personnel to grade this amount of submissions within feasible time limits, not to mention that the work is not very pleasant). However, it is more difficult to define a precise time limit for smaller courses such as spatial data algorithms. One thousand submissions (in 2008) is quite an easy task to handle even by a single instructor and takes probably only a couple of days to grade manually. Even then, we believe the investment is worthwhile if we take also the teacher's and student's point of views into account.

4.2 Teacher's Point of View

The learning results show that the correlation between TRAKLA2 exercises and exams on the spatial data algorithms course is strong and statistically significant. It is even stronger than the correlation in the basic data structures course (Korhonen et al., 2002). However, this is likely to be an artefact of the small sample size on the SDA course. With the smallest sample being only 9 persons, the results of a single person are likely to affect the overall correlation quite a lot. Despite this, it seems that TRAKLA2 exercise results are a good indicator of exam results.

One interesting aspect of the TRAKLA2 results is how much effort students put into them. On the data structures and algorithms course, where TRAKLA2 exercises directly affect the course grade, a large portion of the students get maximum points, even when no further benefit is gained after getting 90% of the points (Malmi et al., 2005). On the spatial data course, TRAKLA2 points have no effect on the course grade. Nonetheless, students seem to want to do as many TRAKLA2 exercises as they can. Even if this is nowhere near the amount of effort students use on the data structures and algorithms course, this indicates that most students are willing to do more work than required. This supports the opinion that this kind of exercises are not only well accepted by the students but also motivate students to do more work than in traditional teaching setups.

4.3 Student's Point of View

The results from the interviews indicate that the students feel that they benefit from the system, even if this is not obvious from the quantitative results. In fact, the overall attitude towards the system was very positive and when asked to give it a grade from 0 to 5, all interviewed rated the system at around 4. During the course of the interviews, most interviewees also asked for more exercises that could cover other algorithms discussed in the course.

In contrast to the majority opinion, one student who had a more advanced background in computer science than the others, felt that the exercises were not a terribly important learning tool for him. Yet, even he said that one or two of the exercises had actually straightened out existing misconceptions he had about the algorithms in question. That is, via the visual algorithm simulation, he was able to learn some details of the algorithms and adjust his incorrect pre-existing mental model of them without delving into implementation.

The interviewees also brought up the notion that this kind of more involved learning with interactive visualizations would lead to a more lasting effect than simply reading. Essentially, they felt that the exercises engaged them more effectively, and they would remember the lessons longer. Thus, the student opinions agree with the findings of Naps et al. (2003).

5 Conclusions

In this paper we have described the experiences we had implementing and using the TRAKLA2 system on a spatial data algorithm course. The implementation of the system was a more challenging task than originally anticipated. One significant contributor to the large amount of effort required was that we were not very familiar with the topic. The effect of the SDA exercises on student learning is similar to the effect of TRAKLA2 exercises on a basic data structures course. Furthermore, the students' attitude to the system seems to be generally positive and they believe that the system helps them to learn.

From the student's point of view, the weaknessess of the system were the quite minimal feedback on incorrect solutions and complex exercise-specific simulation interfaces. Improving the quality of feedback is currently a topic of ongoing research (Seppälä et al., 2005). The difficulty of designing intuitive interfaces for visual algorithm simulation is something that turned out to be more challenging than expected based on previous experience. The mapping of complex mathematical operations to simulation interface actions for the purpose of creating automatically assessed algorithm exercises is also open to more research.

Despite the amount of effort required for creating the exercises, we consider the project a success. Visual algorithm simulation can be used for teaching topics besides basic data structures and algorithms. Furthermore, after the initial work put into the implementation, the system can be used effortlessly. The longer it is in use, the more effort it saves.

References

Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57, New York, NY, USA, 1984. ACM Press. ISBN 0-89791-128-8.

Ari Korhonen and Lauri Malmi. Taxonomy of visual algorithm simulation exercises. In Ari Korhonen, editor, *Proceedings of the Third Program Visualization Workshop*, pages 118–125, The University of Warwick, UK, July 2004.

Ari Korhonen, Lauri Malmi, Pertti Myllyselkä, and Patrik Scheinin. Does it make a difference if students exercise on the web or in the classroom? In *Proceedings of The 7th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education, ITiCSE'02*, pages 121–124, Aarhus, Denmark, 2002. ACM Press, New York.

Mikko-Jussi Laakso, Tapio Salakoski, and Ari Korhonen. The feasibility of automatic assessment and feedback. In *Proceedings of Cognition and Exploratory Learning in Digital Age (CELDA 2005)*, pages 113–122, Porto, Portugal, December 2005. IEEE Technical Committee on Learning Technology and Japanese Society of Information and Systems in Education.

Lauri Malmi, Ville Karavirta, Ari Korhonen, and Jussi Nikander. Experiences on automatically assessed algorithm simulation exercises with different resubmission policies. *Journal of Educational Resources in Computing*, 5(3), September 2005.

Thomas L. Naps, Guido Rößling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodgers, and J. Ángel Velázquez-Iturbide. Exploring the role of visualization and engagement in computer science education. *SIGCSE Bulletin*, 35(2):131–152, June 2003.

Jussi Nikander and Juha Helminen. Algorithm visualization in teaching spatial data algorithms. In *11th International Conference Information Visualization IV2007*, pages 505–510. IEEE Computer Society, July 2007. URL `http://www.graphicslink.co.uk/IV07/`.

Atsuyuki Okabe, Barry Boots, Kokichi Sugihara, and Sung Nok Chiu. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams.* John Wiley & Sons, 2000.

M.Q. Patton. *Qualitative Research and Evaluation Methods.* Sage Publications, 2002.

Otto Seppälä, Lauri Malmi, and Ari Korhonen. Observations on student errors in algorithm simulation exercises. In *Proceedings of the 5th Annual Finnish / Baltic Sea Conference on Computer Science Education*, pages 81–86. University of Joensuu, November 2005.

Using Graphviz as a Low-cost Option to Facilitate the Understanding of Unix Process System Calls

Miguel Riesco, Marian Diaz Fondon, Dario Alvarez
Dep. of Computer Science, University of Oviedo, Spain

albizu@uniovi.es

Abstract

Unix system calls to create and execute processes are usually hard to understand for novice students. In this paper we show how we use graphviz to generate a graphical representation of the behaviour of these system calls to facilitate the comprehension of this important part in the learning of the Unix operating system.

1 Introduction

When teaching Operating Systems, it is usual to employ graphics to better illustrate the different aspects involved. Thus, graphs representing the modules of the operating system, the lifecycle of processes, or the message-based process communication are common.

One of the topics appearing frequently in this subject is the Unix operating system, from diverse points of view: internal structure, command-line user, or system programmer. In the systems programmer view, the API calls of the system are studied, and students develop programs using these calls. Here it is also usual to resort to graphics supporting the explanation of how the system calls work as the program using them is running, to show the dynamic behaviour of a program using system calls, and to visualize the evolution of the data structures involved.

These graphics are generally created manually by the teacher, or taken from text books (where, in turn, they were created by the author manually). Although these static graphs are useful, some kind of animation where the evolution of processes could be seen would be better.

As far as learning the Unix system calls is concerned, we teachers have been explaining how processes are created by using drawings in the blackboard, providing interactive animation by drawing and erasing as we explain. It is important to have some kind of graphic to help students comprehending this part of the API, as this is not something students assume as natural: to the peculiar behaviour of these calls we have to add the fact that there are a number of processes running concurrently, thus making the comprehension more difficult.

This was detected before (Vogt, 1997), but the solution presented is not readily accessible, needs specific systems to generate and visualize animations of processes. A different aproach was done by (Robbins, 2002). However, it is aimed at studying the interaction between *fork* and *dup* system calls. Apart from these, we have not found more systems deemed adequate for our needs. In this paper we present the work we are developing to automatically generate, from a real program that uses process-related system calls, graphics illustrating its behaviour, supporting the teachers explanation of the topic.

2 POSIX process services

The POSIX API offers a reduce set of process-related system calls. There are other ways to get information about processes, but most of the functionality lies in four system calls:

- *fork*: clones the process making the call (the clone is a child process).
- *exec*: changes the program executing the process (does not create a new process, it just substitutes the code the process is running for another one).
- *wait*: waits for the termination of a child process.

- *exit*: Terminates the execution of a process.

Students have problems grasping the behaviour of these system calls, as the natural way would be to have a call creating a new process running the program passed as an argument. To clarify it, graphics such as those in (Jesus Carretero and Perez, 2007) or (Robbins and Robbins, 2003), similar to what is show in figure 1, try to represent the behaviour of the program.

```
main() {
 pid_t pid;
 int i, n=4;
   for (i=0; i<n ; i++ ) {
     pid=fork();
     if (p==0) break;
  }
}
```

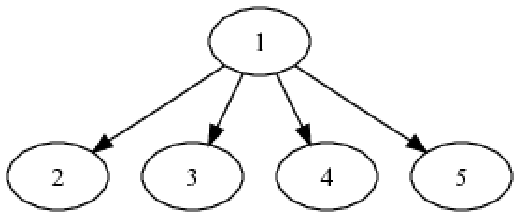


Figure 1 Graphic representing the creation of 4 processes

Even though for simple examples it is easy to create illustrative graphics, for complex programs it is not the case. Besides, trying to correctly represent the dynamic behaviour of processes can take lots of time.

Trying to solve this problem, and having experience with the graphviz tool (Ellson and et al., 2004) for other reasons, we applied it to the generation of graphics representing process-related system calls.

3 Instrumenting programs to generate graphviz graphics

We have developed a function library with names analogous to the original Posix calls (*myfork*, *myexec*, *mywait*, *myexit*). These functions maintain the functionality of the original system call (*myfork* does *fork*, *myexec* does *exec*, etc.), but they also generate one or more lines in graphviz *dot* format to graphically represent the system call. So for example *myfork* is implemented as follows:

```
int myfork() {
 pidf=fork();
 if (pidf==0) {
  ppid=getppid();
  pid=getpid();
  sprintf(cad,"d -> d;\n",ppid, pid);
  store(cad);
 }
 return (pidf);
}
```

Figure 2 shows the graphical representation of the four system calls. Each call gets this representation:

- *fork* is represented with a node for each process containing the process PID, with an edge indicating the father-child relationship.
- *exec* is represented in a similar way, adding the name of the execed program after the process PID.
- *wait* adds a dotted line from the child process to its father (showing the SIGHLD signal sent when the child process ends).

- *exit* uses a dotted node to represent the terminated process.

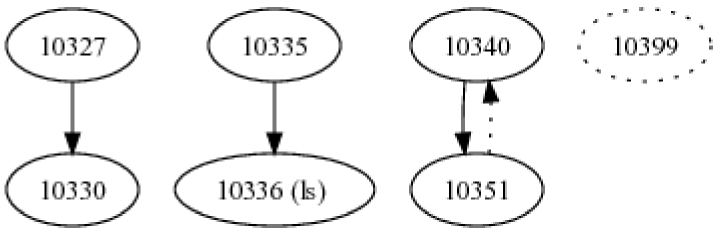


Figure 2 Graphical representation of the fork, exec, wait and exit system calls

To generate the graphical representation of the dynamic behaviour of a real program, the original system calls are replaced with the instrumented library functions mentioned before. A simple program or script does this. The original functionality is preserved and we get its graphical representation.

Once the process ends execution, a graphviz *dot* file is created with the representation of all the executed system calls. This is fed into the graphviz tool, which in turn will generate the corresponding graphic file. Figure 3 shows the graphical representation of a typical program that creates processes recursively.

```
main() {
 int i,p;

   for (i=0; i<3; i++)
     p=myfork();
}
```

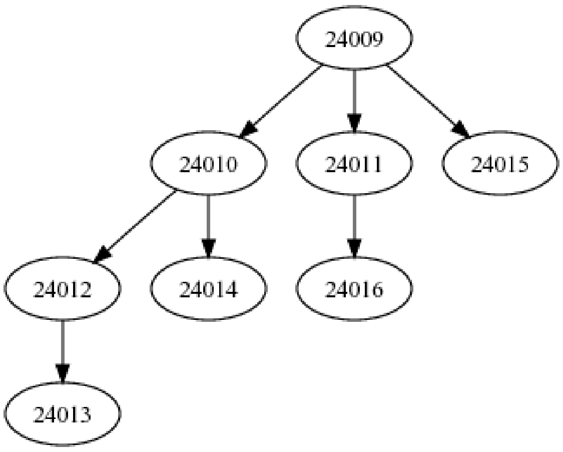


Figure 3 Creating processes recursively and their representation

This kind of graphic is useful in some situations, but we realized that it would be more interesting to analyze how the program evolves as each call is done, and not only at the end of the execution. We developed a second version that allows to visualize this dynamic behaviour. Using the same file generated after the execution of the program, we do the following process:

1. The global *dot* file is divided into as many files as lines are in it. The first one has the first line of the original file, the second one the first two lines, and so on. That is, the ith file has the behaviour of the first i calls (each call has a line in the global *dot* file).

2. n graphics files are created using the n files of the previous step using graphviz.

3. These graphics can be uploaded and then visualized using a web application. With simple controls, the user is able to see the graphical representation of the execution sequence, while showing the source code at the same time. With this support, the teacher can develop an detailed explanation of how the creation and destruction of processes is evolving. This is of great help for the student to grasp the topic.

```
void hdler (int a)
{
 int cr, pid;

 pid=miwait (&cr);

 signal(SIGCHLD, hdler);
}

main() {

 signal(SIGCHLD, hdler);
 pid=myfork();
 if (!pid)
   myexec("ps");
 pid=myfork();
 if (pid)
  do
   pidr=mywait(&cr);
  while (pidr!=pid);
 else {
   myexec("ls");
   myexit(-1);
 }
 myexit(0);
}
```

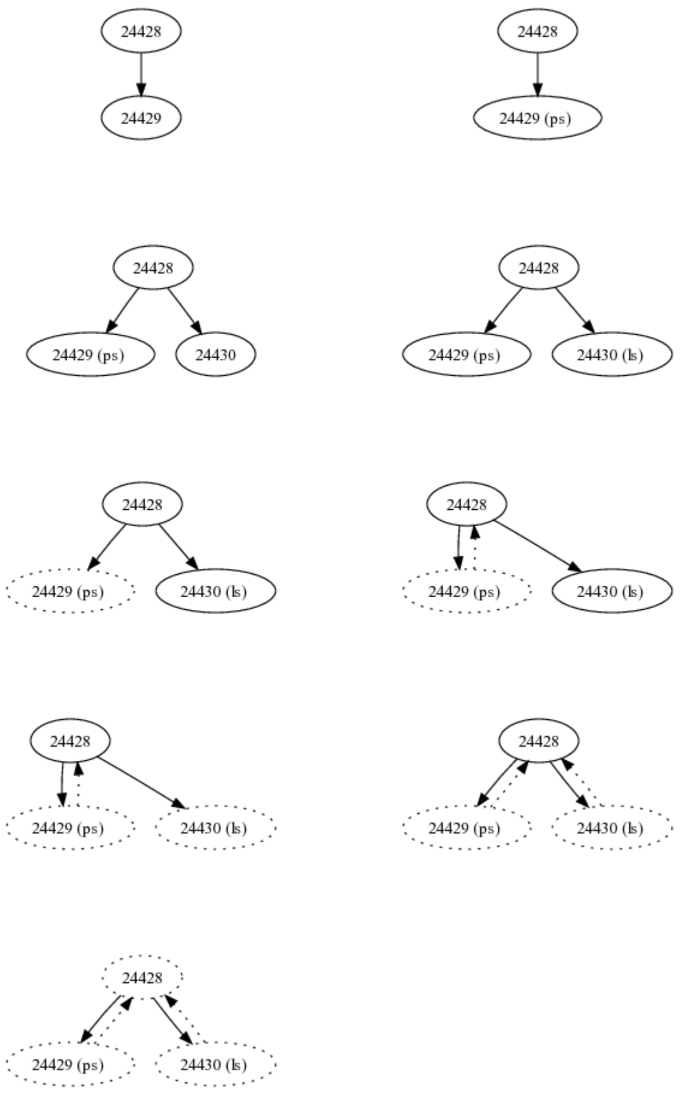


Figure 4 Showing the execution sequence of a program

4 Observed results

To date, we have used this method and tool mainly to support the teachers explanation of the topic, although it could also be used by students to analyze the execution of their own programs.

We do not have yet an exhaustive analysis of the impact in learning the topic. However, we can state two facts:

1. The teachers that have used the method are very satisfied, as they believe it facilitates the construction of examples, as well as the students comprenhension of the topics.

2. The students have accessed the web pages were the teacher-created examples are stored.

In the first week we had 874 page loads. We think this is quite a success, as there are 72 students and 8 different examples.

Therefore, we think this experience is positive. We have the intention of going more deeply and to apply these ideas to other topics in the subject.

5 Future Work

We plan to develop future work along these lines:

The priority is to improve the process of creating and publishing the graphics. Currently each step (creating the original file, dividing it into iterative files, generating the graphics, publishing graphics into web pages) is done independently and semi-manually. So, the first thing to do will be the packaging of the independent steps into one program that automatically generates and publishes graphics in one step from the data of the execution of a process.

Another issue it to develop a tool to visualize the evolution of the program in real time, allowing the user to interact with the running program stepping back and forth (a kind of simple graphical debugger).

Apart from the technical aspect, we have the intention to apply similar ideas to other parts of the subject that could benefit from this kind of support for the explanations. File management is a good candidate. In this case we would visualize the evolution of the data structures involved in performing each system service. We are also studying how to apply this to other topics such as concurrent programming or input/output management, although it is not that obvious.

6 Conclusions

In this paper we have shown a method to graphically represent the behaviour of the POSIX system calls for process management.

Using a free tool such as Graphviz it is even possible to represent simulations of the dynamic behaviour of the processes using these system calls.

The graphical representation of the behaviour help the teachers develop examples for a better explanation of the topic, while the students can analyze the behaviour of the programs in a more convenient way.

The experience has been a positive one. We are still working on the improvement of the graphics-creation process, and to apply the same ideas to other topics in the Operating Systems subject.

References

John Ellson and Emden R. Gansner et al. Graphviz and dynagraph static and dynamic graph drawing tools. Technical report, AT&T Labs - Research, Florham Park NJ 07932, USA, 2004. Also available as `http://www.graphviz.org/Documentation/EGKNW03.pdf`.

Felix Garca Jesus Carretero, Pedro de Miguel and Fernando Perez. *Sistemas Operativos, 2/e.* McGraw-Hill Interamericana, Inc., Madrid, Spain, 2007. ISBN 8448156439.

Kay Robbins and Steve Robbins. *UNIX Systems Programming: Communication, Concurrency and Threads (2nd Edition).* Pearson Education, Inc., Upper Saddle River, New Jersey, USA, 2003. ISBN 0-13-042411-0.

Steven Robbins. Exploration of process interaction in operating systems: a pipe-fork simulator. *SIGCSE Bull.*, 34(1):351–355, 2002. ISSN 0097-8418. doi: http://doi.acm.org/10.1145/563517.563476.

Carsten Vogt. Visualizing unix synchronization operations. *SIGOPS Oper. Syst. Rev.*, 31(3): 52–64, 1997. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/270555.270562.

Dynamic Evaluation Tree for Presenting Expression Evaluations Visually

Essi Lahtinen, Tuukka Ahoniemi
Tampere University of Technology, Department of Software Systems

essi.lahtinen@tut.fi, tuukka.ahoniemi@tut.fi

Abstract

Novice programmers have difficulties with their visual attention strategies when following program visualizations. This article presents work in progress on improving the user interfaces of visualization tools to support students in the visual attention problems. We introduce a user interface solution called the dynamic evaluation tree. The basic idea is to reduce the amount of separate windows of the user interface and thus make it possible to concentrate the visual attention more in one part of the screen. The dynamic evaluation tree has not been implemented yet but we think it would be beneficial to discuss the implementation in the workshop in advance.

1 Introduction

The user interfaces (UIs) of visualization tools are often build with a similar structure. Many tools seem to have the same components in their UI and similar locations for them. We feel that this is partly because the tools are offering multiple different perspectives for the example and no specific design principles are applied for the UI design. Components are in their places just because they always used to be.

However, the effectiveness of a visualization tool in its pedagogical point of view may suffer from the use of multiple components and their placement in the screen. This article references the results of an eye-tracking study by Bednarik (2007). Based on this, we suggest a new way of integrating some of the UI components and improving the user's target of visual attention.

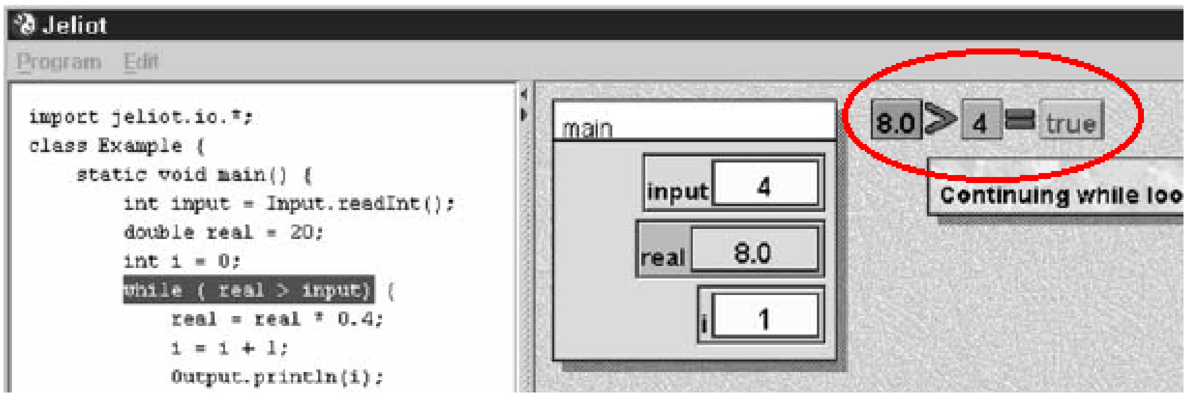
2 A Typical Layout of a Visualization Tool User Interface

A typical visualization tool presents multiple different kinds of actions in turns and parallel during the execution of a program or algorithm. The different kinds of actions can be for instance:

- Control reaches a new statement in the program code or algorithm.
- The values stored in the memory of the computer are referenced or changed.
- The values of expressions are evaluated.
- The program prints output and reads input.

A typical layout of a visualization tool UI presents different kinds of actions in different windows. Different tools have different names for the windows. We list some possibilities:

- *Code window:* Shows the program code or the algorithm that is executed. It typically illustrates the execution by highlighting the line of code or algorithm. It can also be named the algorithm window.
- *Memory window*: Performs most of the visual effects by drawing pictures of the variables and data structures and highlighting parts of the pictures. In the UI of Jeliot (Moreno et al., 2004), this window is named *the theater*.
- *Evaluation window:* This window is activated whenever the code window executes an expression. The values of the operands, the operator, and the value of the whole expression are shown here. An example is marked with a red circle in Figure 1.



```
import jeliot.io.*;
class Example {
    static void main() {
        int input = Input.readInt();
        double real = 20;
        int i = 0;
        while ( real > input) {
            real = real * 0.4;
            i = i + 1;
            Output.println(i);
```

Figure 1: The values of the variable in the expression executed in the code window is copied to the evaluation window for evaluation.

- *Console window:* Prints the input and reads the output of the program.

In addition to the most usual windows mentioned above, there can be other possibilities like *the annotation window* that explains the run of the program in writing (Virtanen et al., 2005). Also the visualization tools that allow user interaction, often have a window for the control buttons.

Depending on the focus of the visualization, it is possible that some of the windows are not necessary and are thus absent. For example, algorithm visualization tools might not need the evaluation window at all since they present the algorithm on a higher abstraction level than individual expressions. Examples of tools that do not need an evaluation window are presented by Malmi et al. (2004) and Naps et al. (2000). Sometimes one window of the visualization tool contains more than one kind of actions. For instance, the theater in Jeliot 3 (Moreno et al., 2004) actually includes both the memory window and the execution window.

3 Results of an Eye-Tracking Study

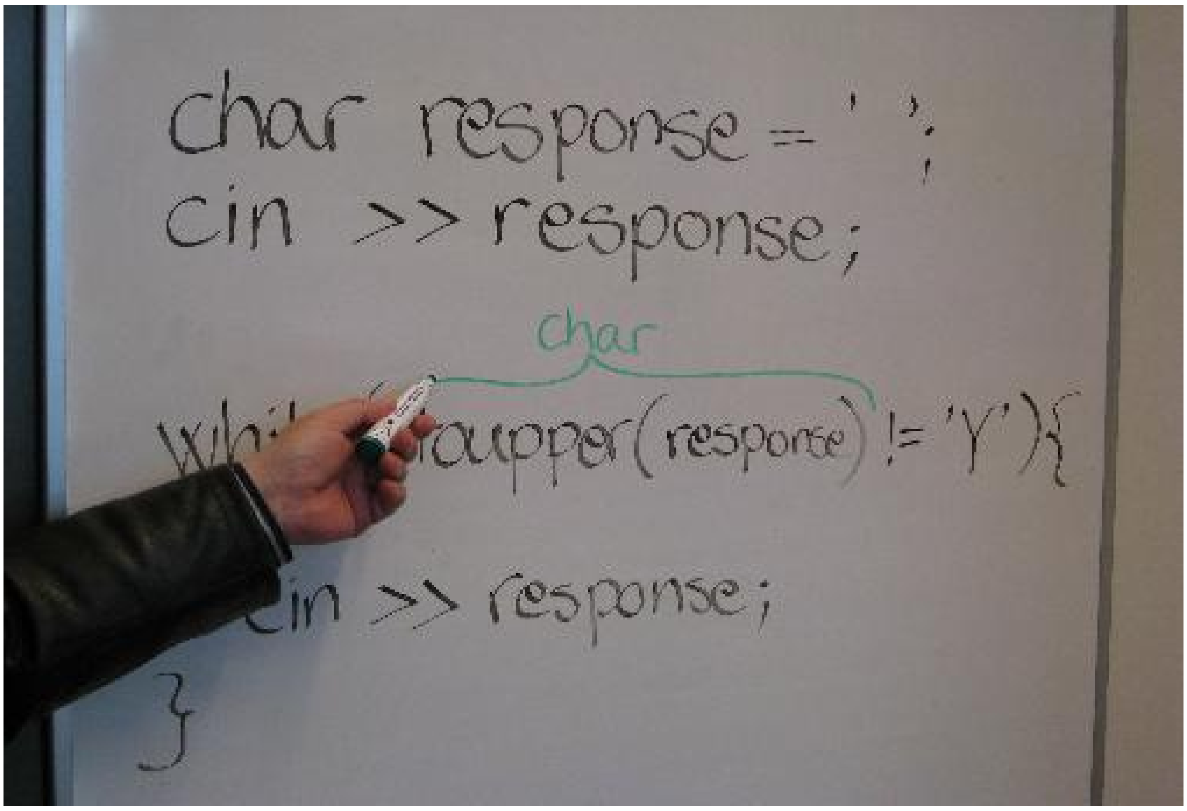
A study on the methods of analyzing visual attention strategies in programming by Bednarik (2007) is partly conducted by tracking the eye movements of programmers using a visualization tool Jeliot 3 (Moreno et al., 2004). The study describes the visual strategies of both expert and novice programmers.

According to the study, expert programmers can better follow the information shown parallel in different windows of the visualization tool. They are able to change their visual attention strategy during a session of using the visualization tool. At the beginning of a session, they often concentrate on the code window and later on in the session on relating the code with the presentations in the other windows. Specifically, the experts follow the code window of the visualization tool more comprehensively than novices.

In contrast, novice programmers use only a couple of visual attention strategies. They either switch their visual attention repeatedly between different windows or concentrate all the time on one of the windows. Since the target audience of visualization tools are mainly novice programmers, this kind of a visual attention strategy should be taken into account when designing the UI of the tools.

4 Dynamic Evaluation Tree

When teachers explain the execution of program code to students, they tend to annotate the program code using curly brackets above or below the code line as seen in Figure 2. It is an easy way to mark the value or the type of an expression. This kind of annotations can be used in many different ways. Table 1 gives some examples. The same notation has also been



```
char response = ' ';
cin >> response;

whi[illegible]oupper(response) != 'Y'){

   cin >> response;
}
```

Figure 2: Using curly brackets to explain on a white board.

used by Kumar (2005) in an applet that generates problems related to expression evaluation. Since this has proven to be a good way to illustrate the execution to students, we suggest it should be tried in a visualization tool too.

The expressions that the evaluation window presents are also shown in the code window since they are, of course, part of the statement in execution. Instead of separating the expression in the evaluation window, we suggest that the evaluation tree could be integrated into the code window. This would reduce the need to switch the focus of visual attention to the other side of the screen and thus should be easier to use for novice programmers.

One possibility for integrating the evaluation of expressions in the code window is to use a presentation similar to the curly brackets in Table 1. In this kind of illustration, the evaluation of an expression would of course not stay in the code window all the time, but appear step by step when the expression is executed and disappear when the execution proceeds to the next line. Thus we call it *the dynamic evaluation tree.*

An alternative way of presenting the data dynamically inside the code window would be to show it in a tooltip window when the cursor of the mouse is placed on an expression in the code window. This could be used in any visualization tool regardless of the use of the dynamic evaluation tree. However, this solution does not help if you want the evaluations to be shown as the user clicks the step button.

5 Discussion and Conclusions

Some tools for functional programming, e.g., WinHipe by Pareja-Flores et al. (2007) and DrScheme by Felleisen et al. (1998), use a similar idea of presenting evaluation of expressions dynamically. It is called the rewriting model of evaluation. In these tools the evaluation is presented as a sequence of rewritten expressions that shows the same information than the curly brackets in the dynamic evaluation tree. The rewriting model works in a very natural way in functional programming languages. WinHipe has also been evaluated and the students

Table 1: Some examples on the use of the curly brackets.

Purpose	Example
1. To show the value of a expression	$\texttt{double width = }\underbrace{\underbrace{\texttt{max(}\underbrace{\texttt{x}}_{1.5}\texttt{, 0)}}_{1.5}\texttt{ + }\underbrace{\texttt{marginal}}_{2.5}}_{4.0}\texttt{;}$
2. To show the type of an expression, especially useful in case of hierarchical data structures like a vector containing structs	$\underbrace{\underbrace{\underbrace{\underbrace{\texttt{studentlist}}_{\texttt{vector<Student>}}\texttt{.at(i)}}_{\texttt{Student}}\texttt{.name}}_{\texttt{string}}\texttt{.at(j)}}_{\texttt{char}}$
3. To explain some error situations	$\texttt{if(}\underbrace{\underbrace{\texttt{response}}_{\texttt{char}}\texttt{ != }\underbrace{\texttt{'Y'}}_{\texttt{char}}}_{\texttt{bool}}\ \Vert\ \underbrace{\texttt{'N'}}_{\texttt{char}}\texttt{)}$

have experienced that the tool is easy to use (Ángel Velázquez-Iturbide et al., 2008). However, we want the original program code to stay in the code window as it is and only add annotations inbetween the lines. Thus, the curly brackets used in a similar way than the rewriting model, suits our needs better than rewriting the expressions.

If the code window includes the dynamic evaluation tree, it is actually no longer merely a code window but more like a multipurpose window. This should not only reduce the constant switching of the focus of visual attention but also relate the evaluation directly to the code. When the evaluation is presented directly inside the actual program code, the user may be able to form a stronger mental association between the code and what it actually does. This way the user could hopefully learn how to read the code better than when using a separate evaluation window.

Since novice programmers have most problems with their visual attention strategies, the dynamic evaluation tree should be most helpful for them. After all, the biggest target audience of visualization tools is novice programmers. Thus, we feel that the idea is worth trying.

The dynamic evaluation tree has not been implemented yet but we are charting the possibilities to add it to the next version of an existing visualization tool, VIP (Virtanen et al., 2005). There will be some technical challenges in the implementation: the code window needs to be "stretched" vertically to make space for the curly brackets and the text above or below them. An other possibility could be to show the curly brackets in tooltip windows on top of the code window. However, this would obscure some code and could thus make the use of the tool difficult. We think it would be beneficial to discuss the implementation in the workshop prior to the implementation phase.

When the dynamic evaluation tree is implemented, it should be evaluated with an eye-tracking study to determine the possible aid with the visual attention strategies. The student still has at least the code window and the memory window to follow. An interesting possibility for further research could be to study whether it is possible to guide the student to develop better visual attention strategies by using the dynamic evaluation tree and other similar solutions in the UI.

One window fits all!

6 Acknowledgments

Nokia Foundation has partly funded this work.

References

Ángel Velázquez-Iturbide, Cristóbal Pareja-Flores, and Jaime Urquiza-Fuentes. An approach to effortless construction of program animations. *Computers & Education*, 50(1):179–192, 2008.

Roman Bednarik. *Methods to Analyze Visual Attention Strategies: Applications in the Studies of Programming.* PhD thesis, University of Joensuu, Joensuu, Finland, 2007.

Mattias Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. The DrScheme project: an overview. *SIGPLAN Not.*, 33(6):17–23, 1998. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/284563.284566.

Amruth N. Kumar. Results from the evaluation of the effectiveness of an online tutor on expression evaluation. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education*, pages 216–220, New York, NY, USA, 2005. ACM. ISBN 1-58113-997-7. doi: http://doi.acm.org/10.1145/1047344.1047422.

Lauri Malmi, Ville Karavirta, Ari Korhonen, Jussi Nikander, Otto Seppälä, and Panu Silvasti. Visual algorithm simulation exercise system with automatic assessment: TRAKLA2. *Informatics in Education*, 3(2):267–288, 2004.

A. Moreno, N. Myller, E. Sutinen, and M. Ben-Ari. Visualizing programs with Jeliot 3. *Proceedings of the International Working Conference on Advanced Visual Interfaces AVI 2004*, May 2004.

Thomas L. Naps, James R. Eagan, and Laura L. Norton. JHAVÉ - An environment to actively engage students in web-based algorithm visualizations. *ACM SIGCSE Bulletin , Proceedings of the thirty-first SIGCSE technical symposium on Computer science education SIGCSE '00*, 32(1):109–113, 2000.

Cristóbal Pareja-Flores, Jaime Urquiza-Fuentes, and J. Ángel Velázquez-Iturbide. Winhipe: an ide for functional programming based on rewriting and visualization. *SIGPLAN Not.*, 42(3):14–23, 2007. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1273039.1273042.

Antti T. Virtanen, Essi Lahtinen, and Hannu-Matti Järvinen. VIP, a visual interpreter for learning introductory programming with C++. *Proceedings of the Fifth Finnish/Baltic Sea Conference on Computer Science Education*, pages 129–134, November 2005.

Work in Progress: Automatic Generation of Algorithm Animations for Lecture Slides

Otto Seppälä, Ville Karavirta
Helsinki University of Technology
Department of Computer Science and Engineering

{oseppala, vkaravir}@cs.hut.fi

Abstract

Algorithm visualizations have not been widely adopted in teaching. One possible reason for this is that visualizations are often developed as standalone systems which can be difficult to integrate into lectures. Recently XML based formats for the two major presentation tools have been introduced. We present a method and a prototype implementation which allows creation of algorithm animations in the ODF format. This allows integrating the animation seamlessly within the lecture material.

1 Introduction

Algorithm visualizations have not been widely adopted in teaching. The problem of integration of visualization in self-study material has been studied in the context of HTML-based hypertextbooks (Rößling et al., 2006). HTML allows integrating animations as Java applets, Flash animations and videos to name a few.

Integrating visualizations in lecture material has been studied a lot less. In a survey study done by the ITiCSE 2002 Working Group (Naps et al., 2003) 79% of the educators listed the "time it takes to adapt visualizations to teaching approach and/or course content" as a substantial impediment for adopting new visualizations to be used on a course. These same difficulties were also found in a recent international survey by Lahtinen et al. (2007). Ben-Bassat Levy and Ben-Ari (2007) argue that tool developers often don't invest enough in how pedagogical software can be embedded into a curriculum. While the most commonly used presentation tools (Powerpoint, OpenOffice Impress) allow embedding e.g. video into the lecture slides, the lecturer is deprived of the possibility of adapting the animation to the specific lecture case.

For lecture use, the development in AA systems has focused on systems that can be used to give presentations. For example, Alvis (Hundhausen and Douglas, 2002), Animal (Rößling and Freisleben, 2002) and MatrixPro (Karavirta et al., 2004) all have features to support use on lectures. However, in most cases, animations created with the AA tools cannot be embedded into the lecture material since they are implemented as independent applications. Instead, they require the educator to switch between a number of programs during classroom presentation. In some algorithm visualization tools it might be possible to overcome this limitation by designing the lecture slides inside the algorithm visualization system. In most cases this is neither desired nor in any way feasible. The fact is that the presentation tools are much more sophisticated than the AA authoring tools.

Interaction provided by the algorithm animation has a major impact on the learning results (Hundhausen et al., 2002). However, in lecture situations, it has been shown that using animations or lecture slides are equally effective (Lawrence et al., 1994). Thus, automatic generation of lecture slides is a valid approach to promote algorithm animation in teaching.

In this on-going research, we explore the idea of lowering the barrier of integration. Our approach is to allow the teacher to create algorithm animations for the presentation tool he/she is already using on lectures. This allows the lecturer to insert the slides seamlessly within the other lecture material used in the same lecture. The recent introduction of XML formats for Microsoft PowerPoint and OpenOffice Impress has made it feasible to generate such formats with external tools. In this paper, we present a proof-of-concept of such a tool that can be used to produce animations of the Kruskal's algorithm in the form of Open Office Impress presentations.

2 Motivation

Think of a typical scenario on a data structures and algorithms course where a teacher is to give a lecture on e.g. Kruskal's algorithm. In order to use an existing visualizations for this algorithm, a suitable presentation must first be found. Here, the teacher can already run into problems. Recent research shows that the existing algorithm visualization are typically of poor quality and concentrate on the simpler algorithms (Shaffer et al., 2007).

Visualizations often concentrate on certain features of the algorithm and ignore others. This is often required to limit visual complexity. As research by Ben-Bassat Levy and Ben-Ari (2007) suggests, the pedagogical style of the visualization might not match the style of the teacher. The final choice of the visualization is likely to be a compromise which addresses most points the teacher wants to address during the lecture. In some cases the slides have to address limitations of the visualization as the visualizations themselves cannot be altered.

Some features of the visualization can also affect the structure of the slides. At least the visualization often has to be included as a "chunk" to avoid constantly switching between the presentation software and the visualization tool.

A whole another problem is that visualizations are often topic-specific tools. For a complete course, the teacher might have to use visualizations provided by a large number of developers. Naturally, this results in visualizations that do not share a uniform look and can cause unnecessary confusion for teacher as well as the students. Additional confusion can also be caused by (even subtle) variations in terminology.

According to results of an international survey, the classroom set-ups vary a lot, with the most typical set-up being a class with a computer and a ceiling-mounted projector (Naps et al., 2003). Thus, the teacher has to make preparations before the lecture. Applets should be downloaded and local webpages created to account for problems in network connections. In some cases the visualization software has to be installed, which might even be impossible in some lecture hall setups. The safest alternative is often to use a personal laptop for giving the presentations. However, PowerPoint or PDF slides typically work with whatever computer is available.

2.1 Why Now?

Until now, the tools for implementing the generation of slides have been missing. One reason for this is that in the past, the presentation tools have used closed proprietary formats. The development of open XML formats for the presentation tools has made implementing such tools much easier. Both of the most popular presentation tools, Microsoft PowerPoint and OpenOffice Impress, have open XML formats, Office Open XML and Open Document Format, respectively. These languages make it possible to generate presentations for these tools with reasonable effort.

In addition, Extensible Stylesheet Language Transformations (XSLT) (Clark (editor), 1999), a language designed to transform XML documents to other formats, makes transforming XML documents into the formats used by presentation tools simpler. The output of XSLT can be another XML format, HTML, or text. There are many tools available to do the XML transformations using XSLT. The most well-known XSLT processors for Java are Xalan and Saxon. This provides a simple way to implement transformations between languages.

3 Technical Description of Proof-of-concept Implementation

Our proof-of-concept implementation generates OpenOffice Impress slides visualizing the behavior of the Kruskal's minimum spanning tree algorithm. Kruskal's algorithm was chosen because it seems for some reason to be rather rarely visualized. The algorithm also uses an interesting data structure - the union-find structure usually implemented as a forest of father-linked trees.

Part of the idea behind the proof-of-concept was to use existing tools to automate parts of the process including graph layout software as well as XSL transformation packages. Figure 1 describes the architecture of our implementation. This whole process is ran using an Ant script. The first step is to execute a Java program. It first generates suitable input data for the Kruskal's algorithm and then executes the algorithm. For each state of the algorithm, the program creates graph descriptions in a format readable by the GraphViz (Ellson et al., 2002) graph layout package. The graph descriptions hold information on colors, line widths, labels etc. Our program also creates the father linked trees that form the union-find structure used by Kruskal's algorithm. These are also laid out by GraphViz. The third and last part of the visualization is the sorted list of graph edges. The Java program outputs a visualization of this list directly in OpenOffice Impress format. In addition to the visualization, the program outputs supplementary information as slide notes that are later combined to the visualization steps.

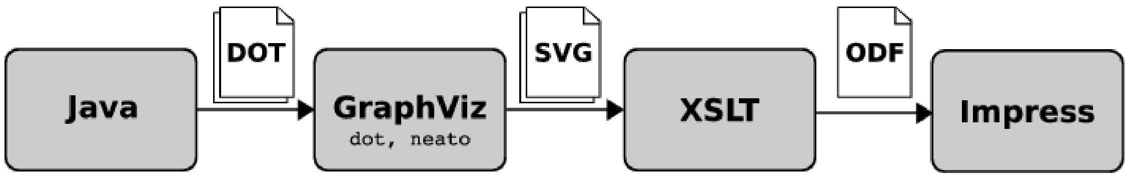


Figure 1: Architecture of our solution

The next step in creating the visualization is running the GraphViz programs dot and neato on the input data. GraphViz is an open-source software package intended for graph drawing. The system includes several different layout algorithms suitable for different kinds of graphs. Dot generates hierarchical layouts and is used to create the union-find graphics. Neato generates spring-based layouts and is used to layout the graph. In addition, the system supports several different output formats like png, jpeg, ps, pdf, and svg. In our program, we decided to use Scalable Vector Graphics (SVG) (W3C, 2001) since it is easy to process in the later steps. SVG is an XML language targeted for describing graphics.

The last step converts the SVG-files into corresponding Impress snippets, which are combined with the parts created directly by the Java code. This is done with XSLT which transforms the multiple SVG-files into one Open Document Format (ODF) presentation. The XSL-stylesheet is divided into two parts that handle different parts of the input. The first one generates the main ODF-document creating the needed styles and pages. It combines each slide from the multiple input files (tree, graph, notes, and edge list). For the SVG-files (used for trees and graphs) we have another XSL-stylesheet. This stylesheet transforms the graphical elements of the SVG into ODF. It also takes care of things like coordinate system transformations, scaling etc.

The output of the process is in Open Document Format (Durusau and Brauer, 2006). ODF is an open, XML-based file format for office applications. The format specifies an XML structure for text documents, spreadsheets, and presentations. In an ODF presentation, each slide includes the slide contents as well as the notes attached to that slide. Listing 1 gives an example of the graphical primitives in one ODF slide. The primitives are the same as in, for example, SVG. However, as can be seen from the listing, the attributes used are from several different namespaces.[1]

[1]This caused some major problems in the implementation, since the ODF specification does not clearly state which attributes should be in which namespace. Especially specifying styles was difficult, partly due to the fact that Impress gives no error messages when using wrong namespaces, it just ignores the attributes. Another problem was with the positioning of the graphical primitives. Although the attributes are from the

```
<draw:g xmlns:svg="http://www.w3.org/2000/svg">
  <draw:polygon draw:style-name="fillwhitestrokewhite" svg:x="1cm" svg:y="1cm" draw:points="
      0,236 0,0 203,0 203,236 0,236" svg:height="11.8cm" svg:width="10.15cm" svg:viewBox="0 0
       203 236">
     <text:p/>
  </draw:polygon>
  <draw:ellipse svg:x="6.7cm" svg:y="3.875cm" svg:width="1.4cm" svg:height="1.45cm" draw:style
      -name="fillredstrokered"/>
  <draw:frame draw:style-name="gr9" draw:text-style-name="P1" draw:layer="layout" svg:x="7.4cm
      " svg:y="4.6cm">
    <draw:text-box>
      <text:p text:style-name="P1">Joensuu</text:p>
    </draw:text-box>
  </draw:frame>
  <draw:ellipse svg:x="8.2cm" svg:y="7.425cm" svg:width="1.4cm" svg:height="1.45cm" draw:style
      -name="fillorangestrokeorange"/>
  <draw:frame draw:style-name="gr9" draw:text-style-name="P1" draw:layer="layout" svg:x="8.9cm
      " svg:y="8.15cm">
    <draw:text-box>
      <text:p text:style-name="P1">Oulu</text:p>
    </draw:text-box>
  </draw:frame>
  <draw:line svg:x1="7.7cm" svg:y1="5.25cm" svg:x2="8.6cm" svg:y2="7.5cm" draw:style-name="
      fillnonestrokeblack"/>
</draw:g>
```

Listing 1: Example of ODF graphical primitives.

Figure 2 shows one slide in a generated example of the Kruskal's algorithm when opened in OpenOffice Impress. On the left, one can see the additional slides in the presentation. On the notes page, the presentation includes questions that the instructor can ask the students, as well as answers to the questions.

4 Conclusions

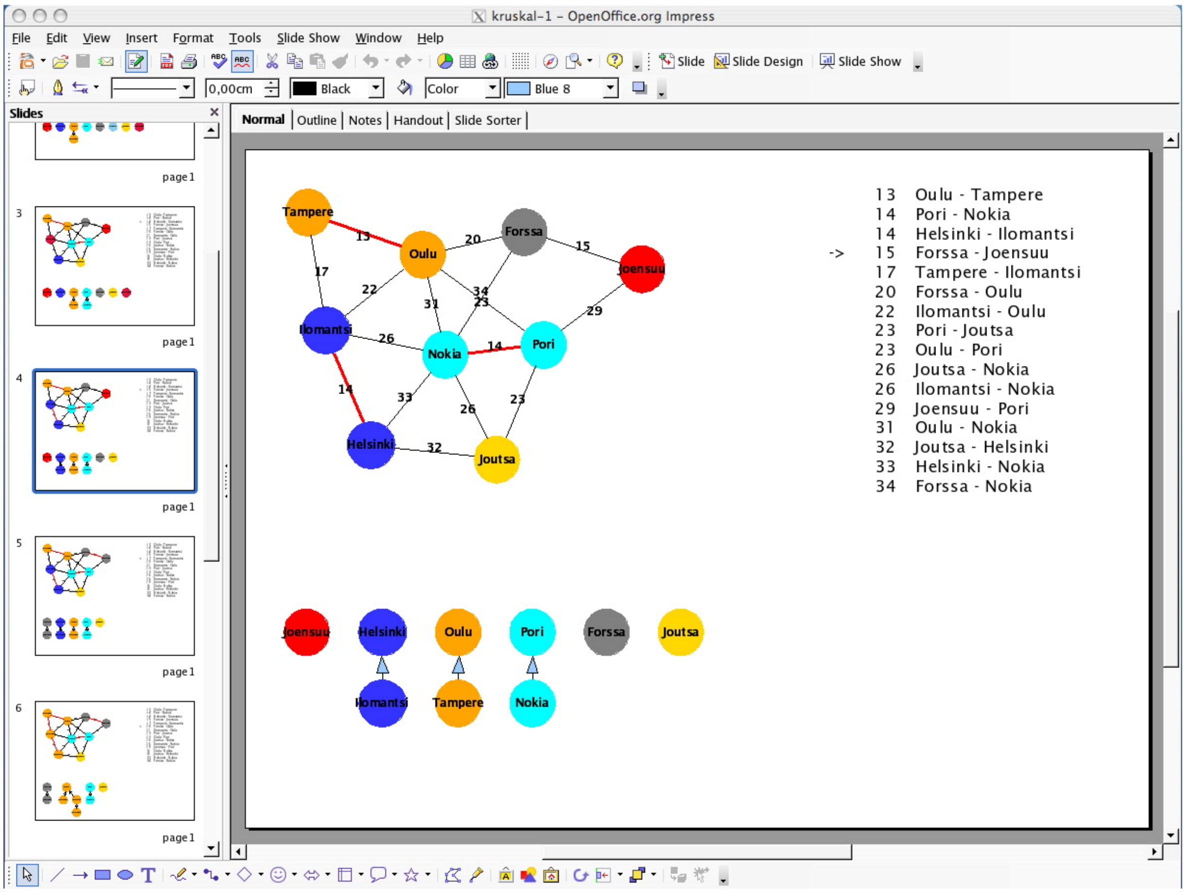
In this work, we have introduced an idea to develop tools to easily create slides for the presentation tools. Such tools would allow instructors to integrate algorithm visualizations into the tools they use on lectures instead of using separate visualization systems. We feel that this type of integration might be a solution to the problem of algorithm visualizations not used as widely as the AV research community hopes.

Our vision is to have automatic tools that output algorithm animations as presentation slides. This approach saves the teacher from switching between an algorithm visualization system and the presentation software. Compared with using, say, applets to visualize algorithms on lecture, the advantage of the slides is that the lecturer can edit them and adapt them to his/her other learning material. This can be, for example, changing terminology, changing colors, translating it to e.g. German, or adding explanations. In addition, the lecturer can alter the parameters for the tool creating the example, allowing the example to be tuned for the current audience.

Besides the visualization, automatically generated lecture material can be accompanied with dynamic documentation in the form of notes for the lecturer to use when showing the animation. These can be things to point out, questions for the audience, and such. This makes it easier for the teacher to make the lecture more interactive.

All the previous can be done in the presentation tool that the teacher is more likely to be familiar with than an algorithm animation system. Although we are algorithm animation system developers ourselves, we have to admit that the AA authoring tools are not of the same high quality as the presentation tools.

svg-namespace, the coordinates used are different.


kruskal-1 - OpenOffice.org Impress
File Edit View Insert Format Tools Slide Show Window Help
Slide Slide Design Slide Show
0,00cm Black Color Blue 8
Slides
Normal Outline Notes Handout Slide Sorter
page 1
Tampere
Oulu
Forssa
Joensuu
Ilomantsi
Nokia
Pori
Helsinki
Joutsa

```
   13  Oulu - Tampere
   14  Pori - Nokia
   14  Helsinki - Ilomantsi
-> 15  Forssa - Joensuu
   17  Tampere - Ilomantsi
   20  Forssa - Oulu
   22  Ilomantsi - Oulu
   23  Pori - Joutsa
   23  Oulu - Pori
   26  Joutsa - Nokia
   26  Ilomantsi - Nokia
   29  Joensuu - Pori
   31  Oulu - Nokia
   32  Joutsa - Helsinki
   33  Helsinki - Nokia
   34  Forssa - Nokia
```

Figure 2: A generated example in OpenOffice Impress.

The limitations of the current proof-of-concept implementation are obvious. The implementation is for a single algorithm, the process requires several software packages to be installed, and the output is only for Open Office Impress. However, from Open Office Impress, the presentations can be exported as Flash or HTML to be easily added to web pages. In addition, they can be saved in Microsoft PowerPoint and PDF formats.

In the future, creating more such generators could be beneficial. In addition, allowing ODF export from some of the existing AV systems is an interesting direction. Furthermore, we see that an online service for generating presentations of different topics might be popular among CS educators. Especially if the service could create slides for both Microsoft PowerPoint and OpenOffice Impress. This also eliminates the need to install the required software packages.

References

Ronit Ben-Bassat Levy and Mordechai Ben-Ari. We work so hard and they don't use it: acceptance of software tools by teachers. In *ITiCSE '07: Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education*, pages 246–250, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-610-3. doi: http://doi.acm.org/10.1145/1268784.1268856.

James Clark (editor). XSL Transformations (XSLT) 1.0 specification. W3C Recommendation, World Wide Web Consortium, nov 1999.

Patrick Durusau and Michael Brauer. Open document format for office applications (opendocument) v1.0 (second edition). Oasis committee specification, OASIS, July 2006.

John Ellson, Emden Gansner, Lefteris Koutsofios, North Stephen C., and Gordon Woodhull.

Graphviz open source graph drawing tools. *Lecture Notes in Computer Science*, 2265/2002: 594–597, 2002.

Christopher D. Hundhausen and Sarah A. Douglas. Low-fidelity algorithm visualization. *Journal of Visual Languages and Computing*, 13(5):449–470, October 2002.

Christopher D. Hundhausen, Sarah A. Douglas, and John T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing*, 13(3): 259–290, June 2002.

Ville Karavirta, Ari Korhonen, Lauri Malmi, and Kimmo Stålnacke. MatrixPro – A tool for on-the-fly demonstration of data structures and algorithms. In *Proceedings of the Third Program Visualization Workshop*, pages 26–33, The University of Warwick, UK, July 2004.

Essi Lahtinen, Hannu-Matti Järvinen, and Suvi Melakoski-Vistbacka. Targeting program visualizations. In *ITiCSE '07: Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education*, pages 256–260, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-610-3. doi: http://doi.acm.org/10.1145/1268784.1268858.

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

Thomas L. Naps, Guido Rößling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodgers, and J. Ángel Velázquez-Iturbide. Exploring the role of visualization and engagement in computer science education. *SIGCSE Bulletin*, 35(2):131–152, June 2003.

Guido Rößling and Bernd Freisleben. ANIMAL: A system for supporting multiple roles in algorithm animation. *Journal of Visual Languages and Computing*, 13(3):341–354, 2002.

Guido Rößling, Thomas Naps, Mark S. Hall, Ville Karavirta, Andreas Kerren, Charles Leska, Andrés Moreno, Rainer Oechsle, Susan H. Rodger, Jaime Urquiza-Fuentes, and J. Ángel Velázquez-Iturbide. Merging interactive visualizations with hypertextbooks and course management. *SIGCSE Bulletin*, 38(4):166–181, 2006. ISSN 0097-8418. URL `http://doi.acm.org/10.1145/1189136.1189184`.

Clifford A. Shaffer, Matthew Cooper, and Stephen H. Edwards. Algorithm visualization: a report on the state of the field. In *SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education*, pages 150–154, New York, NY, USA, 2007. ACM Press. ISBN 1-59593-361-1. doi: http://doi.acm.org/10.1145/1227310.1227366.

W3C. Scalable Vector Graphics (SVG) 1.0 specification. `http://www.w3.org/TR/SVG`, September 2001.

PathFinder: A Visualization eMathTeacher for Actively Learning Dijkstra's algorithm

M. Gloria Sánchez-Torrubia, Carmen Torres-Blanc, Miguel A. López-Martínez
Applied Mathematics Department, Universidad Politécnica de Madrid

gsanchez@fi.upm.es

Abstract

PathFinder is a new eMathTeacher for actively learning Dijkstra's algorithm. In Sánchez-Torrubia et al. (2007) the concept of eMathTeacher was defined and the minimum as well as some additional requirements were described. The tool presented here is an enhanced paradigm of this new concept on Computer Aided Instruction (CAI) resources: an application designed following the eMathTeacher philosophy for active eLearning. The highlighting new feature provided by this application is an animated algorithm visualization panel showing, on the code, the current step the student is executing and/or where there is a user's mistake within the algorithm running. PathFinder also includes another two interesting new features: an active framework area for the algorithm data and the capability of saving/retrieving the created graph.

1 Introduction and Preliminaries

Graphical and dynamic web based tools are more appealing for students than traditional learning materials. It has been confirmed that learners spend much more study time when visualization is involved; however, there has been some skepticism about the real value of visualizations as a pedagogical tool. Many educators think that visual tools enhance their lectures and significantly increase student's comprehension, but such tools are of little effectiveness when students are not actively engaged in the learning process (Naps et al., 2003). Furthermore, when students are not required to wonder about the concepts, to provide answers and to predict what is happening next, they might adopt a passive attitude that is not beneficial at all, and may even be harmful for their training. The analysis presented by Hundhausen et al. (2002) asserts that "*how* students use AV technology, rather than *what* students see, appears to have the greatest impact on educational effectiveness" and their study "suggests that the most successful educational uses of AV technology are those in which the technology is used as a vehicle for actively engaging students in the process of learning algorithms". They concluded that, those who are actively engaged with the visualization have consistently outperformed the other ones who passively viewed them. Thus, in order to avoid a passive attitude, during the execution, the program should interact continuously with the users, forcing them to predict the following step.

In Clear et al. (2001), the authors state that a "consciously designed approach informed by a constructivist view holds the most potential for effective online learning designs"; and that "learners must construct their understanding through an active process building on past experiences and knowledge and that knowledge cannot be simply accepted from others". According to this opinion, we believe that active learning is the only effective way of acquiring knowledge and that students cannot only look how the processes evolves, but they must get involved in the process itself.

The challenge of discovering new ways to motivate students in active learning encouraged us to develop a new kind of web based tools. Our main goal has been to get students involved, as actively as possible, in their learning process. With this objective in mind, we have been developing several interactive Java applets, that allow visualized execution of algorithms (Sánchez-Torrubia et al. (2008a), Sánchez-Torrubia and Gutiérrez-Revenga (2006) & Sánchez-Torrubia et al. (2008b)). Those applets have been designed under the eMathTeacher philosophy and are being used as complementary material for blended learning (bLearning) (Sánchez-Torrubia et al., 2008a) both for teachers on classroom lectures and students when learning by themselves. This way, the power and effectiveness of face to face teaching are boosted with the flexibility and technical capabilities of eLearning, turning out the students into the protagonists of their own learning progression.

In the next sections, we review the definition of eMathTeacher as well as its main requirements. This kind of application introduces a new concept in computer aided education as they can act as genuine virtual trainers extending the teacher's hand through the Web.

1.1 eMathTeacher Definition

An eLearning tool is eMathTeacher compliant (Sánchez-Torrubia et al., 2007) if it works as a virtual maths trainer. In other words: if it is an on-line self-assessment tool that help users to actively learn math concepts or algorithms by themselves, correcting their mistakes and providing them with clues to find the right solution.

They can also be applied as bLearning complementary material for being used both by teachers on classroom lectures and by students when learning maths by themselves. However, the most important feature of these tools is the feasibility of being used for practicing with maths methods or algorithms while the system guides the user towards the right answer.

1.2 Minimum requirements for an eMathTeacher

These, as described in (Sánchez-Torrubia et al., 2007), are the minimum conditions we establish a tool must fulfil to be considered an eMathTeacher :

- Step by step inquiring: for every process step, the student should provide the solution while the application waits in a stand by mode, expecting the user's input.
- Step by step evaluation: just after the user's entry, the eMathTeacher evaluates it, providing a tip for finding the proper answer if it is wrong or executing it if ok.
- Visualization of every step change that happens.
- Easy to use.
- Flexible and reliable: allowing the user to introduce and modify the example and to repeat the process if desired.
- Clear presentation within a nice and friendly graphic environment, helping insight.
- Platform independency and continuous availability (anytime, anywhere).

1.3 Additional requirements for an eMathTeacher

The requirements listed above are mandatory for an application to be considered an eMathTeacher. In addition, there are some other desirable conditions, containing those described in (Sánchez-Torrubia et al., 2008b), that these tools should meet. The main features to be included are:

- Algorithm visualization panel.
- Framework panel showing the current state of the algorithm data structures. It should also allow the user to update those structures.
- Samples library and/or saving/retrieving capabilities. This requirement should include saving, for later analysis, both the basic structure the user is working with, and also the user's interaction history.
- Language menu.

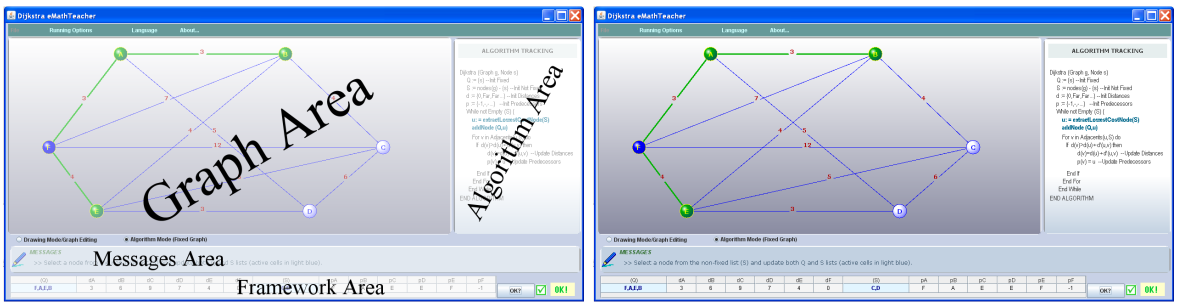
- Integration of different tools as a complete suite, able to cover the whole topic.
- Automatic execution process (optional), especially designed for very complex exercises.
- Capability to find and show alternative solutions once the problem has been solved.
- A theoretical introductory part.
- No installation or maintenance tasks required and light downloading weight.

Other authors (see e.g. Jarc et al. (2000), Rößling and Naps (2002) & Naps et al. (2003)) have already highlighted most of the above listed features as being required for a learning tool to be effective. Last year, we performed a literature search & study (Sánchez-Torrubia et al., 2007), which allowed us to identify a number of systems designed under Java Technology and oriented to help students on learning different Computer Science topics (e.g. IDSV, JHAVÉ, TRAKLA2, JFLAP or AulaWeb Self-Assessment Module). Though those tools seem similar in terms of functionality, there is actually a feature that makes eMathTeachers (i.e. eLearning tools that are eMathTeacher compliant) different: they only execute the current step in case the input is ok and return a customized error message, providing a tip for finding the proper answer, otherwise. This unique characteristic provides eMathTeachers with full interactive learning capabilities, and distinguishes them from other systems so far.

2 PathFinder: an eMathTeacher for Dijkstra's algorithm

Dijkstra's algorithm (Dijkstra, 1959), commonly known as *shortest path algorithm*, solves the problem of identifying the shortest path in a weighted graph from an initial vertex to the other vertices. The central idea behind the algorithm is that each subpath of the minimal path is also a minimum cost path.

Keeping the features described in the preliminaries, we have designed and implemented an application, available at `http://www.dma.fi.upm.es/java/matematicadiscreta/dijkstra`, (see Figure 1) for active learning of Dijkstra's algorithm, including both *with* and *without final node* possibilities. Our PathFinder has been designed under the eMathTeacher philosophy (section 1.1) and meets all the minimum requirements listed in section 1.2 as well as nearly all the additional ones (section 1.3).



```
Dijkstra (Graph g, Node s)
   Q := {s} --Init Fixed
   S := nodes(g) - {s} --Init Not Fixed
   d := {0,Far,Far...} --Init Distances
   p := {-1,-,-...}  --Init Predecessors
   While not Empty (S) {
     u: = extractLowestCostNode(S)
     addNode (Q,u)
     For v in Adjacents(u,S) do
       If d(v)>d(u)+d'(u,v) then
           d(v)=d(u)+d'(u,v)  --Update Distances
           p(v) = u  --Update Predecessors
       End If
     End For
   End While
END ALGORITHM
```

>> Select a node from the non-fixed list (S) and update both Q and S lists (active cells in light blue).

(Q)	dA	dB	dC	dD	dE	dF	(S)	pA	pB	pC	pD	pE	pF
F,A,E,B	3	6	9	7	4	0	C,D	F	A	E	E	F	-1

Figure 1: PathFinder: an eMathTeacher for Dijkstra's algorithm

2.1 Detailed description

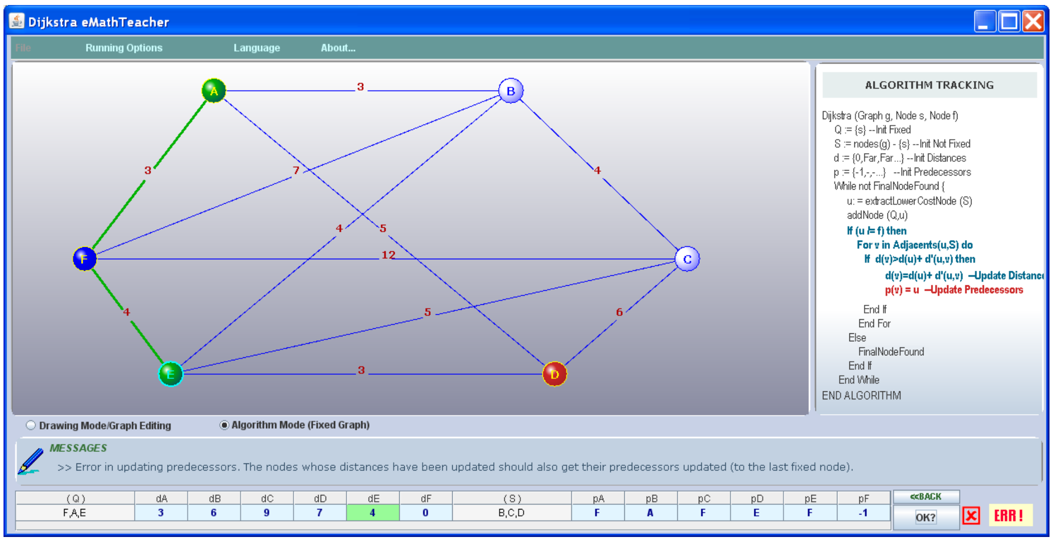
The application runs in a Java Web Start window. The window is split into four areas: graph, algorithm, messages and framework areas (see Figure 1), where **graph** and framework panels are the main working areas. The first one is a panel where the graph is displayed and allows the user to create and edit nodes and weighted edges. The second one is the **framework** area. A table, presenting the current state of the algorithm structures (fixed and unfixed nodes, distances to the initial node and predecessors list), is displayed in this panel.

The **algorithm** area displays the execution code, showing in blue the current step or in red the point where the user's mistake is located, while the **message** panel provides clues to find the right solution or indicates the next step to be done. This panel also offers useful hints when the graph is being edited.

The menu bar presents a graph saving/retrieving option, three execution options and a language selector, currently implemented in Spanish and English.

2.2 Editing the graph

The graph nodes are drawn by left clicking the mouse, and the edges by right dragging between two nodes. When an edge is being created, a text cell appears in the message panel for entering the edge's weight. The nodes can be moved or deleted at any time and the edges can be erased or their weigh modified. The first created node is predefined as the initial node (e.g. A in Figure 2) but the user has also the possibility of changing the initial node and/or defining a final node (e.g. D in Figure 2).



ALGORITHM TRACKING

```
Dijkstra (Graph g, Node s, Node f)
  Q := {s} --Init Fixed
  S := nodes(g) - {s} --Init Not Fixed
  d := {0,Far,Far...} --Init Distances
  p := {-1,-,-...}  --Init Predecessors
  While not FinalNodeFound {
    u: = extractLowerCostNode (S)
    addNode (Q,u)
    If (u /= f) then
      For v in Adjacents(u,S) do
        If  d(v)>d(u)+ d'(u,v) then
            d(v)=d(u)+ d'(u,v)  --Update Distance
            p(v) = u  --Update Predecessors
        End If
      End For
    Else
      FinalNodeFound
    End If
  End While
END ALGORITHM
```

MESSAGES

>> Error in updating predecessors. The nodes whose distances have been updated should also get their predecessors updated (to the last fixed node).

(Q)	dA	dB	dC	dD	dE	dF	(S)	pA	pB	pC	pD	pE	pF
F,A,E	3	6	9	7	4	0	B,C,D	F	A	F	E	F	-1

Figure 2: The PathFinder showing an error on predecessors update

2.3 Executing the algorithm

Once the graph has been introduced and the algorithm mode has been selected, the application checks whether it can be executed or not (as stated in Dijkstra (1959) "we restrict ourselves to the case where at least one path exists between any two nodes", i.e. to connected graphs). If the graph is not connected and thus the algorithm cannot be executed, an error message indicating this fact is displayed, otherwise the algorithm starts.

The execution has three performance options: two modes of interactive running (step by step and iteration verification) and an option of direct execution, able to directly provide the final result (restricted to help on verifying results or to simplify the flow process in case of very complex exercises).

In **step by step verification** option, for every algorithm iteration, the user must update first the fixed and unfixed nodes, then check whether it is correct or not and finally update distances and predecessors and check again the input correctness. **Iteration verification** option is aimed at more advanced students as the user should perform a whole iteration before checking the input correctness. For every verification (in both interactive running modes), when any of the updates is not right, the message panel shows the event: an error message pointing out the problem and providing hints for rectifying is displayed. Simultaneously, in the algorithm panel, the step where the mistake is located changes into red (see Figure 2). If all the updates are right, the application changes the node and/or edge's color and waits for the next user's entry.

2.4 Comparison with previous eMathTeachers and PathFinder analysis

While using the previous eMathTeachers as complementary material for bLearning, we have realized the necessity of implementing an algorithm visualization panel inside the tool for enhancing the student's consciousness of each of the algorithm steps. Also, while encouraging them to write down the algorithm structures, we found the necessity of including those structures inside the tools. As a consequence, when designing PathFinder, we incorporated several new features that, in our opinion, entail huge pedagogical enhancement:

- The algorithm visualization panel (see Algorithm area in Figure 1) provides a better understanding of the algorithm execution code.
- The framework panel (see Framework area in Figure 1) allows users to practice every algorithm step exactly as if they were implementing it *by hand*, but including the graphic panel as well as the feasibility of being corrected and advised by the tool when a mistake happens (this is the main feature of eMathTeacher philosophy: acting as a virtual maths teacher).
- The retrieval option offers the instructor the possibility of preparing selected exercises for the students, and gives the learner the advantage of saving the graph for later review.

In Hundhausen et al. (2002), the authors assert that "AV technology has been successfully used to actively engage students in such activities as what-if analysis of algorithmic behavior, prediction exercises and programming exercises". The eMathTeacher philosophy has been mainly inspired by getting the students involved in the two first activities: what-if analysis of algorithmic behavior (what means understanding the algorithm design in-depth) as well as predicting the algorithm outcomes (that entails a good understanding of the algorithm process). In our opinion, as a consequence of the above described improvements, PathFinder will get the users even more engaged in the two first activities, obtaining as a result an active learning of this algorithm and thus, hopefully, successful educational results.

3 Conclusions and next steps

In this paper, PathFinder, a new tool for actively learning Dijkstra's algorithm, has been presented. Moreover, some new additional requirements for eMathTeacher tools have been introduced. PathFinder is an enhanced paradigm of this new concept on CAI resources. The highlighting new feature provided by this application is an animated algorithm visualization panel. It shows, on the code, the current step the student is executing and also where there is a user's mistake within the algorithm running. Other important new attribute is the active framework area for the algorithm data, designed for encouraging students to active learn the algorithm process, as implemented *by hand*, while the application verifies the correctness of each user's input. Finally, the application runs in a Java Web Start window and this technology allows it to read and write in the client's hard disk, which gives us the feasibility of saving and retrieving graphs. This way, the instructor can load an examples library and the students can save the edited graphs for later revision or modification.

PathFinder has recently been finished, which means that it has not been used by any students yet. Though there are not impact evaluations of it, based on our previous research, we have high expectations regarding its potential effectiveness on helping learning activities. As part of that research, the previous graph eMathTeachers impact has already been evaluated by comparing the rates obtained on the graphs exercise in a Discrete Mathematics final exam.

As detailed in Sánchez-Torrubia et al. (2007), the results showed a deeper understanding of algorithms process in the study group, even considering that those tools offered neither the algorithm visualization panel nor the framework panel.

In the near future, we aim to perform a tool effectiveness evaluation, following the models proposed in (Reeves and Hedberg (2003) & Naps et al. (2003)), as well as measuring its impact on the students' learning. We are also preparing an users survey (covering both students and teachers population) added to a collection of opinions, suggestions for improvement, etc.

Currently, we are designing and developing a whole suite which actually integrates different types of graphs and graph algorithms fitted in it. The design includes all four panels described in the PathFinder and will feature the above mentioned functions together with the possibility of saving the user's interaction history for later analysis and/or (automatic) assessment.

Acknowledges

This work has been partially supported by UPM (Spain) under Project No. IE07 1010-029.

References

Tony Clear, Arto Haataja, Jeanine Meyer, Jarkko Suhonen, and Stuart A. Varden. Dimensions of distance learning for computing education. *SIGCSE Bull.*, 33(2):101–110, 2001.

Edsger Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

Christopher D. Hundhausen, Sarah A. Douglas, and John T. Stasko. A meta-study of algorithm visualization effectiveness. *J. Visual Languages & Computing*, 13(3):259–290, 2002.

Duane J. Jarc, Michael B. Feldman, and Rachelle S. Heller. Assessing the benefits of interactive prediction using web-based algorithm animation courseware. *SIGCSE Bull.*, 32(1): 377–381, 2000.

Thomas Naps, Guido Rößling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger, and J. Ángel Velázquez-Iturbide. Exploring the Role of Visualization and Engagement in Computer Science Education. *SIGCSE Bull.*, 35(2):131–152, 2003.

Thomas C. Reeves and John G. Hedberg. *Interacive Learning Systems Evaluation.* Educational Technology Publication. Englewood Cliffs, 2003.

Guido Rößling and Thomas L. Naps. A testbed for pedagogical requirements in algorithm visualizations. *SIGCSE Bull.*, 34(3):96–100, 2002.

M. Gloria Sánchez-Torrubia and S. Gutiérrez-Revenga. Tutorial interactivo para la enseñanza y el aprendizaje de los algoritmos de búsqueda en anchura y en profundidad. In *Proceedings of the XII J. de Enseñanza Universitaria de la Informática*, pages 573–580, 2006.

M. Gloria Sánchez-Torrubia, Carmen Torres-Blanc, and Juan B. Castellanos. Defining eMath-Teacher tools and comparing them with e&bLearning web based tools. In *Proceedings of the International Conference on Engineering and Mathematics (ENMA)*, 2007.

M. Gloria Sánchez-Torrubia, Carmen Torres-Blanc, and Víctor Giménez-Martínez. An eMath-Teacher tool for active learning Fleury's algorithm. *International Journal Information Technologies and Knowledge (IJ ITK)*, 2(5):437–442, 2008a.

M. Gloria Sánchez-Torrubia, Carmen Torres-Blanc, and Sanjay Krishnankutty. Mamdani's fuzzy inference eMathTeacher: a tutorial for active learning. *WSEAS Transactions on Computers*, 7(5):363–374, 2008b.

Animation and Interactive Programming: A Practical Approach

Phillip Benachour
Department of Communication Systems
Infolab21, Lancaster University, UK

p.benachour@lancaster.ac.uk

Abstract

This paper describes a work in progress of using animation software tools to teach programming principles. The motivation behind this work is to encourage students in higher education who do not see themselves as serious programmers to engage with some of the concepts and methods used in the teaching of programming. In addition this work was used in workshops to engage with local and regional secondary schools focussing on the use of animation as a tool for learning programming principles. The results presented in this paper are based on feedback from first year undergraduate students. Initial feedback from teachers, pupils and schools has been very positive and requests for additional visits have been made. This has created an opportunity to further engage with these schools for additional workshops. Analysis and feedback from these schools will be presented in the workshop.

1 Introduction

The teaching of new concepts and hands on practical skills such us programming can be combined with animation and visuals to great effect. The use of learning objects to deliver programming skills to multimedia students and other disciplines has been explored and implemented (Jones, 2004). Modern programming languages taught at university level is generally seen as a major obstacle for new undergraduates and such a perception of difficulty is commonly cited as a reason for disengagement with the subject as a whole (McCracken et al., 2001). A number of papers in the literature have introduced Actionscript as a suitable tool for teaching introductory programming. For instance, Crawford and Boese (2006) compared Actionscript code with more complicated code in Java for similar programs and concluded that Actionscript not only teaches the fundamental of programming and concept of object-oriented development to the students but also enables them to find the errors in the smaller tasks which is easier to solve. In a more recent paper (Leutenegger and Edgington, 2007), Actionscript has been recommended as a useful step up to high-level languages such as C++. Actionscript is seen to be easier to learn due to the immediate visualization it provides. In addition Leutenegger and Edgington (2007) argued that Actionscript is widely used in the real world e.g for designing animations and 2 dimensional games.

The aim of this paper is to describe and evaluate a work in progress on using animation and interactive programming to engage first year university students, from diverse educational backgrounds and subject disciplines, with programming principles. This work in progress focuses on encouraging students to learn progressive programming tasks from simple commands and assignments to designing a game. The design of a BreakOut computer game is perhaps one of the most useful exercises the students get involved in when learning programming and scripting concepts. Students learn about collision detection between the ball, paddle, bricks and boarders, user interaction by moving the paddle using keyboard controls, update of scores using dynamic text and the use of sound to make the learning of programming experience more enjoyable and engaging.

2 Using Actionscript for Animation and Scripting

Actionscript uses traditional coding techniques but allows the user to see how each piece of code effects the running or execution of the program, allowing the user to have an instant visual understanding of what the code is doing. To help with coding errors Actionscript uses a syntax checker and will inform the user of errors either before or as they run a program. This is in contrast to other programming or scripting languages such as Javascript where errors are not so easily identifiable and the simple omission of a comma can cause the entire program to fail. A Javascript program will also only execute when the whole program is complete, where as with Actionscript code written for a specific action can be run even if the program, as a whole, is unfinished.

2.1 Teaching the use of Increment and Decrement Operators

Perhaps the most obvious and easiest example to work with when beginning to use Actionscript for animation, is to use the increment and decrement operators. Students are encouraged to experiment with moving objects in a two-dimensional and three-dimensional world. The following simple script allows an object to move in the x direction with incrementing values of x by one pixel every time the frame is rendered and displayed.

```
onClipEvent (load) {
  this.\_x = 50; // this sets the initial x position of our clip
}
```

Now try the following code but only for the X-axis:

```
onClipEvent (enterFrame) {
  this.\_x = this.\_x+1; // this moves our clip 5 pixels to the right every frame
}
```

Students are then asked to record their answers on the following questions:

- What value would you assign "this._x" to in order to centre it in the middle of the stage?
- What would you change in the code above to stop your circle from moving?
- What would you change in the code above so that your ball moves in the opposite direction?

The students are then asked to introduce a "y" variable to the Actionscript code and give their movieclip the following action:

```
onClipEvent (load) {
  this.\_x = 50;
  this.\_y = 50;
}
onClipEvent (enterFrame) {
  this.\_x+=1;
  this.\_y+=1;
}
```

The routine above enables students to understand how two-dimensional animation works using the following questions as a guide:

- What happens now and why?
- What would the values of "this._x" and "this._y" be set to put your circle in the centre of your movie clip?
- What sort of increments/decrements would you set "this._x" and "this._y" to in order to get the ball object to travel in the four possible directions from the centre.

2.2 If and If ... Else Statements

Using if and if ... else statements for conditional testing checks whether a condition is true or false. The ball object in the example above can be used to do this. Students are asked to think of ways to stop the ball object moving when it gets to a certain pixel on the screen. A routine like the one below can achieve this task very easily:

```
onClipEvent (enterFrame) {
if (this.\_x<300) {
    this.\_x += 1;
  }
}
```

Once students get the idea that they can modify the code to make the ball object move upwards and downwards once the value of "this._x" reaches 300 pixels, they are asked to experiment with different values of the x co-ordinate and then the y co-ordinate. An alternative way to assess students understanding is to invite them to comment on a piece of code already written such as the one below:

```
onClipEvent (enterFrame) {
  if (this.\_x<300) {
    this.\_x += 1;
  } else if (this.\_x>300) {
    this.\_y += 1;
  }
}
```

Using if and if ... else statements can be used more imaginatively by testing whether an object is moving within a defined space or by testing for collision detection. The design of a breakout game is an example in case where the ball is required to bounce off the edges or a paddle. Students are initially given code which does not take into account the variable ballRadius, once the type the code and see it working they realise that the ball object goes over the edges set. In order to solve this problem ballRadius is included as part of the if statements for the x and y directions as shown in the code below:

```
var screenWidth = 550;
var screenHeight = 400;
var ballRadius = 10;
ball.\_x = ball.\_x + ballSpeedX;
ball.\_y = ball.\_y + ballSpeedY;

if (ball.\_x<ballRadius) {
  ball.\_x= ballRadius;
  ballSpeedX*=-1;
} else if (ball.\_x>screenWidth - ballRadius) {
  ball.\_x = screenWidth - ballRadius;
  ballSpeedX*=-1;
}

if (ball.\_y<ballRadius) {
  ball.\_y= ballRadius;
  ballSpeedY*=-1;
} else if (ball.\_y>screenHeight - ballRadius) {
  ball.\_y = screenHeight - ballRadius;
  ballSpeedY*=-1;
}
```

2.3 Using Functions and loops for layering of bricks in a breakout game

A function is defined using the function keyword, followed by the name of the function, and then a list of parameter names within the following brackets. In the example code below the

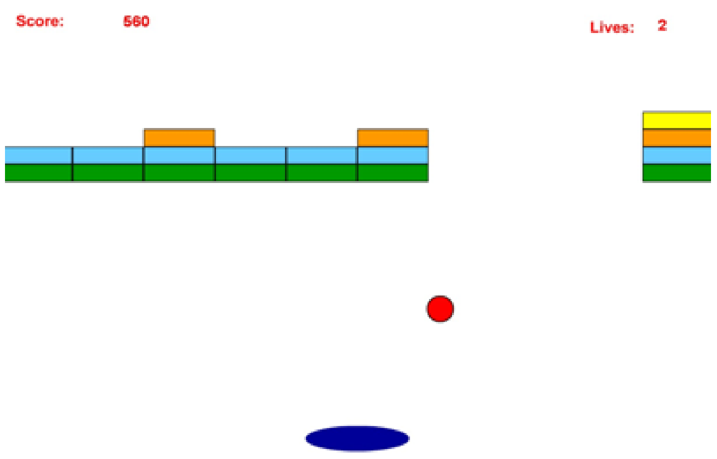


Figure 1: BreakOut Game

function laybricks is passed three parameters which provides the information that allows the layering of bricks.

```
function layBricks(numWide, name, row) {
  eval(name).\_width = screenWidth/numWide;
  eval(name).\_height = 0.25*screenWidth/numWide;
}
```

In this function, a loop is created that lays numWide bricks:

```
function layBricks(numWide, name, row) {
  eval(name).\_width = screenWidth/numWide;
  eval(name).\_height = 0.25*screenWidth/numWide;

  for (i=0; i<numWide; i++) {
  }
}
```

This loop, begins by setting a variable called i, to 0, and then loops numWide times, and creates a new name, based on the original brick name, so that we have a unique variable name for each of the new bricks we are going to create. Figure 1 below shows output produced once the game is completed.

```
nextLevel = 100;
numBricks = 0;

function layBricks(numWide, name, row) {
  eval(name).\_width = screenWidth/numWide;
  eval(name).\_height = 0.25*screenWidth/numWide;

  for (i = 0; i < numWide; i++) {
    brickName = name + i; // trace(brickName);
    duplicateMovieClip (name, brickName, nextLevel++);
    eval(brickName).\_y = 100 + row*eval(name).\_height;
    eval(brickName).\_x = (i + 0.5)*eval(name).\_width;
    numBricks++;
  }
}

layBricks(10, "yellowBrick", 0);
layBricks(10, "orangeBrick", 1);
layBricks(10, "blueBrick", 2);
layBricks(10, "greenBrick", 3);
```

3 Evaluation and Student Feedback

The first year course in Information and Communications Technology is common to all first year undergraduates studying for a communications degree. The course is taught taking into consideration the needs of both potentially highly numerate and technologically aware students as well as students who may have no previous experience of ICT. The first year intake has between 60-70 students, most studying for programmes on Information Technology, Media and Computer Communications. The aim of the survey below is to carry out a quantitative and qualitative evaluation of how effective Actionscript has been in helping students engage with programming and scripting principles, the data and feedback collected was intended to focus on the following three areas:

- Previous experiences of using programming and scripting
- Experience of computer animation, game design, and game play
- Evaluation of Actionscript as a tool to learn programming principles and for application development

From the cohort of students studying the course, 67% said they had some previous experience of programming and 47% with scripting. When asked if they found some programming concepts easy to implement, 82% said that they felt that learning to program without visualisation was hard and slowed their progress. A majority said that they decided to carry on with their courses at school/college because it was too late to change. Many felt that they found difficulty in acquiring fluency in programming.

When considering the distribution of the type of programming languages used, 10%, 16% and 26% said they have had experience of C, C++ and Java respectively. It is also worth noting here that the remaining 48% who have not used C/C++, Java have used other high-level languages such as Pascal, Prolog, C#, VB.NET, and Delphi. We can summarise from this that two-thirds of first year students have some experience of programming using a high-level language at a basic level which is a positive outcome in as far as recruiting students with programming skills. However, this experience was proved to be very basic and engagement was an issue as pointed out in the previous paragraph. Of the 47% of students who said they have scripting experience, 30% said they had used Actionscript before but only at an introductory level (using a button for controlling the start of an animation), 35% said they have used Javascript. The remaining 35% have used VBScript 18%, PHP 12%, ASP 5%. Figure 1 shows the distribution of programming and scripting previously used at school/college by first year students.

All student (100%) of this group found Actionscript an easier tool to visualize and understand basic programming and scripting.

For students who had no previous experience of programming and scripting languages 33% and 53% respectively, engaging them in learning programming principles was rather easier than anticipated. The students had no prior experiences and there was no bias towards a particular programming or scripting language. It was also noted that for this group, the majority tended to have had more experience of using video and audio editing tools with some good experience of Web design and image editing. 80% of this group found Actionscript a useful tool to understand basic programming and scripting.

In order to find out how effective Actionscript has been in engaging and helping student to learn programming principles, success rates of the questions put to the students during the practical sessions were measured. The results of these are shown in Table 1.

4 Conclusions and Further Research

The results attained from the practical assessments has shown that correct responses have progressively improved over time. Although this was somewhat expected it also demonstrated student commitment and engagement as well. The challenges presented when designing the breakout computer game is an example of how well the students have adapted to using Actionscript.

Questions used	**% correct**
What value would you assign this._x to in order to centre it in the middle of the stage?	53%
What would you change in the code above to stop your circle from moving?	46%
What would you change in the code above so that your ball moves in the opposite direction?	58%
What would the values of this._x and this._y be set to put your circle in the centre of your movie clip?	65%
What sort of increments/decrements would you set this._x and this._y to in order to get the ball object to travel in the four possible directions from the centre.	60%
Use if and if ... else statements to control the movement of the ball when it reaches a position on the x-axis	51%
Use if and if ... else statements to control the movement of the ball when it reaches a position on the x-axis and y-axis	60%
Write the code you would add to make the ball bounce from the bottom and top wall (y-axis)	71.4%
Write the code you would use to modify the gameloop code to take into account the height of the ball	85.7%
Reflecting the ball off the paddle and back up the screen	100%
Explain how you would add a 5th layer of bricks with a score of 50 points- to the screen. What additional code would you use?	93%
What code would you add to generate sound for your game?	90%
Recall the different approaches of programming in Javascript and Actionscript. Which programming style do you prefer?	90% said Actionscript

Table 1: Student success rate in the practical sessions

References

Stewart Crawford and Elizabeth Boese. Actionscript: a Gentle Introduction to Pogramming. *Journal of Computing Sciences in Colleges*, 21(3):156–168, 2006. ISSN 1937-4771.

Ray Jones. Designing Adaptable Learning Resources with Learning Object Patterns. *Journal of Digital Information*, 6(1), 2004.

Scott Leutenegger and Jeffrey Edgington. A Games First Approach to Teaching Introductory Programming. In *SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education*, pages 115–118, New York, NY, USA, 2007. ACM. ISBN 1-59593-361-1. doi: http://doi.acm.org/10.1145/1227310.1227352.

Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. *SIGCSE Bull.*, 33(4):125–180, 2001. ISSN 0097-8418. doi: http://doi.acm.org/10.1145/572139.572181.

Animalipse - An Eclipse Plugin for AnimalScript

Guido Rößling, Peter Schroeder
CS Department, TU Darmstadt
Hochschulstr. 10
64289 Darmstadt, Germany

roessling@acm.org

Abstract

AnimalScript, while highly expressive and versatile, is not easy to edit with no editor support. We have developed an Eclipse plugin for editing AnimalScript that includes a text editor, outline, and code assist. We expect that this plugin will make the editing process much easier and faster. The paper presents both technical aspects of the development and the resulting plugin.

1 Introduction

Animal is a versatile system for creating, modifying and presenting algorithm animations and visualizations (AV content). As far as we know, it is currently the only AV system that allows users to create AV content using all of the following approaches:

- *visually* using drag and drop in a novice-friendly graphical user interface (Rößling and Freisleben, 2002),
- *textually* using the highly expressive AnimalScript language (Rößling and Freisleben, 2001; Rößling et al., 2004),
- employing a new Java-based generation API,
- using a set of external applications for generating context-specific animations for trees (Rößling and Schneider, 2006) and graphs / graph algorithms (Naps et al., 2003; Rößling et al., 2007),
- as well as using one of the currently more than 130 animation generators of the built-in *generator* framework (Rößling and Ackermann, 2006). Here, it is important to note that the number of generators does not necessarily indicate the number of algorithms covered, but more the different “flavors” for a given algorithm, such as the choice of the programming language and the output language used for the presentation.

All generation approaches except for the first are directly or indirectly based on using AnimalScript, which is in the process of taking over the role of the preferred representation of Animal AV content from the built-in ASCII notation. The reasons for this development are the human-readable notation of AnimalScript, the ease with which it can be generated from programs and edited manually, and the expressiveness of the language. Since 2008, Animal also includes a window that provides a BNF-based definition of the AnimalScript notation, as well as (since 2006) a small text editor for directly entering or modifying AnimalScript input and visualizing the results.

AnimalScript files contain one command per line, such as a definition of a new graphical object or a transformation of some objects. The animation is organized in steps, each of which can contain one or more commands. If multiple commands are used in a step, the step is surrounded by curly braces { }. Please see (Rößling et al., 2004; Rößling and Freisleben, 2001) for more information about AnimalScript.

Many of the other established AV systems also cover some of the generation approaches listed above. For example, JAWAA (Akingbade et al., 2003) and the GAIGS and JSamba

(Stasko, 1998) visualization engines used by JHAVÉ (Naps and Rößling, 2006) also use a scripting language. JAWAA also offers a visual editor in its current release. JHAVÉ offers a set of content generators that are similar to the approaches offered in ANIMAL's generator framework and can be run off the web. However, they focus on specifying algorithm parameters, and thus do not allow the definition of visual properties such as colors.

While ANIMALSCRIPT can be edited easily using ANIMAL's built-in editor or any arbitrary text editor, the comfort offered by this is somewhat lacking. The internal editor only offers rudimentary editing features; *cut, copy* and *paste* features are only supported by using the underlying operating system support. The editor does not offer a search facility, display of line numbers, indication of recognized syntactical or semantical errors, or syntax highlighting. Thus, editing a longer ANIMALSCRIPT file is awkward and can become frustrating if the system indicates a parsing problem "in line 117". Despite (usually) precise information about the nature of the error, the lack of line numbers, search or "go to line" functions makes locating and fixing the error a tedious and less than enjoyable process.

We decided that his unsatisfying state needed addressing. Essentially, we saw three different approaches to provide better user support: improve the built-in editor to be comparable in comfort to the user's preferred text editor, create a new custom editor for ANIMALSCRIPT content, or provide ANIMALSCRIPT bindings for at least one commonly used text editor. It did not seem useful to invest much effort only to improve the built-in editor so that it would be comparable to, but still different from, a given user's preferred text editor. The same applied to creating a new custom editor. Therefore, we opted to provide ANIMALSCRIPT bindings for at least one commonly used text editor. We now had to decide which text editor to use.

The main target audience for ANIMAL and thus for ANIMALSCRIPT are students and teachers of Computer Science. We decided to base our work on the text editor provided by the Eclipse IDE, as this IDE is used in many Computer Science courses, so that our target users may already be familiar with the basic features of the underlying text editor.

The remainder of the paper is structured as follows. In Section 2, we will briefly summarize the features offered by the Eclipse IDE, focusing on plugins and text editors. Section 3 outlines the plugin for editing ANIMALSCRIPT code using the Eclipse IDE features. Section 4 shows usage examples to illustrate the support for ANIMALSCRIPT provided by the ANIMALIPSE plugin. Finally, Section 5 evaluates the plugin and presents areas for further research.

2 A Brief Overview of the Eclipse IDE

Eclipse (Beck and Gamma, 2003) was presented by IBM in 2001 and turned into open source in 2004. Due to a large number of developers, the platform offers a huge selection of plugins and extensions for different needs, including a large selection of supported programming languages, version control system front-ends for CVS and Subversion, workflow and design components. Probably the best known plugin is the *Java Development Tools*, employed by many students, researchers, developers, and teachers world-wide for writing Java-based programs.

The main components of the Eclipse platform are the *workbench* responsible for the graphical user interface, including the maintenance of the Eclipse windows, and the *workspace*. The workspace is a separate file system that handles the creation, storage and editing of files, including files, directories and projects.

The graphical front-end of Eclipse contains the usual menus and dialogs as well as *editors* and *views*. Editors are used to modify resources - the most well-known is the Java Editor for editing Java class source. Views are responsible for presenting content. Eclipse already offers many different views such as the *Problems, Progress* or *Console* view.

Eclipse *plugins* are Java programs that are loaded by the Eclipse runtime environment and added to the platform. They are connected to Eclipse using "extension points" provided by Eclipse, which describe different aspects of the Eclipse platform that can be extended by a given plugin (Beck and Gamma, 2003). The definition of the extension points used is stated in a XML-based "plugin manifest" that has to be provided together with each plugin.

One of the strengths of the Eclipse editors is the integration of helpful features. This includes syntax highlighting (using either color or font changes, or a combination), the ability to directly jump to a given line in the editor, and marking (recognized) issues. The marking facilities called "annotations" in Eclipse can include a mark in the left or right margin of the text editor next to the affected line. An x-shaped cross in a red circle mark in the left margin is used to indicate syntactical (or other) errors, together with a red mark in the scroll bar to the right. Additionally, the annotated text is indicated, often by a wavy red line, and an appropriate description is placed in the *Problems* view.

The *Outline* view provides a table of contents-like view of the editor contents. For a Java class, this view lists all methods and import statements; clicking on a line directly positions the text editor on the associated content line. Longer components in the text editor may also be folded to reduce visual clutter and make it easier to focus on the current area of work.

Finally, Eclipse editors can also support the user by *content assist*, often also called *code assist* or *code completion.* This feature lets the user choose from a pop-up list of constructs or completions possible at the current text caret position. It can be used both for the insertion of a single keyword or complete structure, such as an *if..else..* statement, and for the choice of a given method to be invoked, including the required invocation parameters. This type of support is only possible if the plugin is aware of the syntax of the underlying language or the set of classes and their methods, respectively. Both aspects of content assist can be very helpful and save time, especially for users who are new to the target language.

3 Animalipse: An Eclipse Plugin for AnimalScript

The Animalipse plugin was expected to provide the following functionality:

- support the creation and editing of AnimalScript as an Eclipse plugin;
- allow easy installation using the Eclipse plugin installation support;
- provide *cut, copy* and *paste* functionality, as well as *undo* and *redo*;
- allow the display of line numbers, animation step folding (showing only one line for a set of commands in the same animation step), automatic indentation of code lines, and syntax highlighting;
- locate and mark errors in the AnimalScript file;
- display a useful overview in the Eclipse *Outline* view;
- and finally provide content assist for the AnimalScript command notation.

An AnimalScript file edited in the plugin shall also be directly runnable from the plugin, so that the user does not have to start the required Animal system externally.

The Animalipse plugin is based on an Eclipse text editor and thus directly inherits some of the requested functionality, such as the support for *cut, copy* and *paste*, as well as the easy installation typical for Eclipse plugins.

3.1 Parsing AnimalScript Content

Some of the features listed above, especially for marking errors, syntax highlighting and content assist, require that the editor "understands" what is being edited. As the editor may also need to request the same piece of information multiple times, we decided to implement a document object model (DOM) for AnimalScript (AS-DOM) to allow for faster and more expressive access to information about the currently selected element or current editing position. The essential structure of the AnimalScript-DOM consists of a root element, metadata about the AnimalScript contents, and a set of animation steps. The steps are placed in ascending order, just as they would be for the animation. Each step can contain one or multiple animation commands.

The creation of a AnimalScript-DOM requires parsing the UTF-8 encoded AnimalScript contents from the text editor. By registering as an observer in the *IDocument* provided by the Eclipse editor classes, the parser can be informed automatically about changes in the code, and thus update its DOM. However, each key press triggers an update event, which would force the system to parse the complete script (again). Therefore, we decided to enforce a waiting interval of at least 2.5 seconds between two parsing iterations to prevent unnecessary continuous parsing of the editor contents. Of course, this interval can be adjusted by the end-user.

Animalipse does not directly use abstract syntax trees to support the parsing process, but parses elements on a line base. This is possible as AnimalScript mandates that each command will occupy exactly one line (and that each input line will contain exactly one command, if one ignores comments or the curly braces used to indicate steps). A command consists of a sequence of language elements separated by an arbitrary amount of whitespace. Each element can for example be numeric, a literal, or a keyword. The internal representation of the parsed elements is similar to an abstract syntax tree, called *ASAST* for AnimalScript-Abstract Syntax Tree. The definition of the tree is created when the plugin is started by parsing a BNF-like definition file, which makes a later adaptation of the AnimalScript language easy.

3.2 Content Assist

The content assist feature of Animalipse uses the content assist components provided by the Eclipse editors. Based on the internal representation of the script and the current position, the plugin creates a list of recommendations for content that could be used to complete the current selection. Compared to many other languages (including Java), this process is difficult for AnimalScript content: the notation used by AnimalScript contains many optional keywords or elements, making the number of possible completions at any given point comparatively high. After the list is populated, it is presented to the user, who is then prompted to choose one of the elements.

Similarly, locating syntax errors in a given AnimalScript input file is also difficult due to the profusion of optional elements. In this case, the large number of branches possible at (almost) any given point in the parsing process leads to a number of “wrong errors”: a command line is only syntactically incorrect if it does not match *any* of the possible syntactical rules. Or, to put it differently, if there is a way to parse a given line without a syntax error, the line is syntactically correct, and all parse errors when trying a different combination of optional elements have to be ignored. Additionally, the parser has to be able to detect when there is “too much” input in the current line: the command has been completely parsed, but contains additional elements that thus do not belong to the text.

Several nodes in the AnimalScript abstract syntax tree can be annotated with context information, as shown in Listing 1. The context definition provides additional details about the context of a given leaf in the AnimalScript abstract syntax tree. Line 1 shows the unique identifier of the regarded context element (*tupleOfTwoNaturalNumbers*), used for defining absolute coordinates. The preceding dollar sign indicates that this is the element to be defined. Each definition line starts with the “at” character @ and can provide the following information:

@info elements provide a user-readable text that describes the element.

@display provides a user-readable rendition of the terminal output in the editor. This is for example necessary to indicate the position(s) of whitespace.

@cursorhint specifies how many character the cursor has to be shifted to the left after inserting the definition shown in *@display* into the editor. In this example, choosing the element will lead to the inclusion of the text `( X , Y )` including all spaces. The value 7 for the cursorhint places the cursor seven positions to the left of its position after the insertion, and thus places it on the first coordinate inside the parentheses.

```
$tupleOfTwoNaturalNumbers
@info=A tuple of two natural numbers
@display=( X , Y )
@cursorhint=7
```

Listing 1: Context Definition Example

The combination of the presented features made the implementation of the parsing and content assist components of the Animalipse plugin far more difficult than we originally anticipated. However, we managed to overcome all obstacles and now have a full-fledged Eclipse plugin for AnimalScript.

3.3 Integration into Eclipse

The integration of the Animalipse plugin into Eclipse uses five different Eclipse extension points, as outlined in Table 1.

Plugin component	**Eclipse Extension Point**
AnimalScript editor	*org.eclipse.ui.editors*
Error marking	*org.eclipse.core.resources.markers*
Starting the animation	*org.eclipse.debug.ui.launchShortcuts*
New Animation Creation Wizard	*org.eclipse.ui.newWizards*
Plugin Preferences	*org.eclipse.ui.preferencesPage*

Table 1: Eclipse Extension Points used for the Animalipse plugin

The AnimalScript editor is based on the *org.eclipse.ui.IEditorPart* interface and extends the Eclipse *TextEditor* class. It handles AnimalScript files with the extension *.asu*. This editor is automatically started whenever the user opens or creates a file with this extension.

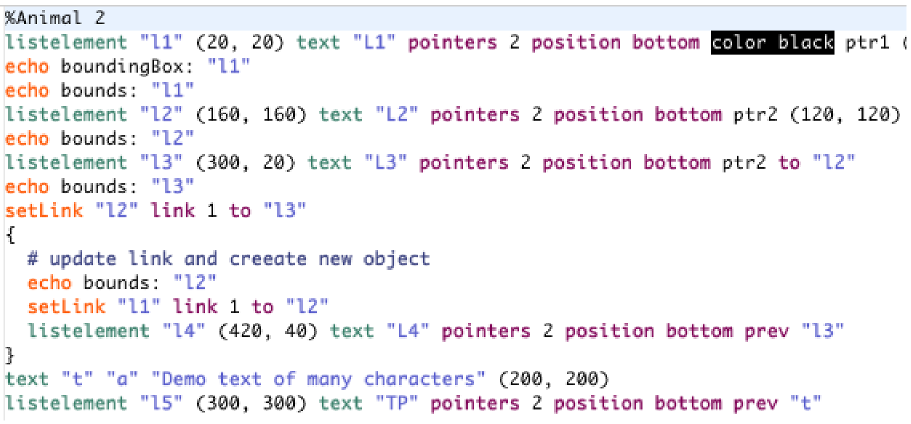
Other components of the plugin, such as code folding or the creation of the overview, similarly implement provided interfaces and extend existing Eclipse classes.

3.4 Installing the Plugin

To install the plugin, the user selects the *Help → Software Updates → Find and Install...* menu entry. After selecting "new features to install", a new remote site has to be created with the address `http://www.algoanim.info/Animal/download/Animalipse/` and confirmed by OK. After finishing the settings, a list of updates should appear and include Animalipse. After confirming all subsequent dialogs, the plugin will be installed and can be used after a restart of Eclipse.

4 Example Features of the Animalipse Eclipse Plugin

New AnimalScript files can be generated inside Animalipse by using the built-in creation wizard. The user first uses the *File* menu, toolbar or menu entry to create a new file of type AnimalScript. In the following dialog, the title of the animation, its width and height and

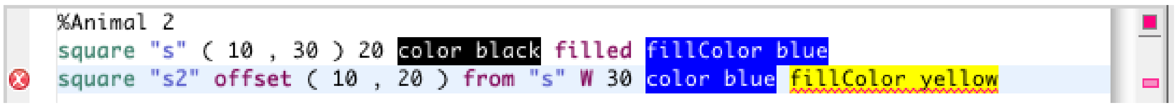


```
%Animal 2
listelement "l1" (20, 20) text "L1" pointers 2 position bottom color black ptr1 (
echo boundingBox: "l1"
echo bounds: "l1"
listelement "l2" (160, 160) text "L2" pointers 2 position bottom ptr2 (120, 120)
echo bounds: "l2"
listelement "l3" (300, 20) text "L3" pointers 2 position bottom ptr2 to "l2"
echo bounds: "l3"
setLink "l2" link 1 to "l3"
{
  # update link and creeate new object
  echo bounds: "l2"
  setLink "l1" link 1 to "l2"
  listelement "l4" (420, 40) text "L4" pointers 2 position bottom prev "l3"
}
text "t" "a" "Demo text of many characters" (200, 200)
listelement "l5" (300, 300) text "TP" pointers 2 position bottom prev "t"
```

Figure 1: The ANIMALIPSE text editor with syntax highlighting

the animation author and title can be specified. This information is then used to create the appropriate ANIMALSCRIPT header and open the resulting file in the ANIMALSCRIPT editor.

Figure 1 shows an example of the ANIMALIPSE text editor. Keywords used to define graphical objects are highlighted in green, operation keywords are shown in orange. All other keywords are shown in purple. Literal values, such as object names and Strings, are shown in blue. Color definitions are placed before a background of the chosen color. The original text is colored in a complementary color to be readable. Font definitions (not included in the example) are placed in italics. Finally, comments - introduced by the hash mark # - are shown in dark green. All color settings can be adjusted in the ANIMALIPSE plugin preferences.



```
%Animal 2
square "s" ( 10 , 30 ) 20 color black filled fillColor blue
square "s2" offset ( 10 , 20 ) from "s" W 30 color blue fillColor yellow
```

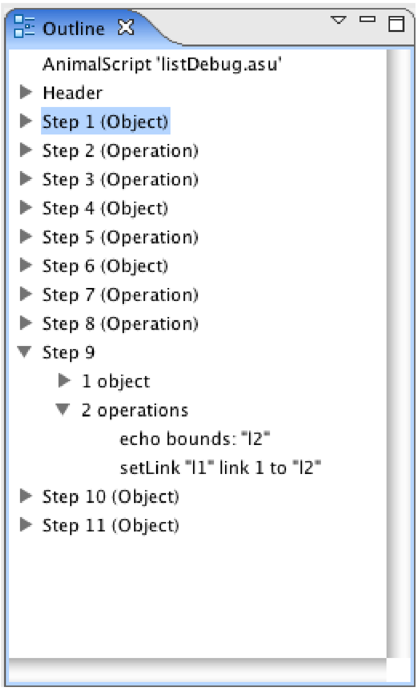
Figure 2: Indication of a syntax error by the ANIMALIPSE plugin

Figure 2 shows a view of a small code snippet with a syntax error (the keyword "filled" is missing, as can be seen by comparing the two code lines). The syntax analysis has to be triggered manually by selecting the *Search for errors...* entry in the context menu of the editor. The incorrectly placed keyword *fillColor* is shown with a wavy red underline. Additionally, the marker in the left margin and the line marker in the scrollbar to the right indicate the error, while the filled red square at the top right shows the presence of (at least) one error. Additionally, but not shown in Figure 2, a short error description appears in the Eclipse *Problems* view.

The outline of the rather simple animation shown in Figure 1 is shown in Figure 3. Each step can be folded or unfolded to show more details. If multiple operations take place in the same animation step, they are shown on separate lines. Clicking on an entry positions the cursor on the appropriate line in the text editor.

Content assist is provided whenever the user requests it explicitly by pressing the CTRL key together with the space key. Additionally, ANIMALIPSE offers content assist if a space character is entered. A small window pops up and offers the list of legal completions at this point, allowing the user to choose one or close the window and continue manually.

Finally, ANIMALSCRIPT content can be run directly by choosing the context menu entry *Run* → *Load in* ANIMAL. This requires that the user has first told the plugin where the ANIMAL jar file can be found by going to *Window* → *Preferences...*, selecting the plugin from the list, and entering or browsing for the location of the ANIMAL jar.



Outline

AnimalScript 'listDebug.asu'
- Header
- Step 1 (Object)
- Step 2 (Operation)
- Step 3 (Operation)
- Step 4 (Object)
- Step 5 (Operation)
- Step 6 (Object)
- Step 7 (Operation)
- Step 8 (Operation)
- Step 9
  - 1 object
  - 2 operations
    - echo bounds: "l2"
    - setLink "l1" link 1 to "l2"
- Step 10 (Object)
- Step 11 (Object)

Figure 3: The ANIMALIPSE outline, showing the structure of the animation

5 Summary and Future Work

In this paper, we have presented the ANIMALIPSE Eclipse plugin for creating, editing and debugging ANIMALSCRIPT-based AV content. The plugin is very easy to integrate into an existing Eclipse installation and behaves similarly to other established plugins, such as the Java Development Tools plugin for Eclipse. Users should therefore find it easy to use the plugin to become more proficient with ANIMALSCRIPT.

The abstract syntax tree model used for the ANIMALSCRIPT language allows extending the language's definition by modifying the BNF-based notation without touching the plugin code. However, a certain measure of caution has to be exerted when editing the file, to prevent the introduction of parsing errors. At the same time, the underlying notation could be exchanged by another language, such as JAWAA (Akingbade et al., 2003), by editing the notation file accordingly.

The other features described in this paper, such as syntax highlighting, searching for errors, and content assist, should also prove helpful. There are some minor issues with some of these components, which are due to the underlying language notation. For example, the choice lists for code assist can become very long, if the user requests assistance near the start of a command. This is due to the large number of optional keywords and components used throughout ANIMALSCRIPT. While this makes programming in ANIMALSCRIPT comfortable ("specify what you need and skip the rest"), it also leads to a large number of possible correct completions.

The error detection is not fully accurate; found errors will lead to parsing errors in ANIMAL, but not all errors during loading in the animation in ANIMAL may be detected by the plugin. For example, the user may request a certain transformation subtype, such as moving the nodes 3 and 4, on a structure that does not support this operation, for example a square. Syntactically, this command is correct, but it will lead to a semantic error when the execution is attempted. Therefore, the plugin cannot detect this type of error unless it were more tightly interwoven with ANIMAL's internal parsing process - which would slow down the processing of files.

Other issues that shall be addressed in the future include highlighting the use of the currently selected identifier or showing context information about elements as a tool tip. Automatic formating of AnimalScript files, similarly to the feature offered by the Java editor, would also be helpful. Finally, it would be interesting to remodel Animal's content generators as Eclipse wizards or create an internal Animal view. However, these aspects require another full-time student working on them as a Bachelor Thesis.

If you are interested in trying out Animalipse, please follow the steps described in Section 3.4. Constructive feedback is appreciated! We would also like to cooperate with others who wish to implement Eclipse plugins for "their" AV notation.

References

Ayonike Akingbade, Thomas Finley, Diana Jackson, Pretesh Patel, and Susan H. Rodger. JAWAA: Easy Web-Based Animation from CS 0 to Advanced CS Courses. In *Proceedings of the 34th ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2003), Reno, Nevada*, pages 162–166. ACM Press, New York, 2003.

Kent Beck and Erich Gamma. *Contributing to Eclipse. Principles, Patterns, and Plugins.* Addison-Wesley Longman, 2003. ISBN 978-0321205759.

Thomas L. Naps and Guido Rößling. JHAVÉ - more Visualizers (and Visualizations) Needed. In Guido Rößling, editor, *Proceedings of the Fourth Program Visualization Workshop, Florence, Italy*, pages 112–117, June 2006.

Thomas L. Naps, Jeff Lucas, and Guido Rößling. VisualGraph - A Graph Class Designed for Both Undergraduate Students and Educators. In *Proceedings of the 34th ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2003), Reno, Nevada*, pages 167–171. ACM Press, New York, 2003.

Guido Rößling and Tobias Ackermann. A Framework for Generating AV Content on-the-fly. In Guido Rößling, editor, *Proceedings of the Fourth Program Visualization Workshop, Florence, Italy*, pages 106–111, June 2006.

Guido Rößling and Bernd Freisleben. AnimalScript: An Extensible Scripting Language for Algorithm Animation. *Proceedings of the 32nd ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2001), Charlotte, North Carolina*, pages 70–74, February 2001.

Guido Rößling and Bernd Freisleben. Animal: A System for Supporting Multiple Roles in Algorithm Animation. *Journal of Visual Languages and Computing*, 13(2):341–354, 2002.

Guido Rößling and Silke Schneider. An Integrated and "Engaging" Package for Tree Animations. In Guido Rößling, editor, *Proceedings of the Fourth Program Visualization Workshop, Florence, Italy*, pages 23–28, June 2006.

Guido Rößling, Felix Gliesche, Thomas Jajeh, and Thomas Widjaja. Enhanced Expressiveness in Scripting Using AnimalScript V2. In *Proceedings of the Third Program Visualization Workshop, University of Warwick, UK*, pages 15–19, July 2004.

Guido Rößling, Silke Schneider, and Simon Kulessa. Easy, Fast, and Flexible Algorithm Animation Generation. In *Proceedings of the 13th ACM SIGCSE/SIGCUE International Conference on Innovation and Technology in Computer Science Education (ITiCSE 2007), Dundee, Scotland*, page 357. ACM Press, New York, NY, USA, 2007.

John Stasko. Building Software Visualizations through Direct Manipulation and Demonstration. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization*, chapter 14, pages 187–203. MIT Press, 1998.

A Java API for Creating (not only) AnimalScript

Guido Rößling, Stephan Mehlhase, Jens Pfau
CS Department, TU Darmstadt
Hochschulstr. 10
64289 Darmstadt, Germany

roessling@acm.org

Abstract

Generating animation content can be tedious and result in "messed-up" code. We present a Java API that can be used for generating AnimalScript-based animations. It was designed to be extendable to other output formats, such as SVG or other scripting languages. Apart from describing the use of the API, we also show a concrete example that was generated using the API to illustrate the API's expressiveness.

1 Introduction

Creating AV content is often a slow and tedious job. Automating this process offers faster - and reusable - generation of new animation content. However, the automation requires either an underlying language notation or a (hopefully well-designed) API, or both. If present, such features can make the content provider's job much easier, and may even allow end-users to create visualization content on-the-fly.

In this paper, we present a Java API for creating animation content which uses AnimalScript as the main output. However, the API can also easily be extended to support other notations or languages, making it an attractive tool for many AV content generators, including those that do not use the Animal AV system.

In the following, we will first briefly describe the underlying AnimalScript language. In Section 3, we then introduce the design for the Java API, followed by an example animation generated using the API. Sections 5 and 6, respectively, present a brief informal evaluation and comments on extending the API to other notations. Section 7 summarizes the work and outlines areas of future research.

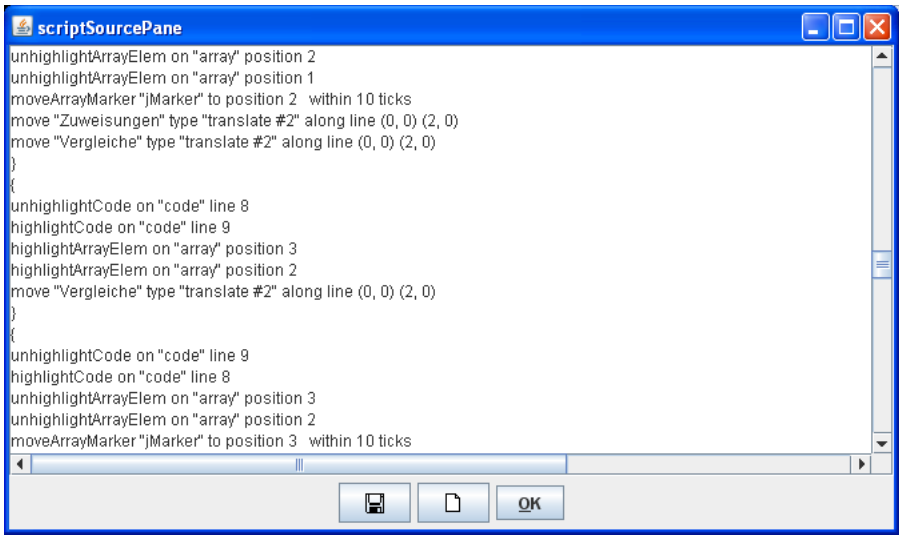
2 A Brief Overview of AnimalScript

AnimalScript (Rößling et al., 2004) is a versatile notation for programming algorithm visualization and animation (AV) content. AnimalScript offers a very flexible placement of elements, using either absolute values or an offset relative to another object's bounding box, text baseline, or an individual node in a polygon or polyline. Almost all animation effects can be assigned a relative starting time as an offset from the beginning of the associated animation step and a duration, which can be specified using either milliseconds or the number of animation frames. Figure 1 shows Animal's built-in editor for AnimalScript code.

Animation effects can also be given a concrete animation "method" as a String parameter, which further describes how the affected object(s) shall be animated. For example, the generic *move* command can be given a method name *translate #2*, which will change the behavior from moving the complete object to moving only the second node of the object(s).

Similarly to all other scripting notations, AnimalScript can easily be entered with a standard text editor. We have recently completed an Eclipse plugin for editing AnimalScript, including code assist and the check for syntax errors (Rößling and Schroeder, 2008).

AnimalScript contains a large selection of optional components in almost all commands, which makes it easier to write for hand-coded scripts. AnimalScript can also be automatically dumped while the underlying algorithm is being executed. However, just as for all other scripting-based approaches, this quickly leads to cluttered code for the actual algorithm. Figure 2 shows an example; here, the actual algorithm code is almost totally obscured by the manually generated visualization code. It may take an "expert" to see that this piece of code actually calculates the step width for Shell Sort.



```
unhighlightArrayElem on "array" position 2
unhighlightArrayElem on "array" position 1
moveArrayMarker "jMarker" to position 2  within 10 ticks
move "Zuweisungen" type "translate #2" along line (0, 0) (2, 0)
move "Vergleiche" type "translate #2" along line (0, 0) (2, 0)
}
{
unhighlightCode on "code" line 8
highlightCode on "code" line 9
highlightArrayElem on "array" position 3
highlightArrayElem on "array" position 2
move "Vergleiche" type "translate #2" along line (0, 0) (2, 0)
}
{
unhighlightCode on "code" line 9
highlightCode on "code" line 8
unhighlightArrayElem on "array" position 3
unhighlightArrayElem on "array" position 2
moveArrayMarker "jMarker" to position 3  within 10 ticks
```

Figure 1: Example ANIMALSCRIPT code

```
int oldDist = -1;
sb.append("  text \"dist\" \"dist:\" offset (40, 20) from \"array\" E");
for (dist = 1; dist < a.length / 9; dist = 3 * dist + 1) {
  toggleStep();
  if (oldDist != -1) {
    sb.append("  hide \"dist").append(oldDist).append("\"");
  }
  sb.append("  text \"dist").append(dist).append("\" offset (10, 0)");
  sb.append(" from \"dist\" baseline");
  oldDist = dist;
}
```

Figure 2: (Bad) Example of integrated ANIMALSCRIPT code generation

Before our research presented in this paper, several content authors had already recognized this situation as bad, and developed their own (specialized) "mini-API" to deal with exactly those contents they needed. However, the limited expressiveness of these APIs and (sometimes) their lack of a good software engineering architecture prevented other content authors from adopting any of these APIs. We therefore set out to develop an "official" Java API for generating ANIMALSCRIPT code that should be easy to use by current and future content authors. In Section 3, we will outline the basic approach for this API.

3 AnimalScript API Design

The main design goals for the ANIMALSCRIPT Java API were the following:

- Provide a "cleaner" way of generating ANIMALSCRIPT code than shown in Figure 2,
- Offer a "common" API that all content generators for ANIMAL can use,
- Use a clean inheritance hierarchy using both classes and interfaces as appropriate,
- Support the current language features of ANIMALSCRIPT in the Java API,
- Provide means to extend the API for future additions to ANIMALSCRIPT,

- Simulate the "optional elements" inside AnimalScript in a sensible way.

We decided to use one comparatively large *Factory* class (Gamma et al., 1995) called "Language". This abstract class provides a large set of methods for creating new objects of the proper type, e.g. there is a method called *newPoint* for creating a new *Point* object. In fact, there is often more than only one way to create a given object, using a different set of creation parameters. In this case, all such Factory methods will usually be mapped to a single (abstract) method. The concrete implementation of these methods is left to the implementing subclass of class *Language*.

It quickly became obvious that the good software engineering decisions for the third bullet point also opened the door for alternative output languages. Therefore, we decided to provide two "layers" of the API. The outer, abstract layer offers all interesting functions to the programmer, but uses an underlying inner, concrete layer to actually map the functions to output commands. In this way, it is easy to implement support for an alternative output format such as *SVG* (Ferraiolo, 2003). The switching between output instances is decided when the concrete instance of the *Language* Factory is created by providing one of the possible implementing subclasses, as shown in Listing 1.

```java
// Generate a new Language instance for content creation
// Parameter: Animation title, author, width, height
Language lang = new ConcreteLanguage("Quicksort Animation",
                 "Name of the author", 640, 480);
// Activate step control
lang.setStepMode(true);
```

Listing 1: Example for creating a new *Language* instance

3.1 Step control

The user can determine if all operations should be executed in sequence, or whether some operations may be grouped together and will be performed in parallel. This is achieved by turning the "step mode" on or off, respectively. If the step mode is turned on, a new step has to be introduced by a call to the *nextStep* method in class *Language*, which can also be passed an *int* value for a delay between consecutive steps, measured in milliseconds. An optional String parameter is used for the hypertext-like "table of contents" shown in Figure 3.

3.2 Defining Graphical Objects

The API for creating graphical objects is based on the definitions used by AnimalScript and the "standardized" XML of an ITiCSE 2005 Working Group (Naps et al., 2005), also found in the *XAAL* system (Karavirta, 2005). Essentially, almost all objects require the following creation parameters: location, value, name, display options and visual properties.

The location specifies the placement of the object and can be either absolute or relative to an other object or object node. The value depends on the type of the object, and can for example by an int array for an object of that type (see Listing 2 for an example of creating a visual object). The display options can be used to declare the object as *hidden* (not visible) or specify a delay after which the object should become visible. Finally, the visual properties describe the outward appearance of the object, such as its color. The visual properties will be examined in more detail in Section 3.3.

```java
// Create a new int[] object (will normally exist before)
int[] arrayContents = new int[] { 1, 3, 7, 5, 2, 6, 8, 4 };
// Parameters: location, value, name, display opt., visual props.
IntArray array = lang.newIntArray(new Coordinates(10, 30),
        arrayContents, "array", null, arrayProps);
```

Listing 2: Example for creating a new Graphical Object, here a visual int array

The API supports a large set of graphical objects: points, squares, triangles, rectangles, polygons, lines, polylines, circles and ellipses (including segments thereof), and text elements. It also supports many of the most relevant data structures used in Computer Science, such as graphs, arrays and matrices with a base type of *int* or *String*, code blocks including indentation, list elements with an arbitrary number of pointers, and three variants each of stacks and queues (conceptual, list- and array-based). All objects can be created with a single method invocation similar to lines 4-5 in Listing 2.

3.3 Defining Visual Properties

Apart from defining a graphical object, such as the *IntArray* shown in Listing 2, the user should also be able to define the object's visual appearance. The typical approach for this is to use either constructor invocation arguments, for example passing in the color of the object, or explicit API invocations to set the values after the "basic" object was created.

For many of the more complex objects, both of these approaches can be cumbersome due to the following reasons:

- Passing in the concrete values to the constructor can result in a bad design or usage issues. If all values are mandatory, that is, there is only one appropriate constructor, the number of parameters needed may become very large. For example, the IntArray in Listing 2 already has color values for its *outline, elements, cell background, element highlight* and *cell highlight*, the latter two of which are used if the user's attention is to be drawn to certain elements or cells. A user who simply wants to create an IntArray now has to worry about five colors (plus the font settings etc.) - or may leave them *null*, which may or may not lead to other problems.

  If the designer tries to help the user by allowing individual colors to be dropped from the list, the API will quickly have a large set of different IntArray constructors, which may also confuse users.

- The user may assign values to the five colors mentioned in the previous item by invoking one API method per color. This leads to a rather broad API with many simple (and similar) methods, as well as to about six additional method invocations - and thus, six more lines of Java code - to get the colors and fonts "right".

- Visual settings cannot easily be reused by either of the two previous approaches. For the second approach, the user could write a method that calls the appropriate API methods with the same settings. However, that still means additional work for the user.

We have therefore decided to follow an approach closer to *Cascading Style Sheets*. Here, the user can define a given combination of visual attributes once, and then reuse it as often as he likes. For example, to create ten text objects that have exactly the same visual settings, the user would first describe the visual properties once, and then pass these visual properties as a constructor parameter to the ten text objects. Listing 3 gives a brief example of how the visual properties for the IntArray object in Listing 2 can be specified. Note that the properties defined in Listing 3 are already used in Listing 2 in line 5.

```
1 // create array properties with default values
2 ArrayProperties arrayProps = new ArrayProperties();
3 // Redefine properties: border black, filled with gray
4 arrayProps.set(AnimationPropertiesKeys.COLOR_PROPERTY, Color.BLACK);
5 arrayProps.set(AnimationPropertiesKeys.FILLED_PROPERTY, true);
6 arrayProps.set(AnimationPropertiesKeys.FILL_PROPERTY, Color.GRAY);
```

Listing 3: Specifying visual properties (here, for array objects) once

On creation of a new visual properties object, as shown in Listing 3, all possible properties for the object are already set to a default value. Therefore, the user only has to overwrite those settings that he wants to adjust. For example, the array properties actually include 11 different properties.

3.4 Animating Graphical Objects

Once a graphical object was defined, it can be animated. The supported animation methods are invoked directly on the underlying graphical object. The parameters required for the animation effects depend on the type of effect chosen. However, they will usually include the following information:

- a *method name* for specifying subtypes of a given animation effect, as "translate #2" used in Figure 1. For example, a *color change* may concern the array's border or its fill color. The method name specifies which of these is actually meant to change;
- an *offset* relative to the start of the current animation step, which is usually 0 to indicate an immediate start;
- and a *duration.* Both offset and duration can be measured in animation frames or *ms.*

4 Example Animation Generated Using the Java API

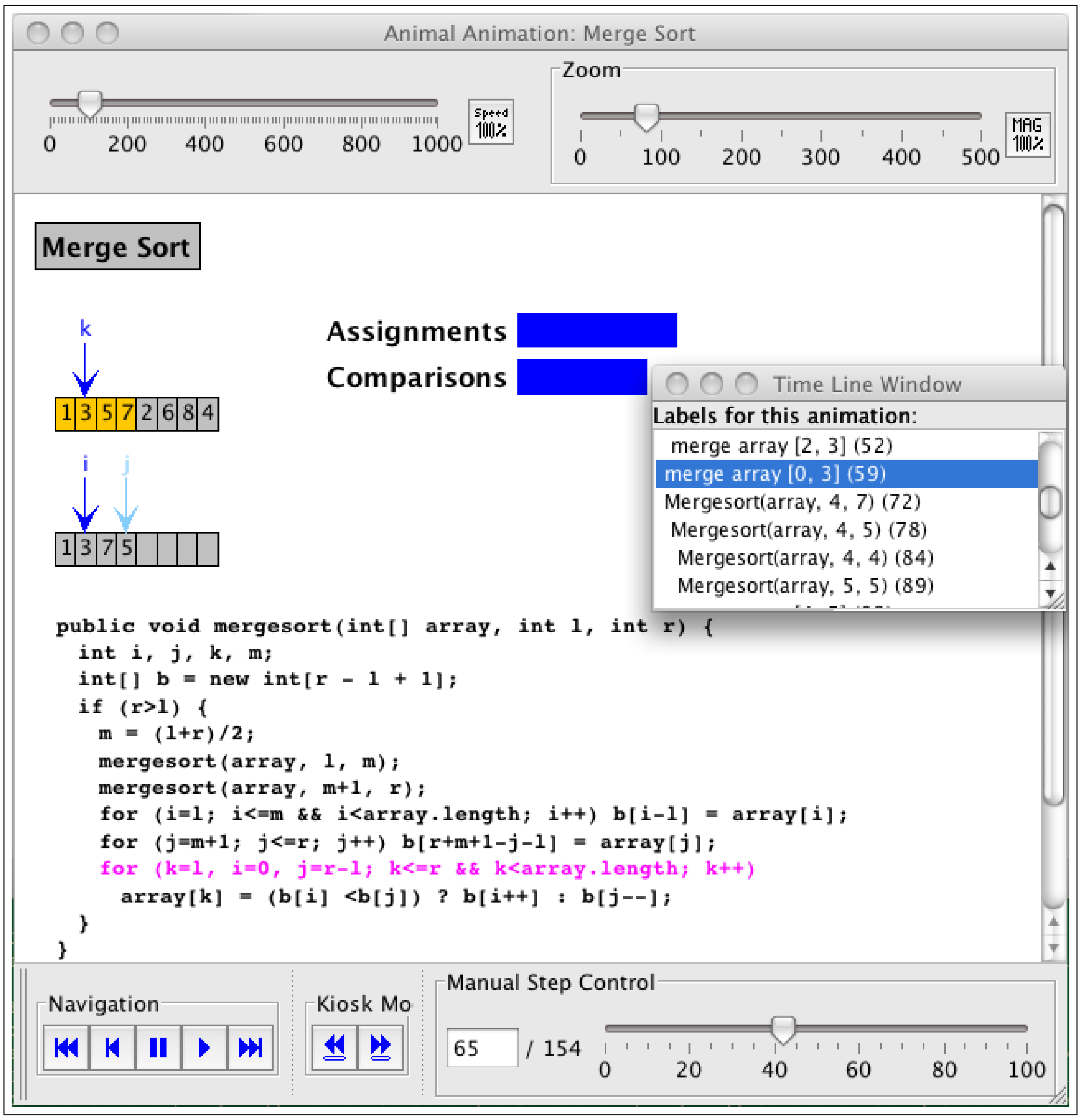
Figure 3 shows an example of a generated animation. The window contains the standard Animal controls for speed and zoom at the top. The animation can be navigated flexibly in both directions and also includes a "kiosk mode", which will display the animation step by step. The user can also jump ahead in the animation by entering the target step or dragging the slider shown on the bottom right. The window overshadowing the animation display contains the labels assigned to the animation, allowing instant access to the associated animation step. In the example, we started from step 59 which starts the merge operation of the first four array elements, and have now reached step 65.

The graphical objects shown in Figure 3 include a boxed text for the title and two int arrays defined similarly to Listing 2. The visual properties used for the arrays are identical to those described in Listing 3. The array markers *i, j,* and *k* are directly installed on the array. They can be moved to another index (using either a "jump" of no duration or a timed "move"), and can also automatically update their position if a change in a cell value occurs, for example by putting a larger number into an array cell.

The source code shown in the example animation is created with a set of API method invocations. The code indentation can be specified separately for each line. Animal will figure out the actual indentation to be used according to the font used for the code lines. The code also highlights the current code line in a configurable color (here, violet).

Finally, the two "counter boxes" for the number of assignments and comparisons also use the AnimalScript API. As there is no "counter box" feature in the API, we instead use a filled rectangle for each box. Whenever an assignment or comparison is made, the associated box is stretched by moving the end point by two pixels per assignment or comparison, as stretching the box by a single pixel would be difficult to see on many larger displays.

Listing 4 shows an excerpt of the concrete code for performing the first two parts of the merge process, when the sorted elements of the left and right subarray are copied into the temporary array. We do not expect the reader to fully understand the code as it is portrayed in this listing, but the main approach should hopefully be understandable. In line 1, we start a new step that is also provided with the "merge array [l, r]" information shown in the list of assigned labels. The parameters *l, r* indicate the current subarray bounds, while *depth* represents the recursion depth and determines the number of spaces to indent the label.



Labels for this animation:
merge array [2, 3] (52)
merge array [0, 3] (59)
Mergesort(array, 4, 7) (72)
Mergesort(array, 4, 5) (78)
Mergesort(array, 4, 4) (84)
Mergesort(array, 5, 5) (89)

```
public void mergesort(int[] array, int l, int r) {
  int i, j, k, m;
  int[] b = new int[r - l + 1];
  if (r>l) {
    m = (l+r)/2;
    mergesort(array, l, m);
    mergesort(array, m+1, r);
    for (i=l; i<=m && i<array.length; i++) b[i-l] = array[i];
    for (j=m+1; j<=r; j++) b[r+m+1-j-l] = array[j];
    for (k=l, i=0, j=r-l; k<=r && k<array.length; k++)
      array[k] = (b[i] <b[j]) ? b[i++] : b[j--];
  }
}
```

65 / 154

Figure 3: Screenshot from a generated animation

Lines 3-4 (as well several other lines in the listing) adjust the counters for the number of assignments and comparisons. In lines 3-4, these are for the initialization and condition of the *for* loop in line 6-12. Inside the loop, the values from the "main" array *array* are copied into the "helper" array *bArray*. The *null* parameters indicate that the effect happens instantaneously without duration or delay. Lines 13 and 25 toggle the display of the current code line from the first to the second and from the second to the final loop shown in Figure 3.

```
 1 lang.nextStep(createMergeLabel(l, r, depth)); // provide a label!
 2
 3 incrementNrAssignments(); // i = l in init of foor loop
 4 incrementNrComparisons(2); // i <=m, i < array.getLength() in for
 5                                    // copy first subarray
 6 for (i = l; i <= m && i < array.getLength(); i++) {
 7   bArray.put(i - l, array.getData(i), null, null); // copy value
 8   incrementNrAssignments();        // counts as one operation
 9   bArray.unhighlightElem(i - l, null, null); // unhighlight
10   incrementNrAssignments();        // i++ in for loop
11   incrementNrComparisons(2);       // comparisons in for loop
12 }
13 code.toggleHighlight("copyLeftside", "copyRightside");
14 lang.nextStep();                   // change step
15
16 incrementNrAssignments();          // j = m + 1 in for
17 incrementNrComparisons();          // j <= r in for loop
18 for (j = m + 1; j <= r; j++) {     // copy second subarray
19   bArray.put(r + m + 1 - j - l, array.getData(j), null, null);
20   incrementNrAssignments();
21   bArray.unhighlightElem(r + m + 1 - j - l, null, null);
22   incrementNrAssignments();
23   incrementNrComparisons();
24 }
25 code.toggleHighlight("copyRightside", "loop");
26 lang.nextStep();                   // next step: merge in loop
```

Listing 4: A subset of the code used for MergeSort

5 API Evaluation

The API presented in this paper is currently used by roughly 120 different algorithm animation content generators. To be fair, many of these generators only differ by nuances, such as the language used for textual output or for program code (e.g., Java versus pseudo code). About 45 of these generators previously used a generation approach similar to the "interwoven" code in Figure 2. Changing these to the API represented a fair amount of work, but was definitely useful, as the modified code is now far more readable and also far shorter.

Currently, a set of students is working with the Java API in a lab about algorithm visualization. We expect to be able to provide more information about their experiences for the final submission to the Workshop, and certainly for the presentation in Madrid, as our summer term has just started two weeks before the submission deadline. The first feedback is positive, claiming that is easy to start working with the API based on the initial English slides we provide at the ANIMAL home page (Rößling, 2008) under "Downloads".

6 Extending the API to other output languages

Extending the API to other output languages is very easy. The programmer first creates a new Java package for the new output language. A new base factory implementation of the abstract *Language* Factory then has to be created and implemented. Tools like Eclipse can help by automatically filling in the methods that have to be implemented; this is currently a set of 26 Factory methods for the objects listed in Section 3.2.

The implementation of the Factory methods then usually requires the creation of additional classes representing the created object, for example a *SVGIntArray* representing an *int[]* in SVG. This class only has to map the existing *IntArray* object into SVG; it does not have to provide any internal representation of the array or other "business logic".

The implementation for the AnimalScript language currently consists of 30 Java classes with a total of 6,336 lines (including all empty lines, comments, package and import statements). By far the largest class is the base Factory class with about 900 lines, many of which are either blank (87) or contain *import* (90) statements or *JavaDoc* comments (344). When we also disregard the declaration of methods, less than 300 lines of actual code are left.

7 Summary and Future Work

We have presented the basic design and use of the AnimalScript Java API. The API is a large help in separating the algorithm from the visualization code. It is easy to use, once the programmer has understood the concept of the *visual properties*, and highly expressive.

In the future, we want to extend the API to other output languages, especially the XML code defined by the ITiCSE 2005 Working Group (Naps et al., 2005), also used in the *XAAL* system (Karavirta, 2005). Other target formats include *SVG* and potentially *ActionScript* (used for Adobe Flash). We would also like to offer the API to any interested party; it can be freely downloaded (Rößling, 2008). This especially concerns AV content creators and the authors of other systems, who may be interested in adopting the API.

References

Jon Ferraiolo. Scalable Vector Graphics (SVG) 1.1 specification. `http://www.w3.org/TR/SVG`, September 2003.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

Ville Karavirta. XAAL - Extensible Algorithm Animation Language. Master's thesis, Department of Computer Science and Engineering, Helsinki University of Technology, 2005.

Thomas Naps, Guido Rößling, Peter Brusilovsky, John English, Duane Jarc, Ville Karavirta, Charles Leska, Myles McNally, Andrés Moreno, Rockford J. Ross, and Jaime Urquiza-Fuentes. Development of XML-based Tools to Support User Interaction with Algorithm Visualization. *SIGCSE Bulletin inroads*, 36(4):123–138, December 2005.

Guido Rößling. Animal2 Home Page. WWW: `http://www.algoanim.info/Animal2`, 2008.

Guido Rößling and Peter Schroeder. Animalipse - An Eclipse Plugin for AnimalScript. In *Proceedings of the Fifth Program Visualization Workshop, Universidad Rey Juan Carlos, Madrid, Spain*, page (in print), July 2008.

Guido Rößling, Felix Gliesche, Thomas Jajeh, and Thomas Widjaja. Enhanced Expressiveness in Scripting Using AnimalScript V2. In *Proceedings of the Third Program Visualization Workshop, University of Warwick, UK*, pages 15–19, July 2004.

A Design of Automatic Visualizations for Divide-and-Conquer Algorithms

J. Ángel Velázquez-Iturbide, Antonio Pérez-Carrasco, Jaime Urquiza-Fuentes
Departamento de Lenguajes y Sistemas Informáticos I, Universidad Rey Juan Carlos, C/ Tulipán s/n, Móstoles 28933, Madrid, Spain

angel.velazquez@urjc.es

Abstract

The paper addresses the design of program visualizations adequate to represent divide-and-conquer algorithms. Firstly, we present the results of several surveys performed on the visualization of divide-and-conquer algorithms in the literature. Secondly, we make a proposal for three complementary, coordinated views of these algorithms. In summary, they are based an animation of the activation tree, an animation of the data structure, and a sequence of visualizations of the substructures, respectively.

1 Introduction

An informal distinction is commonly accepted between program visualization and algorithm animation. The former term describes external representations that are closely tight to program source code. The latter term refers to external representations of the abstract behaviour of a piece of program, typically an algorithm. Given the lower abstraction level of program visualizations, they are frequently generated automatically, whereas the higher abstraction level of algorithm animations forces human intervention.

As effort is one of the main reasons for instructors not to be using visualization software in education, it is worthwhile to further explore different directions for program visualization (Naps et al., 2003). We have addressed a line of research consisting in generating program visualizations based on their underlying algorithm design techniques, e.g. divide-and-conquer or backtracking. Consequently, a student who wants to understand and analyze the behaviour of an algorithm of a common design technique could generate expressive, automatic visualizations of the algorithm.

We have designed an implementation framework (Fernández-Muñoz et al., 2007) to develop several visualization systems, one per design technique. To illustrate the feasibility of this framework, a first system was implemented to visualize recursion, called SRec (Velázquez-Iturbide et al., 2008). Now, we are addressing the design of a visualization system for a proper algorithm design technique, namely divide-and-conquer.

The goal of this paper is to present the design of visualizations adequate for the divide-and-conquer technique. In the second section, we present the result of several studies we performed on the visualization of divide-and-conquer algorithms in the literature. The third section contains our proposal, consisting in three complementary, coordinated views. Finally, we summarize our conclusions and future work.

2 A Survey of Visualizations of Divide-and-Conquer Algorithms

In this section, we show the results of several studies we performed on visualizations of divide-and-conquer algorithms.

We assume that the definition of divide-and-conquer algorithms is well known, but we list here the terms used in the rest of the paper. A problem solved by divide-and-conquer is decomposed into subproblems. They are recursively solved, resulting in subsolutions whose combination gives place to the solution of the original problem.

Divide-and-conquer algorithms often traverse and manipulate a data structure. Each subproblem is constrained to a part of the structure, i.e. a substructure. We only deal here

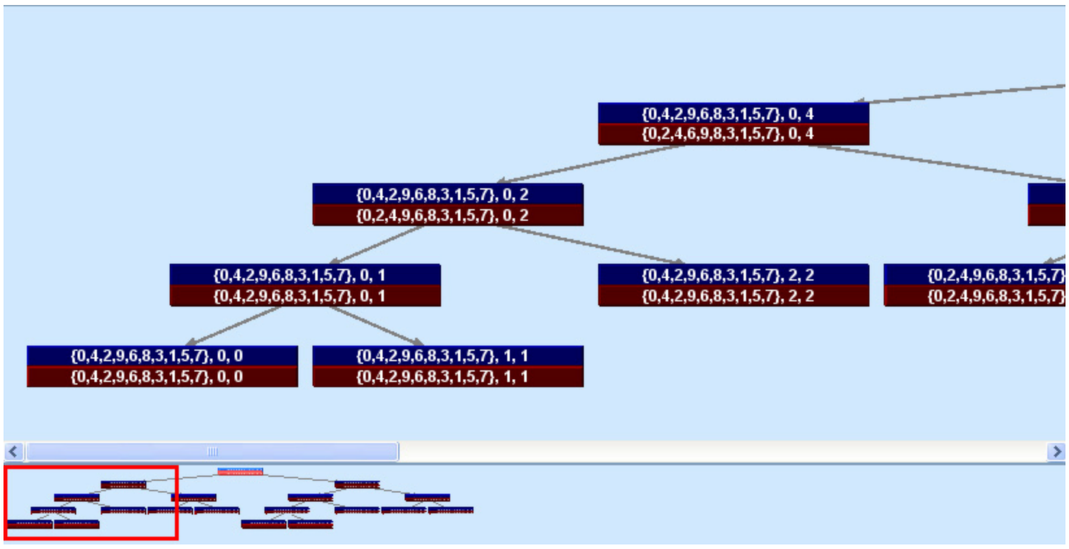


Figure 1: Activation tree for mergesort of {0,4,2,9,6,8,3,1,5,7} displaying array contents and indices

with one- and two-dimensional arrays, thus we often use the terms (sub)arrays, (sub)vectors and (sub)matrices. The most common and efficient way of delimiting subarrays is by using a range, defined with a lower and a higher index.

2.1 Visualizations of Recursion

Divide-and-conquer algorithms are a particular case of recursive algorithms. As a consequence, in a first approach, we tried to make use of visualizations for recursive algorithms. These visualizations are well known in CS: activation (or recursion) trees, the execution stack, traces, and multiple copies (of either code or variables). These visualizations are not equally effective for lineal and for multiple recursive algorithms. In particular, activation trees are more useful to display the behaviour of multiple recursive algorithms, e.g. divide-and-conquer algorithms.

Activation trees have limitations for divide-and-conquer algorithms. They are most effective for algorithms with a few, simple parameters or results. However, they are not as effective for larger data structures. The following two figures illustrate this inadequacy for mergesort. Both figures have been generated with SRec (Velázquez-Iturbide et al., 2008). The system allows the user to select the parameters or results to display; when a method does not return any value but produces side-effects, the original and final value of the parameters are displayed. It also has several facilities (zoom+panning, overview+detail) to handle large-scale activation trees. Finally, a user-defined colouring scheme can be used to differentiate input/output values, and the status of a call in the global process (executed, active or pending). In spite of all of these facilities, the resulting visualizations are not satisfactory.

Fig. 1 shows how the textual representation of arrays produces long nodes and therefore wide and shallow activation trees, difficult to browse and comprehend. In addition, the display of the complete array in every recursive call makes difficult identifying its corresponding subproblem and subsolution. Fig. 2 shows that omitting arrays from the nodes produces a more compact display, but changes of the array contents during the sorting process are not visible.

2.2 Visualizations in Algorithm Animations

Algorithm courses contain many divide-and-conquer algorithms, mainly mergesort and quicksort. Consequently, many creators of animation systems have developed animations for these algorithms. We have reviewed and analyzed the display of quicksort animations contained in

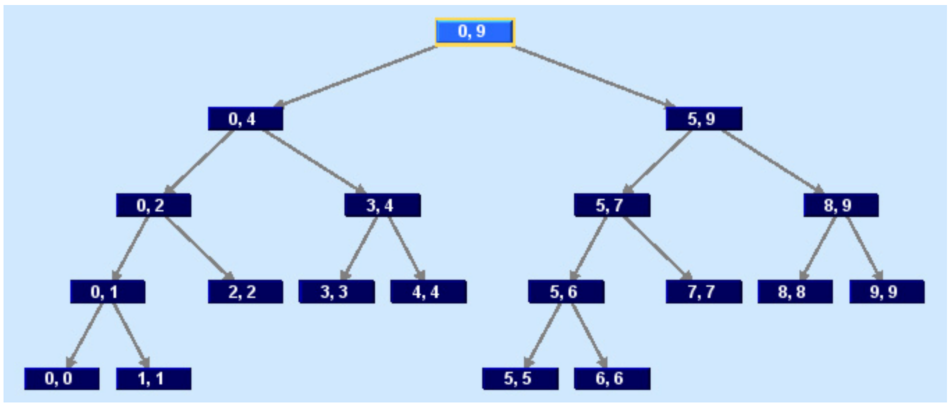


Figure 2: Activation tree for mergesort of {0,4,2,9,6,8,3,1,5,7} only displaying array indices

the reference books by Diehl (2007) and Stasko et al. (1998). Diehl (2007) shows one display (figs. 1.5 and 4.4) and Stasko et al. (1998) contains a higher number (pp. 41, 42, 91, 151, 158, 254, 374, 377, 379). This study reveals that some graphical representations are not useful because of several reasons:

- Some generic graphical representations are too poor. For instance, just displaying the data structure to manipulate is not expressive enough (see p. 42).
- A representation of a vector where each cell is displayed proportional to its size only is useful for certain problems (e.g. sorting). We find representations of cells as vertical bars (figs. 1.5 and 4.4), horizontal bars (pp. 374, 377) or in a "dots view" (pp. 41, 151, 379).

However, these animations also contain elements that can be successfully generalized to other divide-and-conquer algorithms:

- They use boxes to enclose the subarray handled by each recursive call (figs. 1.5 and 4.4).
- They classify the elements with respect to their status in the algorithm history by using shapes (pp. 41, 91, 158) or colours (figs. 1.5 and 4.4, pp. 91, 158, 374).
- The partition tree (pp. 41, 91, 158) is a variation of an activation tree that successfully combines recursion and vector representations. In summary, it consists of a tree isomorphic to an activation tree, where an element of the vector is displayed either as a node of the tree (when it is at its final position) or as a part of a subvector (when it has not been processed yet).

A different analysis of the visualization of recursive algorithms can be found at Stern and Naish (2002). They propose differentiating three kinds of algorithms, depending on how they handle a data structure (namely, algorithms that modify, traverse or construct it), rather than considering their recursion scheme. However, it is not obvious whether their visualizations can be generalized: the visualizations they propose contain vectors for the first class of algorithms, and trees for the other two classes. The visualization included in the article for the former class corresponds to a divide-and-conquer algorithm (namely, quicksort). It displays horizontally the vector and underlies it with horizontal bars that mirror recursive calls.

2.3 Visualizations in Textbooks

A comprehensive study on the visualization of divide-and-conquer algorithms must consider representations used by CS instructors. Consequently, we made a study of some of the most prestigious textbooks on design and analysis of algorithms (Fernández-Muñoz and Velázquez-Iturbide, 2006). The selection was necessarily arbitrary, but we consider it was representative of high-quality textbooks on algorithms (Aho et al., 1983; Alsuwaiyel, 1999; Baase and Gelderl, 1988; Brassard and Bratley, 1996; Cormen et al., 2001; Gonnet and Baeza-Yates, 1991; Goodrich and Tamassia, 2001; Horowitz and Sahni, 1978; Johnsonbaugh and Schaefer, 2004; Levitin, 2003; Manber, 1989; Parberry, 1995; Sahni, 2000; Weiss, 1999)

We summarize our findings, after discarding visualizations specific of any problem:

- It is common to include a visualization illustrating the inductive definition of the recursive algorithm by displaying its elements: problem, subproblems, subresults, and result.
- It is common to include a visualization of the activation tree. There are many variants in its graphical representation:
  - Visualize either a single tree or two parallel trees, where the second one illustrates the auxiliary operation (e.g. partitioning in quicksort).
  - Display either the activation tree or a sequence of visualizations of the data structure. Notice that the latter is an implicit tree, since it corresponds to its traversal.
  - Display either the delimiting indices or the contents of subarrays.
  - Display either the original or the final values contained in a data structure.
  - Display the activation tree in an either ascending or descending layout. We also find the join display of both, representing the advance and return phases of recursion.
- It is common to include a visualization of the data structure, complemented with some representation of the partitioning performed by the divide-and-conquer algorithm:
  - By means of nested boxes enclosing subarrays.
  - As a sequence of successive states of the substructures handled by the successive calls. Each substructure is either aligned according to its delimiting indices or laid out isomorphic to the recursion tree.

3 A Proposal for Automatically Generated Visualizations of Divide-and-Conquer Algorithms

Based on the findings of the previous section, we propose to three (coordinated) views of the behaviour of divide-and-conquer algorithms. We have imposed an additional requirement on our visualizations: they must be applicable to both one- and two-dimensional arrays, i.e. vectors and matrices.

- An animation based on the activation tree. Each node is complemented with a visualization of the substructure it focuses on.
- An animation based on the data structure. It is complemented with a schematic diagram of its partitioning by the algorithm.
- A sequence of visualizations of the substructures.

We elaborate these views in the following subsections.

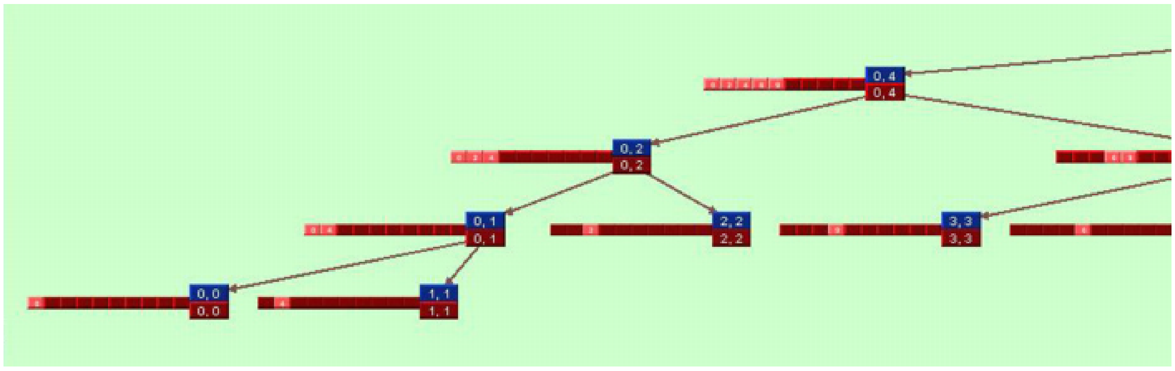


Figure 3: Visualization for mergesort of {0,4,2,9,6,8,3,1,5,7} based on the activation tree

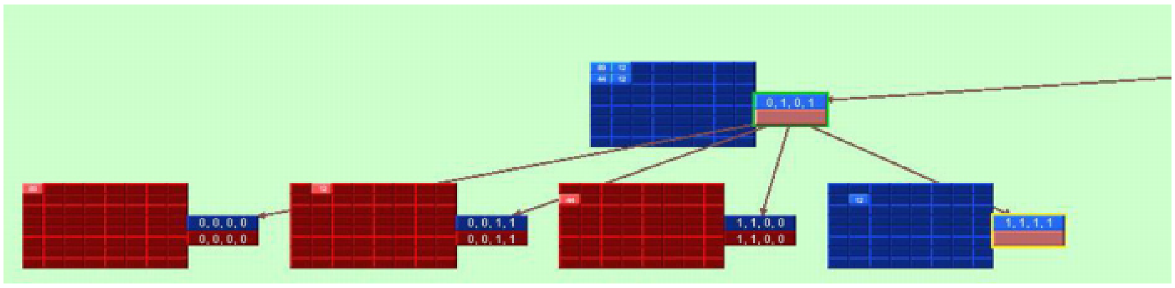


Figure 4: Visualization for transposition based on the activation tree

3.1 Animation Based on the Recursive Process

Fig. 3 shows an example of this view. Opposed to Fig. 1, each array is visualized once in each node. In addition, the application of a user-defined colouring scheme to the array allows to determine at a glance which subarray is the focus of the recursive call, as well as whether it is the subarray state at the entry or exit of the call. For the former issue, we recommend using different tones of the same colour, and for the second one, different colours. In the figures, the blue and red colours are respectively used to represent input and output values.

This view can be applied to matrices as well. Fig. 4 corresponds to a divide-and-conquer algorithm transposing a square matrix (an inefficient one!).

3.2 Animation Based on the Data Structure

This view provides a discrete animation of the successive states of the data structure to manipulate. It displays vectors and matrices in a conventional format. A set of bars is displayed below, that mirror the recursive process by underlying the subarray delimited by each recursive call. Fig. 5 shows an example of this view.

A colouring scheme is applied to the underlying bars, as well as to their associated subarrays. A first colour (red in Fig. 5) is used to mark recursive calls whose execution is over, as well as their corresponding subvectors. A second colour (blue in Fig. 5) is used for recursive calls whose execution is pending, as well as their corresponding subvectors. Tones of the two

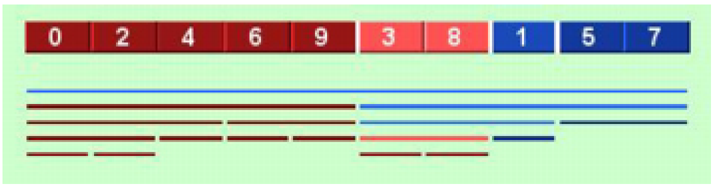
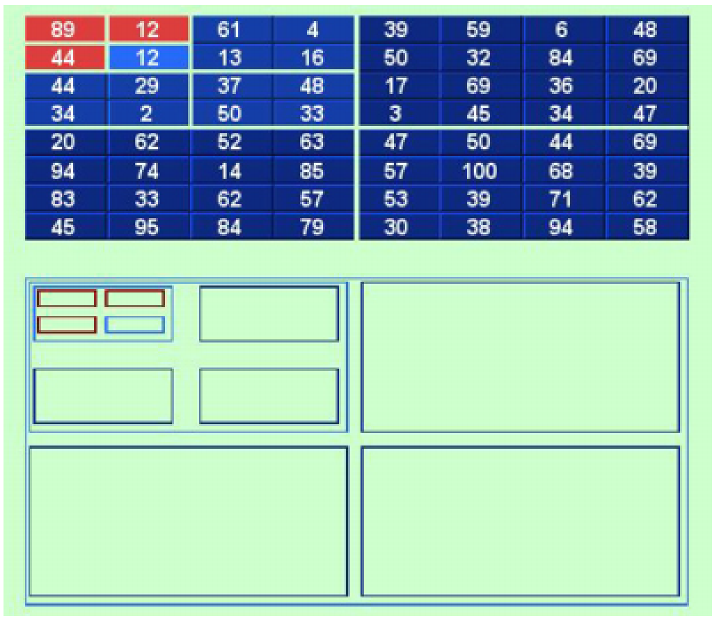


Figure 5: Visualization for mergesort of {0,4,2,9,6,8,3,1,5,7} based on the data structure



89	12	61	4	39	59	6	48
44	12	13	16	50	32	84	69
44	29	37	48	17	69	36	20
34	2	50	33	3	45	34	47
20	62	52	63	47	50	44	69
94	74	14	85	57	100	68	39
83	33	62	57	53	39	71	62
45	95	84	79	30	38	94	58

Figure 6: Visualization for transposition based on the data structure

colours are used to represent the distance of each call in the activation tree to the active call. The active call is always coloured light, and more distant nodes are coloured darker. In Fig. 5, the left part of the array is already sorted, being the active call focused at the subarray 3,8 and about to exit.

This view can be applied to matrices as well. The set of horizontal bars is replaced by a set of nested boxes cueing submatrices. Fig. 6 illustrates this for the algorithm to transpose a square matrix. Here, the algorithm only has completed three base cases and is focused on the fourth one.

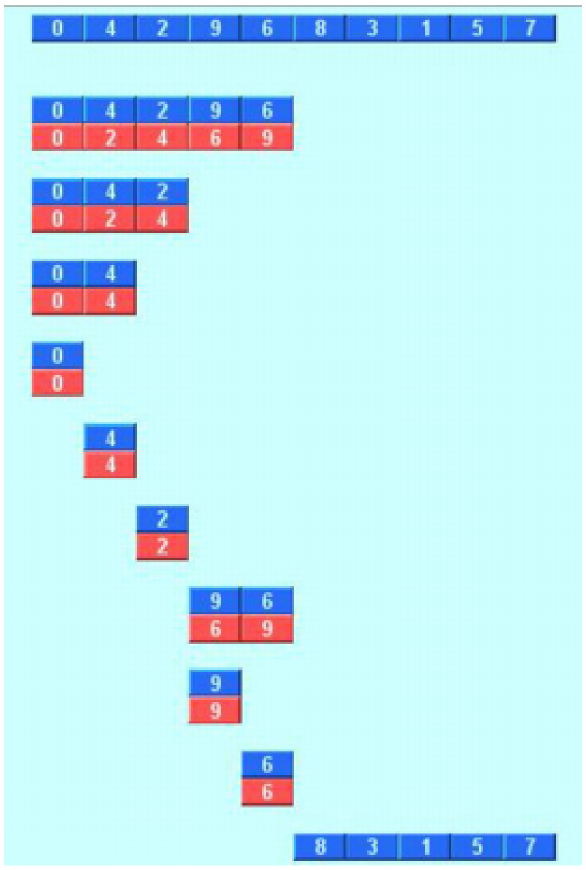
3.3 Sequence of Visualizations of the Data Structure

A third view displays a sequence of visualizations of the data structure, displayed top-down. Every time a recursive call is invoked, a new visualization of the array is displayed at the bottom of this view. In order to highlight the recursive process, each line only contains the subarray focused by its associated call, indented according to its delimiting indices.Every time a recursive call exits, a visualization of the resulting subarray is displayed below the original subarray displayed on call entry. Again, the use of colours allows differentiating them. Fig. 7a shows this view for the mergesort algorithm. The left part of the array is already sorted and a call has been made to sort its right half.

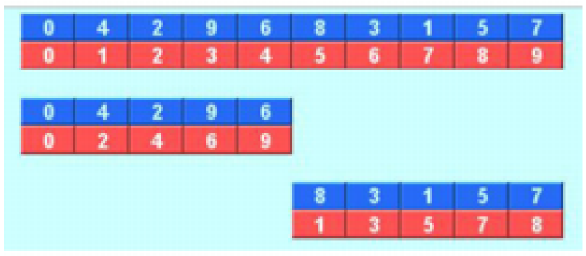
The main advantage of this view is that it allows generating a visualization that mirrors the inductive definition of the recursive algorithm. Fig. 7b illustrates this feature for mergesort. By selecting the animation control to jump over a recursive call and hiding the visualization of its underlying displays, the resulting display just consists of the original array, the two subarrays focused by the two recursive calls, the output subarrays of these recursive calls, and the final array.

4 Conclusions and Future Work

Custom visualizations for particular algorithms, as shown in algorithm animations, can be the most expressive. However, the effort necessary for manual generation of each particular animation is prohibitive. Therefore, we argue for using program visualizations as an alternative, effortless approach.



(a)



(b)

Figure 7: Full (a) and simplified (b) sequence of visualizations for mergesort of {0,4,2,9,6,8,3,1,5,7}

We have shown two results in this paper. Firstly, we have presented the results of examining visualizations of recursion as well as divide-and-conquer visualizations available at the literature. Secondly, we have proposed three program visualizations for divide-and-conquer algorithms. Two of them are respectively based on the animation of activation trees and of the data structure; both are mixed in the sense of displaying code and data elements. A third visualization is a sequence of substructures, and is capable of illustrating the inductive definition of the algorithm.

We have implemented a working prototype of the design presented here. However, more work is necessary to become a fully operational system. Usability evaluations performed by experts (i.e. instructors) and in sessions with students are important to assess their validity, as described in (Velázquez-Iturbide et al., 2008) for the SRec system.

References

V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms*. Addison Wesley, 1983.

M.H. Alsuwaiyel. *Algorithms Design Techniques and Analysis*. World Scientific, 1999.

S. Baase and A. Van Gelderl. *Computer Algorithms*. Addison Wesley, 1988.

G. Brassard and P. Bratley. *Fundamentals of Algorithmics*. Prentice-Hall, 1996.

T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

S. Diehl. *Software Visualization.* Springer-Verlag, 2007.

L. Fernández-Muñoz and J.Á. Velázquez-Iturbide. A study on the visualization of algorithm design techniques (in spanish). In M.Á. Redondo, C. Bravo, and M. Ortega, editors, *VII Congreso Internacional de Interacción Persona-Ordenador*, pages 315–324, 2006.

L. Fernández-Muñoz, A. Pérez-Carrasco, J.Á. Velázquez-Iturbide, and J. Urquiza-Fuentes. A framework for the automatic generation of algorithm animations based on design techniques. In E. Duval, R. Klamma, and M. Wolpers, editors, *Creating New Learning Experiences on a Global Scale - EC-TEL 2007*, volume 4753 of *LNCS*, pages 475–480, 2007.

G.H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures in Pascal and C.* Addison-Wesley, 2nd edition, 1991.

M.T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java.* John Wiley & Sons, 2nd edition, 2001.

E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms.* Pitman, 1978.

R. Johnsonbaugh and M. Schaefer. *Algorithms.* Pearson Prentice Hall, 2004.

A. Levitin. *The Design and Analysis of Algorithms.* Addison-Wesley, 2003.

U. Manber. *Introduction to Algorithms.* Addison-Wesley, 1989.

T.L. Naps, G. Roessling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, and J.Á. Velázquez-Iturbide. Exploring the role of visualization and engagement in computer science education. *SIGCSE Bulletin*, 35(2):131–152, 2003.

I. Parberry. *Problems on Algorithms.* Prentice Hall, 1995.

S. Sahni. *Data Structures, Algorithms and Applications in Java.* McGraw-Hill, 2000.

J. Stasko, J. Domingue, M.H. Brown, and B.A. Price, editors. *Software Visualization.* The MIT Press, 1998.

L. Stern and L. Naish. Visual representations for recursive algorithms. In *33th SIGCSE Technical Symposium on Science Education, SIGCSE 2002*, pages 196–200, 2002.

J.Á. Velázquez-Iturbide, A. Pérez-Carrasco, and J. Urquiza-Fuentes. Srec: An animation system of recursion for algorithm courses. In *13rd Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE 2008*, page In press, 2008.

M.A. Weiss. *Data Structures and Algorithm Analysis in Java.* Addison-Wesley, 1999.

A First Set of Design Patterns for Algorithm Animation

Guido Rößling
CS Department, TU Darmstadt
Hochschulstr. 10
64289 Darmstadt, Germany

roessling@acm.org

Abstract

Design Patterns are extremely helpful in preventing programmers from "reinventing the wheel". However, the algorithm animation area does not yet seem to have any Design Patterns, although there are several design issues that have to be resolved in many systems. We present two Design Patterns that address two central points in flexible algorithm animation systems: reverse playing and conceptual uncoupling to allow for easy extension.

1 Introduction

Design Patterns in the Computer Science context go back to the book by Gamma et al. (1995) which mapped the original concept of the architect Christopher Alexander to software development. The book provides approaches for solving problems that may come up in several different applications by following the same basic "idea" or approach, typically using a set of cooperating classes.

For algorithm visualization (in the following, abbreviated as AV), we could not find a "real" description of design patterns in use, although the problems faced by many systems are also similar. For example, users will often find it interesting or even important to be able to step backwards through the visualization (Anderson and Naps, 2001), without (for them) arbitrary limitations, such as a limited undo stack. Additionally, the process of stepping backwards should be reasonably fast.

Since no AV system is likely ever to be really "complete", means for introducing extensions or reconfigurations need to be provided. An interested developer should find it relatively easy to include a new primitive or data structure, provide a new animation effect or state transition, or do both. However, this should be possible without forcing the developer to take care of everything at the same time. Additionally, the developer may not have—or even want to have—a deeper understanding of the underlying system. Therefore, he or she should not be forced to modify existing components, as this decreases the motivation to "provide just a small change", and also increases the risk of making parts of the systems unusable.

In this paper, we define two initial AV patterns, in the hope that they will serve as a first "stepping stone" for the definition and refinement of additional AV-related design patterns.

Section 2 briefly summarizes the main elements of design patterns. Section 3 presents a pattern for easy navigation in both directions without the need of an undo stack. Section 4 introduces a pattern for enabling the extension of different aspects of an AV system without touching other parts. Section 5 summarizes the patterns in this paper and outlines future AV-related pattern research aspects.

2 An Extremely Brief Description of Design Patterns

Design patterns describe problems that occur many times in different parts of our environment. By describing the core of the solution to the problem, the same basic approach can be used to solve the problem, although the actual code is most likely different each time (Gamma et al., 1995, p. 2). A Design Pattern, as described in the basic book by Gamma et al. (1995), has five essential elements:

The pattern name is used to refer to the pattern. It provides a common understanding of what is referred to, assuming that all readers are familiar with the given pattern.

The intent describes the intention of the pattern in a single sentence.

The problem is a description of the situation that is addressed by the pattern.

The solution describes the components that can be used to solve the problem. It does not describe a concrete implementation, but rather the elements that are used to reach a concrete implementation, to allow for easier reuse and adaptation.

The consequences describe the results and trade-offs of applying the pattern. While the use of a pattern may increase the flexibility of the software, it may also affect the runtime or the amount of memory needed.

The book by Gamma et al. (1995) lists many other aspects of a pattern that can be listed, such as the *motivation* and *sample code*. However, we will not strictly adhere to the format for the sake of clarity and brevity.

We will use the pattern name as the name for the sections. The other elements will appear as subsections.

3 Reverse by Fast Forward

3.1 Intent

Reverse by Fast Forward supports flexible unbounded bidirectional navigation.

3.2 Problem

During a visualization of an algorithm, a user may become confused at some stage. Additionally, the visualization of a user-written algorithm may show a bug that needs to be tracked backed to its origin. Both situations benefit from the ability to easily step back to the previous step or steps. Here, a *step* is taken to be a closed set of operations that happen at the same basic time and represent complete actions. Thus, one move of an object along a line will always be part of exactly one step, even if it is rendered using a set of intermediate animation frames. However, the next step may contain another move of the same object.

The standard approach of supporting stepping backwards is to keep an undo stack of previous visualization states. In some systems, this may be a stack of static images, while other systems need to store a complete object representation. Both situations are usually limited by the amount of memory reserved for the stack, and may be further reduced by the size of the shown content. For example, an animation that covers 200 animation steps may require the storing of 199 previous step states for undo. If intermediate animation frames are to be stored, the number rises very quickly. The same is true if the undo stack tries to mirror the user's movement through the animation, and thus may have to store the same intermediate state multiple times as the user steps back through the animation.

The importance of being able to navigate back to a well-understood step in the display is also discussed in research papers (Rößling and Naps, 2002; Rößling and Naps, 2002). It is, however, not trivial to find a good solution for fast and arbitrary navigation, as shown by the statement that "efficient rewind [is] one of the most 'open questions' in AV" (Anderson and Naps, 2001).

3.3 Solution

We use a technique called "reverse by fast forward", a term coined in a discussion between the author and Amruth Kumar during a SIGCSE session break some years ago.

The main limitations of the classical undo stack have been described above: it may reach the fixed memory limit quickly, consumes much memory, and may redundantly store the same set of objects in different positions. Our proposal may seem counter-intuitive at first:

we navigate backwards by quickly moving forwards from a well-defined position in the AV contents. A similar approach is also used in reverse debugging and checkpointing (Boothe, 2000).

For this approach to work, the following conditions must be met:

- The content must have a certain structure, such as separate steps, to allow for a meaningful definition of "current" and "previous step", as well as for the "start" of the contents.
- The objects and transformations must be encoded in a way that allows executing them multiple times, always producing the same results. In practice, it is enough if a copy of the operations is stored even after they have been executed. "Execute and forget"-like operations, such as in the AV system *JAWAA* (Akingbade et al., 2003), which parse the current command, execute it, and then forget about it, are not suited for this approach.
- Two sets of objects must be stored: the original objects as they were initially defined in the animation, and one set of clones of these objects. Thus, the approach takes twice as much memory for storing objects as the normal approach would require.
- The transformation of the graphical objects can be visualized by the system, but can also be executed "quickly" without executing any visualization code.

Instead of executing a given operation or animation step on the original objects, the system will perform the following steps:

1. Determine the animation step the user wants to reach.
2. Ensure that the chosen step actually exists. In the case of manual step input, the user might provide a step number that does not exist, or may navigate forwards beyond the end or backwards beyond the start of the animation. If the chosen step does not exist, stop at this stage.
3. Clone all original graphical objects and place them in an appropriate data structure, such as a hashtable or list.
4. For all steps between the initial state and the target step, quickly perform all transformations on the cloned graphical objects without visualizing the effects.
5. Once the target step is reached, resume normal operation.

In most cases, executing the actual transformations on the objects without creating any visualization contents or updating the display will be much faster than the visualization. Practical experience with the ANIMAL system (Rößling and Freisleben, 2002) shows that the gap between pressing the "go backward" or "go forward" button and the update of the display is usually not or only barely noticeable, even for large animations.

To increase the performance of the display, the clones from the previous animation step may also be stored. If the user requests the next step to be displayed, the first four steps of the item list can be skipped and execution can directly continue. Additionally, snapshots of steps at certain intervals (e.g., every ten steps) can also be taken to support faster navigation. However, this will also increase the amount of memory needed, and may severely harm the animation speed if the user is actively editing the animation, forcing the system to continuously update the "snapshots" - a situation that does not occur in the other application areas (Boothe, 2000).

Arbitrary animation steps can also be dynamically performed in reverse direction if the animation effects are coded appropriately. This is used in the ANIMAL AV system (Rößling and Freisleben, 2002) to allow fully flexible bidirectional navigation even inside steps, letting "objects fly backwards".

As the pattern requires only the storage of the original and the cloned objects, no UML diagram is given.

3.4 Consequences

The *Reverse by Fast Forward* pattern has a set of consequences:

- The users can always jump to any arbitrary point in the animation. They are not restricted to the *next* or *previous* step, predefined points (such as *start, end*) or a limited number of steps to be reversed.
- Even object-destroying operations, such as scaling by a factor of 0 or multiplying by 0 can be "reversed", as the previous state of the object is retained through the cloning approach.
- The amount of memory used by the application increases, as each graphical object has to be stored twice. As the clones have to be prepared at each step (see 3. in the list above), this may also lead to increased memory fragmentation and garbage collection, which can impact the runtime.

4 Request Handler

4.1 Intent

Request Handler decouples animation effects from graphical objects, making both more flexible and easier to extend.

4.2 Problem

In AV and regular graphics systems, the user typically has (at least) two different abstractions: the *graphical objects* and the *animation* or *editing effects*. Graphical objects may be primitive, such as a text or square, or complex, such as an array or a tree. They may also encapsulate special semantics for some operations, or provide object-specific behavior. For example, changing the fill color of a circle representing a traffic light to red is different from just coloring a "normal" circle; swapping elements requires an appropriate visual rendition to be easily visible to the end-user. Typically, the graphical objects are responsible for storing their current state and can be requested to paint themselves. Animation effects are responsible for changing the graphical objects, optionally also using timing specifications such as the time to wait before the effect starts or its total duration.

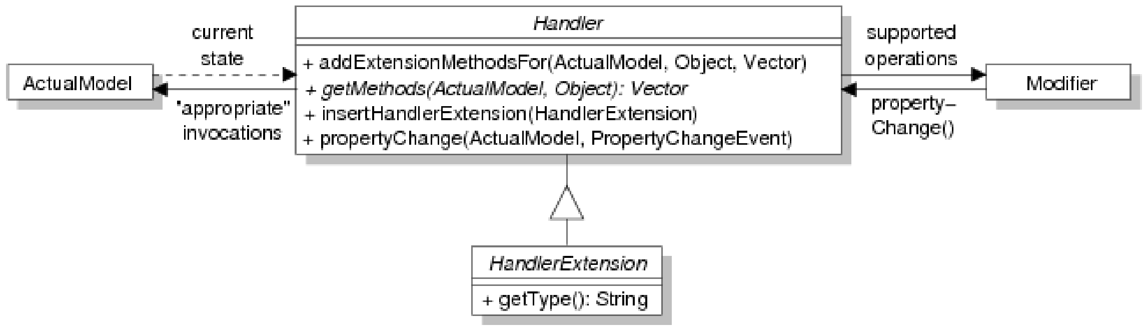
Interested developers should be able to implement a new graphical object without touching the existing animation effect. They should also be able to add a new animation effect without having to modify the existing graphical objects, and should be able to avoid providing a hard link between these entities. Finally, existing animation effects that differ only in small aspects should be modifiable without having to touch the animation effect. For example, if an animation effect for changing a color exists, there should be no need to implement a "fill color" change effect, or even modify the existing animation effect.

A standard approach is to incorporate a design such as *MVC* (Model, View, Controller), where the *Model* is the graphical object, the *Controller* role is assumed by the animation effect, and the *View* is the graphical rendition of the object. However, this does not provide the necessary uncoupling described above.

The intention of the described pattern is also similar to the "expression problem" described by Odersky and Zenger (2005). However, their approach (and other related approaches) requires special mixin features that are not available in Java. Related other approaches as in (Torgersen, 2004) require Java generics, and thus an explicit implementation in separate classes.

4.3 Solution

Use a *Request Handler* class to intercept the direct interactions between graphical objects and transformation effects, as shown in Figure 1. Here, the *ActualModel* entity represents the graphical object. It is aware of its current state and can therefore answer requests by the associated *RequestHandler*.



Handler
+ addExtensionMethodsFor(ActualModel, Object, Vector)
+ getMethods(ActualModel, Object): Vector
+ insertHandlerExtension(HandlerExtension)
+ propertyChange(ActualModel, PropertyChangeEvent)
ActualModel
current state
"appropriate" invocations
supported operations
property-Change()
Modifier
HandlerExtension
+ getType(): String

Figure 1: *Request Handler* architecture

The *Modifier* in Figure 1 is the animation effect. It needs to be aware of the operations that can be performed on the selected graphical object. For example, a generic “color changer” animation effect should not offer a “fill color” change operation for unfilled objects, such as texts, points or lines.

Finally, the *RequestHandler* acts as an intermediate object between the two entities. It can be further refined by subclasses of *HandlerExtension*. The *RequestHandler* offers only four methods:

addExtensionMethodsFor looks for existing extension methods that were implemented in one of the (possibly multiple) HandlerExtension objects and adds them to the Vector returned by *getMethods*.

getMethods returns the Vector of all possible concrete transformations for the combination of *ActualModel* and parameter passed in.

insertHandlerExtension is invoked to add a new *HandlerExtension*.

propertyChange is invoked by the *Modifier* whenever a new animation stage has been reached.

Note that of the four methods in the *RequestHandler* interface, only two are actually concerned with Request Handling, while the other two offer extension support.

Negotiating the change of a given property now works as follows. For the sake of clarity, we assume that the user wants to change the *fill color* of a *circle* object, and that both the *ActualModel* circle and the *Modifier* color changer already exist:

1. The *Modifier* invokes *getMethods(myCircle, x)*, where x is a parameter describing the property to be changed. In our example, this can be any arbitrary *java.awt.Color* instance.

2. The *RequestHandler* detects the underlying transformation type, here a color changer, due to the *Color* parameter passed in.

3. The *RequestHandler* queries the *ActualModel* for its current state. In this way, the handler request can detect whether the circle is filled and can therefore change both its outline and fill color, or whether only the outline color can be adapted.

4. The request handler returns a *Vector* of appropriate operation names to the Modifier, allowing the user to choose one. In our example, the Vector may contain the operation names “color”, “fill color”, and “color + fill color”.

5. At some later time, the actual effect is invoked. The *Modifier* determines the current state based on the initial state (the original fill color), the target state (the target fill color) and the current time (what percentage of the effect has been passed), and interpolates the result. This changed value is passed to the request handler using the *propertyChange* method together with the *ActualModel* instance. In our example, the value is converted into an interpolated color along the RBG line between the original and desired target color, according to the percentage of the color change effect that has currently been reached.

6. The request handler extracts the target state and the transformation information from the *PropertyChangeEvent* passed in, which encodes the name of the “property” to be changed as well as its old and new value. It then maps this change into a set of (often nearly trivial) operations on the *ActualModel.* For example, if the method name is “color”, it will call the graphical object’s *setColor(c)* method, and for “fill color”, it will call *setFillColor(c).*

The *Request Handler* is conceptually similar to the *Adapter* (Gamma et al., 1995, p. 139) and *Mediator* (Gamma et al., 1995, p. 273) design patterns, but differs in a set of key points.

4.4 Consequences

The *Request Handler* pattern has the following consequences:

- The ActualModel classes are decoupled from the Modifier classes. In our example, this means that the graphical objects do not need to be aware of animation as a dynamic change of their internal state. They simply respond to requests to change their internal state (for example, by changing the fill color), and repaint themselves when prompted. Additionally, the animation effects are unaware of the actual objects they modify, and do not need code for handling specific object types.

- The central methods *getMethods* and *propertyChange* are usually trivial to implement. The first needs to figure out, based on the transformation type, what set of operations are possible for the given concrete ActualModel. This is usually very straightforward to implement. The latter method receives both the transformation name generated in *getMethods*, the ActualModel to work on, and the current and target value. Mapping this to appropriate operations on the ActualModel is again usually very simple. For the “fill color” color changer, this operation requires the following steps: recognize that the operation is of a “color change” type; extract the current and target colors from the *PropertyChangeEvent* parameter; extract the method name (“fill color”); based on the method name, decide to call the *setFillColor* model on the ActualModel passed in with the target color.

- As the ActualModel reference is always passed in where it is needed, the *RequestHandler* can be realized as a *Singleton* (Gamma et al., 1995, p. 127) for each graphical object class. This is done in the Animal AV system.

- Using the *HandlerExtension* mechanism, developers can easily add new transformation effect names without needing to modify the code of the original RequestHandler itself.

- The implementation of the *addExtensionMethodsFor* and *insertHandlerExtension* methods can be delegated to a superclass of the actual *RequestHandler* instances, bringing the number of methods to implement down to two per Request Handler.
- If a new graphical object has been implemented, the developer only needs to add a *Request Handler* for this object type. All transformation methods supported by the Handler are then directly usable for the new graphical object - without modifying the code of any transformation object.
- If a new transformation effect has been implemented, the developer has to modify only those Request Handlers that should be able to support the new effect. The effect is then directly usable on the graphical objects without needing to modify their code.
- If a new transformation subtype for a given graphical object shall be supported, the developer only has to add appropriate code to the *getMethods* and *propertyChange* methods. No change of the transformation effect or graphical object classes is necessary. The developer can also put the changes into a new HandlerExtension and register this - in this case, no existing code is touched at all.
- An additional class (the Request Handler) has to be added for each graphical object.

5 Summary and Future Research

In this paper, we have presented two design patterns for AV-related systems. The *Reverse by Fast Forward* can be used to support efficient bidirectional navigation in AV materials. It is also applicable to other materials that match the requirements presented in Section 3 and very easy to implement. The slight delay in execution resulting from the approach is in our experience not or only barely noticeable even when running on older hardware.

The *Request Handler* pattern is used to decouple graphical objects and transformations thereon. It allows the developer to implement new graphical objects without needing to modify existing transformations, or to provide new transformations without modifying the implementation of the graphical objects. After becoming familiar with the underlying concepts, the Request Handler has proven to be highly helpful.

Both patterns have been in active use in the Animal AV system for several years. While grasping them is usually difficult at first for our students implementing new elements for the system, they see the benefits during the implementation phase.

In the future, we hope that other AV researchers will be willing to gather their "best practice" knowledge in the form of design patterns. This shall ultimately help other developers of systems to incorporate tried and proven techniques, and may in the long run even make data exchange between AV systems easier.

References

Ayonike Akingbade, Thomas Finley, Diana Jackson, Pretesh Patel, and Susan H. Rodger. JAWAA: Easy Web-Based Animation from CS 0 to Advanced CS Courses. In *Proceedings of the* 34th *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2003), Reno, Nevada*, pages 162–166. ACM Press, New York, 2003.

Jay Martin Anderson and Thomas L. Naps. A Context for the Assessment of Algorithm Visualization System as Pedagogical Tools. *First International Program Visualization Workshop, Porvoo, Finland. University of Joensuu Press*, pages 121–130, July 2001.

Bob Boothe. Efficient algorithms for bidirectional debugging. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 299–310, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. doi: http://doi.acm.org/10.1145/349299.349339.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

Martin Odersky and Matthias Zenger. Independently extensible solutions to the expression problem. In *Proc. FOOL 12*, January 2005. `http://homepages.inf.ed.ac.uk/wadler/fool`.

Guido Rößling and Bernd Freisleben. ANIMAL: A System for Supporting Multiple Roles in Algorithm Animation. *Journal of Visual Languages and Computing*, 13(2):341–354, 2002.

Guido Rößling and Thomas L. Naps. A Testbed for Pedagogical Requirements in Algorithm Visualizations. *Proceedings of the* 7^{th} *Annual ACM SIGCSE / SIGCUE Conference on Innovation and Technology in Computer Science Education (ITiCSE 2002), Århus, Denmark*, pages 96–100, June 2002.

Guido Rößling and Thomas L. Naps. Towards Improved Individual Support in Algorithm Visualization. *Second International Program Visualization Workshop, Århus, Denmark*, pages 125–130, June 2002.

Mads Torgersen. The expression problem revisited. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Oslo, Norway*, pages 123–143, 2004.

Pedagogical Effectiveness of Engagement Levels - A Survey of Successful Experiences

Jaime Urquiza-Fuentes, J. Ángel Velázquez-Iturbide
Universidad Rey Juan Carlos, C/ Tulipán, s/n, 28933 Móstoles (Madrid), Spain

jaime.urquiza@urjc.es

Abstract

In this paper we survey experiments with program and algorithm visualizations (PAVs) where learning improvements have been detected. We analyze these experiments based on the student's level of engagement with the visualizations. There are some features present in most of these, successful, experiments. Therefore they should be taken into account as important factors affecting pedagogical effectiveness of PAVs, these features are: narrative and textual contents, feedback to students' answers and a student centered approach when designing PAV construction kits.

1 Introduction

Studies about pedagogical effectiveness of PAVs have shown mixed results. The most significant result was reported by Hundhausen et al. (2002), stating that effort dedicated by students in visualization related tasks was more important than visual contents shown by PAVs. They also identified lack of research in some areas, e.g. using narrative and textual contents integrated with PAVs.

Following the idea of going beyond the passive viewing of PAVs, Naps et al. (2002) developed a taxonomy that identified different ways of interacting with PAVs. They called the *engagement levels* taxonomy, an they suggested a hierarchical structure where more engagement should produce educational improvements.

After reviewing PAV literature, and research on some engagement levels (Urquiza-Fuentes and Velázquez-Iturbide, 2007; Urquiza-Fuentes, 2008), the authors feel that educational improvements could depend on other features. This survey studies possible effects of these features.

The rest of the paper is organized as follows. First, in section 2, we describe the study we have carried out, the kind of papers included and their surveyed features. Then, in section 3 we detail the successful experiences, grouped by the engagement levels where the improvement were detected. In section 4 we analyze these experiences from two different point of views. Finally, in section 5 we draw our conclusions and future work.

2 The survey

Literature about PAVs with educational aim is wide, we have focused on successful experiences. These experiences must have detected educational improvements –knowledge acquisition, attitude towards the subject or materials, or programming performance– where PAVs have been used.

Having a look at published experiments, one suspects that just visualizations are not enough to obtain educational improvements. In fact, one of the most significant studies on PAV (Hundhausen et al., 2002) concludes that the way that students use visualizations is more important than what visualizations show to students. Also, there are successful experiences based on providing high quality contents with the visualizations (Hansen et al., 2000), advanced manipulation interfaces (Cross et al., 2007), or adding visualization sessions to regular classes (Moskal et al., 2004).

Our aim is to deepen the effect that these additional features have on the educational improvements. The features that we have taken into account are:

Narrative contents and textual explanations They could help students to understand graphical depictions generated by a PAV system. In addition, when students build their own animations, adding narrative contents engage students in a reflection exercise that could produce learning outcomes.

Feedback on student's actions During animations, students can be asked to predict future steps of the algorithm. Feedback to their answers could reinforce right answers or correct wrong ones. As animations provide inherently feedback in the next step, we will take into account only explicit feedback, for both right and wrong answers.

Extra time using PAV Many tasks in typical learning environments can not be replaced with animation based tasks, therefore to use animations extra time is needed.

Advanced features Some systems provide with advanced contents showing different behaviors of the algorithm, advanced interfaces to manipulate visualizations, or advanced integration with the IDE.

Obviously, we have used the educational improvement reported by each experience and the engagement levels used. Educational improvements can be detected as knowledge acquisition, student's performance when programming their own solutions or student's attitude towards subjects or materials (usually knowledge acquisition is affected by attitude). Experiences design range from studying improvements on one engagement level or a mixture of two of them, to comparative studies.

3 Successful Experiences

We have considered 24 experiences in this survey. In this section, we describe them grouped by the engagement level where the educational improvement has been detected. Table 1 summarizes these experiences.

3.1 Viewing

> *"Viewing" can be considered the core form of engagement, (...) a learner can view an animation passively, but can also exercise control over the direction and pace of the animation, use different windows (each presenting a different view), or use accompanying textual or aural explanations. (...) The remaining four categories all include viewing. (Naps et al., 2003)*

The six experiences related to this level have detected educational improvements in terms of knowledge acquisition. The seventh chapter of Lawrence's dissertation (Lawrence, 1993) detected improvements when using PAVs with textual labels. Crosby and Stelovsky (1995) detected improvements when using multimedia materials made up of visualizations and narrative contents, comparing it with the *no viewing* level.

Kann et al. (1997) made a comparative study among *no viewing*, *viewing*, *constructing* and, *viewing and constructing*. But they only detected significant improvements between *viewing* and *no viewing* students. It is the only viewing experience without textual or narrative contents.

Kehoe et al. (2001) studied the use of PAV in a homework simulation environment, thus students used animations to complete the assignments without time limit.

Kumar's experience (Kumar, 2005) represents an auxiliary use of visualization. The main role of Kumar's system is tutoring students providing them with automatic generated problems. His experience found that using visualizations within the feedback provided by the tutor improves knowledge acquisition.

Finally, Urquiza-Fuentes (2008) investigates the effect of replacing part of exercises sessions with program visualizations sessions during a long term evaluation. The animations had additional textual explanations.

3.2 Responding

"Responding". The key activity in this category is answering questions concerning the visualization presented by the system. (...) In the responding form of engagement, the learner uses the visualization as a resource for answering questions. (Naps et al., 2003)

The three studies of this level compare *responding* with *no viewing* level. The two first experiences detected improvements in knowledge acquisition and were supported by additional narrative contents. Although Byrne et al. (1999) used a plain algorithm animation, the instructor provided the students with questions that had to be answered during the animation. While Grissom et al. (2003) used a system that integrated automatically the questions within the animation.

Finally, Laakso et al. (2005) went beyond simple questions, engaging the students in simulation tasks. Here, the students manipulate a data structure simulating the behavior of a given algorithm, receiving explicit feedback about their simulations. But they also used the *viewing* level, as the students were allowed to see animations related to the algorithm that they had to simulate.

3.3 Changing

"Changing", entails modifying the visualization. The most common example of such modification is allowing the learner to change the input of the algorithm under study in order to explore the algorithms behavior in different cases. (Naps et al., 2003)

The two first experiences mixed *responding* plus *changing* levels, and compared them with *viewing* and *no viewing* levels. They can be found in the same publication Hansen et al. (2000) –studies I, II, IV and V–. Instead of using just isolated animations with additions, they produce high quality materials providing the students with three different animations –conceptual/abstract, detailed and populated– of the same algorithm, asking questions to the students and providing with explicit feedback.

Lawrence studied the effect of changing input data to animations against *no viewing* and *viewing* levels. In the comparative study with the *no viewing* level (Lawrence et al., 1994) she found improvements in knowledge acquisition; the animations had narrative contents and students who worked with them had an additional lab session. She also compared this level with the *viewing* one (Lawrence, 1993), obtaining again improvements in knowledge acquisition without no additional features.

Ben-Bassat et al. (2003) studied the use of a visualization tool for teaching novices java. They found that only medium students improved their knowledge. Moskal et al. (2004) focused on novice students "at risk" of not succeeding in their first programing course. They detected improvements in knowledge acquisition with an extra subject where students worked with an advanced tool to learn OO programming basics.

Ahoniemi and Lahtinen (2007) compared this level with the *no viewing* level. They used animations with additional narrative contents. This experience used homework assignments, therefore working time was not limited.

The last changing experience (Cross et al., 2007) found improvements in programming performance. The instructors provided students with an advanced tool integrated in a Java IDE, while the students in the *no viewing* group used the same environment without visualization features. The students completed programming and debugging tasks with the environment.

3.4 Constructing

"Constructing". In this form of engagement, learners construct their own visualizations of the algorithms under study. Hundhausen and Douglas [27] have

identified two main ways in which learners may construct visualizations: direct generation and hand construction. (...) It is important to note that the Constructing form of engagement does not necessarily entail coding the algorithm. (Naps et al., 2003)

Stasko (1997) designed assignments where students had to construct their own animations. This also included some *changing* activities. He detected that students dedicated more time to study those algorithms for which they had constructed animations.

Urquiza-Fuentes and Velázquez-Iturbide (2007) made a short term comparative study with *viewing* level. Students within the *constructing* group generated animations with textual explanations using an effortless approach, while the others just viewed the same kind of animations, thought generated by the instructors. They detected improvements in students' attitude, *constructing* students remained studding the algorithm more time, and their knowledge acquisition was improved.

Finally, Urquiza-Fuentes (2008) studied the effect of the same construction approach in a long term evaluation. He compared the *constructing* level with *viewing* and *no viewing* levels. He detected improvements in attitude on both comparisons; he also detected improvements in knowledge acquisition when comparing with the *no viewing* level.

3.5 Presenting

"Presenting", entails presenting a visualization to an audience for feedback and discussion. (Naps et al., 2003)

The three experiences studding presenting level include construction tasks, therefore all have additional narative contents. Two of them have focused just on this mixture of tasks (Hundhausen, 2002; Hundhausen and Brown, 2008), while the other compared it with the *viewing* level.

First, Hundhausen (2002) compared constructing and presenting tasks using two different tools: a wellknown algorithm visualization tool, and utilities selected by the students – ranging from slides to crafts–. This observational study detected improvements in attitude of those students who used their own utilities. Using these results, a tool for algorithm animations construction was designed and compared again with construction utilities selected by the students (Hundhausen and Brown, 2008). In this experience, improvements in programming performance were detected on the students who worked with the designed tool.

Finally, Hübscher-Younger and Narayanan (2003) compared *presenting* and *constructing* levels with the *viewing* level. They encouraged students –voluntary task– to generate animations and asked them to evaluate –compulsory task– those generated by the rest of the students. The construction utilities were chosen by the students. They detected improvements in knowledge acquisition of the students who constructed the animations.

4 Discussion

4.1 A global view

Clearly, learning can be enhanced with PAV. The 75% (18/24) of the experiences have detected improvements in terms of knowledge acquisition, together with more than 20% (5/24) detecting improvements in attitude towards the materials used or the subjects affected by the study. Finally, programming skills can also be improved, as they have been detected in more than 8% (2/24) of experiences.

Looking at the successfull engagement levels investigated, there are two ends. *Changing* is the most investigated level with the 37.5%(9/24) of the experiences, while *presenting* is the opposite with 12.5%(3/24). *Responding* is present in the 20.8%(5/24) of experiences, and both *viewing* and *constructing* are present in the 27.2%(7/24) of experiences.

Table 1: Summary of successfull experiences grouped by engagement levels

Experience	**Educational improvement**	**Engagement levels**	**Narrative contents**	**Explicit feedback**	**Extra time**	**Advanced features**
Lawrence (1993) Ch.7	Knowledge acq.	(V)	⋆			
Crosby and Stelovsky (1995)	Knowledge acq.	(V) $\Rightarrow$ NV	⋆			
Kann et al. (1997)	Knowledge acq.	(V) $\Rightarrow$ NV				
Kehoe et al. (2001)	Knowledge acq.	(V) $\Rightarrow$ NV	⋆		⋆	
Kumar (2005) (Feedback of tutoring system)	Knowledge acq.	(V) $\Rightarrow$ NV	⋆			
Urquiza-Fuentes (2008)	Knowledge acq.	(V) $\Rightarrow$ NV	⋆			
Byrne et al. (1999)	Knowledge acq.	(R) $\Rightarrow$ NV	⋆			
Grissom et al. (2003)	Knowledge acq.	(R) $\Rightarrow$ NV	⋆			
Laakso et al. (2005)	Attitude	(R,V) $\Rightarrow$ NV		⋆		
Hansen et al. (2000) Studies I-II-IV	Knowledge acq.	(CH,R) $\Rightarrow$ NV	⋆	⋆		Contents
Hansen et al. (2000) Study V	Knowledge acq.	(CH,R) $\Rightarrow$ V	⋆	⋆		Contents
Lawrence (1993) Ch.6	Knowledge acq.	(CH) $\Rightarrow$ V	⋆			
Lawrence et al. (1994)	Knowledge acq.	(CH) $\Rightarrow$ NV	⋆		⋆	
Ben-Bassat et al. (2003)	Knowledge acq.	(CH)				
Moskal et al. (2004)	Knowledge acq.	(CH) $\Rightarrow$ NV			⋆	IDE interface
Ahoniemi and Lahtinen (2007)	Knowledge acq.	(CH) $\Rightarrow$ NV	⋆		⋆	
Cross et al. (2007)	Prog. perform.	(CH) $\Rightarrow$ NV				Vis. interface
Stasko (1997)	Attitude	(C,CH)				
Urquiza-Fuentes and Velázquez-Iturbide (2007)	Att./Know. acq.	(C) $\Rightarrow$ V	⋆		⋆	
Urquiza-Fuentes (2008)	Att.&Know. acq.	(C) $\Rightarrow$ NV	⋆			
Urquiza-Fuentes (2008)	Attitude	(C) $\Rightarrow$ V	⋆			
Hundhausen (2002)	Attitude	(P,C)	⋆			
Hübscher-Younger and Narayanan (2003)	Knowledge acq.	(P,C) $\Rightarrow$ V	⋆			
Hundhausen and Brown (2008)	Prog. perform.	(P,C)	⋆			

Not all experiences compare two different levels, 20.8%(5/24) of them explores possible improvements within a concrete level. When looking at the comparative experiences, the 73.7%(14/19) have studied the PAV effectiveness against no use of it , the rest –26.3%(5/19)– did it against the *viewing* engagement level.

The use of narrative and textual contents is present in the 75%(18/24) of the experiences. This means that they are an important factor to take into account when designing learning experiences with PAV. While explicit feedback, extra working time or advanced features –high quality contents, advanced interfaces– are present on less than 21% of the experiences.

4.2 Recommendations for designing visualization based learning experiences

As this is not a meta study like (Hundhausen et al., 2002), we can not give formal and scientific evidence of correlations among different engegament levels and educational improvements. But all these experiences give empirical evidence on successful uses of different engagement levels, thus we can extract a number of recommendations for each engagement level.

Just viewing animations can improve knowledge acquisition, but animations should have additional text or narrative contents.

When students **answer questions during the animation**, again they should be provided with additional narrative or textual contents. But explicit feedback is also important, although it is no present in two of the experiences, the questions used in these experiences were predictive ones, thus the correct answer is given in the next steps of the animation.

Allowing the students to **changing input data** is a more active task. Here, narratives and textual contents seem to be less important 62.5%(5/8). The reason could be that researchers were more interested in cognitive work performed by students when choosing input data, rather than explaining students what happens. As this is an explorative task, a strict time limit should be avoided. But also some advanced features as high quality contents –different execution conditions (Hansen et al., 2000)–, the integration with the IDE (Moskal et al., 2004), or the interface used to manipulate animations (Cross et al., 2007), could produce learning outcomes.

When students **construct their own animations**, the construction interface is very important. Thus, providing the students with carefully designed interfaces, or allowing them to choose their own construction kits, have been shown to be effective[1]. Encouraging students to produce their own textual or narrative contents is also positive. Here, most improvements have been detected in attitude towards materials and subjects.

Finally, when students are asked to **present animations**, they also should construct them. Therefore, the construction interface is important again.

4.3 Suggestions for moving among engagement levels

Looking at the experiences, we can analyze what engagement levels have been overcome by others and how. Most of the experiences report on improvements when comparing with the *no viewing* and *viewing* engagement levels.

Coming from the *no viewing* engagement level The *no viewing* level means that no PAVs are being used. Thus, a simple change is to move to the *viewing* level, where knowledge acquisition is improved. It can be a simple movement because there exist a number of PAV collections, but if one wants to generate her own PAVs, the narrative and textual contents should be taken into account.

Moving to the *responding* level is also possible because, again, there are existing PAV collections. This movement can improve attitude and knowledge acquisition. When design-

[1]both represent a student centered approach rather a high technology centered approach

ing your own *responding* experiences the use of narrative contents and explicit feedback is important.

Attitude, knowledge acquisition and programming skills can be improved by moving to the *changing* level. Probably, it will need more time from the students, because this level is often used in a homework environment. Again, narrative contents and explicit feedback –just in case of using this level together with *responding*– are suggested. Also, some experiences have incorporated advanced features, as high quality contents –this means more work for the teacher– and, good integration with the IDE and advanced programming and visualization interface –this means more development effort if one wants to build her own system–.

Finally, moving to the *constructing* level can improve attitude and knowledge acquisition. The construction process should be effortless, and narrative contents should be added.

Coming from the *viewing* engagement level This level means low interaction with visualizations. Thus a simple change is to move to the *changing* level, where knowledge acquisition is improved. In addition to narrative contents and explicit feedback, high quality contents have been shown to be effective.

Moving to the *constructing* level could improve attitude and , as a side effect, knowledge acquisition. Again, the construction process should be effortless, and narrative contents should be integrated in the animations. It can be used together with *presenting* level, improving knowledge acquisition, but students should be free to choose their own construction kits.

5 Conclusions and future work

This is not a meta study, note that we have not included unsuccessful experiences, therefore we can not state if the studied features are significant factors for educational improvements. But we can give some recommendations, we have seen many features present in these successful experiences: narrative and textual contents, feedback to students' answers, and a student centered approach when designing PAV construction kits. Finally, we have identified possible ways to move among engagement levels and its possible effects.

The future work will consider unsuccessful experiences, therefore we will be able to give more formal correlations between engagement levels and educational improvements.

References

T. Ahoniemi and E. Lahtinen. Visualizations in preparing for programming exercise sessions. *Electronic Notes in Theoretical Computer Science*, 178:137–144, July 2007.

R. Ben-Bassat, M. Ben-Ari, and P.A. Uronen. The jeliot 2000 program animation system. *Computers & Education*, 40(1):1–15, January 2003.

M.D. Byrne, R. Catrambone, and J.T. Stasko. Evaluating animations as student aids in learning computer algorithms. *Computers & Education*, 33:253–278, 1999.

M.E. Crosby and J. Stelovsky. From multimedia instruction to multimedia evaluation. *Journal of Educational Multimedia and Hypermedia*, 4:147–162, 1995.

J.H. Cross, T.D. Hendrix, J. Jain, and L.A. Barowski. Dynamic object viewers for data structures. *SIGCSE Bull.*, 39(1):4–8, 2007.

S. Grissom, M.F. McNally, and T.L. Naps. Algorithm visualization in CS education: comparing levels of student engagement. In *Proc. of the 2003 ACM Symposium on Software Visualization*, pages 87–94, New York, NY, USA, 2003. ACM Press.

S.R. Hansen, N.H. Narayanan, and D. Schrimpsher. Helping learners visualize and comprehend algorithms. *Interactive Multimedia Electronic Journal of Computer-Enhanced Learning*, 2(1), April 2000. available at http://imej.wfu.edu/articles/2000/1/02/, 2008.

T. Hübscher-Younger and N.H. Narayanan. Dancing hamsters and marble statues: characterizing student visualizations of algorithms. In *Proc. of the 2003 ACM Symposium on Software Visualization*, pages 95–104. ACM Press, 2003. doi: http://doi.acm.org/10.1145/774833.774847.

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

C.D. Hundhausen and J.L. Brown. Designing, visualizing, and discussing algorithms within a cs 1 studio experience: An empirical study. *Computers & Education*, 50(1):301–326, 2008.

C.D. Hundhausen, S.A. Douglas, and J.T. Stasko. A meta-study of algorithm visualization effectiveness. *J. Visual Lang. and Comp.*, 13(3):259–290, 2002.

C. Kann, R.W. Lindeman, and R. Heller. Integrating algorithm animation into a learning environment. *Computers & Education*, 28(4):223–228, 1997.

C. Kehoe, J.T. Stasko, and A. Taylor. Rethinking the evaluation of algorithm animations as learning aids: An observational study. *Int. J. Hum.-Comput. Stud.*, 54(2):265–284, 2001.

A.N. Kumar. Results from the evaluation of the effectiveness of an online tutor on expression evaluation. *SIGCSE Bull.*, 37(1):216–220, 2005.

M-J. Laakso, T. Salakoski, L. Grandell, X. Qiu, A. Korhonen, and L. Malmi. Multi-perspective study of novice learners adopting the visual algorithm simulation exercise system TRAKLA2. *Informatics in Education*, 4(1):49–68, 2005.

A.W. Lawrence. *Empirical studies of the value of algorithm animation in algorithm understanding.* PhD thesis, Dep. of Computer Science, Georgia Institute of Technology, 1993.

A.W. Lawrence, A.M. Badre, and J.T. Stasko. Empirically evaluating the use of animations to teach algorithms. In *IEEE Symposium on Visual Languages, 1994. Proc.*, pages 48–54. IEEE Computer Society Press, 1994.

B. Moskal, D. Lurie, and S. Cooper. Evaluating the effectiveness of a new instructional approach. *SIGCSE Bull.*, 36(1):75–79, 2004.

T.L. Naps, G. Roessling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, and J.Á. Velázquez-Iturbide. Exploring the role of visualization and engagement in computer science education. *SIGCSE Bull.*, 35(2):131–152, 2003.

J.T. Stasko. Using student-built algorithm animations as learning aids. *SIGCSE Bull.*, 29(1):25–29, 1997.

J. Urquiza-Fuentes. *Generación Semiautomática de Animaciones de Programas Funcionales con Fines Educativos.* PhD thesis, Dep. de Lenguajes y Sistemas Informáticos I, Universidad Rey Juan Carlos, 2008.

J. Urquiza-Fuentes and J.Á. Velázquez-Iturbide. An evaluation of the effortless approach to build algorithm animations with winhipe. *Electronic Notes in Theoretical Computer Science*, 178:3–13, July 2007.

List of Authors

De Backer, Carlos, 11

Ahoniemi, Tuukka, 53, 73
Alvarez, Dario, 67

Barowski, Larry A., 3
Benachour, Phillip, 91
Bruce-Lockhart, Michael, 45

Crescenzi, Pierluigi, 45
Cross II, James H. , 3

Davis, Ruth, 19

Fondon, Marian Diaz, 67

Helminen, Juha, 59
Hendrix, T. Dean, 3

Karavirta, Ville, 37, 79
Korhonen, Ari, 59

López-Martínez, Miguel A., 85
Lahtinen, Essi, 53, 73

Mehlhase, Stephan, 105
Moons, Jan, 11

Nikander, Jussi, 59
Norvell, Theodore, 45

Pérez-Carrasco, Antonio, 113
Pfau, Jens, 105

Rößling, Guido, 29, 97, 105, 121
Riesco, Miguel, 67
Rodríguez, Pilar, 1

Sánchez-Torrubia, M. Gloria, 85
Schaeckeler, Stefan, 19
Schroeder, Peter, 97
Seppälä, Otto, 79
Shang, Weijia, 19

Therón, Roberto, 2
Torres-Blanc, Carmen, 85

Urquiza-Fuentes, Jaime, 113, 129

Velázquez-Iturbide, J. Ángel, 113, 129
Vellaramkalayil, Teena, 29