

# Proceedings of the Fourth Program Visualization Workshop

Hosted by the  
Università degli Studi di Firenze  
Florence, Italy  
June 29-30, 2006



Guido Rößling, Editor



## Foreword

These are the proceedings of the *Fourth Program Visualization Workshop (PVW 2006)*, organized by the Università degli Studi di Firenze, Florence, Italy, on June 28-29, 2006.

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), and at the University of Warwick, UK (2004). All workshops have also 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 2006 offers 21 papers coming from a total of 45 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)
- Ángel Velázquez-Iturbide (Universidad Rey Juan Carlos, Spain)

The local arrangements were organized by Pilu Crescenzi. Without Pilu offering to host PVW 2006 on short notice, the workshop would not have been possible – thank you!

Darmstadt, Germany, June 2006

Guido Rößling



## Contents

<b>Realizing XML Driven Algorithm Visualization</b>	
Thomas Naps, Myles McNally, Scott Grissom . . . . .	1
<b>Animation Metaphors for Object-Oriented Concepts</b>	
Jorma Sajaniemi, Pauli Byckling, Petri Gerdt . . . . .	6
<b>Distributed Framework for Adaptive Explanatory Visualization</b>	
Tomasz D. Loboda, Atanas Frengov, Amruth N. Kumar, Peter Brusilovsky . . . . .	11
<b>Observer Architecture of Program Visualization</b>	
Amruth N. Kumar, Stefan Kasabov . . . . .	17
<b>An Integrated and “Engaging” Package for Tree Animations</b>	
Guido Rößling, Silke Schneider . . . . .	23
<b>An evaluation of the effortless approach to build algorithm animations with WinHIPE</b>	
Jaime Urquiza-Fuentes and J. Ángel Velázquez-Iturbide . . . . .	29
<b>Annotations for Defining Interactive Instructions to Interpreter Based Program Visualization Tools</b>	
Essi Lahtinen, Tuukka Ahoniemi . . . . .	34
<b>Peer Review of Animations Developed by Students</b>	
Rainer Oechsle, Thimo Morth . . . . .	39
<b>SSEA: A System for Studying the Effectiveness of Animations</b>	
Eileen T. Kraemer, Bina Reed, Philippa Rhodes, Ashley Hamilton-Taylor . . . . .	44
<b>Jeliot 3 in a demanding educational setting</b>	
Andrés Moreno, Mike S. Joy . . . . .	48
<b>Visualizations in Preparing for Programming Exercise Sessions</b>	
Tuukka Ahoniemi, Essi Lahtinen . . . . .	54
<b>Visualization of Spatial Data Structures on Different Levels of Abstraction</b>	
Jussi Nikander, Ari Korhonen, Eiri Valanto, Kirsi Virrantaus . . . . .	60
<b>A General Framework for Overlay Visualization</b>	
F. Tihomir Piskuliyski, Amruth N. Kumar . . . . .	67
<b>Work in Progress: A Detail+Context Approach to Visualize Function Calls</b>	
Xiaoming Wei, Keitha Murray . . . . .	72
<b>Providing Data Structure Animations in a Lightweight IDE</b>	
Dean Hendrix, James H. Cross, Jhilmil Jain, Larry Barowski . . . . .	76
<b>Inductive Reasoning and Programming Visualization, an Experiment Proposal</b>	
Taiyu Lin, Andrés Moreno, Niko Myller, Kinshuk, Erkki Sutinen . . . . .	83
<b>Automatic Prediction Question Generation during Program Visualization</b>	
Niko Myller . . . . .	89
<b>Program and Algorithm Visualization in Engineering and Physics</b>	
Michael Bruce-Lockhart, Theodore S. Norvell, Yianis Cotronis . . . . .	94
<b>Integrating Algorithm Visualization Systems</b>	
Ville Karavirta . . . . .	100
<b>A Framework for Generating AV Content on-the-fly</b>	
Guido Rößling, Tobias Ackermann . . . . .	106
<b>JHAVÉ – More Visualizers (and Visualizations) Needed</b>	
Thomas L. Naps, Guido Rößling . . . . .	112



# Realizing XML Driven Algorithm Visualization

Thomas Naps, Myles McNally, Scott Grissom

*U. of Wisconsin-Oshkosh, Alma College, Grand Valley State University*

naps@uwosh.edu

## Abstract

In this paper we describe work in progress on JHAVÉ-II, a new generation of the client-server based algorithm visualization system JHAVÉ. We believe this to be the first algorithm visualization system to be totally XML driven. We describe the XML scripting language visualization authors can use with JHAVÉ-II to define the sequence of graphical snapshots, integrated pop-up questions, synchronized pseudocode, and supplemental information that comprise a particular algorithm visualization. JHAVÉ-II then uses these scripts to render visualizations and support student exploration of algorithms.

## 1 Introduction

Criteria for engaging students with an algorithm visualization (AV) have been previously detailed in (Naps et al., 2003). These criteria are structured into an engagement taxonomy that includes *responding* (to questions about the visualization), *changing* (the visualization by providing new data to the algorithm being visualized), *constructing* (being responsible for the appearance of the graphical rendering done by the visualization), and *presenting* (using the visualization as a component of a written, oral, or hypertextual explanation of the algorithm). An effective instructional algorithm visualization must be much more than a sequence of pretty pictures. The complete instructional package in which AV is used must allow the visualization designer to ask questions of the learner, to strategically allow the learner to provide input to the algorithm, and to supplement the visualization with appropriate text such as synchronized pseudocode and other descriptions of the algorithm.

There is a tremendous amount of data that underlies effective AV – program state data, data set input generation, pseudocode, hypertextual explanations, and generation of non-trivial questions about the algorithm being viewed by the learner. Last year’s ITiCSE working group on the “Development of XML-based Tools to Support User Interaction with Algorithm Visualization” recognized this fact. Its report (Naps et al., 2005) established a framework that defines a direction for future research and development, and raises a number of interesting issues in visualization system design. In this paper we describe a significant first step in addressing some of these issues by describing work in progress on what we believe to be the first instructional AV system based entirely on XML descriptions of underlying data. Although a first step, this system is much more than a prototype. Within the environment we have developed a substantial number of instructional visualizations, including a full collection of sorting, search-tree, and graph algorithms appropriate for a typical data algorithms and data structures course. By using XML as the underlying representation for the data manipulated by these visualizations, we have considerably reduced the amount of developer time required to produce a visualization. We have also better prepared ourselves for the inevitable extensions to the system that we will want to implement in the future.

## 2 JHAVÉ-II Architecture

The JHAVÉ-II architecture for delivering AV extends the original JHAVÉ AV environment described in (Naps et al., 2000). As with the original JHAVÉ, JHAVÉ-II is still an Internet-based system that generates visualizations by executing algorithms on a server, generating a visualization script, and delivering that script to the JHAVÉ client for presentation to the learner. However, unlike the original JHAVÉ, JHAVÉ-II will rely upon XML as the language for defining these scripts.

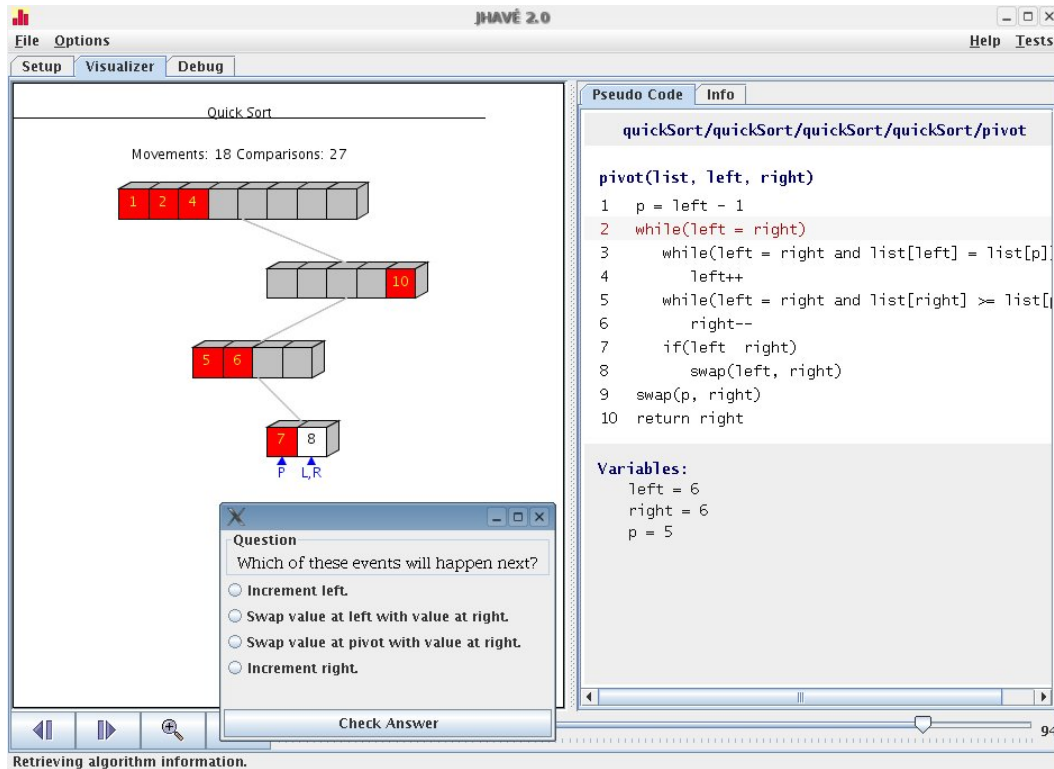


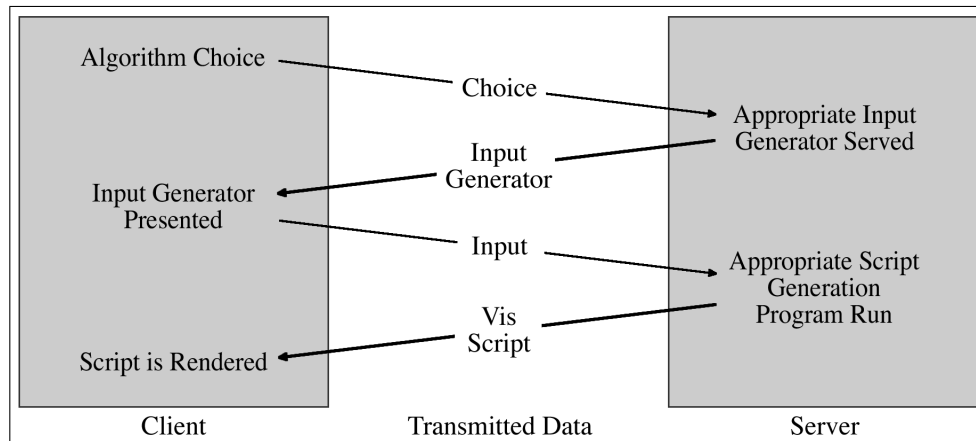
Figure 1: JHAVE-II Screenshot

In this architecture, the server application manages the available algorithms and generates the visualization scripts that the client displays. In a standard session, the learner first launches an instance of the client application, which displays a listing of available algorithms. When the user selects an algorithm from that list, the client sends a request to the server. The server knows what kind of input data the learner must provide for this algorithm and sends an description of an appropriate input generator object to the client. The client uses this to generate a frame with appropriate input areas for the learner. Once the learner fills out these areas, the client returns the input to the server as a data set to use when running the algorithm. The server then runs a program that generates the script for that algorithm and sends a URL back to the client from which the script can be read. The JHAVE-II client instantiates the appropriate visualizer plug-in to parse, render, and present the script to the learner – complete with a standard set VCR-like viewing controls, stop-and-think questions, and information/pseudocode windows. Figure ?? illustrates this for a visualization of the Quicksort algorithm with accompanying pseudocode window and a stop-and-think question.

The data flows during a JHAVE-II visualization session are depicted in Figure ??. Of these, the Vis Script flow is currently implemented in XML. This is the most complex of the flows and is partially described in the next section. The remaining flows (Choice, Input Generator, Input) remain work in progress and are not reported on here.

The original JHAVE client and the plug-ins it supported worked extremely hard at parsing the Vis Script received from the server. That was because, regardless of the plug-in used, the data came to the client in a relatively cryptic format that made sense only to someone familiar with the intricacies of the internals of JHAVE and the plug-in. In JHAVE-II, the XML data the client receives bears meaningful tags that clearly identify the various components of the script. Moreover, because of the broad range of the XML-parsing tools that are widely available, the client's paring of the data is a non-issue. The JHAVE-II client simply uses the JDOM parser classes (available at <http://www.jdom.org>) to verify that it is receiving a syntactically valid script and then build an internal tree representation of the script. This internal tree





**Figure 2:** JHAVÉ-II Data Flows

is then recursively walked to render and present the visualization to the learner. Of the various visualizer plug-ins that were available for JHAVÉ (Animal(Rößling and Freisleben, 2001), GAIGS(Naps and Swander, 1994), and Samba(Stasko, 1997)), only GAIGS is currently supported for use with XML Vis Scripts. Animal support is in development.

### 3 XML Scripting for Visualization

In this section we provide brief descriptions of portions of the XML Vis Script language for JHAVÉ-II. It is essentially a scripting language that captures representations of data structures at the interesting events during an algorithm's execution. These snapshots of the data structures can then be augmented by the various supporting tools of the JHAVÉ-II environment. For each data structure capable of being described by the XML – stacks, queues, arrays, linked lists, trees, and graphs – we have also developed a class that implements the data structure along with a toXML method that can be used by the visualization designer to annotate an algorithm at an interesting event, thereby producing the XML necessary for the visualization script. The process of writing a visualization script-producing program is then to first implement the algorithm you wish to visualize and then annotate the program at its interesting events in a fashion similar to what one does when inserting tracer output to debug a program. The existence of the toXML methods for each data structure make it very painless to produce plain vanilla visualizations, to which can be added stop-and-think questions, synchronized pseudocode and documentation. Although production of such higher quality visualizations certainly requires more programming, we feel that the descriptive nature of the XML tags we use in our scripting language still make it a relatively painless process.

#### 3.1 Overall Script Organization

The current plug-in for JHAVÉ-II specifies the XML for its scripts using XML DTD's (Data Type Definitions). For those not familiar with DTD's, an excellent description is given in (Harold and Means, 2004). In brief such a DTD resembles an Extended Backus-Naur Form (EBNF) description of a language. The DTD for JHAVÉ-II defined in Figure ?? specifies that a visualization is defined by a tagged entity called a show. Each such show consists of one or more snaps (that is, snapshots) followed by zero or more questions. A snap consists of a title, optional documentation and pseudocode URLs, zero or more data structures (stacks, queues, arrays, linked lists, trees, or graphs), and an optional question-reference for the snapshot.

```

< !ELEMENT    show    ( snap+, questions? ) >
< !ELEMENT    snap    ( title,
                        doc_url?,
                        pseudocode_url?,
                        ( tree | array | graph | stack | queue |
                          linkedlist | bargraph | node )*,
                        question_ref? ) >

```

**Figure 3:** High Level Script DTD

### 3.2 Data Structures

Each of the six data structures that can be rendered by the current JHAVÉ-II plug-in has its own DTD definition. As an example, consider the stack definition in Figure ???. The data in a stack consists of a sequence of zero or more list items. The optional bounds tag may be used to specify the position and size of the stack picture that is rendered by the plug-in. The color attribute for a list\_item is used to specify the color of each data item in the stack.

```

< !ELEMENT    stack    ( name?, bounds?, list_item* ) >
< !ELEMENT    bounds   ( EMPTY ) >
< !ATTLIST    bounds   x1 CDATA #REQUIRED
                        y1 CDATA #REQUIRED
                        x2 CDATA #REQUIRED
                        y2 CDATA #REQUIRED
                        fontsize CDATA "0.03" >
< !ELEMENT    list_item ( label ) >
< !ATTLIST    list_item color CDATA "#FFFFFF" >
< !ELEMENT    label    ( #PCDATA ) >

```

**Figure 4:** Stack Data Structure DTD

Of course, non-linear data structures such as trees and graphs have more complicated DTD definitions. Nonetheless the bounds tag and color attribute are used in a consistent fashion throughout all of the data structure definitions.

### 3.3 Documentation, Pseudocode, and Interactive Questions

The support offered by JHAVÉ-II for its plug-ins includes documentation and pseudocode windows. The DTD for documentation window content is merely a reference to a URL that specifies an HTML document. Pseudocode windows are more complicated as they must be synchronized with the state of the data structure that is being viewed by the learner. For example, Figure ?? shows a pseudocode window for a visualization of the quicksort algorithm. In addition to the program listing, note the call stack and the current values of individual variables. The DTD for such a pseudocode window appears in Figure ??.

A DTD for interactive stop-and-think questions has also been defined. Presently four types of questions are supported – true-false, fill in the blank, multiple choice, and multiple selection (multiple choice with more than one right answer).

```

< !ELEMENT doc_url ( #PCDATA ) >
< !ELEMENT pseudocode ( call_stack?,
    program_listing?,
    variables? ) >
< !ELEMENT call_stack ( #PCDATA ) >
< !ELEMENT program_listing ( signature?, line* ) >
< !ELEMENT signature ( #PCDATA ) >
< !ELEMENT line ( ( #PCDATA | replace ) + ) >
< !ATTLIST line line_number CDATA #IMPLIED>
< !ELEMENT variables ( variable* ) >
< !ELEMENT variable ( #PCDATA, replace ) >
< !ELEMENT replace ( EMPTY ) >
< !ATTLIST replace var NMTOKEN #REQUIRED >

```

**Figure 5:** Pseudocode DTD

## 4 Conclusions

In this paper we have described work in progress on JHAVÉ-II, the next generation of the client-server based JHAVÉ AV environment. While this new release will have a number of enhancements, we focused here on the conversion of all JHAVÉ data flows to the XML format. Positive outcomes of this conversion will include enhanced extensibility and ease of visualization development.

## 5 Acknowledgement

This work was supported by a United States National Science Foundation CSLI Grant, DUE-0126494.

## References

- E. Harold and S. Means. *XML in a Nutshell*. O'Reilly, 2004.
- T. Naps and B. Swander. An object-oriented approach to algorithm visualization - easy, extensible, and dynamic. *ACM SIGCSE Bulletin*, 26(1):46–50, March 1994. ISSN 0097-8418. doi: <http://doi.acm.org/10.1145/191033.191052>.
- T. Naps, J. Eagan, and L. Norton. Jhavé – an environment to actively engage students in web-based algorithm visualizations. *ACM SIGCSE Bulletin*, 32(1):109–113, March 2000.
- T. Naps, S. Cooper, B. Koldehofe, C. Leska, G. Rößling, W. Dann, A. Korhonen, L. Malmi, M. McNally, J. Rantakokko, and R. Ross. Evaluating the educational impact of visualization. *ACM SIGCSE Bulletin*, 35(4):124–136, December 2003.
- T. Naps, G. Rößling, P. Brusilovsky, J. English, D. Jarc, V. Karavirta, C. Leska, M. McNally, A. Moreno, R. Ross, and J. Urquiza-Fuentes. Development of xml-based tools to support user interaction with algorithm visualization. *ACM SIGCSE Bulletin*, 37(4):123–138, December 2005. ISSN 0097-8418. doi: <http://doi.acm.org/10.1145/1113847.1113891>.
- G. Rößling and B. Freisleben. Animalscript: An extensible scripting language for algorithm animation. *ACM SIGCSE Bulletin*, 33(1):70–74, March 2001.
- J. Stasko. Using student-built algorithm animations as learning aids. *ACM SIGCSE Bulletin*, 29(1):25–29, March 1997. ISSN 0097-8418. doi: <http://doi.acm.org/10.1145/268085.268091>.

# Animation Metaphors for Object-Oriented Concepts

Jorma Sajaniemi, Pauli Byckling, Petri Gerd

*Department of Computer Science, University of Joensuu, P.O.Box 111, FI-80101 Joensuu, Finland*

`{saja|pbyckli|pgerdt}@cs.joensuu.fi`

## 1 Introduction

Program visualization and animation has traditionally been done at the level of the programming language and its implementation in a computer. For example, variables have been visualized as boxes (representing memory locations), and nested function calls as a stack of frames containing parameters and local variables (representing the call stack implementation in many computer architectures). In object-oriented (OO) context, animation has also been based on UML diagrams that reveal connections between objects and classes and thus represent another level, i.e., relationships between components of an individual program. We know of only one program animation system, PlanAni (Sajaniemi and Kuittinen, 2004), that builds its visualization on general programming knowledge (roles of variables) and uses metaphors to make this knowledge easier to assimilate by learners.

Novices have problems in learning the very basic OO concepts which results in misconceptions leading to either erroneous or suboptimal programming skill (see, e.g., Eckerdal and Thuné (2005); Fleury (2000); Holland et al. (1997)). Program visualization is supposed to enhance learning and prevent misconceptions but the visualizations should be at the same level as the concepts to be learned. Thus visualizations that build upon programming language implementation may easily fail in helping novices to learn programming concepts.

Metaphor involves the presentation of a new idea in terms of a more familiar one (Carroll and Mack, 1999). In contrast to analogy, metaphor is not an exact counterpart but differs from the idea usually both in form and in content. The similarities and differences between the two ideas stimulate thought and can facilitate active learning. This paper applies a metaphor approach to object-oriented programming. Our ultimate goal is to provide novices with metaphors that will help them in learning basic OO constructs. For this purpose, we present new metaphors for such concepts as object, object instantiation, method invocation, parameter passing, and object reference. The metaphors are designed to grasp the basic ideas of object-oriented programming; they do not rely on implementation issues or diagramming techniques designed for expert use.

Section 2 describes the new metaphors and explains how they can be visualized and animated in a program animator. Section 3 discusses visualizations in current program animation systems and compares them with our ideas. Finally Section 4 contains the conclusion.

## 2 Visualization of OO Concepts

An object encapsulates the existence, state and behavior of an entity. Its visualization should reflect these three aspects. The existence is limited by the instantiation of an object and its destruction in garbage collection. The state is manifested in the member variables, and the behavior is a result of method invocations that include the creation and destruction of local variables. The behavior of individual member and local variables can be described by roles (Sajaniemi, 2002) that already have metaphors, e.g., a dog for the role follower, a box for gatherer etc (Sajaniemi and Kuittinen, 2004). For an object, we therefore suggest the metaphor of a *watch panel* with class-dependent fixed “monitors” for its member variables depicted in the form of *role metaphors*.

The instantiation of an object is animated by making a copy of a class-specific *blueprint* found in a *blueprint book* that is created during the processing of class declarations. The

blueprint book lays normally outside the screen and emerges only when needed for object instantiations. Each blueprint occupies its own page in the book, and class variables are located on the same page. Whereas the background color for blueprints is blue, the background color of the class variable area is white. This area becomes visible whenever an object of that class is active. Class variables are depicted with the same role metaphors as member variables.

In this context a meaningful metaphor for method invocation is a temporary *workshop* containing all parameters and local variables, and a *workbench* for the result of the invocation. Because new local variables can be created and destroyed during the invocation, the workshop must be either flexible or large enough to accommodate all variables. If another method of the same object is invoked or a method is invoked recursively, a new workshop is created. Thus the number of co-existing workshops depends on the number of unfinished method invocations. To stress the fact that method invocations are associated with the object's member variables, the workshops are attached to the watch panel depicting the object. Finally, a static method is visualized as a *permanent workshop* with a concrete foundation and a strong roof.

The traditional verbal metaphor for method call is “message passing”. We visualize this metaphor with an *envelope* containing actual parameters. The animation of a method call starts with the creation of the parameter envelope in the invoking workshop, the envelope then flies to the watch panel associated with the called object, a new workshop emerges, and the values in the envelope are transferred to role metaphors of the formal parameters thus giving their initial values. The empty envelope stays on the workbench and is filled with the return value when the method invocation ends. Then the envelope flies back to the calling workshop along a path that was created during the method call.

In Java, pointers are replaced by object references. This concept has been found to be problematic for novices and a well-designed metaphor is needed. Pointers have been traditionally depicted by arrows that are redrawn to point to a new item each time a new value is assigned to the pointer. This arrow metaphor builds on the implementation aspect: pointers are memory addresses and an assignment to a pointer means the setting of a new memory address to the pointer.

In order to avoid this implementation point of view, we suggest a *pennant* metaphor: an object reference is visualized as two pennants with the same unique identity; one pennant is attached to the object reference variable and the other to the referenced object. A null reference is visualized with two pennants lying on the ground. Assignment of a newly created object to an object reference is animated by moving the other pennant to the new object; assignment to an object reference is animated by moving the pennant from the old target to the new target. If two variables refer to the same object, the object has two pennants. As a consequence, an object with no pennants cannot be referenced and is subject to garbage collection, which is animated by a *garbage vehicle* that moves around and stops next to each object, i.e., watch panel, with no pennants. The finalizer is then invoked and the watch panel is finally squashed into the vehicle.

Figure ?? is a sketch of a visualization based on the above metaphors. The animated program models a bank that consists of three bank accounts. The object reference `myBank` is visualized as a pennant whose pair is attached to the `Bank` object. Both of these pennants have the same color that is different from all other pennant pairs. The individual bank accounts are implemented as a linked list and visualized as watch panels with pennants making the linkage. The `next` link in the last account is null represented with two pennants on the ground.

Member variables are depicted with role images: account number stored in the variable `account` is a fixed value; the variable `balance` that gathers the net effect of deposits and withdrawals is a gatherer etc. System defined objects that conceptually encapsulate a single attribute (e.g., String, Date) are visualized just like primitive variables. For example, the latest transaction date stored in the member variable `date` is a most-recent holder.

The active object (account 1476) and its active method (`updateRate`) are enhanced with red color. The workshop for this method invocation contains the parameter `transferDate`

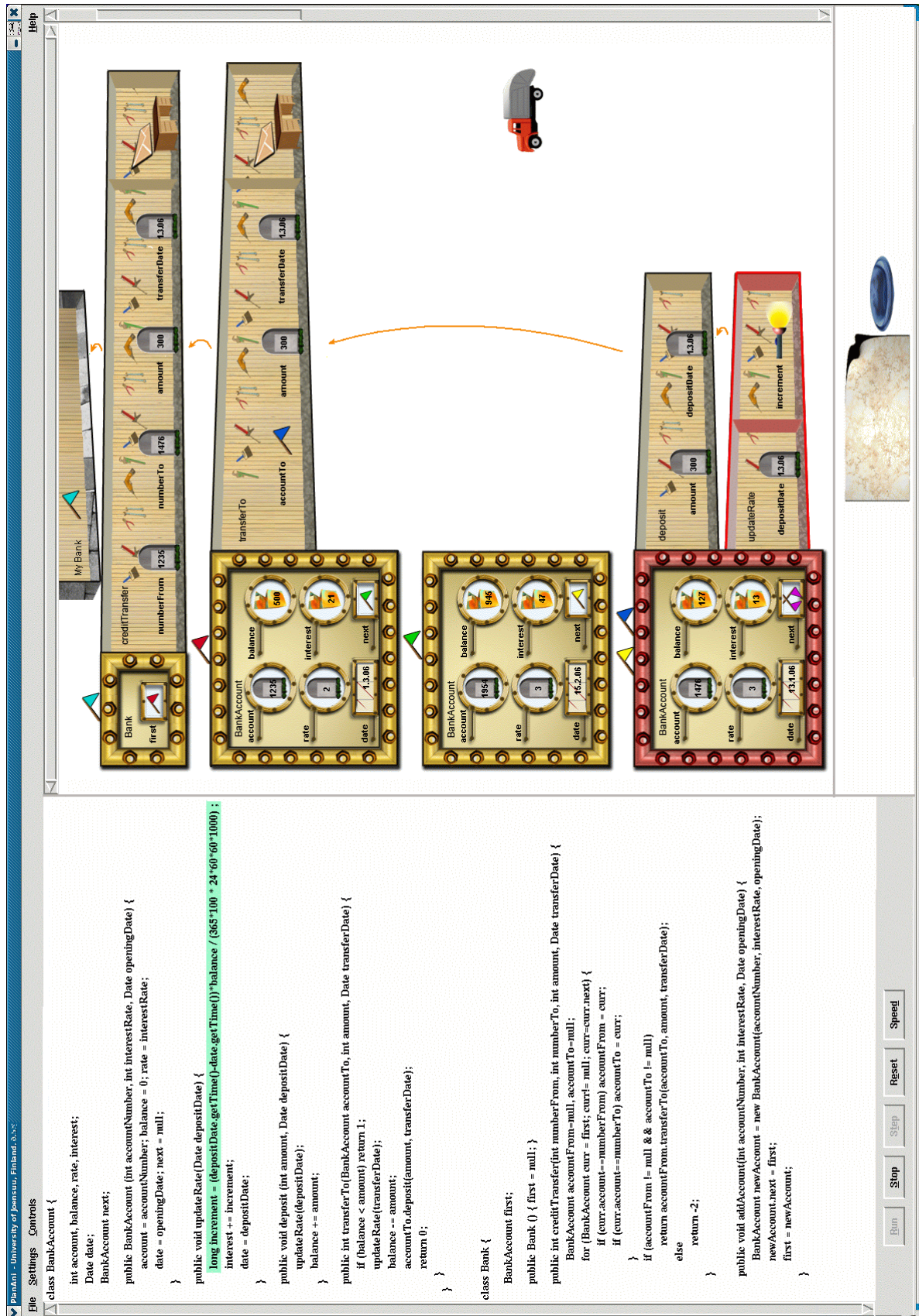


Figure 1: A hypothetical user interface for a program animator using the OO metaphors.

(having the role fixed value) and the local variable `increment` (temporary). This method has been called from the method `deposit` of the same object, which was called from the method `transferTo` of the bank account 1235, which in turn is called from the method `creditTransfer` of the single bank object—called from the static method `main`. The methods `creditTransfer` and `transferTo` return an integer; therefore they have a workbench for the preparation of the return value.

The garbage vehicle moves around but currently there are no objects with no pennants. The blueprint book is not visible at the moment.

### 3 Comparison with Current Visualizations

Current OO program animation systems use basic geometric figures (2D or 3D boxes, cones, arrows, etc) for visualizations. These figures make up a notation language that is new to students. Thus students have to learn simultaneously the new geometric notation language, the new OO concepts themselves, and the connections between these two worlds.

For example, GROOVE (Jerding and Stasko, 1994), Jeliot 3 (Moreno et al., 2004), OGRE (Milne and Rowe, 2004), and OOP-Anim (Esteves and Mendes, 2003) use geometric figures to represent OO constructs such as class and object, relationships between classes and objects, relationships within class hierarchies etc. On the other hand, JACOT (Leroux et al., 2003), JAN (Lohr and Vratislavsky, 2003), and JavaVis (Oechsle and Schmitt, 2002) use UML notations for the same purposes. Even though UML is a standard notation language, it is new to novices and must be learned in addition to the OO concepts themselves. None of these systems uses metaphors for OO concepts.

JACOT, JAN and JavaVis use UML sequence and object diagrams also to animate method calls; method instances are not visualized. GROOVE visualizes method calls similarly to our suggestion but method instances are not attached to the corresponding objects. In OOP-Anim, each object is depicted as in UML and contains both the member variables and names of all methods; a method invocation is animated by changing the color of the invoked method within the object and the whole method is executed in a single step. Neither parameters nor local variables are represented in the visualization; moreover, several instances of the same method cannot be visualized. Jeliot 3 animates method invocations with a special method instance area that represents the implementation-based method call stack. None of these system uses anything similar to our workshop metaphor.

The above systems visualize object references with arrows or miniature pictures of the referenced object. This is in contrast to our pennant metaphor where the referencing variable has a unique identity that does not change if the target of the reference changes.

### 4 Conclusion

We have presented new metaphors for classes (blueprint book), objects (watch panel), method invocation (workshop), parameter passing (envelope), return value (workbench) and object reference (pennant). We have also presented visualizations for these metaphors and program animation that uses the metaphors.

The visualizations and animations do not scale up to large programs but this is not a problem. We do not suggest using the visualizations in, e.g., debugging or comprehending large programs. Instead, we do suggest that in elementary programming education, roles of variables are first introduced with their role metaphors using the PlanAni program animator. When the role metaphors are familiar, the OO metaphors can be introduced by animating a few, carefully selected OO programs. With appropriate student engagement this will give students a correct mental model of the relationships between OO concepts—a mental model that builds up on a spatial representation for program execution. The visually rich visualizations of the metaphors are expected to consolidate this mental model so that even though the visualizations do not scale up on computer screen, the metaphors do scale up in novices' mental representations.

In future, we are planning to add the new metaphors into the PlanAni program animation environment and study their effects on novices' mental models of OO concepts.

## Acknowledgments

This work was supported by the Academy of Finland under grant number 206574.

## References

- J. M. Carroll and R. L. Mack. Metaphor, computing systems, and active learning. *International Journal of Human-Computer Studies*, 51:385–403, 1999.
- A. Eckerdal and M. Thuné. Novice Java programmers' conceptions of "object" and "class", and variation theory. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education ITiCSE'05*, pages 89–93. ACM, 2005.
- M. Esteves and A. Mendes. OOP-Anim, a system to support learning of basic object oriented programming concepts. In *CompSysTech' 2003 - International Conference on Computer Systems and Technologies*, 2003. Available at <http://ecet.ecs.ru.acad.bg/cst/Docs/proceedings/S4/IV-6.pdf>.
- A. E. Fleury. Programming in Java: Student-constructed rules. In *Proc. of the 31th SIGCSE Technical Symposium on CS Education*, pages 197–201, 2000.
- S. Holland, R. Griffiths, and M. Woodman. Avoiding object misconceptions. *SIGCSE Bulletin*, 29(1):131–134, 1997.
- D. F. Jerding and J. T. Stasko. Using visualization to foster object-oriented program understanding. Technical Report GIT-GVU-94-33, Georgia Institute of Technology, Atlanta, GA, USA, July 1994.
- H. Leroux, C. Mingins, and A. Requile-Romanczuk. JACOT: A UML-based tool for the runtime-inspection of concurrent Java programs. In R. Filman, M. Haupt, and K. Mehner, editors, *1st Workshop on Advancing the State-of-the-Art in Run-Time Inspection*, 2003. Available at <http://www.st.informatik.tu-darmstadt.de/pages/workshops/ASARTI03/LerouxASARTI03.pdf>.
- K.-P. Lohr and A. Vratislavsky. Jan - Java animation for program understanding. In *Proceedings of the 2003 IEEE Symposium on Human Centric Computing Languages and Environments*, pages 28–31, 2003.
- I. Milne and G. Rowe. Ogre: Three-dimensional program visualization for novice programmers. *Education and Information Technologies*, 9(3):219–237, 2004.
- A. Moreno, N. Myller, E. Sutinen, and M. Ben-Ari. Visualizing programs with Jeliot 3. In *Proceedings of the International Working Conference on Advanced Visual Interfaces*, pages 373–376, 2004.
- R. Oechsle and T. Schmitt. JAVAVIS: Automatic program visualization with object and sequence diagrams using the Java debug interface (JDI). *Lecture Notes in Computer Science*, 2269:176–190, 2002.
- J. Sajaniemi. An empirical analysis of roles of variables in novice-level procedural programs. In *Proceedings of IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)*, pages 37–39. IEEE Computer Society, 2002.
- J. Sajaniemi and M. Kuittinen. Visualizing roles of variables in program animation. *Information Visualization*, 3:137–153, 2004.



# Distributed Framework for Adaptive Explanatory Visualization

Tomasz D. Loboda, Atanas Frengov, Amruth N. Kumar, Peter Brusilovsky

<i>University of Pittsburgh</i>	<i>College of New Jersey</i>
<i>Pittsburgh, PA 15260</i>	<i>Mahwah, NJ 07430</i>
<code>{tol7,peterb}@pitt.edu</code>	<code>{afrengov,amruth}@ramapo.edu</code>

## Abstract

Educational tools designed to help students understand programming paradigms and learn programming languages are an important component of many academic curricula. This paper presents the architecture of a distributed event-based visualization system. We describe specialized content provision and visualization services and present two communication protocols in an attempt to explore the possibility of a standardized language.

## 1 Introduction

Research teams working on modern algorithm and program visualization environments have been exploring a wide range of new directions in answer to the challenge of making visualization more effective, useful, and graphically rich while decreasing the effort required to maintain high quality. For example interactive questions engage students into working with visualization (Naps et al., 2000) turning them from passive observers to active learners. Explanatory narratives (Dancik and Kumar, 2003)(Kumar, 2002)(Shah and Kumar, 2002) help students understand ideas that have been presented visually. Adaptive visualization helps focus student attention on an individual's least understood concepts (Brusilovsky and Su, 2002). The multitude of research directions has led to a situation that is typical for a research-intensive field: no team has expertise in all aspects of modern visualization and no existing tool supports all desired functionalities. In this situation, the traditional method of producing visualization environments – as monolithic software applications working on a specific platform – restricts further progress in the field. The high price of developing each desired functionality combined with the inability to integrate functionalities already developed by different teams makes it impossible for each team to explore feature-rich environments.

A possible solution is to decompose monolithic visualization environments into several reusable, communicating components. This solution was proposed by the ACM SIGCSE Working Group (Naps et al., 2005). The architecture suggested by the group separates production of the visualization trace from its interactive rendering and possible enhancement with questions and narration. A set of XML-based protocols provides the connection between these identified components.

Our research groups at the University of Pittsburgh and Ramapo College fully support this vision. Our current NSF-supported project is focused on increasing the value of visualization with explanations and adaptation components. The distributed XML-based architecture could significantly help us disseminate the results of our research since virtually any open visualization environment can be enhanced with explanations and personalization. To support work on the newly distributed architecture our groups have attempted to restructure our work on adaptive explanatory visualization in terms of the new architecture. This paper presents our first attempt to implement explanatory visualization in a distributed framework following the ideas of the ACM ITiCSE Working Group (Naps et al., 2005). We start by presenting the ideas and the original implementation of event-based explanatory visualization, which is the foundation of our work. Then we present our distributed architecture focusing on both components and communication protocols. The paper concludes with a discussion of current and future work.

## 2 Event-Based Visualization

The core concept behind the proposed distributed framework for visualization is the flow of "interesting events" that are produced by a program or an algorithm. An "interesting event" is simply an event that is important for understanding some concept and should be presented visually, for example, a variable assignment or removal of an item from a linked list. To increase the pedagogical value of visualization, an event could be extended with explanation (narration), a question that challenges the student to predict the result, etc. The ability to represent the flow of interesting events in a standardized XML format allows one to decompose a monolithic visualization application into a system of several independent components such as producers, enhancers, and players within the events flow.

## 3 Architecture

Following the proposal of the ACM SIGCSE Working Group (Naps et al., 2005) we have implemented an architecture of a distributed model of communication between content providers and content visualizers (Figure 1). The idea behind it is straightforward. *Content Provider* (CP) outputs an XML stream containing code, events, and explanations (*c-XML* for *content-XML*). *Content Visualizer* (CV) expects an object oriented *v-XML* (*visualization-XML*). *Value-Added Service Provider* (VASP) translates *c-XML* into *v-XML* and enriches the stream with pertinent questions that the CV is capable of presenting.



Figure 1: The proposed architecture

The two formats originate from different perspectives. *c-XML* represents concerns of content provision. *v-XML* is shaped to best suit presentation purposes. The current work attempts to explore the possibility of having only one format. Because of that *Format Translator* (FT) is an optional component of the framework. At the same time, the presence of FT is depicting a real-life scenario where a system already capable of generating an XML-based visualization specification does not comply with the standard format. FT enables such a system to become a part of the distributed framework and use a non-native visualization engine of its choice (or even multiple engines at the same time).

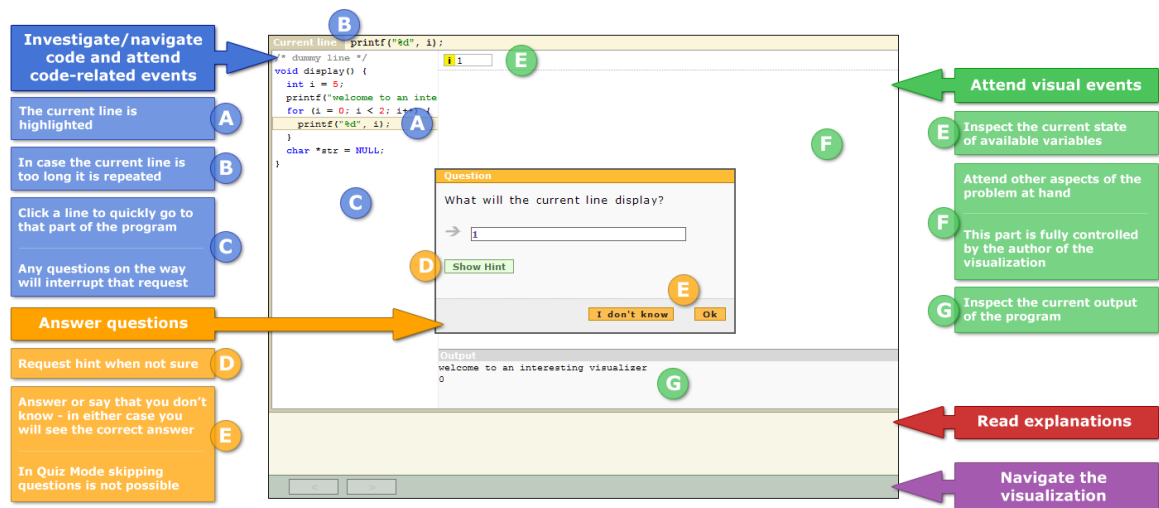
## 4 Content Visualizer

From the end user point of view the implemented system (named IMPROVE) is an educational application that supplements learning of programming skills. It does so by visualizing and explaining the execution of snippets of code in a given programming language. The user interface is divided into five areas: Code, Visualization, Output, Explanations, and Navigation (Figure 2).

The system is interactive. The student can step through the execution of the program forward or backward. Jumping to a specific line of code is also possible. If a question is encountered on the way to the user-designated line the execution stops there and the student is presented with a prompt. That prevents the student from missing a question. As the student goes through the program execution the current line is highlighted, the output of the program is updated, and relevant visualizations are presented (currently only variable-related). Many steps will also involve presenting textual explanations. The system is capable of asking questions. Four types of these are currently supported: *one-of-many* (radio buttons), *multiple-choice* (checkboxes), *short-text-input* (one line text control), and *long-text-input* (multiple lines

text control). When a question is posted the student can skip it, attempt to answer, or say "I don't know" (certain system modes restrict skipping a question).

The system can work in one of four modes: *Exploration* (free navigation; no questions asked), *Challenge* (free navigation; questions are asked, but can be skipped; feedback is presented after each answer), *Evaluation* (forward-restricted navigation; questions cannot be skipped; feedback is presented after each answer), and *Quiz* (forward-restricted navigation; questions cannot be skipped; no feedback is presented after an answer). The first two modes let students interact with the material on a stress-free basis. The last two modes are designed to be used by teachers to administer quizzes (with or without feedback).



**Figure 2:** The user interface of IMPROVE visualization engine. The system is working in Quiz Mode and is waiting for the student to answer a question.

The visualization engine treats everything as an *action* (or event in the event-based visualization paradigm). The engine distinguishes two types of them: *non-step-based* and *step-based*. The first type is executed instantaneously (or consists of a single step). An example of such an action would be *show-object*. The second type of action takes a given amount of time to execute (or consists of a number of steps). An example could be *move-object* which would change the object's location in an animated manner (as opposed to e.g., *set-object-location*). At this time the visualization engine supports the following nine actions: *set-current-line*, *show-explanation*, *hide-all-explanations*, *print*, *define-variable*, *assign-variable*, *post-increment*, *ask-question*, and *wait*. The IMPROVE system is available at <http://kt2.exp.sis.pitt.edu:8080/improve>.

## 5 Content Provider

Problets are web-based tutors for C/C++/Java/C# programming languages that have traditionally generated problems, their solutions, feedback, and visualization within a monolithic system. The point of this research was to test whether the problems, feedback and visualization generated by problets could be rendered by a non-native visualizer such as IMPROVE. To this end, a servlet was developed (Figure 1) to (a) convert problets from an application to a service, and (b) translate the output of problets into *c-XML*.

For each programming problem, a proplet randomly generates a program and annotates each line of code with its line number. It executes the program using a custom interpreter (Kumar, 2002), and automatically generates narration of the step-by-step execution of the program (Kumar, 2005), indexing each line of explanation with the line number, program objects and program events that contribute to the explanation. Finally, the proplet provides

the learner interactive controls to visualize the step-by-step execution of the program (it uses the explanation to drive such visualization). The servlet (Figure 1) converts the explanation generated by a proplet into *c-XML* event stream that can be obtained and visualized by any distributed client. More information on proplets is available at <http://www.proplets.org>.

## 6 Communication Protocol

Our vision of the future involves many specialized content provision and content visualization services. Those services could possibly appear and disappear on an ad-hoc basic. Such dynamic framework of responsibilities delegation has to be based on a standard communication protocol. One of our most important research agendas is to move towards defining such a protocol. In Section ?? we present the visualization specification expected by our visualization engine (*v-XML*). In Section ?? we present the content generated by our content provider (*c-XML*). For the purpose of illustration we use a one-line C language code snippet printing "Hello World!" in the console. The translation between these two formats is based on XSLT, a language for transforming XML documents into other XML documents (W3C, 1999). The transformations themselves will not be presented due to space constraint. The translation process is discussed in Section ??.

### 6.1 Visualization Specification (*v-XML*)

The XML format expected on the *CV* side consists of five parts presented below. It is object-based in that it first defines objects and then actions that take those objects as arguments. In *v-XML* actions drive the execution sequence.

```
<!-- 1. Lines of code -->
<code-lines>
  <item id="ln1">printf("\s", "Hello World!");</item>
</code-lines>

<!-- 2. Explanations -->
<explanations>
  <item id="expl1">
    <code>printf</code> function will display <code>Hello World!</code>.
  </item>
</explanations>

<!-- 3. Questions -->
<questions>
  <item id="q1" type="short">
    <text>
      What will be displayed in the current line?
    </text>
    <answer>Hello World!</answer>
    <hint><code>printf</code> function takes several parameters</hint>
    <feedback>
      Here, <code>printf</code> function is instructed to display
      a string. In this case this string will be <code>Hello World!</code>.
    </feedback>
  </item>
</questions>

<!-- 4. Output -->
<output>
  <item id="out1">Hello World!</item>
</output>
```

```

<!-- 5. Actions to be executed and the execution sequence -->
<code-execution>
  <action type="set-current-line" line-id="ln1"/>
  <action type="show-explanation" explanation-id="expl1"/>
  <action type="ask-question" question-id="q1"/>
  <action type="print" output-id="out1"/>
  <action type="wait"/>
</code-execution>

```

## 6.2 Content Specification (c-XML)

The XML format generated on the *CP* side consists of four parts presented below. In *c-XML* explanations drive the execution sequence.

```

<!-- 1. Lines of code -->
<code-text>
  <line id="1" indentation="1">printf("Hello World!");</line>
</code-text>

<!-- 2. Explanations -->
<explanation>
  <event line-num="1" context="Variable" event-type="OUTPUT">
    <CD>printf</CD> function will display <CD>Hello World</CD>
  </event>
</explanation>

<!-- 3. Output -->
<output>
  <single-output line-number="1">
    <text>Hello World!</text>
  </single-output>
</output>

<!-- 4. Errors -->
<errors>
  <single-error line-number="20" object-name="ptr">
    <error-name>
      Dangling Pointer: Dereferencing pointer before initializing/allocating
    </error-name>
  </single-error>
</errors>

```

## 6.3 Format Translation and Question Generation

Translation of lines of code, explanations, and program output elements is essentially XML element name-matching. Providing the questions, actions, and execution sequence is more complicated. Problers generate an explanation for every line in the order of program execution. The order of these explanations is used to infer program execution sequence. Explanations provided by the server are also processed to identify actions to be taken. **Event-type** attribute of each **explanation/event** element is used here. Each output element provided by the server contains text printed on the console while the program executes. This text is translated into a question about output that a particular line will generate. *CV* does not use **errors** element at this moment. Therefore, *FT* discards it during the translation process. This element may be used in the future to generate questions about bugs in the provided code.

## 7 Conclusions and Future Work

This is a proof-of-concept system. In order to extend it to cover more programming constructs we plan to (a) identify objects and events that are of interest in various programming contexts (e.g. building a dynamically linked list) and (b) design a general, flexible, and extensible protocol to transmit this information from *CP* to *CV*.

Further, we plan to extend that protocol to adapt visualizations to the needs of the learner, i.e. focus on visualizing parts of material that are not yet well understood. We plan to represent the learner's progress in terms of concepts (e.g. *for-loop* or *variable-declaration*). Proplets currently associate concepts with interesting events. *VASP* would consult the student model service to retrieve information about learner progress within each of those concepts. On the presentation side, as the learner interacts with the visualization (i.e., observes animations, reads explanations, answers questions, etc.) *CV* would record their progress through the student model service.

Another direction would be to develop an authoring tool to help teachers create visualizations more easily. Direct authoring can be an alternative to creating automated content provision services. Such "hand-crafted" visualizations could then be accessed from a Web-enabled repository and presented by *CV*.

## Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 0426021.

## References

- Peter Brusilovsky and H.-D. Su. Adaptive visualization component of a distributed web-based adaptive educational system. *Intelligent Tutoring Systems*, 2363:229–238, 2002.
- G. Dancik and Amruth N. Kumar. A tutor for counter-controlled loop concepts and its evaluation. *Frontiers in Education Conference (FIE 2003)*, 2003.
- Amruth N. Kumar. Model-based reasoning for domain modeling in a web-based intelligent tutoring system to help students learn to debug C++ programs. *Intelligent Tutoring Systems (ITS 2002)*, pages 792–801, 2002.
- Amruth N. Kumar. Explanation of step-by-step execution as feedback for problems on program analysis, and its generation in model-based problem-solving tutors. *Instruction, Cognition and Learning (TICL) Journal*, page to appear, 2005.
- Thomas L. Naps, J. R. Eagan, and L. L. Norton. JHAVE - an environment to actively engage students in web-based algorithm visualizations. *ACM SIGCSE bulletin*, 32(1):109–113, 2000.
- Thomas L. Naps, Guido Rößling, Peter Brusilovsky, J. English, D. Jark, V. Karavirta, C. Leska, M. McNally, A. Moreno, R. J. Ross, and J. Urquiza-Fuentes. Development of XML-based tools to support user interaction with algorithm visualization. *ACM SIGCSE bulletin*, 37(4):123–138, 2005.
- H. Shah and Amruth N. Kumar. A tutoring system for parameter passing in programming languages. *ACM SIGCSE bulletin*, 34(3):170–174, 2002.
- W3C. XSL Transformations (XSLT) Version 1.0. <http://www.w3.org/TR/xslt>, 1999.

# Observer Architecture of Program Visualization

Amruth N. Kumar, Stefan Kasabov

*Ramapo College of New Jersey, 505 Ramapo Valley Road, Mahwah, NJ 07430, USA*

amruth@ramapo.edu

## 1 Abstract

We propose Observer architecture for program visualization. The principles of Observer architecture are modular, model-driven visualization with one-directional coupling, hierarchical delegation, message-passing and archival by visualizers. The architecture is scalable. The resulting visualization can be distributed and modified independent of the model. The Observer architecture has been implemented in online tutors for programming called problets.

## 2 Introduction

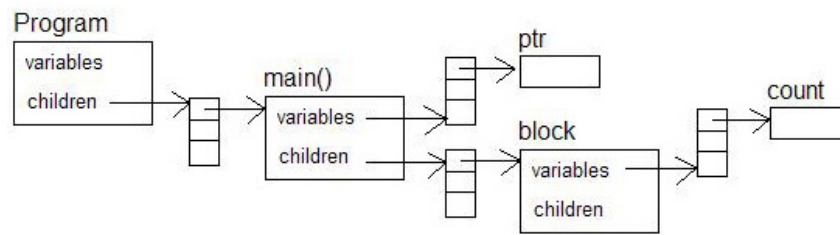
Researchers have found that on problem-solving transfer tasks (Mayer and Anderson, 1991), animation with narration outperforms animation only, narration only, or narration before animation. Similarly, on recall tasks, narration with visual presentation outperforms narration before visual presentation (Baggett, 1984). These results support a dual-coding hypothesis (Paivio, 1990) that affirms two types of connections among stimuli and representations: representational connections between stimuli and the corresponding representations (verbal and visual), and referential connections between verbal and visual representations.

Computer Science researchers have similarly concluded that visualization must be accompanied by textual explanation in order to be pedagogically effective (Naps et al., 2000)(Stasko et al., 1993). Explanations help learners understand what they see (Brusilovsky, 1994). Complementing visualization with explanation is one of the 11 recommendations made for improving the effectiveness of visualization in a recent study (Naps et al., 2003). In fact, one of the researchers states: "perhaps, our focus should change from algorithm visualization being supplemented by textual materials to textual materials being augmented and motivated by algorithm visualization" (Naps et al., 2000). This is in fact, what we have undertaken: our programming tutors already provide textual explanation of the step-by-step execution of programs (Kumar, to appear). We have now supplemented the text explanations with program visualization. We had proposed a model-based scheme of visualization at a previous visualization workshop (Kumar, 2002b). In this paper, we will describe the resulting architecture, called Observer architecture.

In the Observer architecture, visualization of a program is driven by observation of its execution. Observer architecture relies on the availability of an observable model of the executing program. We will present the details of such a model in section ???. We will present the principles of Observer architecture of visualization in section ??? and provide an example of its implementation in online tutors that we have developed for programming, called problets ([www.problets.org](http://www.problets.org)). We will enumerate the advantages of Observer visualization in section ??? and end with discussion in section ???.

## 3 The Domain Model

Our programming tutors, called problets, automatically build a model of the program presented in each problem (Kumar, 2002a). The model consists of the structure and behavior of the program. The structure describes the programming objects and constructs, and how they are interconnected within a program. The behavior describes the execution semantics of the constructs used in the program. Consider the following C++ pointer code with a nested block:



**Figure 1:** Domain model of a C++ program involving pointers

```
void main(){
    int * ptr;
    {
        int count = 5;
        ptr = &count;
    }
    cout << *ptr;
}
```

The structure of the domain model for this program is shown in Figure ?? . In the figure, variables (such as global variables) and functions such as `main()` are components of the program. The pointer variable `ptr` and a block are components inside the function `main()`. The variable `count` is a component of the block. The behavior of a variable is modeled as a state diagram, with *declared*, *assigned* and *referenced* as nodes/states and possible operations (such as assignment) as arcs. Problots build such a model for each program and simulate it to generate 1) the output of the program; 2) semantic and run-time errors in the program, if any; 3) narration of the step-by-step execution of the program (Kumar, to appear).

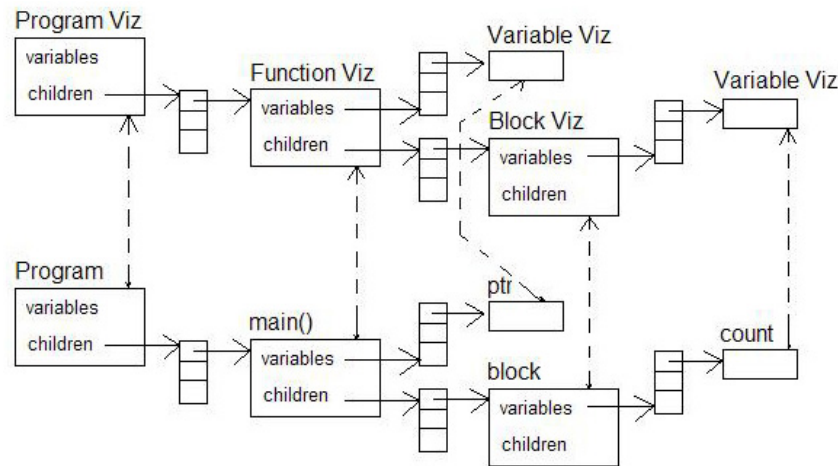
## 4 Principles of Observer Architecture of Program Visualization

In the Observer architecture of program visualization, each component in the program model is an observable object. Each of these objects is coupled with an Observer object whose responsibilities are to track and visualize the changes in the state of the observable object. The overall structure of the visualization objects is isomorphic to the structure of the model. This architecture is inspired by the Model-View-Controller pattern used for graphical user interface construction (Gamma et al., 1995).

The principles behind observer visualization are:

- **Model-driven:** Each visualization object is driven by a domain object, called model, that can be simulated to derive the model's behavior. The visualization and model are separate objects with mutually exclusive responsibilities.
- **Modular:** Each visualization object is responsible for visualizing only the model with which it is coupled.
- **One-directional Coupling:** A visualization object renders itself by first consulting the model with which it is coupled, detecting any change in the state of the model, and reflecting this change in its rendering. In other words, the visualization object follows the corresponding model; the visualization object does not effect changes in the model; the model does not actively drive the visualization, and may not even be aware of the existence of the visualization object.





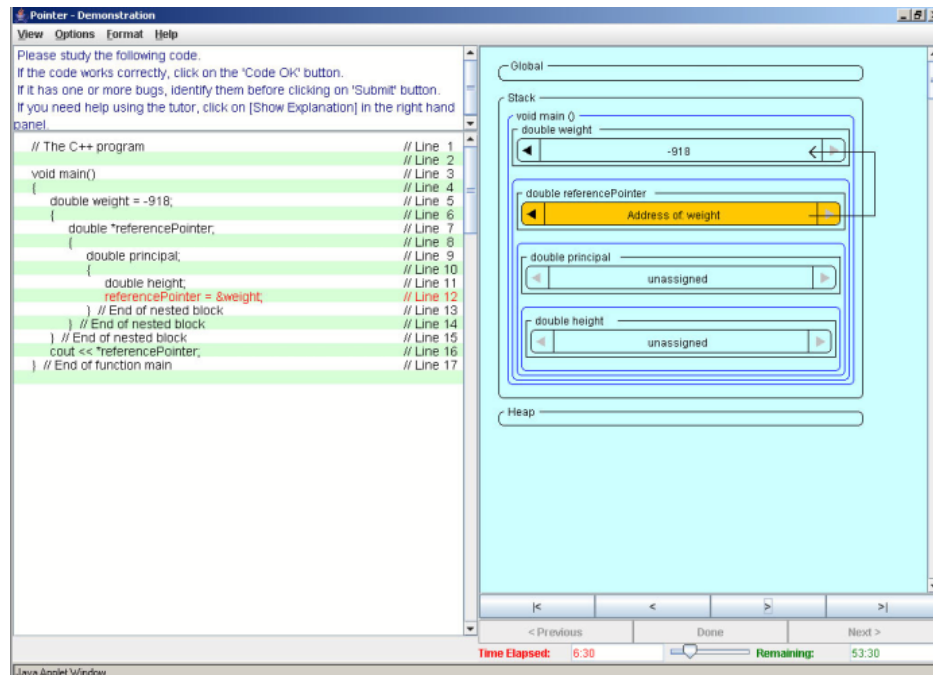
**Figure 2:** Visualization of the Program model shown in Figure ???. "Viz" objects are observers.

- **Hierarchical delegation:** Each visualization object delegates its components to render themselves before rendering itself, e.g., the function visualizer for `main()` delegates the pointer visualizer and the block visualizer to render themselves before rendering itself.
- **Message-passing for coordination:** Since each visualization object is modular, it is often necessary to coordinate among two or more visualization objects, e.g., when animating the assignment of the value of one variable to another, or depicting the assignment of a pointer to point to a variable or heap object. Message-passing is used for such coordination. In message-passing, a visualization object that needs to coordinate with another visualization object passes a message up and down the visualization hierarchy. A visualization object that receives the message acts on it if it is the intended target of the message, and passes it onwards if it is not.
- **Archival:** The visualization object maintains a history of the state changes of its model for future inspection by the learner, e.g., a variable visualizer maintains a history of all the values previously assigned to the variable. This design simplifies the model, allowing it to focus solely on the semantics of program execution.

Some of these principles, viz., separation of model and view, and one-directional coupling between the model and view have been proposed in the Matrix framework for building algorithm visualization (Korhonen and Malmi, 2001). Our work may be seen as an extension of the Matrix architecture in that we have proposed hierarchical delegation, message-passing for coordination and archival functions. Whereas a static repository is used in the Matrix architecture to keep track of the visualizations, we use hierarchical delegation.

The visualization objects created by proplets for the components in the program model shown in Figure ?? are shown in Figure ??. Dashed arrows connect observable objects in the model to their corresponding observers in the visualization hierarchy. The direction of the dashed-arrows denotes the direction in which data flows from the observable to the observer. The visualization of the program is obtained by rendering the observer objects.

Figure ?? shows the snapshot of the visualization generated by proplets using Observer architecture for a C++ program involving multiple levels of nested blocks and a pointer. In the snapshot, the data space of the executing program is shown as consisting of a global space, stack and heap. The activation record of the function `main()` is shown in the stack, and the variables of nested blocks are shown nested in this activation record. Message-passing was used by the visualizer to draw the arrow connecting `referencePointer` to the variable



**Figure 3:** Snapshot of the visualization of a program with multiple levels of nested blocks

`weight`. Note that left and right scroll controls are provided for variable visualizers so that the learner can examine the archived values of each variable.

As is typical of visualization environments, the following four learner controls are provided at the bottom: 1) one step forward; 2) one step backward; 3) fast forward to the end; and 4) rewind to the start. Since interaction is one of the keys to effective visualization, we did not provide auto-play facility. The visualization of step-by-step execution of the program code in the left panel (the currently executing line of code highlighted in red) is coordinated with the visualization of the data space in the right panel (the variables affected by the currently executing line highlighted in orange).

## 5 Advantages of Observer Architecture

There are several advantages to the Observer architecture of visualization:

- Observer-architecture is model-driven, i.e., visualization is driven by a model of the domain that can be simulated. Model-driven visualizations are capable of capturing more of the semantics of the domain being modeled (Naps et al., 2005). They support custom input data sets, one of the ways to improve the effectiveness of visualization (Naps et al., 2003) - in the case of program visualization, this includes visualizing learner-written programs. They reduce instructor overhead, one of the impediments to adoption and use of visualization (Naps et al., 2003), because, instead of specifying the visualization, the instructor can specify the program for which the visualization is desired, and let the model-driven observer architecture automatically create the visualization for the program.
- The architecture is scalable - when new constructs are added to the domain, visualization can be extended by adding corresponding observer objects in the visualization hierarchy and composing them as prescribed by the model. This facilitates incremental development of the visualization system, which is a plus when dealing with a large domain such as programming.

- A recent study suggests that eliminating the tight coupling between the visualizer and the program model might facilitate "easier and more widespread development of effective visualization systems" (Naps et al., 2005). In this vein, we have been working with the University of Pittsburgh to develop a distributed client-server architecture for code visualization (Loboda et al., 2006). Since the coupling between the program model and visualizers in the Observer architecture is one-directional, this architecture is well-suited for a distributed client-server deployment.
- The visualization presented in Figure 3 was designed for students in the junior/senior level Programming Languages course, hence the mention of stack, heap and global space, all components of the data space during program execution. Since the architecture defines a clear separation between observable and observer objects, we can develop alternative visualizations of the same program by simply developing alternative observer objects, e.g., a visualization for Computer Science I, wherein, the variables are shown as independent objects in the visualization space rather than as components embedded in scope objects. In other words, visualization can be modified independent of the model - Observer architecture enjoys the benefits of the Model-View-Controller pattern (Gamma et al., 1995).

## 6 Discussion

We have proposed Observer architecture for visualization. The architecture is general and domain-independent - it can be used for algorithm visualization just as well as program visualization, as long as an appropriate model is available. This may be seen as a disadvantage of Observer architecture for program visualization: the model needed for program visualization is a language interpreter, constructing which can be a daunting undertaking.

The best known model-driven program visualization system is Jeliot 3 for Java (Ben-Ari et al., 2002). It uses a modified version of the source code interpreter DynamicJava ([www.koala.ilog.fr/djava](http://www.koala.ilog.fr/djava)) as its model. It uses an intermediate code and an interpreter for the intermediate code to coordinate the model with the visualizer. The objects in the intermediate code interpreter, such as Value and Variable maintain a reference to their corresponding visualization objects, called actors, such as ValueActor and VariableActor. Therefore, Jeliot 3 "pushes" data from the model to the visualizer rather than have the visualizer passively observe the model, as proposed in the Observer architecture.

Problets are driven by parameterized templates. The user can execute and visualize a new program by simply entering its template into a probet. By defining a clear separation between observable and observer objects, Observer architecture makes it easy for the developer to provide multiple visualizations of a programming construct. When the developer provides alternative visualizations, the user can select among them as easily as selecting from a menu. The user will be able to customize the visualization to the extent that the individual visualization objects allow them, e.g., visualization of a variable may permit the user to change its appearance or animation scheme. Since the visualization objects are composed to build the overall visualization, the user can customize the visualization of each type of program object independent of the visualization of the other objects.

## 7 Acknowledgments

Partial support for this work was provided by the National Science Foundation's Educational Innovation Program under grant CNS-0426021.

## References

- P. Baggett. Role of temporal overlap of visual and auditory material in forming dual media associations. *Journal of Educational Psychology*, 76:408–417, 1984.

- M. Ben-Ari, N. Myller, E. Sutinen, and J. Tarhio. Perspectives on program animation with jeliot: In diehl, s. (ed.). *Software Visualization. LNCS*, 2269:31–45, 2002.
- P. Brusilovsky. Explanatory visualization in an educational programming environment: connecting examples with general knowledge. In *Proceedings of the 4th International Conference on Human-Computer Interaction, EWHCI'94, Berlin: Springer-Verlag*, pages 202–212, 1994.
- E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
- A. Korhonen and L. Malmi. Design pattern for algorithm animation and simulation. In *Proceedings of the First Program Visualization Workshop*, pages 89–100, 2001.
- A.N. Kumar. Model-based reasoning for domain modeling in a web-based intelligent tutoring system to help students learn to debug c++ programs. In *Proceedings of Intelligent Tutoring Systems (ITS 2002)*, pages 792–801, 2002a.
- A.N. Kumar. Explanation of step-by-step execution as feedback for problems on program analysis, and its generation in model-based problem-solving tutors. *Technology, Instruction, Cognition and Learning (TICL) Journal*, to appear.
- A.N. Kumar. Model-based animation for program visualization. In *Proceedings of the Second Program Visualization Workshop*, pages 37–44, 2002b.
- T. Loboda, A. Frengov, A.N. Kumar, and P. Brusilovsky. Distributed framework for adaptive explanatory visualization. In *Proceedings of The Fourth Program Visualization Workshop*, June 2006.
- E. Mayer and R.B. Anderson. Animations need narrations: An experimental test of a dual-coding hypothesis. *Journal of Educational Psychology*, 83:484–490, 1991.
- T. Naps, G. Rößling, P. Brusilovsky, J. English, D. Jarc, V. Karavirta, C. Leska, M. McNally, A. Moreno, R.J. Ross, and J. Urquiza-Fuentes. Development of xml-based tools to support user interaction with algorithm visualization. *SIGCSE Bulletin*, 37(4):123–138, 2005.
- T. L. Naps, J.R. Eagan, and L.L. Norton. Jhave - an environment to actively enhance students in web-based algorithm visualizations. In *Proceedings of the 31st SIGCSE Technical Symposium, Austin, TX*, pages 109–113, March 2000.
- T. L. Naps, R. Fleischer, M. McNally, G. Rößling, C. Hundhausen, S. Rodger, V. Almstrum, A. Korhonen, J.A. Velazquez-Iturbide, W. Dann, and L. Malmi. Exploring the role of visualization and engagement in computer science education. *SIGCSE Bulletin*, 35(2):131–152, June 2003.
- A. Paivio. Mental representations: A dual coding approach. *New York: Oxford University Press*, 1990.
- J. Stasko, A. Badre, and C. Lewis. Do algorithm animations assist learning? an empirical study and analysis. In *Proceedings of the INTERCHI 93 Conference on Human Factors in Computing Systems, Amsterdam*, pages 61–66, April 1993.

# An Integrated and “Engaging” Package for Tree Animations

Guido Rößling, Silke Schneider  
 Technische Universität Darmstadt  
 Darmstadt, Germany

roessling@acm.org

## Abstract

This paper describes a prototypical system that combines several aspects of engagement as defined in (Naps et al., 2003) for the topic of tree and tree algorithm animations.

## 1 Introduction

The ITiCSE 2002 Working Group on *Improving the Educational Impact of Algorithm Visualization* proposed the following engagement taxonomy for active AV use (Naps et al., 2003):

- 1: No viewing** – AV content is not used at all;
- 2: Viewing** – AV is used, but the user’s involvement is centered on consuming the content or controlling the progress (forward / pause / faster / ...);
- 3: Responding** – the user is asked certain questions while the content is presented, usually asking him or her to predict the next state of the animation;
- 4: Changing** – the user can modify the visualization, usually by specifying the input data set or selecting actions that cause an update of the visualization;
- 5: Constructing** – the user generates a new animation of a given algorithm;
- 6: Presenting** – the user presents a visualization to an audience for feedback and discussion. The content of the visualization does not have to be created by the user.

The Working Group also stated two hypotheses: first, that there is no significant difference regarding learning outcomes between the two lower levels; and second, that there will be a significant difference in learning outcomes when comparing any other “higher” level with a “lower” level. Thus, each step in the hierarchy (apart from the first two) is supposed to increase the possible improvement in learning outcomes.

In this paper, we present a new Java application for animating trees and tree algorithms. Apart from the engagement levels 1 and 2 supported by all AV tools, the application also supports levels 3 (responding) and 4 (changing), and can be used for level 6 (presenting).

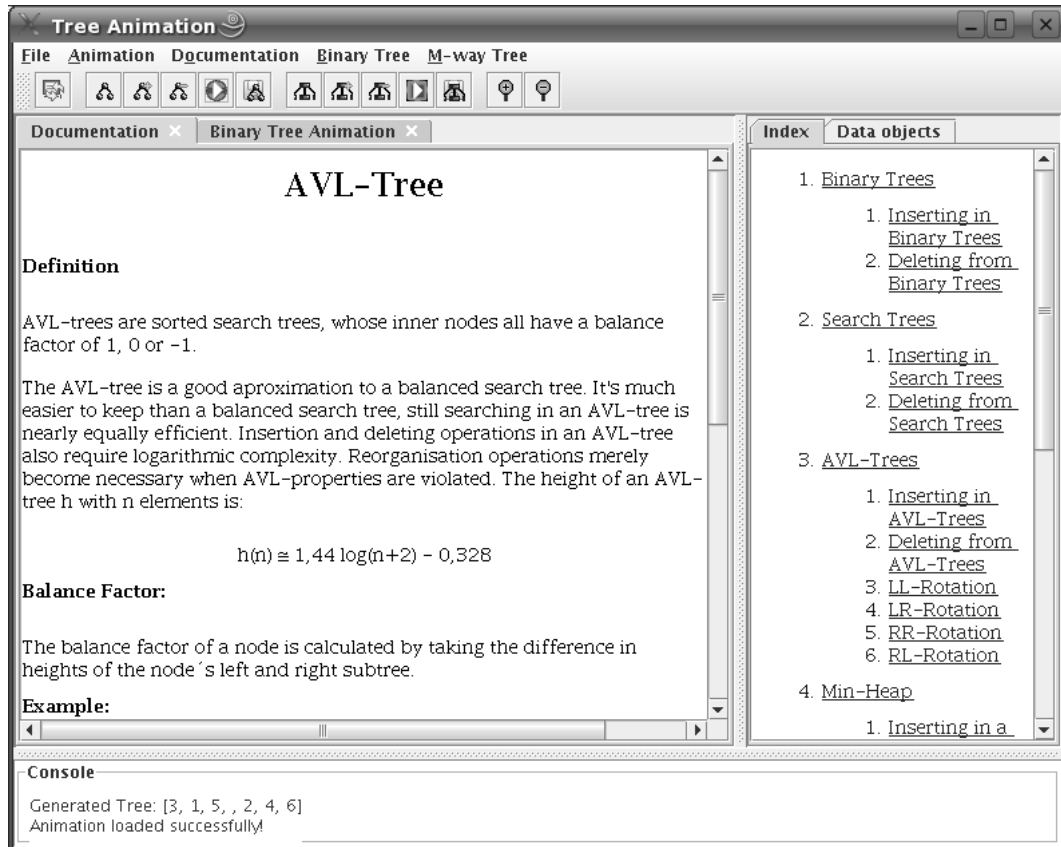
## 2 Visualizing and Creating Tree Animations

Figure ?? shows the welcome screen for the application. The application consists of four areas. At the top is the control area, containing the application’s menu bar and (draggable) tool bar for creating or modifying the current trees. The main part of the window is taken up by the *documentation* and *animation area*, which shows the current content.

The right side contains an index of the web pages as a reference. This page is essentially identical to the welcome page initially shown in the documentation tab of Figure ?. The *Data objects* view provides access to the current instances of the two supported tree types, binary and m-way trees. Finally, a console at the bottom of the window displays the output of the operations.

The buttons in the toolbar can be used for performing tree operations. The first and the last two buttons shown in Figure ?? can be used to load a new animation, and zoom in and

out of the animation, respectively. The two groups of five buttons each are used for creating a new tree, inserting a new key, removing a key, showing the generated animation, and saving the animation to disk. The first set of buttons is for binary trees, while the second group is for m-way trees, as indicated by the node form used for the icons.



**Figure 1:** The tree visualization application with index and AVL tree documentation

The application currently supports the following types of trees:

- binary trees,
- binary search trees,
- AVL trees,
- Heaps, implemented as *min* or *max heaps*,
- and m-way search trees with the implementation of a B tree.

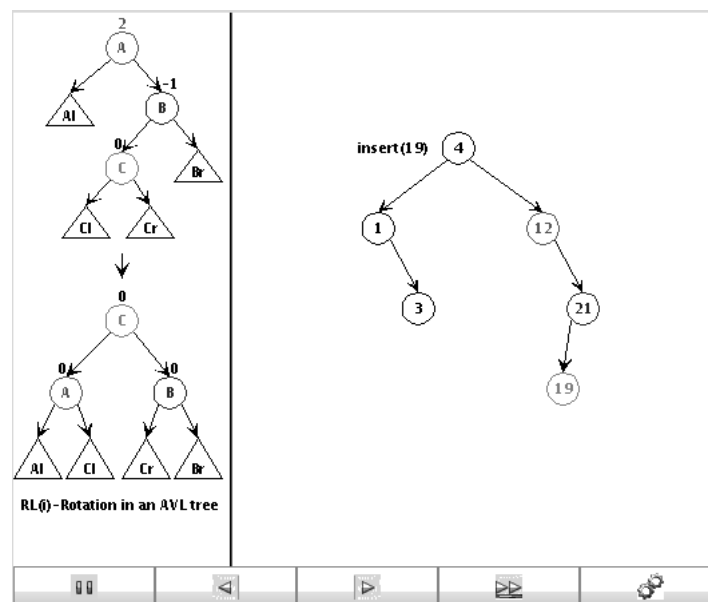
The trees were modeled as described in the literature, e.g. by Goodrich and Tamassia (2005). It is easy to add new implementations, for example for red-black trees. To do so, the implementation has to inherit from the appropriate class - for red-black trees, this is the class *tree.SearchTree*. Additionally, some of the methods implemented by the super class may have to be overridden. This usually concerns the methods for inserting or deleting keys, reordering the tree after operations, and in some cases, for adapting how the nodes are painted.

The user is free to create a new tree type by selecting either binary or m-way trees from the menu. He can then decide whether documentation shall be included in the display, and whether interactive questions should be asked during execution, as described by the 2002 ITiCSE Working Group (Naps et al., 2003). The user then works with the tree using the buttons shown in the toolbar of Figure ?? to insert or remove keys.

The “play” button displays the current state of the tree as an animation. The animation is generated in ANIMALSCRIPT and displayed by a customized front-end of ANIMAL. Figure ?? shows an excerpt of an AVL tree animation. In the Figure, the insertion of the key 19 will cause a  $RL(i)$  rotation to rebalance the AVL tree. This double rotation is known to be often poorly understood by students. Therefore, the animation incorporates a schematic display of the state of the (sub-)tree before and after the operation, shown on the left side of Figure ??.

Even after the animation is started, the user can continue modifying the tree using the buttons described above. The application creates add-on visualization code for the current animation to reflect the results of the actions taken by the user. This can then be loaded in using the “incremental loading” button shown at the bottom right of Figure ?. Instead of parsing the complete animation, only the added code will be parsed and added to the end of the current visualization, boosting the speed of the display.

The application described so far thus combines the generation of a tree according to the user’s operations and the stepwise animation of the output. Therefore, this places the application on the levels 2 (viewing) and 4 (changing) of the Working Group engagement taxonomy (Naps et al., 2003).



**Figure 2:** Excerpt of the AVL animation illustrating a  $RL(i)$  double rotation

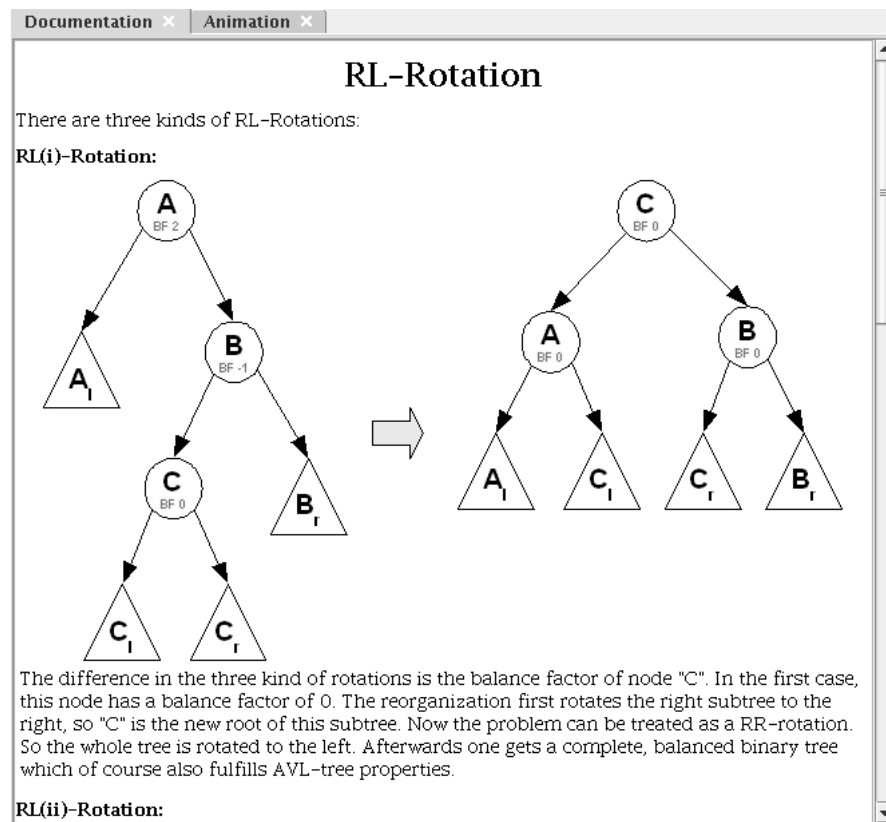
### 3 Embedded Documentation

Rößling and Naps (2002) stated the importance of incorporating hypertext explanations of the visual display. This documentation is meant to describe the operation of the underlying visualization engine, the mapping of the algorithm to the visualization, and should be “ideally adaptive to the current state of the algorithm”.

The approach up to this stage is similar to some existing AV systems, especially *Matrix-Pro* (Karavirta et al., 2004). However, some of the planned features, especially considering embedded documentation as shown in Figure ??, are not easily implemented in these systems. Basing the work on ANIMALSCRIPT makes generating content like this relatively easy.

Users of the application can use the built-in documentation provided in HTML format. Figure ?? shows part of the HTML page explaining the  $RL(i)$  rotation animated in Figure ?. This HTML documentation does not refer to actual values, but provides one page for each possible operation (insertion and removal of tree nodes and all balancing or reordering operations).

Additionally, the user can decide to turn on additional documentation to explain the individual operations. The documentation is adaptive to the current state of the algorithm and also mentions the concrete affected elements for any operation.



**Figure 3:** Example documentation: performing a  $RL(i)$  rotation on an AVL tree

This documentation is included in the animation, to prevent the user from having to switch views. When it appears, it is placed at the top of the animation window. The documentation is currently available for all supported tree types except for B-Trees. It will document both insertion and removal operations in one of four levels of detail:

**NO\_DOCUMENTATION** turns off all documentation features within the animation;

**EXPERT** offers a single line of rather terse documentation;

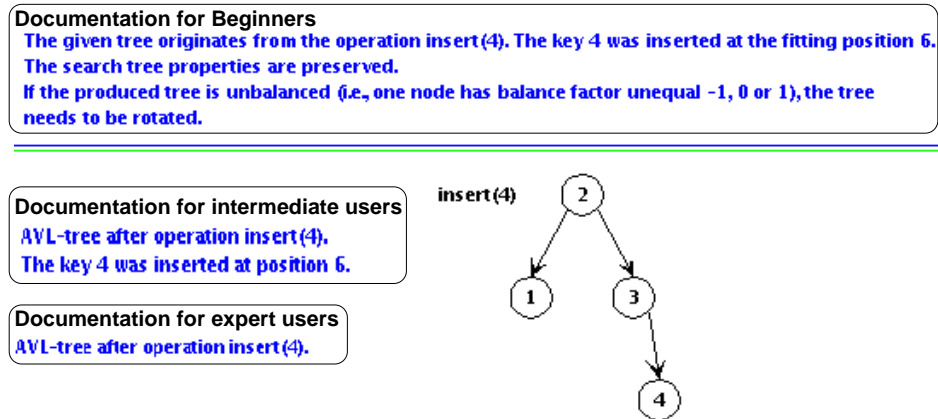
**INTERMEDIATE** offers two lines of documentation for each operation;

**BEGINNER** offers up to four lines of documentation for each insertion or removal.

Figure ?? shows the three documentation levels (*NO\_DOCUMENTATION* is not shown, as it is by definition empty). The basic animation content is the documentation for beginners, with the tree to the right. The documentation for intermediates and experts was grafted to this Figure to allow for easy comparison. The documentation, and indeed the whole application, is prepared for internationalization, and already addresses English and German. Adding more languages is fairly straightforward, as it mainly requires translating the resource text files.

The combination of live tree exploration with embedded *and* external documentation brings the application close to the realm of hypertextbooks described by Ross and Grinder (2002), also addressed by an ITiCSE 2006 Working Group. It also makes it easy to use the application to present tree algorithms to an audience, making the system applicable to level 6 (presenting) of the engagement taxonomy for its (admittedly) narrow focus.





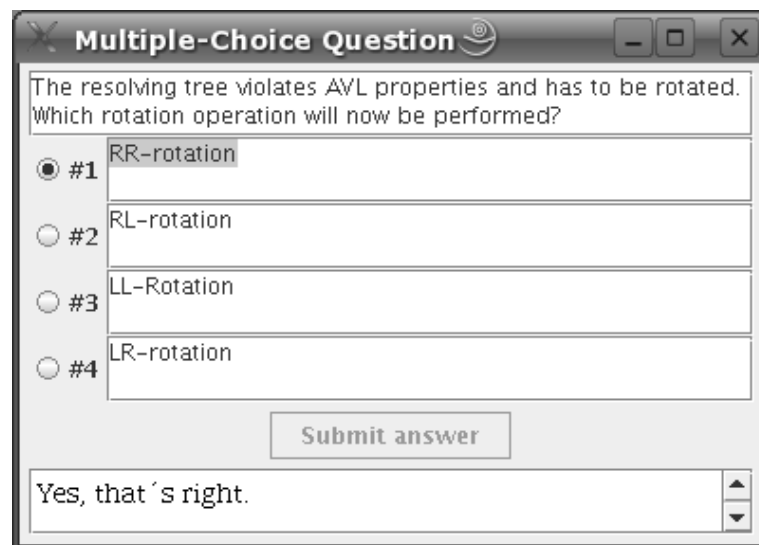
**Figure 4:** Documentation levels in the animation: beginner, intermediate, and expert

## 4 Interactive Prediction Support

The *responding* category in the engagement taxonomy expects the animation to be interrupted by questions. These questions will typically prompt the user to predict the next step performed by the algorithm, or ask him or her to describe the current visualization state.

For this end, we have added the *avInteraction* package to the application, as described by Rößling and Häussge (2004). The functionality of this package is similar to the “interactive prediction” supported by *JHAVÉ* (Rößling and Naps, 2002). The *avInteraction* package can easily be extended or adapted to other systems. Additionally, it offers other interesting features, such as skipping questions once a specified number of “related” questions were answered correctly.

Figure ?? shows an example pop-up window that prompts the user to predict which rotation will occur in the next step of the AVL tree reorganization. Currently, only AVL trees and B-trees incorporate interactive prediction, although this can be changed easily.



**Figure 5:** Interactive Prediction: determining the correct type of rotation

## 5 Summary and Further Work

In this paper, we have presented an extensive application for creating tree visualizations. Based on the underlying ANIMAL system, generating the animation code is fairly simple. The application merges several requirements from past Working Group reports and research papers: both “static” and “live” documentation are included, and interactive questions can be activated to make the learner’s session more engaging. Finally, the extensive collection of HTML pages describing the underlying data structures and algorithms makes this a promising learning tool for trees.

We have incorporated some new ideas into this prototype. First, this is one of the first systems that we are aware of that uses the actual values in its built-in documentation, and supports different levels of documentation detail. The ability to add code to the visualization without having to re-parse from the beginning is also a new feature, and definitely a premiere for the ANIMAL system.

In the future, we want to further explore the connection between the ideas used in this system and the hypertextbook concept (Ross and Grinder, 2002). We also plan to add some of the tried and tested features, especially the “incremental loading”, into the ANIMAL AV system.

We are also looking for educators interested in trying out the application. We would especially appreciate colleagues who are willing to take part in an evaluation of the system regarding learning outcomes, as our teaching obligations currently do not include access to the data structures course, and therefore keep us from the key clientele.

## References

- Michael Thomas Goodrich and Roberto Tamassia. *Data Structures & Algorithms in Java*. Wiley & Sons, 2005.
- Ville Karavirta, Ari Korhonen, Lauri Malmi, and Kimmo Stålnacke. MatrixPro - A Tool for Ex Tempore Demonstration of Data Structures and Algorithms. In *Proceedings of the Third Program Visualization Workshop, University of Warwick, UK*, pages 27–33, July 2004.
- 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, 2003.
- Rockford J. Ross and Michael T. Grinder. Hypertextbooks: Animated, Active Learning, Comprehensive Teaching and Learning Resources for the Web. In Stephan Diehl, editor, *Software Visualization*, number 2269 in Lecture Notes in Computer Science, pages 269–284. Springer, 2002.
- Guido Rößling and Gina Häussge. Towards Tool-Independent Interaction Support. In *Proceedings of the Third Program Visualization Workshop, University of Warwick, UK*, pages 99–103, July 2004.
- Guido Rößling and Thomas L. Naps. A Testbed for Pedagogical Requirements in Algorithm Visualizations. *Proceedings of the 7<sup>th</sup> Annual ACM SIGCSE / SIGCUE Conference on Innovation and Technology in Computer Science Education (ITiCSE 2002), Århus, Denmark*, pages 96–100, June 2002.

# An evaluation of the effortless approach to build algorithm animations with WinHIPE

Jaime Urquiza-Fuentes and J. Ángel Velázquez-Iturbide  
*ViDo Research Group, Rey Juan Carlos University, Móstoles, Spain*

{jaime.urquiza,angel.velazquez}@urjc.es

## 1 Introduction

Algorithm visualizations (AV) are used in computer science education since the early eighties (Baecker and Price, 1998). There are various surveys on using visualization as an aid for computer science education (Grissom et al., 2003; Hundhausen et al., 2002; Naps et al., 2003a,b; Stasko and Lawrence, 1998). In spite of their educational potential, they have not been incorporated into the mainstream of computer science education. This lack of use has two main reasons: from the instructors' point of view, animations are not usually easy to use, deploy and adapt to the course (Naps et al., 2003a; Ihantola et al., 2005); from the students' point of view, more interaction, than just viewing animations, is needed to obtain learning improvements (Grissom et al., 2003; Hundhausen et al., 2002; Naps et al., 2003b).

Naps et al. (2003b) defined a taxonomy of engagement levels for the different ways of interaction between the students and the animations. The pedagogical effectiveness of these engagement levels in AV has been analyzed (Grissom et al., 2003; Hundhausen et al., 2002; Naps et al., 2003b). The general conclusion is: the higher the engagement of students with AV technology, the better the learning outcomes. For instance Grissom et al. (2003) found learning improvements, at the understanding level of Bloom's taxonomy (Bloom et al., 1959), with AV extended with stop-and-think questions, the responding engagement level. We have found in the literature neither studies about AV technologies improving learning further than the understanding level of Bloom's taxonomy, nor studies about engagement levels in AV further than responding.

We have developed an effortless approach to build and maintain algorithm/program animations (Urquiza-Fuentes and Velázquez-Iturbide, 2005b). Thus we have extended the WinHIPE IDE with visualization facilities to produce web-based algorithm/program animations. These animations consist of four components: the animation itself (a sequence of visualizations), the source code, the description of the algorithm implemented, and the description of the problem solved by the algorithm.

The main aim is to minimize the work needed to produce the animation. Animations are built from a set of static visualizations representing the execution stages of a program, which are automatically generated. Apart from the typical edition-compilation-execution process, the user will have to select which visualizations will form part of the animation, and type the problem and algorithm descriptions. We have designed an information visualization technique called R-Zoom (Urquiza-Fuentes and Velázquez-Iturbide, 2005a) that helps on the task of selecting the visualizations, and typing text is not a complex task. Thus, we have an animation generation process very similar to the edition-compilation-execution process of a program, where the additional tasks are not complex; this is why we call it an effortless approach.

Following the framework provided by the engagement levels, we have conducted a controlled evaluation to test if our effortless approach of building AV (construction engagement level) improves learning. In our evaluation, we have compared the viewing engagement level against the constructing engagement level. Learning improvement has been measured in terms of the comprehension and application levels of Bloom's taxonomy. We have completed this evaluation with measurements of efficiency and students' satisfaction.

The rest of the paper is structured as follows. Section two describes the evaluation:

participants, variables studied, method and procedure. Then results of the evaluation are shown in section three. Finally, in section four, conclusions and future work are described.

## 2 Description of the evaluation

This evaluation attempts to find if there is any performance difference between viewing algorithm visualizations (viewing engagement level) and constructing algorithm visualizations (constructing engagement level). Improvements will be measured in terms of Bloom's taxonomy, with tasks related to the comprehension and application levels.

The context of this evaluation is an Algorithm Design and Analysis course at the Rey Juan Carlos University, where a group of students had to build algorithm animations using our effortless approach. In the evaluation, the tree breadth traversal algorithm was used.

### 2.1 Participants

Fifteen different subjects participated in the evaluation, thirteen were male and two female. Participation in the evaluation was voluntary. All of the subjects were students from the Algorithms course.

Participants were randomly divided in two groups: the control group and the experimental group (CG and EG respectively for the rest of the paper). Both groups were asked about their previous knowledge about the algorithm, only one student having previous knowledge. Therefore both groups belong to the same population and further results can be compared.

### 2.2 Variables

The independent variables of the evaluation were: pedagogical effectiveness and efficiency, and users' opinion about both approaches (viewing and building). The dependent variables were: the answers to a number of questions about the algorithm (mapped to the comprehension and application levels of Bloom's taxonomy), the time expended in studying the algorithm, the time used to complete a knowledge test about the algorithm, and the user's subjective opinion.

Learning improvements related to the comprehension level were measured with the following questions:

- What are the main ideas of this algorithm?
- Given the following tree, write the result of applying the algorithm to it.
- Given the following list, write the tree to which the algorithm was originally applied (note that the tree was balanced).
- Which is the existing relationship among the nodes of the tree, if the result is a strictly ascending ordered list?

Learning improvements related to the application level were measured with the following question: What modifications should be done on the algorithm to change the traverse direction to right-to-left?

Users' opinion was measured with a questionnaire where students were asked:

- if they thought that *building* (or *viewing*) algorithm animations had helped them in understanding the algorithm, and
- if they thought that algorithm animations are easy to *build* (or *use*) with WinHIPE.

## 2.3 Method and procedure

The evaluation was divided into two sessions: a training session where the IDE was shown to the students, and the experimental session where knowledge about the algorithm was evaluated. Participation was ten and thirteen students respectively.

The training session was two hours long. The instructor demonstrated the tool, he generated two web-based animations with WinHIPE as an example, and students generated two more animations. The animations used were unrelated to the algorithm that would be used in the experimental session. None of the students appeared to have problems using the tool. At the end of this session a questionnaire about the tool was completed by the students thus, we got their first impression about the tool.

The experimental session also was two hours long, and two weeks after the training session. First, we explained to the students that we were carrying out the evaluation, and that their participation would be voluntary. Next, we randomly formed the CG ( $n=7$ ) and EG ( $n=6$ ), and we checked that all students in the EG had attended the previous training session. Students of both groups were asked about their previous knowledge about the algorithm. Then, we gave the students all the materials they were allowed to use to study the algorithm, which was a textual description of the algorithm for both groups, and:

- a number of web-based algorithm animations to be viewed, built with WinHIPE, for the CG, and
- the source code of the algorithm, to build web-based algorithm animations, for the EG.

Students of both groups were asked to study the algorithm using the materials, until they thought that they had enough knowledge about it. Then, they completed the knowledge test and another questionnaire to collect their subjective opinion about their viewing/building learning experience.

## 3 Results of the evaluation

Learning effectiveness was tested by means of two levels of Bloom's taxonomy: comprehension and application. Both levels were graded in the range  $[0.0, 1.0]$ . Four questions were asked to test performance related to the *comprehension level*. The first question asked students to identify the main ideas underlying this algorithm; these ideas were: (1) operations with lists, (2) left-to-right direction in tree traversing, and (3) accumulation of recursive operations with subtrees. Students of both groups performed the same identifying ideas (1) and (2), while idea (3) was only identified by 14% of students (1/7) in the CG, but by 83% of students (5/6) in the EG ( $p = 0.013$ ). We did not find significant differences in performance in the second, third and fourth questions. Thus, the average grades for the understanding level were 0.88 for the EG and 0.73 for the CG, a 16% of learning improvement.

We found significant differences ( $p = 0.03$ ) in the answers to the question related to the *application level*. The CG obtained an average grade of 0.33, while the EG obtained an average grade of 0.77, a 60% of learning improvement.

We also measured the time expended with the materials and the time used to complete the knowledge test. With respect to the time expended by students using the materials to study the algorithm, students of the EG expended an average time of 49 minutes, while students in the CG expended an average time of 18 minutes. Difference of this time between groups was significant ( $p < 0.05$ ). But no differences were found ( $p = 0.194$ ) in the time used to complete the knowledge test between the EG and CG.

The students' first impression (after the training session) about WinHIPE and the building process was very good. None had previously used WinHIPE. All of them ( $n = 10$ ) thought that the web-based animations were easy to build, and that building animations would help them in understanding the algorithms.

This opinion was maintained by students after the experimental session. Answers to the questionnaire about users' satisfaction showed that students in the EG agreed with both ideas: building algorithm animations helped them understanding the algorithm, and web-based animations were easy to build with WinHIPE. All of the students in the CG agreed with: web-based animations helped them in understanding the algorithm, and web-based animations were useful and easy to use. We also asked these students about what approach would be more helpful in learning algorithm concepts: viewing or building. 71% (5/7) thought that both approaches were equally helpful: two of them just said this, but three also said that both approaches should be used together.

## 4 Conclusions and future work

We have developed an effortless approach to build and maintain algorithm animations, and we have made a short term evaluation of it. This evaluation compares the learning improvements achieved with two engagement levels: viewing (viewers) and constructing (builders), where our approach is used. Learning improvements were measured in terms of two levels of Bloom's taxonomy: understanding and application.

Results show that, at the understanding level, builders obtained slightly better results than viewers, 16% of improvement; at the application level, builders improved learning the 60% more than viewers. Builders expended much more time than viewers, but they did not complain about it: they thought it was necessary and fruitful to use this amount of time.

According to the students' opinion, builders believed that building algorithm animations had helped them in understanding the concepts of the algorithm, and thought that animations were easy to build. Most of the viewers thought that viewing and building were equally helpful, but half of them thought that both engagement levels should be used together.

We realize that the generalization of these results is limited because of: the low number of students (15), the short period of time evaluated (two sessions of two hours), and the topic used (just the tree breadth traversal algorithm). But we think that this is a promising result because: we have empirical evidence of learning improvements at the understanding and application level of Bloom's taxonomy, using our effortless approach together with the building engagement level; and students felt comfortable with this approach, perceiving it as effective and helpful.

As future work, we plan to conduct a long term evaluation of this approach with functional program animations; also, effortlessness have to be evaluated from the instructor's point of view (Ihantola et al., 2005) so, we will evaluate the usability of the building/management process of these web-based algorithm/program animations and collections of them.

## 5 Acknowledgments

This work is supported by the research project TIN2004-07568 of the Spanish Ministry of Education and Science. Also authors want to thank: Carlos Lázaro-Carrascosa for his valuable help during the experimental session, and students of the Algorithm Design and Analysis course of the Rey Juan Carlos University for participating in the evaluation.

## References

- R. Baecker and B. Price. The early history of software visualization. In J.T. Stasko, J. Domingue, M.H. Brown, and B.A. Price, editors, *Software Visualization*, pages 29–34. MIT Press, Cambridge, Massachusetts, USA, 1998.
- B. Bloom, E. Furst, W. Hill, and D.R. Krathwohl. *Taxonomy of Educational Objectives: Handbook I, The Cognitive Domain*. Addison-Wesley, 1959.

- S. Grissom, M.F. McNally, and T. Naps. Algorithm visualization in CS education: comparing levels of student engagement. In *Proc. 2003 ACM Symposium on Software Visualization*, pages 87–94, New York, NY, USA, 2003. ACM Press.
- C.D. Hundhausen, S.A. Douglas, and J.T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing*, 13(3):259–290, 2002.
- P. Ihantola, V. Karavirta, A. Korhonen, and J. Nikander. Taxonomy of effortless creation of algorithm visualizations. In *ICER '05: Proc. 2005 International Workshop on Computing Education Research*, pages 123–133, New York, NY, USA, 2005. ACM Press. doi: <http://doi.acm.org/10.1145/1089786.1089798>.
- 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. ITiCSE 2003 working group reports: Evaluating the educational impact of visualization. *ACM SIGCSE Bulletin*, 35(4):124–136, June 2003a.
- T. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, and J.Á. Velázquez-Iturbide. ITiCSE 2002 working group report: Exploring the role of visualization and engagement in computer science education. *ACM SIGCSE Bulletin*, 35(2):131–152, June 2003b.
- J.T. Stasko and A. Lawrence. Empirically assessing algorithm animations as learning aids. In J.T. Stasko, J. Domingue, M.H. Brown, and B.A. Price, editors, *Software Visualization*, pages 419–438. MIT Press, Cambridge, Massachusetts, USA, 1998.
- J. Urquiza-Fuentes and J.A. Velázquez-Iturbide. R-zoom: A visualization technique for algorithm animation construction. In *Proc. IADIS International Conference Applied Computing 2005*, pages 145–152. IADIS Press, 2005a.
- J. Urquiza-Fuentes and J.A. Velázquez-Iturbide. Effortless construction and management of program animations on the web. In Rynson W.H. Lau, Qing Li, Ronnie Cheung, and Wenyin Liu, editors, *Advances in Web-Based Learning - ICWL 2005, 4th International Conference*, volume 3583 of *LNC3*, pages 163–173, Berlin, Germany, 2005b. Springer Verlag.

# Annotations for Defining Interactive Instructions to Interpreter Based Program Visualization Tools

Essi Lahtinen, Tuukka Ahoniemi  
*Tampere University of Technology*  
*P.O.Box 553, FIN-33101 Tampere, Finland*

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

## Abstract

An interpreter based visualization tool can be used for creating visualization exercises that engage the student to work with the visualization instead of just watching it passively. This article describes how to make the interpreter instruct the student interactively. The idea is to use the interpreter for testing the code that the student has written and give the instructions according to the test results. This only requires minor work for implementation of the visualization tool.

## 1 Introduction

To engage the student to learn when using a program visualization, the visualization needs to activate the student to take part in it (Naps et al., 2003). Thus the developers of program visualization tools should include features that enable interactive communication between the student and the visualization tool.

This paper introduces a way to make an interpreter based visualization tool to give feedback about the students' own programming solutions. First we introduce an existing technique of annotated instruction of a program visualization tool called VIP. Then we introduce our work in progress and discuss the idea.

## 2 Program Visualizations

There are numerous ways of implementing program visualizations. At its simplest the visualization can be a line of moving pictures, e.g. implemented with a technique like Flash. The problem of this kind of approach is that the interaction between the user and the tool is very limited. Visual interpreters that work on any program code are easier to adjust to communicate with the user. This kind of program visualization tools, e.g. Jeliot 3 (Moreno et al., 2004) and VIP (Virtanen et al., 2005), are available for free use on introductory programming level.

Even if the program visualization tools include features that enable interaction with the user, they are most often used as *illustrative visualizations* (Lahtinen and Ahoniemi, 2005) just to present concepts and run some example programs. Using the visualization tool as an exercise environment is more beneficial for the students (Ahoniemi and Lahtinen, 2006).

## 3 The Instructions and the Annotations

A program visualization tool typically shows the visualized program code in a window. This window can include short comments as a part of the code. The purpose of the visualization is to explain the run of the program to the student. To aid this the visualization tool can include extra instructions to describe the program and to draw the students attention to the essential.

To make sure that the additional instructions do not make the source code unnecessarily long, they can be placed in a separate instruction window. The instruction window can show only the instructions related to the line of code that is currently executed. This way the



**Table 1:** Annotations that are implemented in VIP.

Annotation	Meaning
<code>/*@</code>	A normal instruction shown every time the associated statement is executed.
<code>/*@&lt;nr&gt;</code>	The instruction is shown on the <nr>:th time when the statement is executed.
<code>/*@n</code>	The instruction is shown on all the times when no other number is specified.
<code>/*@odd</code> <code>/*@even</code>	The instruction is shown on the odd/even numbered times when the statement is executed. (Useful, e.g., in function calls inside a loop structure.)
<code>/*@init</code>	The instruction is shown before the program starts running. Before VIP starts the program run, it goes through all these instructions highlighting the line they are associated with. (Can be used to explain libraries, namespaces etc. lines never ran.)
<code>/*@lock on</code> <code>/*@lock off</code>	Editing lock on/off. The locked lines can not be edited in the VIP code editor. Neither do they lose their instructions like unlocked code does after editing. The code is editable as default.

students who need more explanation can follow long instructions and the students who do not need it can skip reading since the instruction is separate from the actual visualization.

The teacher can write the instructions in the visualized program source code inside comments that contain some special characters to mark them. This way the code can be compiled normally to test that it works properly. The visualization tool will just parse the beginning of the comment to see, whether it will be displayed in the code window or in the instruction window. This is also easy for the teacher since only one text file is required for making a visualization. It is also possible to make the instructions versatile by using some markup language, e.g. HTML. This way the teacher can emphasize some parts of the instruction by using colour or italic, or clarify the text with a little picture.

### 3.1 The Existing Annotations

To improve the expression of the instructions, the visualization tool developer can add annotations to define the visibility of the instructions. For example, a certain instruction could be shown only when the execution reaches the statement for the first time. When the execution returns again to that statement there will be a new instruction. This way the teacher can build the instructions more useful and interesting for the student. E.g. when executing a function call statement on the first time the program reaches that line, the teacher would probably want to explain how the parameter passing happens. When the function call finishes and the execution comes back to the same line the teacher can explain the passing of the return value of the function.

VIP visualization tool includes a few simple annotations that are explained in Table ???. In VIP the instructions are always related to the code line right after the instruction. When using multiple annotated instructions for one line of code the interpreter shows the first instruction whose annotation is valid.

The existing annotations are connected to the execution order of the program. Thus they enable writing wide explanations of the program for the students but do not really support interactivity. If the visualization tool provides a possibility to edit the visualized program source code the tool can also be used for visualization exercises described by Lahtinen and Ahoniemi (2005).

### 3.2 Work in Progress

The instructions can be developed further so that the values of the variables in the program code can be examined and the instruction text is shown only if the internal state of the

```

/*@lock on */
#include <cstdlib>
#include <iostream>

void sort( int array[], unsigned int size ) {
/*@lock off */
    // Press the "edit code" button and write your implementation here!
/*@lock on */
}

int main() {
    const int AMOUNT = 3;
    int numbers[ AMOUNT ];

    std::cout << "Enter " << AMOUNT << " integers: ";
    for( int i = 0; i < AMOUNT; ++i ) {
        std::cin >> numbers[ i ];
    }

    sort( numbers, AMOUNT );    // The function sorts the elements of the array ascending

    std::cout << "The integers sorted: " << std::endl;
    for( int i = 0; i < AMOUNT; ++i ) {
        std::cout << numbers[ i ] << std::endl;
    }

/*@hide on */
    int _vip_array1[] = { 3, 2, 1 };    int _vip_array2[] = { 2, 1, 3 };
    sort( _vip_array1, AMOUNT );        sort( _vip_array2, AMOUNT );
    bool _vip_test1 = true;              bool _vip_test2 = true;

    for( int _vip_i = 0; _vip_i < AMOUNT - 1; ++_vip_i ) {
        if( _vip_array1[ i ] > _vip_array1[ i + 1 ] ) {
            _vip_test1 = false;
        }
        if( _vip_array2[ i ] > _vip_array2[ i + 1 ] ) {
            _vip_test2 = false;
        }
    }

    if( _vip_test1 && _vip_test2 ) {
        _vip_out << "Your function seems to work correct! Well done! You can continue with the next exercise..." << endl;
    } else if( !_vip_test1 && _vip_test2 ) {
        _vip_out << "Test sorting an array that is in a reverse order. Your fuinction might need a correction there." << endl;
    } else {
        _vip_out << "The function does not seem to work very well. Review page 123 of the course material and try again." << endl;
    }
/*@hide off */
    return EXIT_SUCCESS;
}
/*@lock off */

```

**Figure 1:** An example of the use of hidden code annotations.

**Table 2:** The annotations in development.

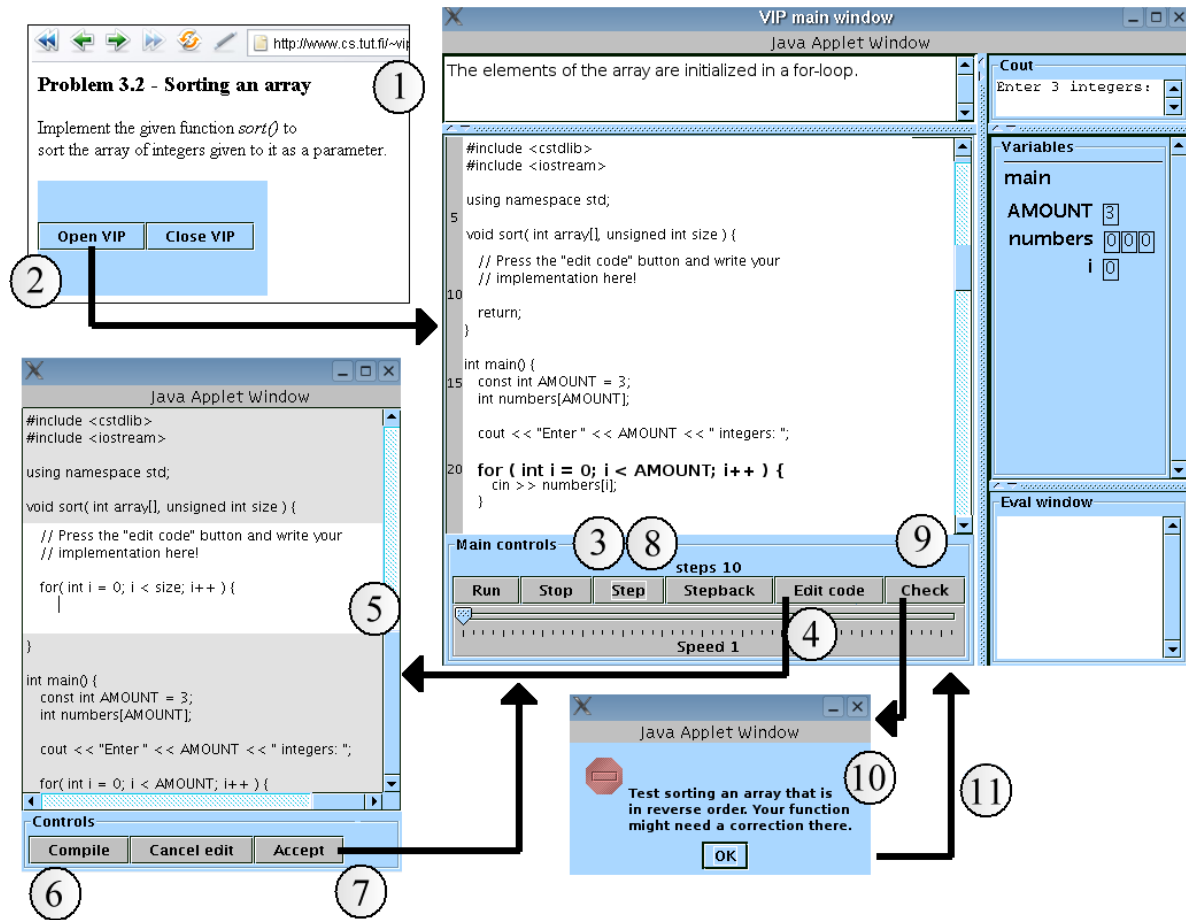
Annotation	Meaning
/*@hide on	Hidden code for testing the students solution. Is not run normally or visualized
/*@hide off	in the visualization window. Is also locked.

visualized program is certain. For example, the teacher can specify a “well done” text to be shown if the student has succeeded to give the program an input that led to a certain state. Or in a *utilizing visualization* if the student has succeeded to modify the program in a right way he can be granted for the exercise.

The annotations to enable this need to include statements that are interpreted. Our idea is to use the visual interpreter for interpreting these too. An annotation can mark some parts of the code hidden. In the hidden code, the teacher can test the program. E.g., if the task for the student is to implement a function, the hidden test code can run the function a couple of times and display the instruction text for the student according to the results of the tests.

Figure ?? shows an example code of a programming exercise visualization where the student is supposed to implement a function to sort the integers in an array. The teacher has implemented a main function that calls the function that the student is supposed to design, write and test. The hidden code in the end of the example first tests the students solution and then instructs the student to the right direction. The instruction texts are written to a special output stream named `_vip_out` whose content is displayed in a pop up window. The new annotations used in this code are introduced in Table ??.

Figure ?? shows how the student will interact with the visualization tool in the visual-



**Figure 2:** An example of how the visualization exercise could work.

ization exercise described in Figure ?? . The phases are marked in Figure ?? as follows: The assignment is presented for the student in a website (1) with a button to open the visualization tool (2). In the visualization tool window the student can start studying the given code and visualize it step-by-step using the “step” button (3). By pressing the “edit code” button (4) the student can proceed to writing his solution in the code editor. The student can edit only the unlocked code on the white background (5) (the function to be implemented). After making the changes he can try to compile the code with the “compile” button. When he is satisfied with his solution he returns to the visualization window by accepting the changes (7).

After editing the code the student probably tries to run his own code in the visualization tool either step-by-step or using the “run” button (8). In both these situations only the visible code is executed and the student will see how the code he developed works. If he thinks the task is completed the student can run the check (9). As the “check” button is pressed both the visible and the hidden code are executed and a pop-up window containing the results is shown (10). The result window shows the text that was written to the special output stream when running the tests in the hidden code.

All of the windows presented in Figure ?? are separate, so the student can still return to the visualization tool or the code editor (11). He will still be able to see the assignment description in the first window and the instructions in the pop up window.

## 4 Discussion

The functionality introduced here is easy to add to an existing interpreter since it uses the same interpreter to implement the tests in the hidden code. After this addition the visualization

tool can be used e.g. for *utilizing visualizations* and *problem-solving visualizations* (Lahtinen and Ahoniemi, 2005).

This annotation technique is also easy for the teacher. He can check the whole visualization code and the test code with a normal compiler because everything extra is placed inside a comment. The extra output stream can be defined as `cout` while testing. Also the whole exercise can be created just by writing one text file which is easy for the teacher. Of course designing the exercises requires time.

A flaw of this approach is that it can be used only for certain kinds of exercises. For example testing a function is easy, testing an `if`-statement can be done if the code inside the `if`-statement modifies some of the variables, but testing an `if`-statement that only outputs some text needs a different approach. The `if`-statement can be run many times like the function is in Figure ?? if the hidden code contains a loop structure.

It is of course possible to add many other types of tests to an interpreter too. E.g., tests for the output of the program or tests for finding certain expressions in the source code. However, these kinds of tests can not be run using only the interpreter. The first one requires an extra process to run the interpreter with different inputs and compare its outputs. The second one requires parsing the source code for a different purpose.

When implementing tests like this to a visual interpreter also the effectiveness of the interpreter should be considered. A visual interpreter can be implemented with the idea that it does not need to be very effective, since the visualization needs to be observed and thus is not supposed to be too fast. In this case it should be checked that the interpreter is effective enough to run the tests in the hidden code in a reasonable time. At least the student should be shown some kind of a progress bar to follow while waiting.

## 5 Conclusions

It is easy to add annotated instructions to an interpreter based visualization tool. With some imagination these instructions can be used for creating visualization exercises that engage the student to work with the visualization. Adding this little feature to the tool widens the pedagogical uses of the tool greatly.

As our future work we will use the developed visualization exercises as a part of the course material and gather experiences on their usage. We intend to evaluate them by measuring the students learning results.

## References

- Tuukka Ahoniemi and Essi Lahtinen. Visualizations in Preparing for Programming Exercise Sessions. *Proceedings of the Fourth Program Visualization Workshop*, June 2006.
- Essi Lahtinen and Tuukka Ahoniemi. Visualizations to Support Programming on Different Levels of Cognitive Development. *Proceedings of The Fifth Koli Calling Conference on Computer Science Education*, pages 87–94, November 2005.
- 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.
- 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.
- Antti T. Virtanen, Essi Lahtinen, and Hannu-Matti Järvinen. VIP, a visual interpreter for learning introductory programming with C++. *Proceedings of The Fifth Koli Calling Conference on Computer Science Education*, pages 129–134, November 2005.

# Peer Review of Animations Developed by Students

Rainer Oechsle, Thiemo Morth  
*University of Applied Sciences, Department of Computer Science,  
Trier, Germany*

oechsle@fh-trier.de, mortht@fh-trier.de

## Abstract

This paper describes the project “Visual Knowledge Communication”, a joint project that started recently. The partners are psychologists and computer scientists from four universities of the German state Rhineland-Palatinate. The starting point for the project was the fact that visualizations have attracted considerable interest in psychology as well as computer science within the last years. However, psychologists and computer scientists pursued their investigations independently from each other in the past. This project has as its main goal the support and fostering of cooperation between psychologists and computer scientists in several visualization research projects.

The paper sketches the overall project. It then discusses in more detail the authors’ subproject which deals with a peer review process for animations developed by students. The basic ideas, the main goals, and the open questions are described.

This paper is a work-in-progress report. We expect useful recommendations and hints by the participants of the workshop.

## 1 Introduction

Visualizations are becoming increasingly important for different kinds of communication. Newspapers, flyers, TV programs, and web pages today contain several times more pictures, graphics, diagrams, graphs etc. than they did several decades or years ago.

Visualizations also have attracted considerable interest in different scientific disciplines within the last years. In psychology, the role of pictures and animations as a learning aid was investigated in an impressive number of research projects (Lowe, 2001), (Lowe, 2003), (Lowe and Schnotz, 2006), (Schnotz and Lowe, 2003). An interesting research field in learning with the help of pictures focuses on the fundamental differences between verbal and visual communication. One of the most significant differences is the fact that visualizations are usually more concrete than verbal descriptions can be. That means, it is not possible to describe a certain fact in a general way; a picture always illustrates a fact by an example. E.g., a drawn triangle is always a specific one; it is not possible to draw a general triangle. So the question is whether and how pictures and animations help in learning, especially whether and how the learners are able to generalize the perceived learning material.

In computer science, there are several fields that deal with visualizations. The area of computer graphics has achieved major breakthroughs. With the help of an off-the-shelf PC (near) photorealistic views with different kinds of light sources can be computed and rendered in real-time. The most prominent applications for these techniques are all kinds of games. In the future, it is expected that the field of virtual, augmented and mixed realities will become increasingly important.

A second field is scientific visualization. The objective of this field is the finding of appropriate representations for a huge amount of data gathered from different kinds of measurements in physics, chemistry, geology etc.

Last but not least, the subject of this workshop, namely program and algorithm visualization, is a third field in computer science that deals with pictures and animations. The field covers, e.g., algorithm animation techniques for computer science education as well as appropriate visualization techniques of huge (legacy) programs in order to get an overview of the existing dependency and usage relations of the different parts (modules, classes, ..) of the source code.

In the past, psychologists and computer scientists pursued their investigations independently from each other. On the one hand, psychologists often are not aware of the current trends in computer science. Computer scientists, on the other hand, are focusing primarily on technology. The consideration of human perception, thinking, and learning is often neglected by computer scientists.

This situation was the starting point for a joint project called “Visual Knowledge Communication” that started recently. The partners are psychologists and computer scientists from four universities of the German state Rhineland-Palatinate: the University of Applied Sciences in Kaiserslautern-Zweibruecken, the University of Koblenz-Landau, the University of Trier, and the University of Applied Sciences in Trier.

In section ?? of this paper an overview of the overall project is given. The ideas of the authors’ subproject are then described in more detail in section ??.

## 2 Project “Visual Knowledge Communication”

The project “Visual Knowledge Communication” is funded by the Research Ministry of the German state Rhineland-Palatinate in Mainz. It started in March / April 2006. As its goal the project is supposed to investigate different problems in the area of visual knowledge communication. The project consists of six subprojects. Each subproject is driven by a separate research group. The subprojects are independent of each other. However, each research group will cooperate with and get advice from one or two of the other research groups. Detailed information about the cooperation relationships is listed in the project plan. We omit this aspect here.

In the following table the six subprojects and their goals are briefly described:

Title	Leader	Goal
Interactive Mixed-Reality-Visualization in Learning Systems	S. Mueller, University of Koblenz-Landau, Department of Computer Science	Investigation of useful applications for mixed realities, especially for interactive learning systems.
Knowledge Acquisition with Interactive Animations	K. Wender, University of Trier, Department of Psychology	Investigation of fundamental learning mechanisms caused by interactive animations, especially study of the conditions for effective, abstract knowledge acquisition.
Forms of Visualization and Dynamic Adaptation for Information Retrieval Interfaces	J. Krause, University of Koblenz-Landau, Department of Computer Science	Study of the design of graphical user interfaces and the dynamic adaptation of user interfaces to specific users and specific situations, especially user interfaces for information retrieval systems.
Learning Systems with Personalized Graphical User Interfaces	B. Reuter, University of Applied Sciences in Kaiserslautern-Zweibruecken, Department of Economics	Investigation of possibilities for the personalization of graphical user interfaces for learning systems.

Title	Leader	Goal
Emotional Effects of Static and Dynamic Pictures in Science Education of Secondary Schools	A. Mueller and W. Schnotz, University of Koblenz-Landau, Department of Psychology	Investigation of the conditions for positive effects of decorative pictures in science education of secondary schools.
Animations Developed by Students and Assessed by a Peer Review Process	R. Oechsle, University of Applied Sciences in Trier, Department of Computer Science	See following section.

### 3 Subproject “Animations Developed by Students and Assessed by a Peer Review Process”

#### 3.1 Basic Idea

In (Naps et al., 2003) a taxonomy has been defined comprising six levels of engagement with respect to algorithm and program visualization (AV):

1. No use of AV technology.
2. Viewing: Controlling the direction and pace of the animation.
3. Responding: Answering questions about the AV content.
4. Changing: Modifying the AV content.
5. Constructing: Building a new visualization of a given algorithm or data structure.
6. Presenting: Presenting an AV content to an audience for feedback and discussion.

Our subproject has its focus on the highest levels of engagement. But instead of presenting an algorithm animation to an audience, we propose an alternative way to deepen the involvement of the students: we want the students to do a peer review in order to mutually assess their animations. Thus we replace level 6 of the engagement taxonomy by level 6’:

**6’ Peer Reviewing:** The developed animations are assessed by a peer review process.

The review process is supposed to follow well-known procedures (Sitthiworachart and Joy, 2004): development and submission of animations, review of animations by peer students (each animation is reviewed by  $N$  other students,  $N = 2, 3, 4$ ; each student thus has to review  $N$  animations), and finally, each student has to study not only the reviews of her own animation, but also the other reviews for these animations that she had reviewed.

The basic question that this project is supposed to answer is: In which way does this approach really enhance the knowledge and understanding of computer science subjects compared to other approaches? Can it be carried out in an efficient way (i.e. is the effort justified with respect to the learning results)?

#### 3.2 Project Plan

We foresee three stages for our project:

1. Assessment and selection of appropriate tools or systems that students shall use in order to develop their animations.
2. Development / adaptation of a web-based peer review system.

3. Realization of several animation peer reviews. This part of the project will be supported by Karl Wender's psychologist research group from the University of Trier.

The web-based peer review system will not be developed from scratch. We rather plan to integrate the peer review of animations into our own web-based system that we are currently developing for peer reviews and a semi-automatic assessment of student programs (style checking, testing). There are no significant open questions for stage 2 of our subproject. This does not hold for the other two stages.

### 3.3 Open Questions

Because the project started only several weeks ago, we do not have any results yet. We would like to use the workshop to discuss our open questions:

- We plan to use the animation peer review in the students' first year. To us it is not clear right now whether we will find an animation environment that is suited for this goal. Besides general animation tools like PowerPoint and Flash, we are currently looking at specialized algorithm animation tools like Animal (Rößling and Freisleben, 2002) and Jawa (Pierson and Rodger, 1998). We do not have a complete list of requirements yet (Pollack and Ben-Ari, 2004). We are unsure whether there is an animation system that students can use without a considerable learning effort. It is not clear whether the animations should be developed in an interactive way, with the help of a scripting language, or with the help of a Java library. We have not yet decided if the animation should be a "dumb animation" (i.e. just moving pixels), or if the animation should be embedded within a real algorithm. We fear that there is the following trade-off: if an animation system is easy to use, the students will not have much control over the appearance; if an animation system offers many degrees of freedom, the learning effort for the animation system as well as the implementation effort for a specific animation will be high.
- We are still unsure about the experiments that we will carry out. Especially, which subjects shall be animated by the students (sorting algorithms, tree and graph algorithms)? How much time will we spend for the experiments? Against which other learning methods do we compare the learning results (for example, constructing animations with / without peer reviews, constructing animations with / without presentations)? Should the animations only be reviewed by those students who developed themselves animations for the given subject? Or is it better to let the animations be reviewed by students who have not been studying the animation subject before? How will we measure the learning progress?

## 4 Summary

This paper described the project "Visual Knowledge Communication", a joint project with psychology and computer science partners from four universities of the German state Rhineland-Palatinate. The paper gave an overview of the overall project. It then discussed in more detail the authors' subproject which deals with a peer review process for animations developed by students. The basic ideas, the main goals, and the open questions have been described.

This paper is a work-in-progress report with many questions yet unanswered. We are looking forward to discuss these questions with the participants of the workshop. We expect to get many interesting and helpful recommendations and hints.

## 5 Acknowledgments

We would like to thank the Research Ministry of the German state Rhineland-Palatinate, Mainz, for funding the project "Visual Knowledge Communication". We also thank our



partners from the different subprojects for their ideas and contributions. Especially, we would like to thank Wolfgang Schnotz, University of Koblenz-Landau, for having initiated the project “Visual Knowledge Communication”. Finally, we would like to express our thanks to the two anonymous reviewers for their comments.

## References

- Richard K. Lowe. Understanding information presented by complex animated diagrams. In J.F. Rouet and A. Biarreau, editors, *Multimedia learning: Cognitive and instructional issues*, pages 65–74. Pergamon, 2001.
- Richard K. Lowe. Animation and learning: Selective processing of information in dynamic graphics. *Learning and Instruction*, 13:157–176, 2003.
- Richard K. Lowe and W. Schnotz, editors. *Learning with Animation: Research and implications for design*. Cambridge University Press, 2006.
- T. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, and J. Velazquez-Iturbide. Exploring the role of visualization and engagement in computer science education. *ACM SIGCSE Bulletin*, 35(2): 131–152, June 2003.
- W. Pierson and S. H. Rodger. Web-based animation of data structures using jawaa. In *29th SIGCSE Technical Symposium on Computer Science Education*, pages 267–271, 1998.
- S. Pollack and Mordechai Ben-Ari. Selecting a visualization system. In *Proceedings of the 3rd Program Visualization Workshop*, pages 134–140. University of Warwick, UK, 2004.
- G. Rößling and B. Freisleben. Animal: A system for supporting multiple roles in algorithm animation. *Journal of Visual Languages and Computing*, 13(2):341–354, 2002.
- W. Schnotz and Richard K. Lowe. External and internal representations in multimedia learning. *Learning and Instruction*, 13:117–123, 2003.
- J. Sitthiworachart and M. Joy. Effective peer assessment for learning computer programming. In *9th Annual Conference on the Innovation and Technology in Computer Science Education (ITiCSE)*, pages 122–126, 2004.

# SSEA: A System for Studying the Effectiveness of Animations

Eileen T. Kraemer, Bina Reed, Philippa Rhodes, Ashley Hamilton-Taylor  
*Computer Science Department, The University of Georgia, Athens, GA USA*

eileen@cs.uga.edu

## 1 Introduction

Many programmers, instructors, and programming students have a strong intuitive belief that visualization is valuable for communicating information about the state and behavior of programs. However, although some empirical studies of the benefits of such visualizations have shown benefits, results have been mixed (Lawrence, 1993; Hansen et al., 2000; Lawrence et al., 1994; Kehoe et al., 2001; Tudoreanu et al., 2002) or inconclusive (Byrne et al., 1999; Jarc, 1999) overall. Further research has sought to explain these results (Hundhausen et al., 2002; Naps et al., 2003). One factor that has been overlooked in many studies is the effect of the design of the animation itself on viewer comprehension.

In our work we seek to investigate attributes of algorithm animations and other program visualizations that affect how well the user can understand the concepts the designer intends to convey. We believe that effective PV systems must support perceptually appropriate graphical design, layout, and animation, as well as good pedagogical design. We are working to identify and evaluate perceptual, attentional, and cognitive features of program visualizations that affect viewer comprehension. This work is performed in the context of a larger project that involves observational studies of instructors, empirical studies of the perceptual properties of low-level animation actions (Davis et al., 2006) through the VizEval environment (Rhodes et al., 2006) and the development of improved presentation and interaction techniques for program visualization in the context of computer science education.

In this paper we describe SSEA, a System for Studying the Effectiveness of Animations, an environment designed to support the empirical study of program visualizations. SSEA was created as a testing environment for studying the effects of various attributes of visualization design on viewer comprehension. Researchers can create a suite of animations in SSEA with a design characteristic in mind. SSEA allows users to view, interact with, and answer questions about an animation, and records the viewer's interactions and responses to questions. Researchers can then examine the log files generated and perform analysis of the responses and timings with respect to the attribute being examined.

## 2 SSEA System Description

SSEA integrates a visualization interface with a question panel, pop-up questions and a monitor. The monitor automates data collection of user interactions and events. A SKA (Taylor and Kraemer, 2002) module manages the display of the animation and adjusts the visualization to any users interactions. SKA (Support Kit for Animation) is an algorithm animation system designed to support the instructional task.

A user's interaction may begin with a questionnaire, if the researcher so specifies. The questionnaire typically collects demographic information. The main SSEA user interface is seen in Figure 1. It includes an animation area (A), a pseudocode display (B), animation controls (C), and a question area (D). Animated graphics depicting the underlying algorithm or other aspect of program behavior are shown in the animation area. The pseudocode display is also animated, highlighting the code that is currently being executed by the algorithm. The animation area and pseudocode area are synchronized by SKA. A high resolution monitor is required to view the multiple areas of SSEA in their entirety.

The user can control the animated display. Specifically, the user can select the data set, control the speed, start, stop, pause and step the animation. In addition the user may move

the slider seen in section C of figure 1 to select a point in animation time at which to restart the animation.

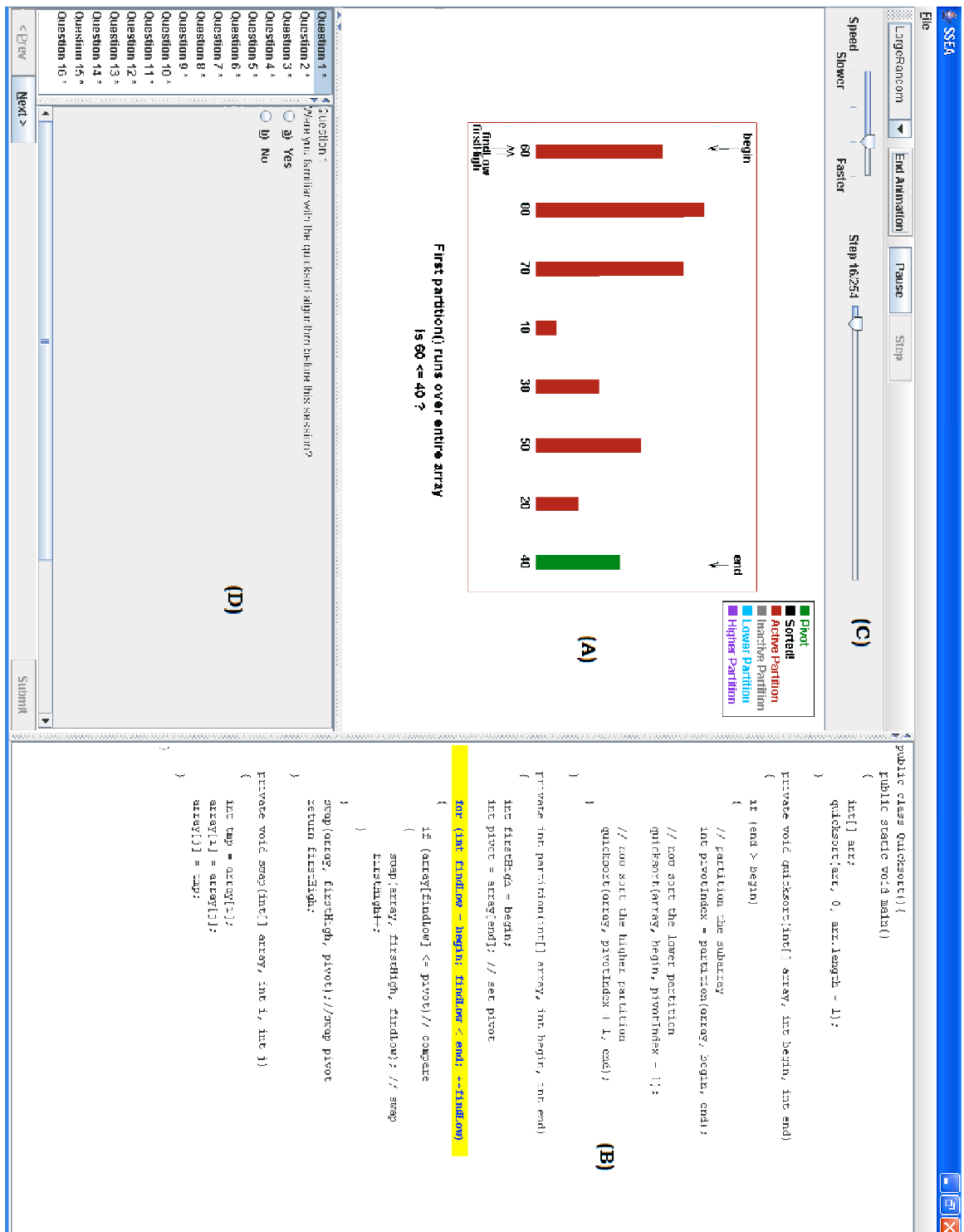
The questions seen in area D of figure 1 may be answered by the user at any time. A listing of all questions is displayed to the left side. The selected question and its choices are displayed to the right. An asterisk next to a question number in the list indicates that the question is still unanswered. The user may also return to a question and change his answer. After all questions have been answered a *submit* button becomes enabled. When the user clicks *submit* the session is complete. Both the intermediate choices and the final answers for all questions are recorded in the log. In our studies we used this question mode to solicit answers to “traditional” questions. For example, in a quicksort algorithm we might ask about the order in which operations are performed, or the contents of the array after some operation or set of operations has been performed.

In addition to these traditional questions the experiment designer may also specify popup questions. These questions are associated with the animation of the algorithm on a particular data set. The popup questions appear only the first time the user runs the animation of the algorithm with the associated data set. We used these questions to solicit answers about what the user just saw - which two bars were just compared, or which two bars just exchanged values, for example. To prevent users from using the highlighted code to supply the correct answer to the popup questions, the popup box is positioned over the area containing the corresponding code and is not movable, nor does any action proceed, until a response is provided by the user. The popup text then updates to indicate the correct answer to the question. The user may then close the popup and continue with the animation. User answers to these popup questions are also recorded in the log.

### 3 Findings using SSEA

To date, we have conducted two empirical studies using SSEA. In the first study we evaluated the effect of two attributes of an algorithm animation: presence of cueing (flashing) to indicate to the viewer that two data elements have been compared, and type of animation (arcing move or grow/shrink) to indicate that two data elements have exchanged values. We evaluated the impact of these attributes both on perception of the animated changes and on viewer comprehension of the depicted algorithm, as measured by the number of correctly answered questions in two question sets: “traditional” (comprehension) questions and “popup” (perception) questions. While no significant effect on comprehension was observed for either flash cueing or exchange motion, we note that comparison and exchange behaviors were redundantly cued in the animation studied. Significant effects were found for flash cueing and “move” versus “grow” in the perceptual questions displayed in popup windows. In summary, it was not clearly shown whether the perceptual benefits of flash cueing and the “move” depiction of exchange carry over to comprehension of the algorithm. In practice, the degree to which such perceptually beneficial techniques aid in comprehension may rely on the extent to which the cued behaviors are redundantly encoded in the depicted algorithm.

The use of popups that ask the user to answer questions about what they have just seen would seem to encourage users to pay closer attention to an animation. In a second study we looked at the effect of the use of popup questions, the type of questions (reactive vs. predictive) and the use of feedback on viewer perception (ability to answer popup questions) and comprehension (ability to answer traditional questions). Analysis of the resulting data indicates that providing feedback to pop-up questions increases the participant’s ability to correctly answer both pop-up questions and traditional test questions. However, the use of the popup questions themselves adversely impacted the user’s ability to correctly answer traditional test questions, though not significantly. We speculate that the demands of the frequent popup questions may have caused “over-attention” to detail, preventing the user from mentally stepping back to observe the higher-level behavior of the algorithm.



**Figure 1:** SSEA screen shot. (A) Animation area where visualization of underlying algorithm is displayed. (B) Pseudo code display: highlights code being executed by underlying algorithm, which is synchronized with animation area. (C) Animation controls: viewer controls playback of algorithm animation. (D) Question listing: viewer responds to questions designed by the researchers to evaluate the viewer's comprehension of the algorithm.

## 4 Conclusions and Future Work

The SSEA environment provides good support for carrying out empirical studies of program visualizations. Two studies have been conducted to date. Support for sound actions has been added to the SKA package and additional studies that look at the roles of voice-over and non-speech audio in animations are under development.

## References

- Michael Byrne, Rich Catrambone, and John T. Stasko. Evaluating animations as student aids in learning computer algorithms. *Computers and Education*, 33(4):253–278, 1999.
- Elizabeth Thorpe Davis, Kenneth Hailston, Eileen Kraemer, Ashley Hamilton-Taylor, Philippa Rhodes, Charalampos Papadimitriou, and Bath-Ammi Garcia. Examining perceptual processing of program visualization displays to enhance effectiveness. In *Annual Meeting of the Human Factors and Ergonomics Society*, 2006. in submission.
- S. Hansen, N.H. Narayanan, and D. Schrimpscher. Helping learners visualize and comprehend algorithms. *Interactive Multimedia Electronic Journal of Computer-Enhanced Learning*, 2(1), 2000.
- Chris Hundhausen, Sarah Douglas, and John T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing*, 13(3):259–290, 2002.
- Duane Jarc. *Assessing the benefits of interactivity and the influence of learning styles on the effectiveness of algorithm animation using web-based data structures courseware*. PhD thesis, The George Washington University, 1999.
- Colleen Kehoe, John Stasko, and Ashley Taylor. Rethinking the evaluation of algorithm animations as learning aids: An observational study. *International Journal of Human-Computer Studies*, 54(2):265–284, 2001.
- A. Lawrence. *Empirical Studies of the Value of Algorithm Animation in Algorithm Understanding*. PhD thesis, Georgia Institute of Technology, College of Computing, 1993.
- Andrea Lawrence, Albert Badre, and John T. Stasko. Empirically evaluating the use of animations to teach algorithms. Technical Report GIT-GVU-94-07, Georgia Institute of Technology, Atlanta, GA, 1994.
- T. Naps, S. Cooper, B. Koldehofe, C. Leska, G. Rößling, W. Dann, A. Korhonen, L. Malmi, J. Rantakokko, R. Ross, J. Anderson, R. Fleischer, M. Kuittinen, and M. McNally. Evaluating the educational impact of visualization. *ACM SIGCSE Bulletin*, 35(4):124–136, 2003.
- Philippa Rhodes, Eileen Kraemer, Ashley Hamilton-Taylor, Sujith Thomas, Matthew Ross, Elizabeth Thorpe Davis, Kenneth Hailston, and Keith Main. Vizeval: An experimental system for the study of program visualization quality. In *Visual Languages and Human-Centric Computing*, 2006. in submission.
- Ashley Hamilton Taylor and Eileen Kraemer. Ska: Supporting algorithm and data structure discussion. *ACM’s 33rd SIGCSE Technical Symposium on Computer Science Education*, 34(1):58–63, 2002.
- Mihail Tudoreanu, Rong Wu, Ashley Hamilton-Taylor, and Eileen Kraemer. Empirical evidence that algorithm animation promotes understanding of distributed algorithms. In *Proceedings of the 2002 IEEE Symposium on Human-Centric Computing, Languages and Environments (HCC’02)*, Washington, D.C., pages 236–243, 2002.

# Jeliot 3 in a demanding educational setting

Andrés Moreno<sup>1</sup>, Mike S. Joy<sup>2</sup>

<sup>1</sup>*Department of Computer Science, University of Joensuu, Joensuu, Finland*

<sup>2</sup>*Department of Computer Science, University of Warwick, Coventry, UK*

`Andres.Moreno@cs.joensuu.fi`, `M.S.Joy@warwick.ac.uk`

## Abstract

We report the preliminary findings of a qualitative investigation into how students approach a program visualization tool, and whether the approach depends on how they are taught to use the tool. Volunteer students in an undergraduate programming course were divided into two groups. One group was taught programming concepts explicitly using the tool, and required to use it to solve weekly exercises and projects, whereas the other group only used the tool on a voluntary basis. We identify those aspects of using the tool which the students find beneficial, and discuss the limitations of the animations provided by Jeliot 3.

## 1 Introduction

Software Visualization (SV) tools are intended to be used in the early stages of the learning path of a programmer, teaching them the basics of programming (Moreno et al., 2004), algorithms (Korhonen et al., 2001), and the software development cycle (Walker et al., 1998). SV tools convey important information by means of graphics or animations. Although it is believed that visual representations are useful, previous evaluations of Software Visualization tools suggest that such tools do not significantly improve the learning outcome (Hundhausen et al., 2002). Nonetheless, some experiments (Kehoe et al., 2001) have shown that they motivate the students when learning algorithms or basic programming skills.

### 1.1 Jeliot 3

Jeliot 3 (Moreno et al., 2004) is a program animation tool oriented towards novice programmers, in which animations represent the step by step execution of Java programs. Each step in the execution is made explicit, and the resulting animation is a simulation of how the virtual machine interprets the program code. The animation takes place in a “theater” that is divided in four separated areas. The central and main area is the evaluation one, to which messages, method calls, values and references are moved to and from the other areas as the evaluation proceeds. Students can program in Jeliot 3, and later they can visualize their program and follow its execution path.

## 2 Literature Review

Ben-Bassat Levy et al. (2002) studied the pedagogic advantages of Jeliot 2000 (the predecessor to Jeliot 3) compared with the use of an Integrated Development Environment (IDE), TurboPascal, to trace programs. Their study divided high school students in an Introduction to Programming Course into two groups. Both of them attended the same lectures, but in their laboratory sessions one group used Jeliot 2000, and the other used TurboPascal. The results identified an improvement in understanding amongst those who used Jeliot 2000. The largest improvement came for the mediocre students – Jeliot 2000 represented a viable working model of the execution of a program that they could understand and follow. However, the strongest and weakest students appeared not to gain much from the tool, since it was either too simple or too complex for them.

In another experiment, qualitative data were gathered from 35 students who were taking part in a second course on programming and were using Jeliot 3 (Kannusmäki et al., 2004).

Students were asked to reflect on their Jeliot 3 usage while solving programming and debugging tasks. The findings noted the problems that Jeliot 3 presented to those students with some experience in programming and to those without previous use of Jeliot 3. Again, novice students were more positive regarding the usage of the tool.

Kehoe et al. (2001) argue that a different approach was needed to demonstrate the effectiveness of algorithm animations. They hypothesize that the pedagogic value of algorithm animation tools would be more apparent when used in an interactive setting, such a homework assignment. They claimed that the pedagogic value of such tools would increase if accompanying instruction is provided at the same time as the animation. Further, a meta-analysis notes the importance of how the animation is administered to the students, rather than what is actually visualized (Hundhausen et al., 2002).

Petre (1995) discusses the skills required to understand visualizations, and how novices and experts extract different information from them. The type of training given in using the visualization influences the development of the student's skill to notice secondary notation cues. The acquisition of this skill distinguishes novices from experts; hence, it is considered fundamental to understanding visualizations.

### 3 Purpose

The idea for our investigation was to extend the experiment done by Ben-Bassat Levy et al. (2002). However, rather than comparing two different tools, this investigation focuses on students' use of a single visualization tool in two different learning contexts, and compares their attitudes towards the tool and their usage of the tool. More importantly, we have sought to check whether the information conveyed by the animation is correctly understood by both groups, even if they have been using the tool differently. We may expect using the tool in different contexts should produce different effects on the students, both in their performance using the tool (Kehoe et al., 2001) and in their understanding of the taught concepts (Petre, 1995). We formulate the following hypotheses:

- *H1*: Jeliot 3's animations are comprehended better by those students who have been required previously to solve tasks with Jeliot 3. For example, they should be able to follow and reproduce what it is happening in the animation theater in greater detail than the students that have not been explicitly instructed.
- *H2*: Students' expectations from the tool should be different depending on how they have approached it, and we can distinguish two possible uses of the tool, either as a learning aid, or as a debugger.

### 4 Methods

The participants in the study consisted of 6 Maths undergraduate students in an introductory course on programming, who voluntarily used Jeliot 3 as a programming environment for their weekly tasks and projects. They were divided in two groups, the "voluntary" group and the "normal" group. The former consisted of two students who attended an extra session each week during which where Jeliot was used exclusively. One of the students reported having prior programming experience of VisualBasic, and one other was highly proficient in the Microsoft Office suite, but none had previous Java programming experience.

All participants were taking part in a 10-week long "introduction to programming" module for Mathematics undergraduate students taught at the University of Warwick. The module lecturer collaborated with the experiment by encouraging students to use Jeliot 3 and by using it in the initial lectures and lab sessions.

The tool was deployed as a Java web-start application. A web-page<sup>1</sup> was set up to host the application, along with the on-line documentation.

<sup>1</sup><http://www.cs.joensuu.fi/jeliot/warwick/>

## 4.1 Procedures

For half an hour each week, the “voluntary” group of students completed a set of extra mini-tasks, designed to explain one or more concepts using Jeliot 3. These taught concepts were closely related to the material taught during that week’s lecture, and with what they were expected to use when solving the weekly assignments. Students completed these mini-tasks individually in a controlled laboratory environment, with the researcher (the first author) present to help them to understand the visualization and to make progress with the mini-tasks. Students’ answers to these mini-tasks were collected, and the researcher checked their answers for programming and animation misconceptions, and gave further explanation to the students as to why their answers were incorrect or what the animation meant. This was the only difference between the two groups.

Both the “normal” and the “voluntary” group used Jeliot 3 at the weekly lab sessions (2 hours long each), and at home to complete their tasks. The only support given was that specifically requested during the lab sessions.

During the first four weeks, the researcher was present in the ordinary lab sessions as a teacher assistant, interacting with all the students taking part in those session (no more than 20 students), and writing down his own observations, and the attitudes and problems students found in Jeliot 3.

After the 4th week, when the students were supposed to move on to more complicated project work, semi-structured interviews were carried out with the six students from both groups.

## 4.2 Data Analysis

The qualitative data analyzed in this report consists of students’ responses during the interviews. The core of the interview explored students’ attitudes towards Jeliot 3, for example their knowledge about programming concepts, and whether they would continue to use Jeliot in future assignments. Assessing their knowledge was performed in two steps. They were asked firstly to explain which steps Java went through when creating an object, and secondly to describe the animation of an object being created as they watched it in Jeliot. In total, six interviews were audio recorded and transcribed.

A list of sentences was extracted from the transcripts and grouped according to how Jeliot was useful to the students, and how they used it. As expected, two main categories emerged: Jeliot as a learning aid, and as a debugger. Subsequently, the researcher’s notes from the lab sessions were incorporated and used to further explain the reasonings given by the students and to illustrate the behavior of all the students taking part in the lab sessions, whether using Jeliot 3 or not.

# 5 Results

## 5.1 Jeliot as a learning aid

All of the students mentioned the existence of a “gap” in the course. At different points of the course, they realized that the requirements needed to follow the rhythm of the course and the practical assignments changed abruptly. The gap appeared between theory and the application of that theory; however, a gap also appeared when students were not able to grasp a new concept. Arrays and objects, but also the basic syntax, seemed to be troublesome, and students asked for further explanations on those topics.

Students’ answers suggested that the animations involving arrays and objects were the most useful ones. Only one student, from the “normal” group, did not mention Jeliot to be of help for the gap. All the other students used Jeliot 3 when they tried to close a gap which had opened in the course, and they found that Jeliot 3 was useful then, no matter which group they were from.



Even if they claimed that Jeliot helped them to understand concepts, they failed to reproduce the execution path that follows an object creation. Only two of the students from the “normal” group produced an acceptable description of the allocation process. Both of them were aware that Jeliot had helped them.

The other students showed some misunderstandings when describing the animation of an object allocation. The `this` reference, and argument passing from the constructor call to the constructor frame, caused most of the problems. For example, some of them expected parameters to be passed “by value” straight to the object. Nonetheless, the animations corresponding to basic statements, e.g., assignments or variable declarations, were correctly described by all subjects but one from the “normal” group.

## 5.2 Jeliot as a debugger

All but one of the participants who belonged to the “voluntary” students used Jeliot 3 to complete the assigned tasks during the four observed lab sessions.

Most of them followed an iterative approach when completing the tasks: (i) read instruction, (ii) write method, and (iii) create and run short test. The first iterations of this cycle can be understood as part of the learning process. The following iterations had a clearer debugging aim. They used Jeliot to visualize and identify bugs.

When dealing with project work which consisted of several files, they had to temporarily abandon Jeliot 3 because it does not support classes a set of classes stored in different files. However, they came back to use Jeliot 3 when they did not understand why their programs were not working.

It was while using Jeliot 3 as a debugger that students found out the current limitations of the tool. Most of them suggested the possibility of being able to “jump through the animation” in order to visualize only what they were interested in.

None of the students mentioned the usage of two Jeliot 3 features that could be activated from different menus of the tool— the ability to insert a breakpoint, and the ability to call a method different than the main method.

## 5.3 Lab session observations

From observations during the lab session, we noted that the complete novices all exhibited a common behavior. Students appeared to have two different goals when attending the course. Their main aim is to pragmatically complete the course by solving the assigned tasks, and their secondary goal is to understand how their solution is achieved. It was noted that neither the animation (if they were using Jeliot 3) nor the console output (if they were using the system Java implementation rather than Jeliot 3) were used to understand the basics of programming, unless explicitly taught. Students found it difficult to understand compound assignments, e.g., `a+=3;`. Some of the students using Jeliot 3 could not decipher the steps taken in its animation. Only after the animation was explained to them, could they make sense of it.

Students also grew tired of the length of the animations, and would run them at full speed. However, at the third lab session, one student from the “voluntary group” identified non-syntactic bugs in his program while visualizing the animation at full speed.

## 6 Discussion

According to our hypothesis *H1*, students from the “voluntary” group should show a greater vocabulary and understanding than the “normal” students. The two “voluntary” students showed a similar level of detail when describing the animation, but they also showed several misconceptions about the object creation animation, which suggests that the voluntary mini-

tasks might not have been of great help. One possible explanation could be that the half hour the “voluntary” students additionally spent working with objects was not enough.

Hypothesis *H2* would be supported if we observed the “voluntary” group exhibiting different behaviour to the “normal” group. In this case, students from both groups were able to use Jeliot to try to understand new concepts and to resolve bugs in their code, and we do not have evidence to support *H2*.

In this small investigation we have not been able to identify any clear distinction between students who received extra education on using Jeliot as a tool to assist in learning programming, and those who used it to learn programming but without any special help. For example, it did not matter if the “voluntary” students had been taught explicitly how Jeliot 3 creates an object, or if the “normal” students had seen the object creation animation many times before. It appears that they did not contrast the knowledge they have, with the information that the animation keeps repeating.

The number of repetitions of the same animation may desensitize students to the importance of the animation itself, and reduce it to a “movie of moving boxes”. They were able to follow the boxes, and discover when they have been misplaced, but whilst this is useful to identify bugs, the underlying meaning of the animation may not have been assimilated.

## 7 Conclusions and Future Work

This investigation reflects on how students have used an educational tool in the context of solving programming assignments in a programming course for undergraduates. Two groups of students were observed using the tool. The first group were given explicit instruction in its use, and required to use it to solve mini-tasks related to the weekly assignment. The other group did not have any help apart from the one they requested during the lab sessions.

From these initial results, it appears that Jeliot 3 animations are hard for novice students to understand. The transfer of knowledge from the tool to the student is not successful. Even students who have been explained explicitly the meaning of the animations have problems understanding or applying this later. Despite this issue, both groups of students found that the help they could get from the animation was useful to debug their programs. Moreover, all the students found Jeliot 3 easy to use, and most the students who started using Jeliot 3 continued to use it.

Results from the investigation are inconclusive, but they spread some more light in why Software Visualization tools are not as widely used in education as expected. In the case of Jeliot 3, we believe that adding verbal explanations to the animation (Mayer, 2001) and “stop and think” questions as in JHAVE (Naps, 2005) might solve some of the issues found in this investigation.

Mayer’s multimedia theories (Mayer, 2001) claim that learning can be improved when two different channels are used to convey information. In multimedia, this means that the simultaneous combination of audio and graphics will optimize the cognitive load of the learner. An alternative, and simpler, approach is to combine graphics with textual captions. Such extended animations would provide novices with important information to help them comprehend the secondary notation used in an animation (Petre, 1995), and to build a correct mental model of a program computation.

“Stop and think” questions are those which are woven into the animation. They evaluate student comprehension by asking specific questions about the running algorithm (e.g., which is the next value of the variable stepper?). Such questions have been found to help retain students’ attention during an animation and to encourage them to process recently acquired data into meaningful information (Naps, 2005).

Finally, we believe that students get benefits from using Jeliot 3. However, Jeliot 3 is not flexible enough to support the different levels of users’ knowledge and cognitive skills (e.g. some students grasp the concepts faster than others), and the different usage patterns (e.g.

comprehending or debugging). These three factors would imply the need to model the student, so that content (a combination of animation questions and explanations) could be adapted to the student skills and the task they are performing with Jeliot. Moreover, explanations could help weaker students by providing them with extra and valuable learning material.

## Acknowledgments

The authors thank the participants and the staff at the University of Warwick, specially the students at MA117 and their teacher Petr Plechac. We also thank the rest of the Jeliot team and Justus Randolph, who helped us in designing this experiment.

## References

- Ronit Ben-Bassat Levy, Mordechai Ben-Ari, and Pekka A. Uronen. The Jeliot 2000 program animation system. *Computers & Education*, 40(1):1–15, 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, 2002.
- Osku Kannusmäki, Andrés Moreno, Niko Myller, and Erkki Sutinen. What a novice wants: Students using program visualization in distance programming course. In A. Korhonen, editor, *Proceedings of the Third Program Visualization Workshop*, pages 126–133, The University of Warwick, UK, July 2004.
- Colleen M. Kehoe, John T. Stasko, and Ashley Talor. Rethinking the evaluation of algorithm animations as learning aids: an observational study. *International Journal of Human Computer Studies*, 54(2):265–284, 2001. URL [citeseer.ist.psu.edu/kehoe99rethinking.html](http://citeseer.ist.psu.edu/kehoe99rethinking.html).
- Ari Korhonen, Lauri Malmi, and Riku Saikkonen. Matrix - concept animation and algorithm simulation system. In *ITiCSE '01: Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education*, page 180, New York, NY, USA, 2001. ACM Press. ISBN 1-58113-330-8. doi: <http://doi.acm.org/10.1145/377435.377694>.
- Richard E. Mayer. *Multimedia Learning*. Cambridge University Press, 2001.
- Andrés Moreno, Niko Myller, Erkki Sutinen, and Mordechai Ben-Ari. Visualizing programs with Jeliot 3. In *Proceedings of the 8th International Working Conference on Advanced Visual Interfaces*, pages 373–376, 2004.
- Thomas L. Naps. JHAVÉ – Addressing the need to support algorithm visualization with tools for active engagement. *IEEE Computer Graphics and Applications*, 25(5):49–55, 2005. ISSN 0272-1716. doi: <http://dx.doi.org/10.1109/MCG.2005.110>.
- Marian Petre. Why looking isn't always seeing: readership skills and graphical programming. *Commun. ACM*, 38(6):33–44, 1995. ISSN 0001-0782. doi: <http://doi.acm.org/10.1145/203241.203251>.
- Robert J. Walker, Gail C. Murphy, Bjorn Freeman-Benson, Darin Wright, Darin Swanson, and Jeremy Isaak. Visualizing dynamic software system information through high-level models. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 271–283, New York, NY, USA, 1998. ACM Press. ISBN 1-58113-005-8. doi: <http://doi.acm.org/10.1145/286936.286966>.

# Visualizations in Preparing for Programming Exercise Sessions

Tuukka Ahoniemi, Essi Lahtinen  
*Tampere University of Technology*  
*P.O.Box 553, FIN-33101 Tampere, Finland*

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

## Abstract

Visualizations are widely researched and used in teaching but the results of their benefits in learning are vague. We introduce an experiment of using visualizations in learning introductory programming. The aim was to support students in their preparation for the exercise sessions by using visualizations. The students' preparation consists of two phases that both are supported: reviewing the subject and a homework assignment. Thus this is also a novel approach to using programming visualizations and integrating them to the course content.

The experiment shows positive results especially among the students with no prior programming experience and the students who consider the programming course challenging. We conclude that integrating the use of visualizations to students' preparation for exercise sessions leads to better learning, more meaningful studying, and ultimately to better preparation. Therefore we also suggest this as a possible way for integrating visualizations to the course.

## 1 Introduction

The learning problems in programming are often connected to more advanced issues than individual concepts, so the learning materials and situations should also be directed to develop more advanced programming skills (Lahtinen et al., 2005). One of the biggest learning problems of the novice programmers is that they have to handle abstract concepts of which they do not have a concrete model in their everyday life (Robins et al., 2003). Thus, providing interactive visualizations as extra material for the students is a good way to concretize the subject in the beginning.

The most common use of visualizations is demonstrating a code example as an illustrative visualization. We wanted to make students participate in the visualization and integrate the use of visualizations to the students' preparation and homework assignments for their weekly exercise sessions. The effects of this approach were tested in a real learning situation by in-class tests.

## 2 Background

The research done on the field of visualizations has resulted in instructions on how to build visualizations so that they will be pedagogically as beneficial as possible. For instance, Naps et al. (2003) recommend that the visualizations should engage the student to participate in the visualization actively. As possible ways to do this it is suggested, e.g., that the visualizations should enable the user to provide his own input for the program and that there should be an interactive prediction in the visualization tool (Rößling and Naps, 2002). To increase the interactivity of the visualizations they can also be built to support all six stages of cognitive development listed in Bloom's taxonomy (Lahtinen and Ahoniemi, 2005).

Despite all these recommendations and ideas on how to improve visualizations, the reports on their usage are diverse. A wide study conducted by Hundhausen et al. (2002) states that it is more important how the visualizations are used than what their content is. Hundhausen (2002) reports that visualizations can actually distract the students' attention away from the subject. On the other hand according to Ben-Bassat Levy et al. (2003) visualizations benefit the students with learning problems. This was also our main interest of research.

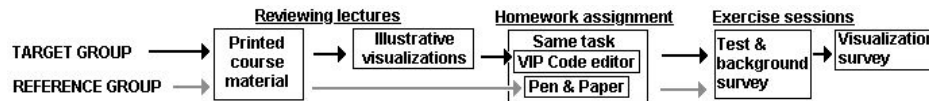


Figure 1: Organization of the experiment.

### 3 The Experiment

This experiment took place on an introductory course for programming (*CS1*) in Tampere University of Technology. The prerequisites for the course are limited to only basic knowledge of computer literacy and it is the first programming course for the students. The programming language used on the course is C++. There are weekly lectures and exercise sessions. The students ought to complete a small homework assignment prior the exercise session. The homework assignment requires them to familiarize themselves with the basics of the new subject. This usually also means reviewing the content of the lectures with the course material.

The idea of visualizations was familiar to the students already before the experiment. We had supported the students' own studying by providing visualizations on the course web page. The printed course material contains web addresses of the visualization examples and the visualization tool – VIP (Virtanen et al., 2005) – was also demonstrated on a lecture.

The experiment took place on the fourth and the fifth week of the course. On the first week of the experiment (the fourth week of the course) the exercise sessions dealt with loop structures and on the second week arrays. These weeks were chosen because both of the subjects are typically difficult for novice students (Lahtinen et al., 2005) and they are easy to visualize.

#### 3.1 The Method

We used two random groups of about 30 students who had enrolled for the exercise sessions. The target group used visualizations when preparing for the exercise session and the reference group did not. The organization of the experiment is illustrated in Figure ??.

##### 3.1.1 Settings before the Exercise Session

Both groups had the printed course material for reviewing before the exercise sessions. Besides the printed course material, the students in the target group were provided an extra web page with instructions on how to review with the visualization examples and links to the examples. On both weeks the reviewing material contained two *illustrative visualizations* (Lahtinen and Ahoniemi, 2005) to clarify the concepts.

The actual homework assignments were exactly the same for both groups. The only difference was that the students worked on them using different tools. The reference group had the assignment available on the course web site. Most students in the reference group had used pen and paper to write the code and the answers to the questions. Some of them had also used a regular code editor and a compiler.

The web page provided for the target group contained the homework assignment as text just like for the other students. In addition, there was a link to a visualization tool where the student could start working on the code. VIP (Virtanen et al., 2005) contains a code editor where the student can write his own solutions, compile them and run them as a visualization.

##### 3.1.2 Settings in the Exercise Session

On the experiment weeks, there was a short written test in the beginning of the exercise sessions to measure the students' learning. The students were not notified about the test in advance. They were not allowed to look at the materials and they returned their answers anonymously. The task was to write really small programs similar to the ones they had

	Target group	Reference group
novices & strugglers	12	17
others	9	10

**Figure 2:** The focused subset (highlighted with grey) was the novices and the strugglers of both groups. The amounts of students are shown in the table.

implemented in their homework assignments. The time was limited to only five minutes because the tasks tested the very basics and therefore would have been easily implemented in the time – assuming the subject was well learnt. We also wanted to have more variation inside the groups by limiting the time. Only the best students would complete the whole test.

Besides the small test, all the students responded to a short survey for background information, e.g., about their previous programming experience and how they felt about their progress on the course. Also the amount of time used, both on reviewing the subject and on doing the actual homework assignment, was asked. The students in the target group also answered another survey concerning the use of the visualizations as a supporting tool for exercise preparation. The survey form was attached to the test so that the background information can be connected to the test answers.

### 3.2 The Homework Assignment

The exercise sessions in the first week dealt with loop structures. In the homework assignment there was a simple example of a `while`-loop. The task was first to find out what the piece of code does and to understand how it works. Then the students had to modify the code to implement an other kind of a functionality. The task reaches the level application (3) of Bloom’s taxonomy of cognitive development, since it requires ability to apply one’s knowledge in a new situation. Thus the assignment version implemented in the visualization tool is a *utilizing visualization* (Lahtinen and Ahoniemi, 2005).

The subject in the second week was arrays. To widen the perspective of visualizational aid we chose this homework assignment differently: The students familiarized themselves with a given complex loop structure handling two arrays and answered questions related to it. The task requires identifying and analyzing the components of the code, so it is on the level analysis (4) of Bloom’s taxonomy. Thus the version implemented in VIP is an *analyzable visualization* (Lahtinen and Ahoniemi, 2005).

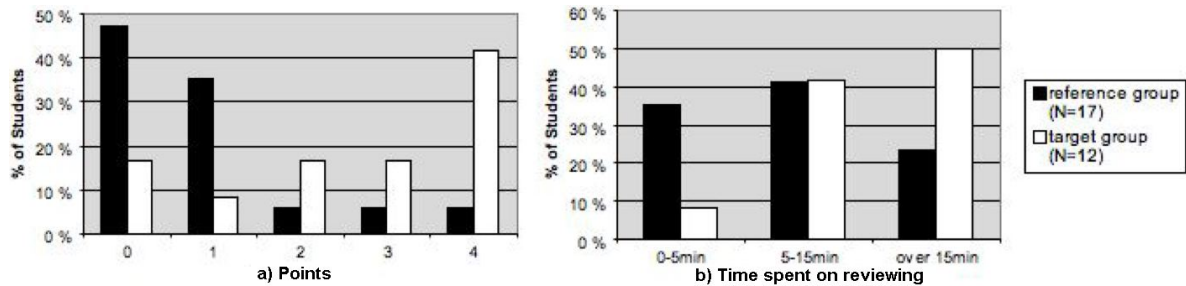
## 4 Results

On the first week, there were 21 students present in the exercise session of the target group and 27 students in the reference group, i.e. altogether 48 students. On the next week the corresponding numbers are 21, 22 and 43.

As visualizations are mainly targeted for the novices and the students who have learning difficulties we constricted the comparison of the groups to only the novices (no previous programming experience) or the ones finding the course subjects so far difficult or very difficult (here called *the strugglers*). The division and the numbers of the students in the groups is illustrated in Figure ??.

The results are divided into two parts: the effects on learning results and the effects on studying behaviour. The first represents the students’ knowledge on the subject measured in the test as the second represents how the students prepared for the exercise session.

According to the results from the first week, the use of visualizations benefits learning: We found a statistically significant difference of the mean values of the test grades between the groups. The results from the second week are analogous and support the results from the first week. Because of the smaller difference in the second week, this section mainly concentrates on representing the results from the first week.



**Figure 3:** Results from the first week of the experiment concerning the novices and strugglers: (a) Distribution of the grades of the first task of the test and (b) Time spent on reviewing.

#### 4.1 The Effects on Learning Results

The effects on learning results were analyzed by rating the students' answers for the test. For example, on the first week all three tasks were graded on a linear scale with points from 0 to 4 resulting the maximum of 12 points. On the second week the maximum was only 8.

The loop tasks seemed to be difficult for the students to complete in the given five minutes. The mean result was altogether only 3.5 out of 12 points (standard deviation 2.5). An independent samples T-test was used to analyze the difference between the groups. The means for the focused subset of novices and strugglers are 3.6 points (standard deviation 2.2) for the target group and only 1.7 points (standard deviation 1.5) for the reference group. This shows a significant statistical difference (over 95%). Even if the comparison is done to the whole groups (instead of only the focused subset) there is a small analogous difference between the groups.

In the next week, the corresponding means of the novices and strugglers are 3.1 points out of 8 points (standard deviation 2.3) for the students in the target group and 2.3 points (standard deviation 1.9) for the ones in the reference group. The trend is same as on the earlier week.

As the students carried out the tasks in the test sequentially, they all started with the first task. Figure ??a shows the percentage values of each grade in this task. Only the novices and the strugglers are taken into account. Almost all students in the target group (10 out of 12 = 83%) got at least one point and even 42% full 4 points as the reference group had the same numbers in 53% and 6%. The same phenomenon can be observed in the results of the second week.

#### 4.2 The Effects on Studying Behaviour

Since the novices and the strugglers were the only ones whose learning results are different, it is logical that they are the only ones' whose studying behaviour was influenced by the visualizations. Thus this subsection concentrates only on the novices and strugglers of the groups.

According to the students' answers to the survey about their preparation, the students in the target group had used more time than the students in the reference group. Both the time spent on reviewing the subject and the time spent on doing the homework exercise were higher. The difference was bigger in reviewing the subject. The comparison between the time usage on reviewing the subject in the first week of the experiment is shown in Figure ??b.

More than a third of the students in the reference group spent less than 5 minutes in reviewing. More than 90% of the students in the target group spent longer than 5 minutes. It is clear that the students using the visualization tool concentrated longer even though the statistical significance between the groups can not be stated.

Also the feedback of the survey about visualizations as a preparation tool resulted in plain positive feedback. Students wrote comments like "Though having read the specified course

material, I really understood the subject after using the visualization examples.”

### 4.3 Comparing the Results of the Two Weeks

The experiment was not done in a strictly controlled situation but in a normal teaching group so some circumstances varied between the two weeks of the experiment. E.g., there were more absent students on the second week. The subjects on the two weeks were different so we also had a new type of homework assignment and a different test on the second week. All of these factors have influenced the results.

On the first week, the homework assignment was a *utilizing visualization* and on the second week an *analyzable visualization*. One important reason for the difference in the results can be that *utilizing visualizations* engage the student to produce his own code where as *analyzable visualizations* engage the student to observe the code intensively. The test performed in the class room was about producing their own code. So on the first week the preparation and the test were more similar than on the second week.

The in-class test was not announced in advance so on the first week of the experiment no one expected it. On the second week the students might have assumed that there could be a test again. Thus the students may have prepared better for the exercise session. This can also be one of the reasons why the statistical difference was not achieved on the second week.

## 5 Discussion

Even if the circumstances between the weeks of the experiment varied, it is advantageous that the experiment was done in a real learning situation. We captured the students’ experiences in a situation where they act as they would act normally when studying. Thus the results can better be applied to planning teaching in the future.

The results show that the use of visualizations helped the students who have most challenges in learning programming (the novices and the strugglers). They learnt more if they used visualizations when preparing for the exercise sessions. The students who had earlier experience in programming already had a mental model about the subject and thus the use of visualizations was not so helpful. Also the students who felt that the subject was easy could form the mental model without using visual materials. Hence, they did not benefit of the use of visualizations so much either.

Another result was that the students who used visualization examples along with the normal course material spent more time on reviewing the subject than the others. Studying obviously became more interesting as a new visual perspective was provided.

So what really can be concluded from the results is that visualizations do aid learning, but it is not sure whether this results directly of their usage. It can also result from the fact that when using visualizations, the studying itself is more interesting and the students use more time on it and thus learn better. However, it is not important, if the visualizations improve the learning results directly. The most important result is that they do improve them.

The difference between the two weeks of the experiment – the week when the students did a *utilizing visualization* exercise and the week when they did an *analyzable visualization* exercise – also supports the recommendation from Naps et al. (2003) that the visualization should engage the student to work actively. *Utilizing visualization* makes the student produce their own code where as *analyzable visualization* only makes them analyze code written by someone else. The engagement to the visualization is more intense with a *utilizing visualization*. Also the learning results from the week when the *utilizing visualization* was used are better.

Using visualizations in students’ preparation for exercise sessions had definitely a positive outcome because of the better learning. The exercise sessions ran smoother because students were better prepared due to the increase in their motivation. This also shows that using visualizations in preparing for exercise sessions is a working way of integrating visualizations to the rest of the course content.



The problems and considerations of this kind of approach are technical issues and the time spent by the teacher. Implementing tasks with visualizations requires quite advanced tools that have to be available for every student. Also preparing the tasks with a visualization tool takes more effort from the teacher than without a visualization tool.

When planning new ways to use visualizations in a course the teacher should also bear in mind that not all want to use new kinds of learning tools. As the use of visualizations mainly benefit the novices and the strugglers, it can be annoying for the students that do not need it. Some of the students might not like visual learning style or just have their own idea on how to work. Thus we recommend that the use of visualization tools is optional.

## 6 Conclusions

We cannot say whether the better learning results originate from the pedagogical impact of the visualizations or from the fact that the visualizations made the students study for a longer time. Either way, using visualizations improved the students' learning and preparation for the exercise sessions which was the purpose. Therefore, we recommend both using visualizations in teaching and using the exercise sessions to integrate the visualizations to the other parts of the course.

## References

- Ronit Ben-Bassat Levy, Mordechai Ben-Ari, and Pekka A. Uronen. The Jeliot 2000 program animation system. *Computers & Education*, 40(1):1–15, 2003.
- Christopher 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.
- Christopher D. Hundhausen, Sarah A. Douglas, and John T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages & Computing*, 13(3):259–290, 2002.
- Essi Lahtinen and Tuukka Ahoniemi. Visualizations to Support Programming on Different Levels of Cognitive Development. *Proceedings of The Fifth Koli Calling Conference on Computer Science Education*, pages 87–94, November 2005.
- Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. A study of the difficulties of novice programmers. *ITiCSE 2005, Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, pages 14–18, June 2005.
- 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.
- A. Robins, J. Rountree, and N. Rountree. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2):137–172, 2003.
- Guido Rößling and Thomas L. Naps. A Testbed for Pedagogical Requirements in Algorithm Visualizations. *ITiCSE 2002, Proceedings of the 7th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, June 2002.
- Antti T. Virtanen, Essi Lahtinen, and Hannu-Matti Järvinen. VIP, a visual interpreter for learning introductory programming with C++. *Proceedings of The Fifth Koli Calling Conference on Computer Science Education*, pages 125–130, November 2005.

# Visualization of Spatial Data Structures on Different Levels of Abstraction

Jussi Nikander<sup>†</sup>, Ari Korhonen<sup>†</sup>, Eiri Valanto<sup>‡</sup> and Kirsi Virrantaus<sup>‡</sup>

*Helsinki University of Technology*

*Department of Computer Science and Engineering<sup>†</sup> / Department of Surveying<sup>‡</sup>  
Finland*

`{jtn, archie}@cs.hut.fi, {Eiri.Valanto, Kirsi.Virrantaus}@tkk.fi`

## Abstract

Spatial data structures are used to manipulate location data. The visualization of such structures faces many challenges that are not relevant in the visualization of one-dimensional data. The visualized data can be represented using several different types of visual metaphors. These metaphors can be divided into several different levels of abstraction depending on the purpose of the visualization. This paper proposes a division of data structure visualization into four levels of abstraction, and shows how these abstractions can be taken into account in the visualization of spatial data structures.

## 1 Introduction

Spatial data structures are structures that manipulate spatial data. Spatial is a term, which is used to refer to located data, for objects positioned in any space (Laurini and Thompson, 1992). Spatial data is used in many areas of computer science, like Geographic Information Systems (GIS), robotics, computer graphics, virtual reality, as well as in other disciplines like finite element analysis, solid modeling, computer-aided design and manufacturing, biology, statistics, VLSI design, and many others. Algorithms that manipulate spatial structures are called Spatial Data Algorithms (SDA).

Spatial Data Structures are based on regular non-spatial data structures like arrays, lists and tree structures, as well as the algorithms that manipulate these fundamental structures. However, the complexity of spatial data structures comes from the multidimensionality of spatial data. Two- and three dimensional variations of efficient algorithms make algorithm development more challenging. In the GIS literature there is a core set of fundamental as well as advanced spatial algorithms and data structures. However, no comprehensive textbook exists on the topic.

Spatial data and algorithms for manipulating such data are an integral part of geoinformatics, a branch of science where information technology is applied to cartography and geosciences. Geoinformatics is closely related to cartography, and therefore illustrations, such as maps and other diagrams, are often used. For people in this field of study, graphics are a familiar and natural way of illustrating the work.

Software visualization (SV) is a branch of software engineering that aims to use graphics and animation to illustrate the different aspects of software (Stasko et al., 1998). It is typically divided into two subdivisions: program visualization and algorithm visualization. In both subdivisions it is important to be able to differentiate between the various *levels of abstraction* in the visualization. For example, a linked list implemented using two arrays can be visualized by showing the arrays and their contents. Such a visualization shows the implementation-level details of the list, but makes it hard to grasp its logical structure, which can easily be seen when the list is visualized as nodes connected by references. In this paper we will concentrate on the algorithm visualization (AV) side of SV.

One of the main uses for SV is in pedagogy, and numerous SV systems have been developed for teaching. For examples, see the work of Hundhausen and Douglas (2000), Rößling et al. (2000) or Malmi et al. (2004). In pedagogy, however, visualization does not have any intrinsic value. As noted by Hundhausen et al. (2002), the learners must be actively involved in activities where algorithm visualization is used in order to get better learning results.

The graphic nature of SDA and its applications makes SV a natural tool for teaching the topic. The data handled by spatial algorithms is multidimensional, and therefore the natural visualization for the solution and the process is a map or a diagram. Without using graphics, it is almost impossible to explain spatial algorithms or data structures. Thus, we are currently extending the TRAKLA2 visualization system to include SDA. We are focusing on the structures and algorithms required in geoinformatics, a branch of science where information technology is applied to cartography and geosciences. We are unaware of any other work that applies SV techniques to spatial data structures.

In this paper, we will propose a division of algorithm visualization into different levels of abstraction and describe how these levels can be mapped into the visualization of data structures in general. We will extend the category fidelity and completeness from the taxonomy of Price et al. (1993) in order to give a better insight into the various needs of pedagogical systems that “take liberties to provide simpler, easier-to-understand visual explanations”. The idea is to name four different levels of “visual metaphors that present the behavior of the underlying virtual machine”. We will also show how these levels can be taken into account when utilizing software visualization to teach SDA for geoinformatics students.

The rest of the paper is organized as follows. In section ?? we will introduce our model of different levels of abstraction in data structures. In section ?? we will discuss how this model can be applied to the visualization of SDA. Finally, Section ?? discusses how the different levels of visualizations can be used in teaching SDA.

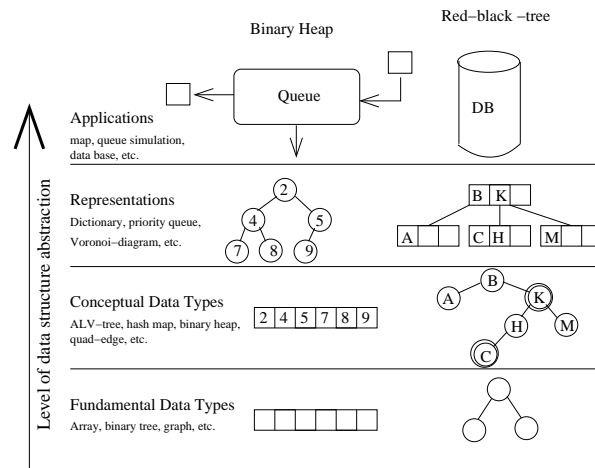
## 2 Model for Data Structure Abstractions

A data structure (DS) can be defined as a collection of variables, possibly of several different data types, connected in various ways. An abstract data type (ADT) is a set of (abstract) items with a collection of operations defined on them. Separate definitions for the logical (ADT) world and for the physical (DS) world are essential for many reasons. First, distinguishes the design from the implementation. The definition of an ADT does not specify how the data type is implemented and implementation level details are hidden from the end user of the ADT. This hiding of the implementation details is known as encapsulation. Second, the concept of an ADT is an important principle used for managing complexity through abstraction. The irrelevant implementation details can be ignored in a safe way. This is also the way humans deal with complexity. We use metaphors to assign a label to an assembly of concepts and then manipulate the label in place of the assembly. Third, reusability is one of the key principles in software engineering that can be promoted by designing general purpose data structures suitable for many tasks, i.e., by implementing data types suited for several algorithms.

In software visualization, we are interested in illustrating the organization of the related data items whether they are abstract or not. The same data items can be *represented* in several levels of abstraction depending on the purpose of the visualization. However, this need for granularity is not only biased to these two extremes (DS and ADT). For example, a binary heap is a priority queue ADT that is implemented as an array. Between these two levels of abstractions, it has a well known representation in which a binary heap can be visualized as a binary tree.

In the following, we will fine tune the concept of data type abstractions in order to create a model that gives a deeper insight into the world of data structure implementations and the conventional visual notations used in text books. This is done by introducing two new levels of abstraction that we call *fundamental data types* (FDT) and *conceptual data types* (CDT).

Let us start with an example illustrated in Figure ??, which shows data structures visualized on four different levels of abstraction. The fundamental data type (FDT) used to implement a binary heap is an array. This is the lowest level of abstraction we will consider (FDT level). The FDT level structures can be utilized to construct conceptual data types



**Figure 1:** Levels of abstraction in data structure visualization

(CDTs), i.e., an array can be used to implement a binary heap by imposing a set of semantics to it: the child nodes of the node  $i$  are in positions  $2i$  and  $2i + 1$ , and for each node the heap property “the parent is smaller than or equal to the children” holds. The heap property is easier to see, however, if the data structure is shown using the binary tree representation instead of the array representation (representation level). Finally, a binary heap can be utilized, for example, in a network router simulation (application level). Different packets can have different priorities, and the priority queue could be abstracted into a black box where the internal structure is not shown at all. The packets go in from the right, and forwarded packets come out from the left. If packets are dropped, they are shown below the visualization.

In general, a fundamental data type (FDT) is a generic data type (skeleton) that is used to implement a particular data structure. Examples of FDTs include arrays, linked lists, binary trees, trees, and graphs. These are kind of archetypes for creating structures, and all the data structures used in text books can be implemented in terms of these archetypes. For example, an adjacency list is a composition of an array and a number of linked lists.

From the visualization point of view, *FDT level* is the lowest level of abstraction that we are interested in presenting. Still, it is an abstraction. For example, an array is a commonly used term in computer science to denote a contiguous block of memory locations, where each memory location stores one fixed length, and fixed-type variable. However, an array can also refer to the FDT composed of a (homogeneous) collection of variables, each variable identified by a particular index number. Most programming languages do not directly support this latter form of definition due to its abstract nature. Thus, there are possibly several different *implementations* for arrays. However, the low-level abstraction behind each of them is common for all of the implementations, and can be represented in a uniform way<sup>2</sup>.

As we can see, FDTs do not have semantics. The definition of an array does not dictate the data types nor the order of the items within. In software visualization, however, we need the actual data and possibly some other restrictions in order to give a meaning for the illustration. The idea is that a single FDT can be reused many times to represent several conceptual data structures all having different semantics. For example, an array can be the basic building block for both hash table and binary heap. These basic building blocks allow us to have a single representation for all arrays independent of the data they contain. A data structure defined in terms of FDTs is called a conceptual data type (CDT).

In *CDT level* we have data structures that are implementations for *conceptual models* that encapsulate particular ADTs. This conceptual model defines the structures (FDTs) used as well as the type constraints that lay the foundation for the implementation. The

<sup>2</sup>Of course, there are several ways to illustrate arrays, but the canonical visual notations commonly used are all equal in such a way that a person can easily grasp the idea of an array from each of them.

corresponding data structure is an implementation that defines also the operations needed to change the structure. For example, a red-black-tree is a CDT that has the structure of binary tree, and implements an ADT called dictionary (operations search, insert, and delete). The conceptual model is defined as follows: A red-black tree is a binary search tree in which 1) each node is either red or black; 2) the root is black; 3) children of red nodes are black; and 4) each path from root to a leaf contains equal number of black nodes. An example tree is illustrated in Figure ?? (red nodes are shown with double circles). A typical implementation for this data structure makes rotations and color changes in order to establish the operations defined in the dictionary ADT in logarithmic time. However, while visualizing the data structure, we are most interested in the layout and the question whether it satisfies the constraints above. The concept of rotation, for example, can be expressed in terms of changes in this layout (i.e., moving nodes and arcs between nodes). Of course, the actual code can be illustrated as well, for example, as a pseudo-code.

A single data structure can often be represented in several ways. Thus, we need another abstract visualization level called *Representation level*. In representation level, the data structure is supposed to be illustrated through a canonical view in such a way that it is the most appealing for human to grasp. It is also possible to use a number of representations simultaneously in order to make it easy to understand idea of the data structure.

For example, a red-black tree can be used to implement a B-tree (Guibas and Sedgewick, 1978), and therefore it can be visualized as such as well. In Figure ??, a 2-3-tree representation is used for the red-black tree in CDT level. Similar to this, there were the two choices for binary heaps: the array and binary tree representations.

The top level of abstraction is called *application level*. In this level, we illustrate the data in domain specific ways and typically omit most or all of the data structure's details. For example, B-trees are often used as database indexes or storage structures as illustrated by the database symbol in Figure ?. Therefore, on the application level, the data structure visualization is analogous to the definition of an ADT. The interface is defined, but implementation is hidden.

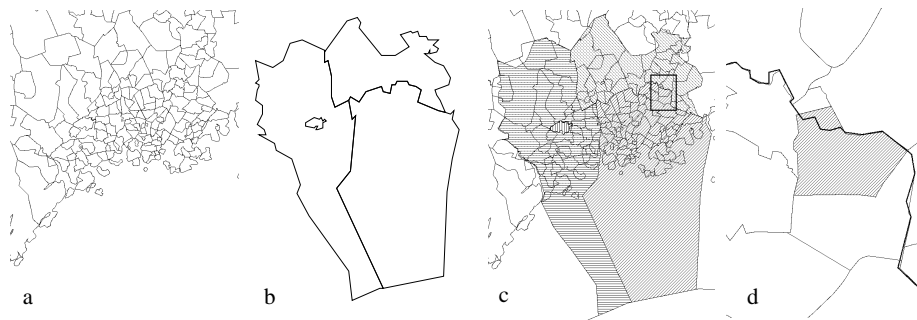
Software visualization techniques and tools can be used in many other disciplines as well. However, there are a few abstractions that have proved to be particularly important for many disciplines. The idea behind the categorization of these abstractions is that we could utilize the same components and concepts over and over again and take advantage of the prior work done. It should be noted, however, that the levels of abstraction are not always distinct.

Some application areas are intrinsically visual. For example, in GIS applications, the typical representation is a map. However, the underlying data structures and their visual counterparts are basically the same as, for example, in software visualization. Thus, we need to make a distinction between the application level visualization and the illustrations for the underlying data structures and algorithms. However, no matter what kind of application we have in mind, it can be reduced to those basic building blocks through the levels of abstraction, as we will see in the next section.

### 3 Visualization of Spatial Data Algorithms

Spatial data algorithms process spatial data which is naturally visualized using different maps. In this section, we show how the model from previous section is applied to spatial data algorithms. Specifically, we present a common geoinformatics problem, called map overlay, as an example and describe how it can be visualized using different abstraction levels.

On the application level, we typically illustrate the application data in some form. In GIS applications both source and result data sets can typically be visualized as maps of some type. Thus, application level visualization can give good overview of the problem, particularly as specialists in geoscience and cartographic fields are used to handle and understand maps. However, while visualizations in GIS applications give a good illustration of the data, they



**Figure 2:** Map overlay problem a) postal code area, b) municipality boundary source maps c) map overlay result, d) detail where municipality boundary divides a postal code area into two polygons.

omit all details of the data structures used.

The problem of combining two different maps to produce a new map with information from both maps' domain is called map overlay. The interest often lies in discovering areas where some properties coincide. For example squirrel habitat areas on a map layer could be combined to a vegetation layer to get a map presenting areas where squirrels are found in coniferous forests. Map overlay is one of the fundamental operations on geographic information systems and we use it as an example throughout this section. The description of map overlay as a combination of two map layers belongs to the highest abstraction level in our conceptual framework (Figure ??).

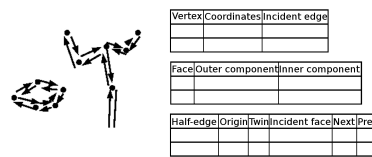
We have two maps presenting four urban municipalities in Helsinki metropolitan area and postal code areas in the same region, respectively. Our problem is to find which municipality each postal code area belongs to and to locate areas where postal code areas cross municipality boundaries. Map overlay can provide a solution to these questions. Figure ?? illustrates the source maps as well as the resulting overlay.

In order to solve a problem with a computer, the problem has to be represented formally in some way. Typically, mathematical presentations are used. Such a presentation also gives a canonical view of the data that is independent of any computerized data structure implementation. However, such a views are typically easy for a human to grasp and accurately show the connections between different data elements. Therefore, such presentations belong to the representation level. In geoinformatics, examples of mathematical presentations are Voronoi diagrams and their dual the Triangular Irregular Network (TIN) (Okabe et al., 2000).

In GIS, data is represented using two models: either vector or raster model. In vector representation, data is presented as points, lines or polygon objects, which have spatial location data associated to them. Polygons can be used to construct polygon maps, lines can be used in graphs, and point sets can be structured into TIN-models or Voronoi-diagrams. The advantage of TINs and Voronoi diagrams is that discrete measured points make a structure that supports, for example, interpolation and visualization. A raster model utilizes a regular grid that divides the represented area into equal sized cells. Each cell contains an associated value that represents the values of the respective area.

The mp overlay problem can be solved by applying the vector or raster model. In our example, we use map overlay operation in vector format. On the representation level, the areas to be overlayed can be represented as polygons. Thus, we have two polygon maps which are overlayed by intersecting the polygon boundaries. The map overlay first uses some method, for example the line-sweep algorithm, to locate intersecting lines. The intersection points are then used to create new polygons, which form the resulting map. Figure ?? d shows a detail on Helsinki boundary where the map overlay divides a postal code are into two polygons. The rectangle in Figure ?? c highlights the location of the detail.

To achieve an implementable and efficient solution on a computer, a suitable data structure



**Figure 3:** DCEL structure presentation of municipality boundaries

is needed for storing the mathematical model. Typically there are several possible data structures that can be used to store each model. Since these structures implement a mathematical model they have a well-established formal set of semantics, and therefore are conceptual data types. For map overlay, one possible CDT level structure is Doubly Connected Edge List (DCEL) (de Berg et al., 2000), which can be used to store the polygons.

The DCEL consists of vertices, polygon faces and half-edges that connect vertices. Each vertex has a location and connection to a edge that originates at the vertex. Every polygon face has information of a half-edge on its outer boundary and a list containing each hole in the face. Each edge is divided to two half-edges. A half-edge has information of its origin vertex and twin. Destination is not needed because it is the same vertex as twin's origin. In addition, some next and previous half-edge is connected to each half-edge as well as the incident polygon face. Figure ?? shows the simplified municipality boundaries as DCELs.

Using DCEL map overlay is solved by first combining both map layer's DCEL structures into an invalid DCEL. This DCEL is then examined and modified to form a valid structure.

Finally, the conceptual data types used to implement the mathematical models require FDT level structures in the implementation. The DCEL, for example, contains three types objects: vertices, faces and half-edges. Therefore, DCEL implementation requires at least three FDTs for storing the data. A natural way to build a DCEL is by three arrays containing the corresponding information. On the lowest abstraction level, basic canonical representations used to visualize spatial data structures are identical to canonical views of any other data structure.

## 4 Discussion

In this paper, we have described how software visualization can be applied to spatial data structures and algorithms. Our example, map overlay, was from the field of geoinformatics which, as a discipline, is very visual in nature. Most geoinformatics problems involve visualization in the problem description, solution, and the solution process. For the people in the field, visualizations are a natural and extremely common way for processing and representing data. Therefore visualizations are also a very good tool for teaching aspects of geoinformatics, including SDA. Currently, we are extending the TRAKLA2 algorithm visualization environment to include SDA visualizations. Our goal is to create a number of algorithm simulation exercises for use in teaching geoinformatics. The work is currently in implementation phase.

Since spatial data structures contain two-dimensional data, traditional algorithm visualization techniques can be sub-par. Traditionally, the focus of the visualization has been to show the composition of the data structures. A graph visualization, for example, shows the vertices and edges in a visually appealing fashion, and a tree visualization is optimized to show the relations between tree nodes. When the data stored in the structure is one-dimensional (like numbers or strings), this also shows the relation between the data elements.

When the data is two-dimensional, however, the relations between different parts of the structure do not typically give a comprehensive view of the relations between data items. DCEL, for example, is typically used to represent vector maps in geoinformatics. Depending

on the level of abstraction used and the type of visualization selected, the information one can gain from a graphical view of DCEL varies a lot. When viewed as a arrays, the visualization is good for understanding the minute details of the implementation, but does not give a very good view of the data it represents. If viewed as a graph such as in Figure ??, the visualization gives a good view of the data being represented, but omits most of the implementation-level details.

In order to effectively visualize some SDAs, we clearly need several visualizations simultaneously. One visualization can, for example, be on the CDT level, showing the inner composition of the data structure in question, and the other on Representation level, giving an easily understood view of the data being represented. This effectively expands the *multiple views* category of (Price et al., 1993), which measures the degree of multiple synchronized views the visualization can provide. The Price's taxonomy discusses views of different granularity. Our work, on the other hand, uses views on different levels of abstraction.

In this paper, we have showed that views on different levels of abstraction can be a of great help, when visualizing spatial data structures or algorithms. Furthermore, such abstractions are likely to be of use in general algorithm visualization. Therefore, in our view, future algorithm visualization systems should contain support for showing different abstraction levels.

## References

- Mark de Berg, Marc Van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*, chapter 2.2, pages 29–33. Springer, 2000.
- Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21. IEEE Computer Society, 1978.
- Christopher Hundhausen and Sarah A. Douglas. SALSA and ALVIS: A language and system for constructing and presenting low fidelity algorithm visualizations. In *Visual Languages*, pages 67–68, 2000. URL <http://citeseer.ist.psu.edu/hundhausen00salsa.html>.
- Christopher D. Hundhausen, Sarah A. Douglas, and John T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages & Computing*, 13(3):259–290, June 2002.
- Robert Laurini and Derek Thompson. *Fundamentals of spatial information systems*. Academic Press, 1992.
- 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.
- Atsuyuki Okabe, Barry Boots, Kokichi Sugihara, and Sung Nok Chiu. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. John Wiley & Sons, 2000.
- Blaine A. Price, Ronald M. Baecker, and Ian S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.
- Guido Rößling, Markus Schöler, and Bernd Freisleben. The ANIMAL algorithm animation tool. In *Proceedings of the 5th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education, ITiCSE'00*, pages 37–40, Helsinki, Finland, 2000. ACM Press, New York.
- John T. Stasko, John B. Domingue, Marc H. Brown, and Blaine A. Price. *Software Visualization: Programming as a Multimedia Experience*. MIT Press, Cambridge, MA, 1998. ISBN 0-262-19395-7.



# A General Framework for Overlay Visualization

F. Tihomir Piskuliyski, Amruth N. Kumar  
*505 Ramapo Valley Road, Mahwah, NJ 07430, USA*

tpiskuli@ramapo.edu, amruth@ramapo.edu

## 1 Abstract

If visualization is more effective when accompanied by narration, why not superimpose visualization on narration? This might result in better transfer of learning. We will present a general framework for such superimposed visualization, called overlay visualization. The objectives for the design of our framework are 1) to separate the application from the visualization; and 2) to separate the specification from the rendering. We will describe a few applications of overlay visualization for programming and provide examples from our implementation of overlay visualization for software tutors called problets. The advantages of overlay visualization include: less cognitive load on the learner, and automatic support for both path and state visualization.

## 2 Introduction

Researchers have found that on problem-solving transfer tasks (Mayer and Anderson, 1991) animation with narration outperforms animation only, narration only, or narration before animation. Similarly, on recall tasks, narration with visual presentation outperforms narration before visual presentation (Baggett, 1984). These results support a dual-coding hypothesis (Paivio, 1990) that suggests two types of connections among stimuli and representations: representational connections between stimuli and the corresponding representations (verbal and visual), and referential connections between verbal and visual representations.

What if visual presentation is superimposed on narration? In programming problems, what if visual presentation is superimposed on the program code? We conjecture that this will promote better referential connections between visual and verbal representations and result in better transfer from visual representation to the concepts being learned. To support such visualization, we have developed a framework of what we will henceforth refer to as overlay visualization.

Overlay Visualization is the superimposition of graphics on the material to be visualized. In the context of programming, it is superimposing graphics on code. We will first describe a general framework for overlay visualization in section ???. Next, in section ??, we will discuss some applications of overlay visualization for program visualization. We will illustrate with examples from our implementation of overlay visualization for programming tutors called problets ([www.problets.org](http://www.problets.org)) (Kumar, 2006a). In section ??, we will discuss the advantages of overlay visualization and compare it with prior work.

## 3 A General Framework for Overlay Visualization

**Objectives:** We had two objectives for our overlay visualization framework: 1) to separate the application from the visualization; and 2) to separate the specification from the rendering.

While separating the application from the visualization, we wanted to ensure two objectives: 1) maximize the flexibility of visualization; and 2) minimize the overhead of specifying such visualization. Separating the specification from the rendering and using a declarative representation for the specification increases the flexibility of the framework. The specification can be coded by hand, generated automatically by a program, or specified by the learner using mouse gestures. In all these cases, the visualizer that translates the specification into visualization would be the same.

We identified two layers of separation between the application and the visualization: a layer of visual primitives and a layer of graphical primitives.

**Graphical Primitives:** The graphical primitives that we have implemented so far are: 1) Box - draws/animates a box, 2) Arrow - draws/animates an arrow, 3) Ellipse - draws/animates an ellipse, 4) Text - draws/animates a string of text.

**Visual Primitives:** The visual primitives that we have identified and implemented so far are: 1) Point at certain words or lines of code using arrows, 2) Highlight lines of feedback and/or code using boxes, 3) Circle lines of feedback and/or code using ellipses, 4) Connect two circled segments of code with an arrow, and 5) Animate a given segment of text.

**Visual Specification:** A visual specification consists of the type of the visual primitive, the line in the code for which the visual primitive needs to be created, whether the visual primitive should be drawn or animated, the order in which the visual primitives must be drawn/animated, and the color in which the visual primitive must be rendered. Since the visual specification is declarative in nature, it can be generated in several ways, as mentioned earlier:

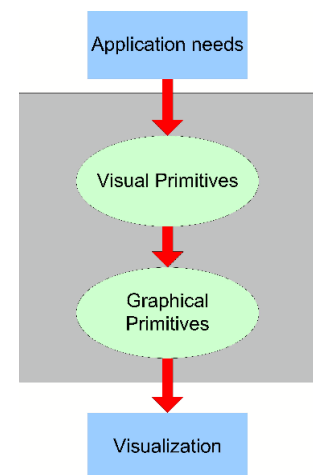
- **Automatically generated by the application:** The application that uses the overlay visualization can automatically determine the lines of code that must be visualized, the visual primitives with which they should be visualized, and the order in which they should be visualized, and generate the visual specifications accordingly.
- **Specified by the learner:** The learner can use mouse gestures to specify the visual primitives that should be used, the lines of code for which the primitives should be used, and the order in which they should be rendered.

**Visualizer:** The visualizer (shown as a dark box in Figure ??) gets a declarative list of visual specifications as input, e.g., a list of specifications to illustrate the execution of a `for` loop, `while` loop, or `switch` statement. The Visualizer creates the visual primitives corresponding to the visual specifications. The visual primitives in turn create the graphical primitives needed to render them. During rendering, the Visualizer sequences the rendering of the visual primitives, which in turn delegate the rendering to their corresponding graphical primitives.

## 4 Applications of Overlay Visualization

Overlay visualization can be used for various purposes in program visualization: to visualize the control flow in a program, visualize the data flow in a program, cluster the code by functionality, set off missing stages in the lifetime of a variable, etc. In each case, the visualization serves to focus the attention of the learner on the segments of code that are of immediate interest. We will present examples of control flow visualization, data flow visualization and error highlighting from our implementation of overlay visualization for programming tutors called problets.

**Control flow Visualization** clarifies the order in which the lines of code in a program are executed. Control flow visualization is especially helpful when the program involves selection statements, repetition statements and function calls. An example of control flow visualization



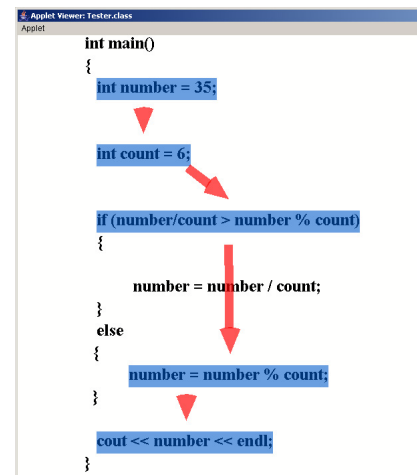
**Figure 1:** The architecture of the overlay visualization system

generated by the overlay visualizer is shown in Figure ???. The specification provided to the visualizer was:

1. CONNECT line 2 to line 5
2. CONNECT line 5 to line 8
3. CONNECT line 8 to line 15
4. CONNECT line 15 to line 18

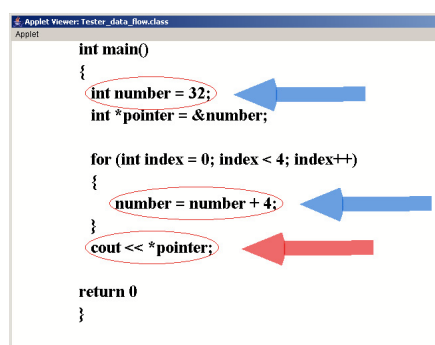
We plan to have problets automatically generate this specification as they execute the program, by keeping track of the line numbers of the lines of code that are executed.

**Setting off an error:** When a program object does not go through the correct sequence of state transitions, it may end up in an error state, e.g., a variable is not assigned before it is referenced, a pointer is not allocated before it is de-referenced, or a loop is not updated in its body. Overlay visualization can be used to highlight or set off the missing state transition and clarify the origin of the error.



**Figure 2:** An Example of Control Flow Visualization

**Data flow visualization** clarifies the sequence of transformations applied to one or more variables in a program. Data flow visualization is especially helpful when operations are applied to a variable in non-contiguous sections of a program. It complements data space visualization provided in systems such as Jeliot 3 (Ben-Ari et al., Springer-Verlag, 2002) and problets (Kumar and Kasabov, 2006) - whereas data space visualization provides a snapshot of the latest values of all the variables in a program, data flow visualization displays the sequence of operations performed on one or more variables that can explain their latest value. An example of data flow visualization generated by the overlay visualizer is shown in Figure ???. The specification provided to the visualizer was:



**Figure 3:** An Example of Data-Flow Visualization

1. CIRCLE line 2 in red
2. POINT to line 2 in blue
3. CIRCLE line 7 in red
4. POINT to line 7 in blue
5. CIRCLE line 9 in red
6. POINT to line 9 in red

Once again, we plan to have problets automatically generate this specification while executing the program, by keeping track of the lines of code in which a particular

**Clustering code:** When analyzing the behavior of a program, clustering together logically related lines of code greatly helps comprehension, e.g., drawing boxes around each loop in a program with nested loops (e.g., see (Kumar, 2006a)). Overlay visualization can be used for this purpose. Whereas code and data flow visualization map time onto space, i.e., map behavior of the program at discrete events of time on to the text of the code, clustering maps space onto space, i.e., maps discrete lines of code into related clusters.

## 5 Discussion

There are several advantages to using overlay visualization:

- **Less Cognitive load:** There is less cognitive load if both the code and the visualization are displayed on the same panel. The student doesn't have to alternate between two separate panels, and mentally put together the information from the two panels. His/her attention will be focused on both the code and the visualization at the same time.
- **Execution History:** As compared to code visualization, where only the currently executing line of code is highlighted (e.g., (Loboda et al., 2006)), overlay visualization automatically supports the display of execution history. Including execution history is one of the ways to improve the effectiveness of visualization (Naps et al., 2003). In other words, overlay visualization supports path visualization (execution history), just as easily as state visualization (a current snapshot of the program being executed).
- **Active learning:** One of the recommendations for improving the effectiveness of visualization is to let the learner construct his/her own visualization (Naps et al., 2003), (Hundhausen et al., 2002). This will promote visual as well as active learning. With overlay visualization, students can construct their own visualization using mouse gestures. For example students can specify Connect and Point by dragging the mouse, highlight by double-clicking the mouse, etc. The user interface can translate mouse gestures into visualization specifications, which can then be compared with the correct specifications to provide feedback to the learner.

Overlay visualization is meant to be used as a supplement to, and not a substitute for the traditional types of visualization used for program analysis, such as data space visualization (snapshot of all the variables and their values), data flow visualization (highlighting the flow of data from one variable/object to another), code visualization (highlighting the line of code that is currently being executed), control space visualization (flowchart of the program, call graph, UML diagram, etc.), and control flow visualization (highlighting the path of execution of a program) provided in visual debuggers (e.g., Retrovue <http://www.retroview.com/>, whyline (Ko and Myers, 2004)), program visualizers (e.g., Jeliot 3 (Moreno et al., 2004)) and concept visualizers (e.g., (Bowdidge and Griswold, 1998) Malloy and Power (2005)).

Overlay visualization is proposed as a tool, not as a specific type of program visualization - as a matter of fact, it can be used for many of the traditional types of program visualization mentioned above. We believe that it is especially effective for data flow and control flow visualization since it superimposes the visualization on the program code, thereby reducing the student's cognitive load. Overlay visualization is not even restricted to program visualization - it can be used to connect the sequence of arguments/stream of thought in text, highlight clues in a puzzle, highlight the structure of a web page, etc. Meta-data can be used to automatically generate such overlay visualization.

Representation of the program text is the most basic form of program comprehension (Pennington, 1987). Since overlay visualization is superimposed on program text, it is especially suitable for novice programmers. Traditionally, program visualization systems visualize the program for the student. With overlay visualization, it is also easy to have the student specify the visualization using two of the most basic facilities: the program text, which is the most basic form of representation of the program (as compared to data space, flowchart, class graph, UML diagram, etc. which require a deeper understanding on the part of the student), and mouse gestures, which are the most primitive form of user interaction. So, overlay visualization is especially amenable to active learning, and for use by novice programmers.

We plan to implement a user interface which will translate mouse gestures into visualization specifications. We plan to evaluate the effectiveness of overlay visualization in problems in fall 2006.

## 6 Acknowledgments

Partial support for this work was provided by the National Science Foundations Educational Innovation Program under grant CNS-0426021.

## References

- P. Baggett. Role of temporal overlap of visual and auditory material in forming dual media associations. *Journal of Educational Psychology*, 76:408–417, 1984.
- M. Ben-Ari, N. Myller, E. Sutinen, and J. Tarhio. Perspectives on program animation with jeliot: In diehl, s. (ed.). *Software Visualization. LNCS*, 2269:31–45, Springer-Verlag, 2002.
- R. W. Bowdidge and W. G. Griswold. Supporting the restructuring of data abstractions through manipulation of a program visualization. *ACM Trans. Software. Engineering. Methodology*, 7(2):109–157, April 1998.
- C. D. Hundhausen, S. A. Douglas, and J. T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing*, 13(3):259–290, June 2002.
- A. J. Ko and B. A. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, Vienna, Austria, CHI '04*. ACM Press, New York, NY, pages 151–158, April 24 - 29 2004.
- A.N. Kumar. Generation of problems, answers, grade and feedback - case study of a fully automated tutor. *Journal of Educational Resources in Computing (JERIC)*, 2006a.
- A.N. Kumar and S. Kasabov. Observer architecture of program visualization. *Proceedings of the Fourth Program Visualization Workshop, Florence, Italy*, June 29-30 2006.
- T. Loboda, A. Frengov, A.N. Kumar, and P. Brusilovsky. Distributed framework for adaptive explanatory visualization. *Proceedings of the Fourth Program Visualization Workshop, Florence, Italy*, June 29-30 2006.
- B. A. Malloy and J. F. Power. Exploiting uml dynamic object modeling for the visualization of c programs. In *Proceedings of the 2005 ACM Symposium on Software Visualization*, pages 105–114, May 14 - 15 2005.
- E. Mayer and R.B. Anderson. Animations need narrations: An experimental test of a dual-coding hypothesis. *Journal of Educational Psychology*, 83:484–490, 1991.
- A. Moreno, N. Myller, E. Sutinen, and M. Ben-Ari. Visualizing programs with jeliot 3. In *Proceedings of the Working Conference on Advanced Visual interfaces, Gallipoli, Italy*, ACM Press, New York, NY, pages 373–376, May 25 - 28 2004.
- T. L. Naps, R. Fleischer, M. McNally, G. Rößling, C. Hundhausen, S. Rodger, V. Almstrum, A. Korhonen, J.A. Velazquez-Iturbide, W. Dann, and L. Malmi. Exploring the role of visualization and engagement in computer science education. *SIGCSE Bulletin*, 35(2):131–152, June 2003.
- A. Paivio. Mental representations: A dual coding approach. *New York: Oxford University Press*, 1990.
- N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19, 1987.

# Work in Progress: A Detail+Context Approach to Visualize Function Calls

Xiaoming Wei and Keitha Murray

*Computer Science Department*

*Iona College, New Rochelle, New York, 10801*

`{xwei,kmurray}@iona.edu`

## Abstract

We describe our work in progress to use Java3D to animate dynamic function calls. Our main goal is to develop a visualization tool to help both lower level and upper level computer science students understand function call trace. A detailed visualization of a function call, such as the assignment of variables, allocation of memory, execution of control statements, etc. can help lower level students understand underlying data structures and algorithms. At the same time, a global view of the hierarchical call chain can provide greater insight for higher level students. We choose to use Java3D in consideration of visualizing larger and more complex Java programs.

## 1 Introduction

Software Visualization (SV) is defined as the method using typography, graphic image, sound and cinematography to demonstrate and help users understand computer software. Early in 1981, Baecker Baecker (SIGGRAPH 1981, Los Altos, CA) presented an algorithm animation video "Sorting out Sorting". Since then, many software visualization systems or tools Akingbade et al. (2003); Cappos and Homer (2001); Naps et al. (2000); Röbling and Freisleben (2002) have been developed and evaluated by researchers based on effectiveness of teaching Ithantola et al. (2005); Pollack and Ben-Ari (2004).

The problem we would like to solve is the dynamic visualization of function calls. Viewing an application's call trace in graphical form can be an elucidating educational experience. For novice CS students, doing so can help them understand an application's internal behavior. Advanced students will be able to obtain information for program optimization. For example, by optimizing those functions that are called most often, you can get the greatest performance benefit from the least amount of effort. The paper is organized in the following manner. In section 2, we discuss the background and motivation for this work. In section 3, we introduce our work in progress.

## 2 Background

We can roughly classify SV into three categories as algorithm visualization, program visualization and the combination of the two. Algorithm visualization is the visualization of high level abstraction of data, operations and the semantics of a program. Program visualization displays the underlining source code and data structures of a program. Most of the algorithm visualizations are created by visualizing a specific data structure or algorithm. They are usually based on a higher abstraction level than the source code. Such tools are very helpful when teaching algorithm concepts, such as Quicksort and AVL-trees. For students in introductory and intermediate programming courses, we find that it is relatively easy for them to understand the fundamental algorithms in a higher level pseudo code or script language, but much more difficult to transform the pseudo code into source program. One of the reasons is that the concepts of variables, memory, function calls, etc. are hard to imagine, in other words, they can not visualize them.

We believe when teaching introductory and intermediate computer science courses, showing how an algorithm or data structure works is not good enough since the implementation of the algorithm is also important. The question is how to help students understand the

dynamic aspects of programs better? Jeliot 3 Moreno et al. (5th Annual Finnish / Baltic Sea Conference on Computer Science Education. November 17 - November 20, 2005) is an animation system that visualizes the dynamic execution of Java programs by showing the current state of the program (e.g. methods, variables and objects) and animations of the expression evaluations. Kannusmaki et al. Kannusmaki et al. (2004) give a thorough evaluation of using this tool in a secondary programming course. Jeliot 3 can be a great tool to teach novice students in their first programming course. However, for intermediate and advance students, an adaptive visualization tool is needed since the interests of these two groups of students are different.

As Java programs get larger, they become difficult to understand and to debug. In order for advanced computer science students to understand and further improve their programs, they have to read through the code and follow function calls for various inputs. And throughout the process, it is very easy for them to lose the global picture of the function call stack. Most of the current SV systems display the call trace either by using a fixed 2D window or using a separate pop up window for each function call. However, this leads to the problem of overlapping data for a program with many function calls. Joshi, et. al. Joshi et al. (2004) designed a tool to visualize dynamic call graph using a helix cone tree structure. The runtime behavior of executing Java programs is captured and recorded in an XML file. A viewer loads the file and renders it in the helix cone tree format. The user can also interact with the visualization result in the viewer. A static view of the function call trace will help students understand the structure of the program. However, using this approach, we will not be able to observe the internal behavior of each function. We would like to design a dynamic and step by step animation of the function call chain. In this way, novice students can interact with the current function call, while advanced students will be able to concentrate more on the global view of the call chain.

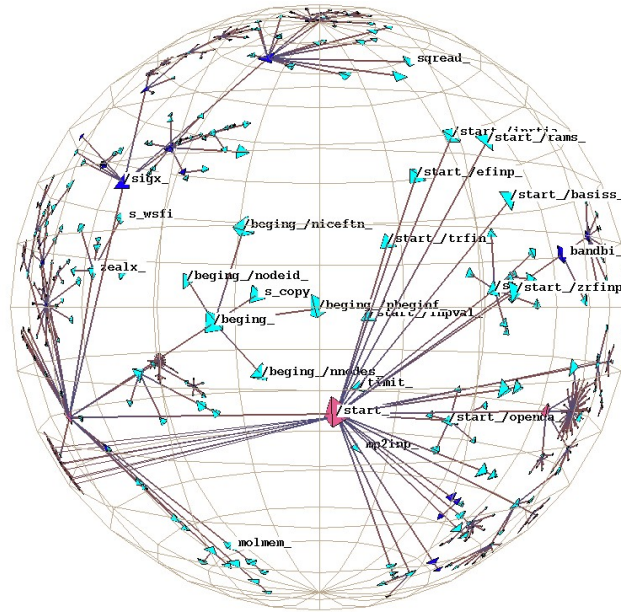
### 3 Current Work

We plan to use Java3D to set up a visualization environment. The approach is as follows. A global data structure is defined to represent the hierarchical structure of function calls. Each function call inserts a node into the data structure and Java3D will update the visualization interface according to the new data structure. To visually represent the hierarchical structure, we plan to use a hyperbolic graph layout representation Lamping et al. (1995). Hyperbolic trees are defined in 3D non-Euclidean space; they are very useful and an efficient approach to visualizing hierarchical structures.

Figure ?? shows such an example. While traditional methods such as paging (divide data into several pages and display one page at a time), zooming, or panning show only part of the information at a certain granularity, hyperbolic trees show detail and context at once. Students can interact with the result by rotating, selecting and scaling in the visualization interface. In the meantime, we will keep a timer to allow users to control the speed of the program. In this way, the visualization is synchronized with the running program. We will be able to animate the detailed information inside each function.

The reason to build this detail+context function visualization tool is to help both novice and experienced computer science students. Novice students concentrate more on the execution of statements and how the values of variables change. To help them, we allow the user to interactively choose one function node and display the function details in the manner of a fisheye view. Advanced students concentrate more on the overall structure of the program and care more about the complexity and how to improve and debug the program. Having a global view helps them identify loops in the function call chain easily. The students will be able to debug and optimize an application through a understanding of the call chains and their respective frequencies.

During the Spring semester, we are currently teaching a graduate level Information Vi-



**Figure 1:** An Example of Hyperbolic Tree. T. Munzer Munzner (October 20-21 1997) use hyperbolic tree to show the static function call graph structure for a mixed C/FORTRAN scientific computing benchmark. The color indicates whether a particular global variable was untouched(cyan), referenced(blue), or modified(pink). Such display can help software engineers see which parts of a large and/or unfamiliar program might be modularizable.

sualization class. The first half of the class is devoted to a discussion of basic concepts in computer graphics, such as geometric transformation, texture mapping, lighting and shading and the introduction of techniques used in information visualization, such as parallel coordinates, cone-tree, hyperbolic tree, multi-focal display, etc. In the second half of the class, each student is required to implement a final course project. The work proposed in this paper is one project we are working on with a student. The student will work through the summer semester to complete the project. Because of the limitations of time, the visualization tool, at the current stage, will be part of the user program that needs to be visualized. At a later stage, we would like to separate them. Eventually, we would like to have a system that will be able to load in different Java programs and animate the function calls automatically.

## References

- A. Akingbade, T. Finley, D. Jackson, P. Patel, and S. H. Rodger. Jawaaw: easy web-based animation from cs 0 to advanced cs course. In *Proceedings of the 34th SIGCSE Technical symposium on Computer Science Education*, pages pp. 162–166, 2003.
- R. Baecker. Sorting out sorting, SIGGRAPH 1981, Los Altos, CA.
- Justin Cappos and Patrick Homer. Dscats: Animating data structures for cs 2 and cs 3 courses. Technical report, University of Arizona, Tucson, AZ, 2001.
- P. Ihantola, V. Karavirta, A. Korhonen, and J. Nikander. Taxonomy of effortless creation of algorithm visualizations. In *ICER*, pages 123–133, 2005.
- J. Joshi, B. Cleary, and C. Exton. Application of helix cone tree visualization to dynamic call graph illustration. In *Third Program Visualization Workshop*, pages 68–75, 2004.



- O. Kannusmaki, A. Moreno, N. Myller, and E. Sutinen. What a novice wants: Students using program visualization in distance learning course. In *Third Program Visualization Workshop*, pages 126–133, 2004.
- J. Lamping, R. Rao, and P. Pirolli. A focus+context technique based on hyperbolic geometry for visualizing large hierarchies. In *In Proceedings of the Conference on Human Factors in Computing Systems*, pages pp. 401–408, 1995.
- A. Moreno, N. Myller, and R. Bednarik. Jeliot3, an extensible tool for program visualization, 5th Annual Finnish / Baltic Sea Conference on Computer Science Education. November 17 - November 20, 2005.
- Tamara Munzner. H3: Laying out large directed graphs in 3d hyperbolic space. In *Proceedings of the 1997 IEEE Symposium on Information Visualization*, pages pp. 2–10, October 20-21 1997.
- T. L. Naps, J. R. Eagan, and L. L. Norton. Jhave: An environment to actively engage students in web-based algorithm visualizations. In *Proceedings of the 31th SIGCSE Technical symposium on Computer Science Education*, pages 109–113, 2000.
- S. Pollack and M. Ben-Ari. Selecting a visualization system. In *Third Program Visualization Workshop*, pages 134–140, 2004.
- G. Rößling and B. Freisleben. Animal: A system for supporting multiple roles in algorithm animation. *Journal of Visual Languages and Computing*, 13(3):341–354, 2002. URL [citeseer.ist.psu.edu/oling02animal.html](http://citeseer.ist.psu.edu/oling02animal.html).

# Providing Data Structure Animations in a Lightweight IDE

Dean Hendrix, James H. Cross, Jhilmil Jain, and Larry Barowski

*Department of Computer Science and Software Engineering*

*Auburn University*

*Auburn, Alabama 36849 USA*

`{hendrtd, crossjh, jainjhi, barowla}@auburn.edu`

## Abstract

Motivated by the need to bridge the gap between implementation and the conceptual view of data structures, new dynamic object viewers have been integrated into jGRASP.<sup>3</sup> These viewers provide multiple synchronized visualizations of data structures as the user steps through the source code in either debug or workbench mode. This tight integration in a lightweight IDE provides a unique and promising environment for learning data structures. Preliminary evaluation of use by CS2 students indicates the viewers can have a significant positive impact on student performance.

## 1 Introduction

Although many visualization techniques have been shown to be pedagogically effective, they are still not widely adopted. The reasons include: lack of suitable methods of automatic-generation of visualizations; lack of integration among visualizations; and lack of integration with basic integrated development environment (IDE) support. To effectively use visualizations when developing code, it is useful to automatically generate multiple synchronized views without leaving the IDE. The jGRASP IDE (<http://jgrasp.org>) provides object viewers that automatically generate dynamic, state-based visualizations of objects and primitive variables in Java. Such seamless integration of a lightweight IDE with a set of pedagogically effective software visualizations should have a positive effect on the usefulness of software visualizations in a classroom environment. Multiple instructors have reported positive anecdotal evidence of their usefulness. We conducted formal, repeatable experiments to investigate the effect of these viewers for singly linked lists on student performance and we found a statistically significant improvement over traditional methods of visual debugging that use break-points. Similar experiments, but which focus on binary search trees, are currently underway.

## 2 Related Work

The approach we have taken for the state-based viewers in jGRASP to automatically generate the visualization from the user's executing program and then to dynamically update it as the user steps through the source code in either debug or workbench mode. This 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

<sup>3</sup>The jGRASP research project is funded, in part, by a grant from the National Science Foundation.

against modifying the user's 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 user's actual program during routine development.

### 3 Motivation

All Computer Science, Software Engineering, and Wireless Engineering majors at Auburn University are required to take the COMP 1210 course (an objects-early CS1 in Java) followed by the COMP 2210 course (a Java-based CS2). Data structures and algorithms are abstract concepts, and the understanding of this topic and the material covered in class can be divided into two levels: a) Conceptual - where students learn concepts of operations such as create, add, delete, sort etc; and b) Coding - where students implement the data structure and its operations using any programming language (Java in our case). Attrition from our computing majors is most noticeable during the CS2 course.

We conducted paper-based surveys and one-on-one interviews in Fall 2004 and Spring 2005 to understand the aspects of the CS2 course that students find most difficult (Jain et al., 2005a). One result of the surveys was a clear indication that students did not find fundamental concepts difficult to understand but had much more trouble with the implementation. This survey result was supported by data from the course grades. About 75% of students indicated that they had an appropriate level of expertise in Java to complete the requirements of CS2. The basic problem was that students have difficulty transitioning from static textbook concepts to dynamic programming implementation (Shaffer et al., 1996). Thus, there is a need to bridge the gap from concepts to implementation.

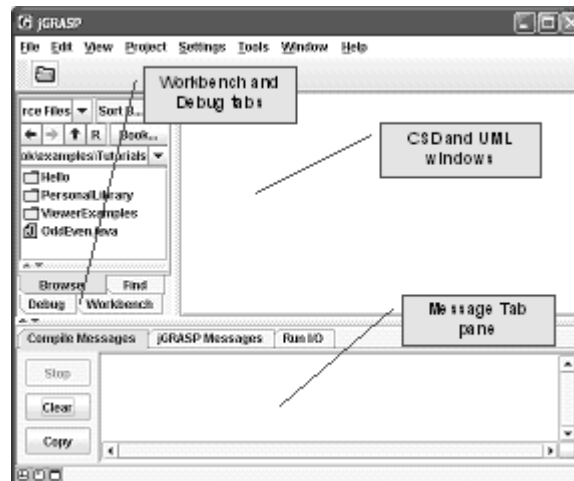
Felder and Silverman (1988) report that between 75-80% of students are visual learners. Most students will retain more information when it is presented with visual elements (pictures, diagrams, flowcharts, etc). In programming, visual learners can benefit from creating diagrams of problem solutions (e.g., flowcharts) before coding. Similarly, visual representations of data structure states should help in data structure understanding. Thus, it would be beneficial to have a tool that enables students to visualize both the conceptual and the implementation aspects of data-structures.

We surveyed over 21 tools that are used for the purpose of data structure visualization (Jain et al., 2005b) and found that most tools (more than 14 in our survey) focused on conceptual understanding. We found that only 7 implementation level tools were available to help students during program comprehension and debugging activities. None of these implementation tools fulfilled all of our goals, viz.,

- serve the dual purpose of classroom demonstration and development environment (i.e. can be used for lab exercises and assignments)
- provide automatic generation of views
- provide multiple and synchronized views
- provide full control over the speed of the visualization
- bridge the gap between abstract learning and code implementation

### 4 jGRASP Object Viewers

During execution, Java programs will usually create a variety of objects from both user and library classes. Since these objects only exist during execution, being able to visualize them in a meaningful way can be an important element of program comprehension. Although this visualization can be done mentally for simple objects, most programmers can benefit from seeing more tangible representations of complex objects while the program is running.



**Figure 1:** jGRASP Virtual Desktop

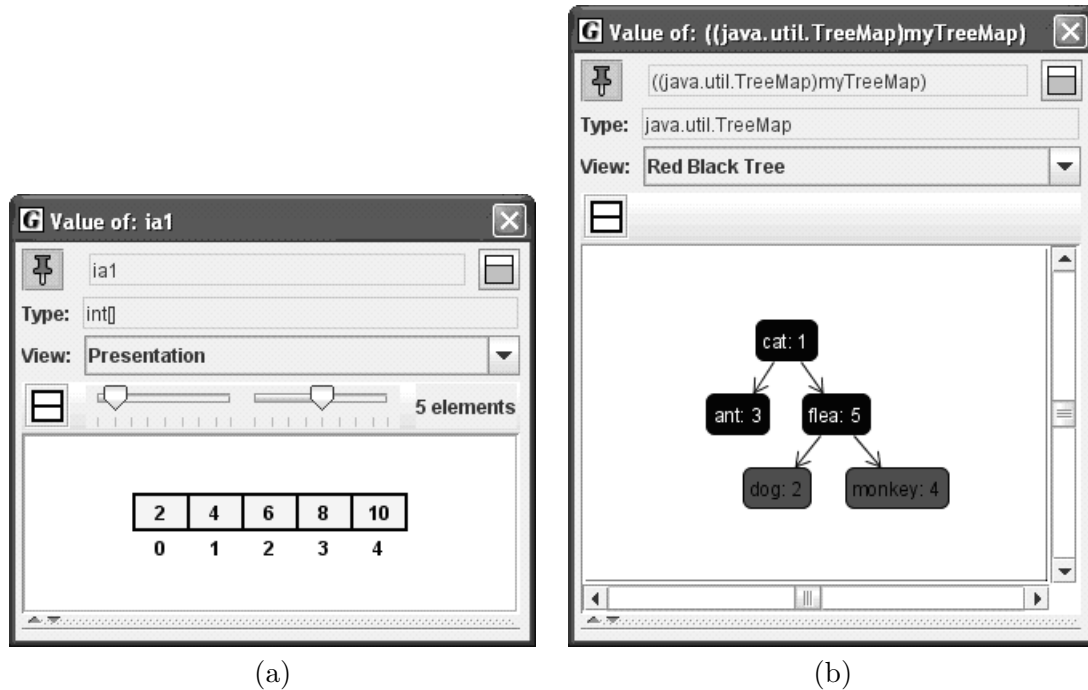
Beginning with version 1.8, the jGRASP IDE provides a family of dynamic viewers for objects and primitives. These viewers are the most recent addition to the software visualizations provided by jGRASP. The purpose of a viewer is to provide one or more views of a particular class of objects. When a class has more than one view associated with it, the user can open multiple viewers on the same object with a separate view in each viewer. These viewers are tightly integrated with the jGRASP workbench and debugger and can be opened for any item in the Workbench or Debug tabs from the Virtual Desktop (see Figure 1). Since the jGRASP integrated debugger is used to collect the runtime information necessary to render the visualizations, a program must run in the debugger or from the jGRASP workbench for its data structures to be visualized. A separate viewer window can be opened for any primitive, object, or field of an object that is currently active on the workbench or in the debugger tab by simply dragging and dropping an icon from the debugger or workbench to the jGRASP desktop. Thus, these viewers are effortless with respect to the amount of work required of the student to create and use them.

All objects have a basic view, which is the same as the view shown in the workbench and debug tabs. This view shows all the values associated with the object in a collapsible hierarchy. Depending on their data type, some objects will have additional views. Figures 2a and 2b show object viewers for an array of integers (int) and an instance of `java.util.TreeMap`. Each is shown in a presentation view which is intended to be similar to a textbook depiction or to what an instructor might draw on the board. jGRASP provides presentation viewers for arrays, strings, and classes from the Java Collections Framework.

## 5 Animated Verifying Viewers

Viewers fall into two categories: non-verifying and verifying. The non-verifying viewers assume that the structure of the object being viewed is correct, and generally use method calls to elaborate the structure. When a structure gets beyond a certain size, the non-verifying viewers will examine only the part of the structure that is on-screen. Because of this, they can be used to examine large structures without slowing the debugging process excessively. The non-verifying viewers would generally be used to examine the contents of a structure in the context of an algorithm that uses it, rather than to examine the workings of the data structure itself. Viewers for “built-in” data structures (e.g., arrays, JCF classes) are all non-verifying. Non-verifying viewers are discussed in further detail in (Hendrix et al., 2004).

The purpose of the verifying viewers is to aid in the understanding of the data structures themselves, and to assist in finding errors while students are developing their own implemen-



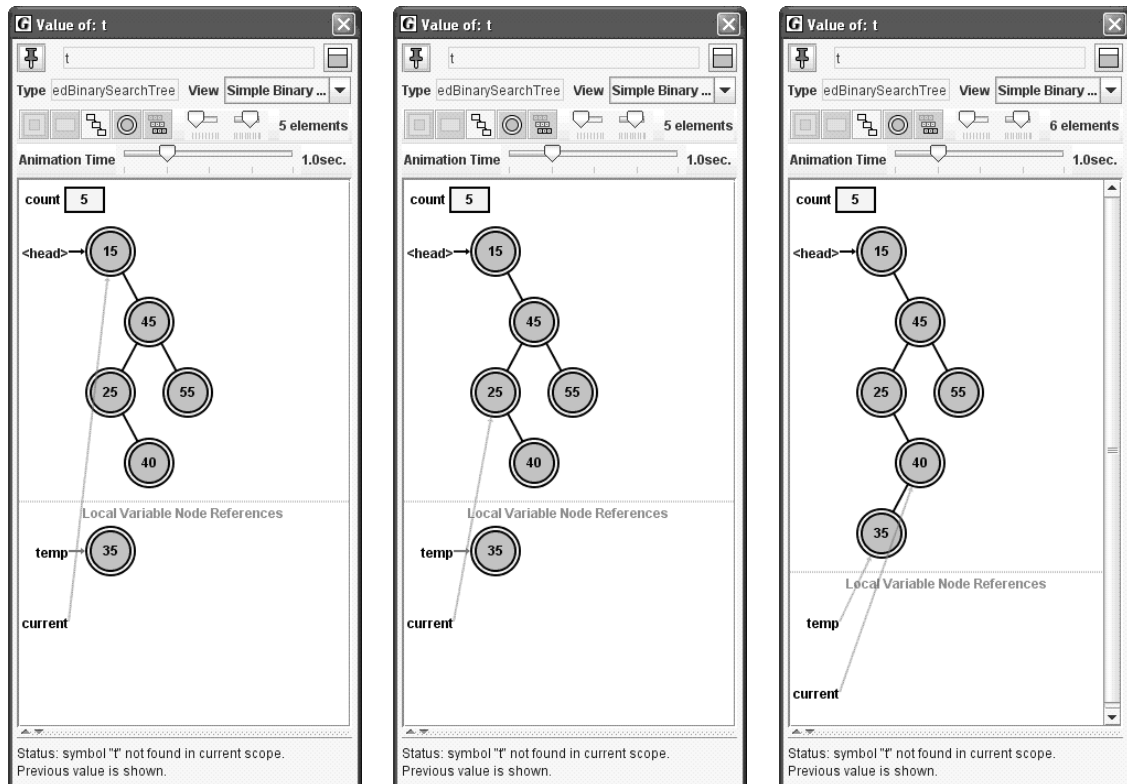
**Figure 2:** (a) Viewer for an array of ints. (b) Viewer for an instance of `java.util.TreeMap`.

tation of a data structure. To further this intended use, any local variables of the structure's node type are also displayed, along with any nodes to which they are linked. Links between these local variable nodes or structure fragments and the main structure are displayed. This allows mechanisms of the data structure such as finding, adding, moving, and removing elements to be examined in detail by stepping through the code.

As an additional aid to understanding the mechanisms of the data structure, the verifying viewers animate structural changes. In order to do this, they store a representation of the entire data structure at each update (viewer updates happen at a breakpoint or after a step in the debugger). At each update, the value from the previous update (which may or may not be the same as the current value) is examined for changes. If any nodes in the structure have moved, the viewer enters into animation mode. In this mode, an "animation update" occurs at regular intervals. During animation, the previous structure value and previous local variable nodes and structure fragments (which may or may not be present any longer) are displayed. Node locations are interpolated so that they move smoothly from their old locations to the new ones, within and between the main structure and local variable nodes and structure fragments. At the end of animation, the new structure value and new local variable nodes and structure fragments are displayed.

Animated verifying viewers for data structures are currently created by extending the base viewer classes provided with the jGRASP. When placed in a viewer directory, these viewers are available to any program executing in debug or workbench mode. A user can simply drag and drop the object reference anywhere on the screen. The viewer will be automatically updated as the user steps through the code. If multiple viewers are implemented for the same class, the user simply makes a selection from a drop down list in the viewer window. We are currently working towards a viewer mechanism which will attempt to identify the type of structure, if any, defined by a user's class, and then map the internal fields of the class onto an appropriate category of viewer class (e.g., linked list). This will drastically reduce the need to manually extend the base viewer classes. When the user opens a viewer, the goal is for jGRASP to determine the inherent data structure of the object and display the most appropriate view.

Figure 3 shows three frames from an animation sequence generated by a verifying viewer.



**Figure 3:** Snapshots from an animated verifying viewer for a “textbook” binary search tree.

The viewer was opened on an instance of a binary search tree class used in the CS2 textbook. These frames depict the insertion of a new element (35) into an existing binary search tree. Using this viewer, students are able to watch the pointer (*current*) walk down the tree nodes to find the proper insertion point, and then watch as the new node “slides” into place as the left child of 40. All this is done as they are stepping through the code, thus making an immediate connection between the abstract behavior demonstrated in class and the concrete implementation embodied in the code.

## 6 Evaluation

We are currently conducting controlled experiments to test the following hypotheses:

1. Students are able to code more accurately (with fewer bugs) using the jGRASP data structure viewers.
2. Students are able to find and correct “non-syntactical” bugs faster using jGRASP viewers.

Two experiments focusing on linked lists have already been performed and statistically significant results were obtained (Jain et al., 2006). Data analysis from these experiments show that animated verifying viewers increase both accuracy and speed for students during development and debugging of their linked list code. Two follow-on experiments have just now been performed, but the data analysis is not complete. These experiments focused on binary search trees rather than linked lists.

### 6.1 Tree Experiment 1

The hypothesis being tested was that students will be more productive during development (will code faster and with greater accuracy) using the jGRASP data structure viewers. Stu-

dents were asked to implement one operation for linked binary search trees. The class `LinkedListBinaryTree.java` from the class textbook was used in this experiment. Students were provided a detailed description of the programming assignment and the grading policy. Students were required to work independently and were timed (although there was no time limit to complete the assignment). The independent variable was the visualization medium (coding using jGRASP viewers vs. without viewers). The dependent variables were: time taken to complete the assignment, and the accuracy of the assignment.

The control group implemented the method `levelOrder()` using the jGRASP visual debugger without the viewers. The driver program provided to this group contained a `toString()` method so that they could print out the contents of the list without writing additional code. The treatment group implemented the same method using the jGRASP visual debugger with the object viewers. Since our algorithm for `levelOrder()` traversal required three different data structures, we provided the students with three viewers (for `LinkedListBinaryTree`, `LinkedListQueue` and `ArrayUnorderedList`). The driver program given to this group did not contain the `toString()` method, so the subjects had to use the viewers in order to see the contents of the data structures. The machines in the lab were set up with permissions such that only the treatment group had access to the viewers.

## 6.2 Tree Experiment 2

Our hypothesis was that students are able to detect and correct logical bugs in less time when using jGRASP viewers. A Java program that implemented a linked binary search tree was provided. The program contained a total of 5 logical errors, one in each of the following five methods `addElement()`, `removeElement()`, `find()`, `preorder()`, and `postOrder()`. Students were asked to find and correct all the logical errors. The independent variable was the visualization medium (finding errors using jGRASP viewers vs. without viewers). The dependent variables were: number of bugs found, number of bugs accurately corrected, and number of new bugs introduced in the program while performing the experiment. Both the groups were first required to identify and document errors. Next, similar to experiment 1, the control group corrected the detected errors using the jGRASP visual debugger without the viewers and the treatment group corrected the errors using the jGRASP visual debugger with the object viewers.

While the data analysis for these two experiments is not yet complete, anecdotal evidence from students suggests that the positive results from the linked list experiments will be replicated in the binary search tree experiments.

## 7 Conclusion

jGRASP object viewers automatically generate dynamic, state-based visualizations of objects and primitive variables in Java. Multiple synchronized visualizations of an object, including complex data structures, are immediately available to users within the IDE. Multiple instructors have used these viewers in CS1 and CS 2 and have reported positive anecdotal evidence of their usefulness. Formal, repeatable experiments with linked lists have indicated statistically significant positive results on student performance. Follow-on experiments with binary search trees have just been completed, and anecdotal evidence and student feedback suggest that they will yield similar positive results.

## References

- R. M. Felder and L. K. Silverman. Learning and teaching styles in engineering education. *Engineering Education*, 78:674–681, 1988.
- J. Hamer. A lightweight visualizer for java. In *Proceedings of Third Program Visualization Workshop*, pages 55–61, July 2004.

- D. Hendrix, J. Cross, and L. Barowski. An extensible framework for providing dynamic data structure visualizations in a lightweight ide. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, pages 387–391, February 2004.
- J. Jain, N. Billor, D. Hendrix, and J. Cross. Survey to investigate data structure understanding. In *Proceedings of the International Conference on Statistics, Combinatorics, Mathematics and Applications*, Auburn, Alabama USA, December 2005a.
- J. Jain, J. Cross, and D. Hendrix. Qualitative assessment of systems facilitating visualization of data structures. In *Proceedings of 2005 ACM Southeast Conference*, Kennesaw, Georgia USA, March 2005b.
- J. Jain, J. Cross, D. Hendrix, and L. Barowski. Experimental evaluation of animated-verifying object viewers for java. In *SoftViz 2006 (submitted)*, 2006.
- 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 Program Visualization Workshop*, pages 126–133, July 2004.
- T. Naps. Jhave: supporting algorithm visualization. *IEEE Computer Graphics and Applications*, Sep/Oct:49–55, 2005.
- C. Shaffer, L. S. Heath, and J. Yang. Using the swan data structure visualization system for computer science education. In *Proceedings of SIGCSE 1996*, pages 140–144, 1996.
- A. Zeller. Visual debugging with ddd. *Dr. Dobbs's Journal*, July, 2001.



# Inductive Reasoning and Programming Visualization, an Experiment Proposal

Taiyu Lin<sup>1</sup>, Andrés Moreno<sup>2</sup>, Niko Myller<sup>2</sup>, Kinshuk<sup>1</sup>, Erkki Sutinen<sup>2</sup>

<sup>1</sup>*Advanced Learning Technologies Research Centre, Department of Information Systems,  
College of Business, Massey University, New Zealand*

<sup>2</sup>*Department of Computer Science, University of Joensuu, Joensuu, Finland*

t.lin@massey.ac.nz, amoreno@cs.joensuu.fi, nmyller@cs.joensuu.fi,  
kinshuk@ieee.org, sutinen@cs.joensuu.fi

## Abstract

We lay down plans to study how Inductive Reasoning Ability (IRA) affects the analyzing and understanding of Program Visualization (PV) systems. Current PV systems do not take into account the abilities of the user but show always the same visualization independently of the changing knowledge or abilities of the student. Thus, we propose IRA as an important skill when comprehending animation, which can be used to model the students and thus to adapt the visualization for different students. As an initial step we plan to check if IRA correlates with ability to answer program related questions during program visualization. We discuss the possible benefits of using IRA modeling in adaptive PV.

## 1 Introduction

Inductive reasoning is one of the important characteristics of human intelligence: Thurston regarded inductive reasoning as one of the seven primary mental abilities (Selst, 2003) that are accounted for intelligent behaviors. Merriam-Webster on-line dictionary defined induction as an “inference of a generalized conclusion from particular instances”. Pellegrino and Glaser (1982) noted that the induction reasoning ability can be extracted in most aptitude and intelligent tests and is the best predictor for academic performance. Harverty et al. (2000, p. 250) cited several other researches that viewed inductive reasoning as an significant factors for problem solving, concept learning, mathematic learning, and development of expertise, and Heller et al. (2001) research showed that inductive reasoning is necessary ability for extracting the knowledge of problem solving in physics.

Despite its recognized importance underlying the learning process, little effort is spent on research to support the inductive reasoning process taking place in virtual learning environments (Lin et al., 2003). This paper describes a project to incorporate modeling and support of learner’s inductive reasoning ability in Jeliot - a program visualization system to help learning Java. If an inductive reasoning ability proves to be a good indicator of programming learning, program visualization tools could adapt their contents according to the students inductive reasoning ability. The paper is a continuation of the road map laid down in (Bednarik et al., 2005) and can be seen as an initial step towards an adaptive or smart program visualization system.

This paper starts with an introduction to the research done in inductive reasoning ability (Section ??). Following in Section ?? there is a description of Jeliot 3 and the modifications done to support the experiment described in Section ??. We end the paper with a discussion of the possible implications of the findings we may come across.

## 2 Inductive Reasoning Ability

The term induction is derived from the Latin rendering of Aristotle’s epagoge that is the process for moving to a generalization from its specific instances (Rescher, 1980). Bransford et al. (2000) pointed out that generalizations aimed at increasing transferability (ability to apply to a different context) can result in (mathematical) models, or global hypothesis in

terms of Harverty et al. (2000). These can be later applied to a variety of contexts in an efficient manner.

Zhu and Simon (1987) pointed out that the learners have to induce how and when to apply the problem solving method in worked-out examples. In reality, the learners also need to induce where to apply with a contextual awareness. The induction here requires the learners to (1) recognize the similarities and differences of the parameters in the current and the experienced contexts, (2) recognize and match pattern of current context to the experienced context(s), and/or (3) recognize/create the hypothesis/method that can be applied to solve the problem.

Point (1) requires information filtering, encoding and classification. In the early stage of an inductive reasoning task, one needs to see what attributes are relevant to the task at hand. Selected attributes are then encoded into meaningful chunks so that classifications or categories can be created. It is an iterative process to proceed from the selection of attributes to the forming of categories. In situations where a classification is sought, e.g., when solving a well-defined problem, the meaningfulness of classification is already a sufficient condition for one to stop this iteration, whereas if one is not confident with what the classifications should be, e.g. exploration of a novel domain, one may need to extend this iterative process to include hypothesis testing.

The pattern finding in point (2) is detection of co-variation from a stack of samples or past experiences as explained by Holland et al. (1987). In the research done by Heller et al. (2001), the instructors believed that the attainment of problem solving skill in physics comes from reflective practices to extract knowledge from previous experiences of working on other problems or from sample problem solutions. Point (3) corresponds to the hypothesis generation activity in categorization of Harverty et al. (2000). Hypothesis generation differs from mere guessing in an important ways - having a rationale behind the hypothesis. The rationale of the hypothesis is primarily derived from the observed pattern. Use of analogy also plays an important role for inductive reasoning.

### 3 Jeliot 3

Jeliot 3 (Moreno et al., 2004) is a program visualization system that animates the execution of Java programs. It has been used as an lecturing aid by teachers and a learning aid by students. Jeliot is currently used in classrooms (Ben-Bassat Levy et al., 2003) and in distance education (Kannusmi et al., 2004). Different problems have been identified in these contexts.

In classrooms, the mediocre students seem to benefit from the utilization of the program visualization tool the most. However, for the weakest and strongest students, Jeliot 3 in its current form might not be that helpful and useful. On the one hand, the visualization in Jeliot is still too complex and hard to grasp for the weakest students. On the other hand, the strongest students do not need the visualization as much and would like to use it only as needed or their learning might be even harmed because of this.

In distance education, students are studying alone and can have problems with learning and motivation. For visualization tools, this means that the use of the system should be both motivating and give students support in order to overcome barriers or even become a partial replacement of a teacher. As there is no teacher explaining the program visualization and the related programming concepts for the students, the tool should assume this role and provide explanations and hints to students as needed.

In order to know how to support students during the learning process on individual basis, we plan to use adaptation based on both students performance and cognitive traits.

#### 3.1 Question generation in Jeliot

Students can be engaged and motivated to explore and make sense of it by asking prediction questions during the visualization (Byrne et al., 1999; Naps, 2005). Furthermore, interaction

and question answering during multimedia learning have a positive influence on problem-solving ability in the domain (Evans and Gibbons, 2006).

We have implemented an automatic prediction question generation facility into Jeliot 3 (Myller, 2006). Question generation is achieved by analyzing the program trace (MCode) before it is visualized because the correct answer is also available. During the visualization the question is popped up before the evaluation of the selected expression has started. The question display and processing is based on the work of Rößling and Häussge (2004), namely *AVInteraction*. Students' answers are saved into a database together with the information related to the question such as related programming concepts. We aim to use the information collected about student's performance to analyze the skill development in different programming concepts and model student's cognitive traits in order to adapt the visualization to the abilities of the student.

### 3.2 Modeling the student in Jeliot

Student modeling provides the means to track the students' activities while using a learning tool and analyze students behavior and possible abilities and known concepts. Our devised system will contain both a domain model, and a student model. The domain model will contain a reduced set of Java concepts derived from the Java Learning Object Ontology (JLOO) (Lee et al., 2005). This set of concepts will still retain the "pre-requisite" relationships amongst concepts suggested in the JLOO.

To record the student's performance, the system will use an overlay user model: the model will consist of pairs containing a concept from the domain model and the student's ability to understand it. Abilities will be measured as the number of times the student has been asked about a certain concept, and number of correct and incorrect answers.

In the system, when a student answers correctly two or more questions related to the same Java concept, the user model will mark that concept as understood. We believe that already the second correct answer will minimize the "chance factor", nevertheless it does not necessarily mean that the concept is fully understood. Only concepts that have their "pre-requisite" concepts understood will be available for the system to generate questions.

## 4 Experiment Design

In this section, we propose an experiment to investigate the relationship between the students' IRA determined with psychometric test and their learning performance while using the previously described program visualization environment, Jeliot 3.

The students of a Programming I course in ViSCoS project, a 2 year-long web-based Computer Science study program, will be the experiment participants. At the beginning of the course students will complete Web-IRA, a task to analyze the IRA of the students. Web-IRA consists of questions which are divided into three categories: (1) series extrapolation where students are asked to find out what is the next item in a series - testing the ability to generate hypothesis, perform comparison (Bailenson et al., 2002); (2) analogical reasoning where students are often given a pair of related items and are asked to use this as an analogy to find another pair of related items - testing the ability to filter attribute relevance and map relevant attribute from one context to another (Sternberg, 1977), and (3) exclusion where the students are asked to detect irregularity among the items - testing the ability to classify and test hypothesis (Laumann, 1999).

The programming course is divided in weekly lessons, and concepts are distributed across them. For the experiment purposes, user model concepts will be also tied with the week they are explained, i.e. before a concept is learned students are asked to answer questions related to the concept (pre-test) and after the students have learned about the concept they will be presented with questions related to the same concept (post-test).

On week 2, they will be introduced to Jeliot 3 and the experiment. Each student will use a web start version of Jeliot, which will connect to an on-line version of the student model. Jeliot 3 will be updating the user model as the questions are answered.

Following weeks will introduce concepts at a fast pace. Students will be given a brief introduction to them and asked to visualize the program using Jeliot 3 and answer the questions. To motivate students to perform as well as they can points are granted based on the correct answers. As said before, only a reduced set of concepts will be used in the domain model. For the experiment, we will have two or three concepts that will make Jeliot generate questions about them. For example, in week 4 *Logical Statements* and *Comparison Operations* will be the chosen concepts.

At the end of the course, the Web-IRA could be completed again by the students, to test the reliability of the test itself.

The experiment is designed in a way that the student's behavior can be analyzed in terms of inductive reasoning ability. However, two major sources of bias can be predicted: student's prior knowledge and the case when a student does not make mistakes. If the student knows a concept and commits no mistake we can not analyze the effort required to induce/learn the concept. During the analysis, the first bias can be removed statistically by data (grade) from other courses students had taken. The analysis will not include data gathered from student who made no mistake.

After removing the potential biases, the following procedure will be carried out in the analysis:

1. Data gathered in Web-IRA will be standardized; and student's data gathered in Jeliot 3 will be matched with information from data gathered in Web-IRA.
2. The standardized Web-IRA will allow separation of students into groups, for example, groups of low, medium, or high IRA.
3. Pattern of behaviors in different groups can be searched by either statistical mean (average number of mistakes), decision-tree based classification algorithm such as ID-3 (Ross, 2000), or neural network.

## 5 Discussion and Future Work

Co-relating IRA and programming visualization comprehension is just another step in acknowledging the importance of cognitive skills in programming. If the link between IRA and programming abilities can be determined, it could have several implications in the Computer Science field. One of the possible implications of the result is that Web-IRA could be used to predicting the success of students taking programming courses. Moreover, it could be interesting to study later how much Jeliot 3 could help those students with low IRA.

But the results could imply as well that if reasoning patterns can be found in Jeliot 3, they can be built into the pedagogical module of Jeliot 3 which would then be able to classify students into groups (low, medium, or high IRA). It entails that Jeliot 3 alone will be able to model students' IRA without the need of data from Web-IRA.

### 5.1 Personalization of Program Visualization tools

The experiment described here is part of a bigger effort to make program visualization tools aware of personal differences. A investigation done on Jeliot 3 (Moreno and Joy, 2006) showed that students could not always fully understand the animations. To help students understanding the animations and the concepts behind, it was suggested to add explanations and questions to the animation. This will require a proper modeling of the student's ability and previous knowledge. Otherwise, the expanded animation will not cover the needs of a specific

student. For this experiment, we will implement the student model that will gather the information about a student's knowledge, and decide which questions are more suitable for the students. A further iteration in the development of the tool will add textual explanations of the animations and the programming concepts involved in them.

If a link between IRA and program visualization comprehension were to be found, animations, questions and explanations could be tailored to the student's IRA. It is hypothesized that students with lower IRA will benefit of further explanations or hints. For example, animations of certain concepts could be ignored after a number of times the student has seen it. The number of repetitions should be linked to the student's IRA; a student with high IRA should not require the same information repeated as many times as a student with lower IRA. However, this will require further experimentation to fully prove that hypothesis.

## References

- Jeremy N. Bailenson, Michael S. Shum, Scott Atran, Douglas L. Medin, and John D. Coley. A bird's eye view: biological categorization and reasoning within and across cultures. *Cognition*, 84:1–53, 2002.
- Roman Bednarik, Andrés Moreno, Niko Myller, and Erkki Sutinen. Smart program visualization technologies: Planning a next step. In *Proceedings of the 5th IEEE International Conference on Advanced Learning Technologies*, pages 717–721, Kaohsiung, Taiwan, 2005.
- Ronit Ben-Bassat Levy, Mordechai Ben-Ari, and Pekka A. Uronen. The Jeliot 2000 program animation system. *Computers & Education*, 40(1):1–15, 2003.
- John D. Bransford, Ann L. Brown, Rodney R. Cocking, M. S. Donovan, and James W. Pellegrino. *How people learn: Brain, mind, experience, and school*. National Academy Press, Washington, US, 2000.
- Michael D. Byrne, Richard Catrambone, and John T. Stasko. Evaluating animations as student aids in learning computer algorithms. *Computers & Education*, 33(4):253–278, 1999.
- Chris Evans and Nicola J. Gibbons. The interactivity effect in multimedia learning. Accepted to *Computers & Education*, 2006.
- Lisa A. Harverty, Kenneth R. Koedinger, David Klahr, and Martha W. Alibali. Solving inductive reasoning problems in mathematics: No-so-trivial pursuit. *Cognitive Science*, 24(2):249–298, 2000.
- Patricia Heller, Kenneth Heller, Charles Henderson, Vince H. Kuo, and Edit Yerushalmi. Instructors' Beliefs and Values about Learning Problem Solving. WWW-page, from <http://groups.physics.umn.edu/physed/Talks/Heller%20PERC01.pdf> (Retrieved 15 Oct, 2003), 2001.
- John H. Holland, Keith J. Holyoak, Richard E. Nisbett, and Paul R. Thagard. *Induction: Processes of inference, learning, and discovery*. The MIT Press, London, UK, 1987.
- Osku Kannusmäki, Andrés Moreno, Niko Myller, and Erkki Sutinen. What a novice wants: Students using program visualization in distance programming course. In Ari Korhonen, editor, *Proceedings of the Third Program Visualization Workshop (PVW 2004) Research Report CS-RR-407*, pages 126–133, Warwick, UK, 2004. Department of Computer Science, University of Warwick.
- Lisa L. Laumann. *Adult age differences in vocabulary acquisition as a function of individual differences in working memory and prior knowledge*. PhD thesis, West Virginia University, 1999.

- Ming-Che Lee, Ding Yen Ye, and Tzone I Wang. Java learning object ontology. In *Proceedings of the 5th IEEE International Conference on Advanced Learning Technologies*, pages 538–542, Kaohsiung, Taiwan, 2005.
- Taiyu Lin, Kinshuk, and Ashok Patel. Cognitive trait model for persistent student modelling. In D. Lassner and C. McNaught, editors, *Proceedings of EdMedia 2003*, pages 2144–2147, Norfolk, USA, 2003. AACE.
- Andrés Moreno and Mike Joy. Jeliot 3 in a Demanding Educational Setting. In *Proceedings of the Fourth International Program Visualization Workshop*, Florence, Italy, 2006.
- Andrés Moreno, Niko Myller, Erkki Sutinen, and Mordechai Ben-Ari. Visualizing Program with Jeliot 3. In *Proceedings of the International Working Conference on Advanced Visual Interfaces, AVI 2004*, pages 373–380, Gallipoli (Lecce), Italy, 2004.
- Niko Myller. Automatic Prediction Question Generation during Program Visualization. In *Proceedings of the Fourth International Program Visualization Workshop*, Florence, Italy, 2006.
- Thomas L. Naps. JHAVE – Addressing the Need to Support Algorithm Visualization with Tools for Active Engagement. *IEEE Computer Graphics and Applications*, 25(5):49–55, December 2005.
- James W. Pellegrino and Robert Glaser. Analyzing aptitudes for learning: Inductive reasoning. In R. Glaser, editor, *Advances in instructional psychology (Vol. 2)*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1982.
- Nicholas Rescher. *Induction*. University of Pittsburgh Press, Philadelphia, US, 1980.
- Peter Ross. Rule induction: Ross quinlan’s id3 algorithm. WWW-page, from <http://www.dcs.napier.ac.uk/~peter/vldb/dm/node11.html> (Retrieved 23 Oct, 2003), 2000.
- Guido Rößling and Gina Häussge. Towards Tool-Independent Interaction Support. In Ari Korhonen, editor, *Proceedings of the Third International Program Visualization Workshop, Warwick, England*, pages 110–117, 2004.
- Mark Van Selst. Intelligence. WWW-page, from <http://www.psych.sjsu.edu/~mvselst/courses/psyc235/lecture/chapter14intelligence.pdf> (Retrieved 14 Oct, 2003), 2003.
- Robert J. Sternberg. *Intelligence, information processing, and analogical reasoning: the componential analysis of human abilities*. Wiley, New York, 1977.
- Xinming Zhu and Herbert A. Simon. Learning mathematics from examples and by doing. *Cognition & Instruction*, 4(3):137–166, 1987.

# Automatic Prediction Question Generation during Program Visualization

Niko Myller

*Department of Computer Science, University of Joensuu, Joensuu, Finland*

`niko.myller@cs.joensuu.fi`

## Abstract

Based on previous research it seems that the activities performed by and the engagement of the students matter more than the content of the visualization. One way to engage students to interact with a visualization is to present them with prediction type questions. This has been shown to be beneficial for learning. Based on the found benefits of the question answering during the algorithm visualization, we propose to implement an automatic question generation into a program visualization tool, Jeliot 3. In this paper, we explain how the automatic question generation can be incorporated into the current design of Jeliot 3. In addition, we provide various example questions that could be automatically generated based on the data obtained during the visualization process.

## 1 Introduction

According to Hundhausen et al. (2002), the activities performed by and the engagement of the students matter more than the content of the visualization. Thus, a research program has been laid down in which the level of engagement and its effects on learning with algorithm or program visualization are being studied (Naps et al., 2002). One of the ways to engage students to interact with a visualization is to present them with questions, which ask the students to predict what happens next in the execution or visualization (Naps et al., 2000). This has been shown to be beneficial for learning as well (Byrne et al., 1999; Naps, 2005). Furthermore, interaction and question answering during learning have been found to have a positive influence on problem-solving ability in the domain (Evans and Gibbons, 2006).

Based on the found benefits of the question answering during the algorithm visualization, we propose to implement an automatic question generation into a program visualization tool, *Jeliot 3* (Moreno et al., 2004). In this paper, we explain how the automatic question generation can be incorporated into the current design of Jeliot 3. In addition, we provide various example questions that could be automatically generated based on the data obtained during the visualization process. Finally, conclusions and future directions are presented.

## 2 Previous Work

Dancik and Kumar (2003) have developed a system, called *Problets*, that generates exercises related to programming concepts (e.g. loops, pointers etc.) from language independent templates, thus supporting multiple programming languages. These exercises present a program and ask the user to identify the lines that generate output and determine what is the output during the execution of the program. In pointer exercises, user needs to identify the code lines that are either syntactically or semantically erroneous. These exercises are delivered in the form of a applet that is connected to a server that handles the exercise generation and stores information related to the performance of the user. This is done in order to analyze what kind of exercises to present to the user.

When compared to the question generation in Jeliot 3, we can identify certain similarities and differences. Both ask questions related to the execution of a program. However, Problets are related to the program code, whereas questions in Jeliot can be related to the program code and visualization to give more variation in the question types as seen in Section ??.

Jeliot also support dynamic aspects of the program execution, for example, user can give input to the program and the questions are adjusted accordingly because they are based on

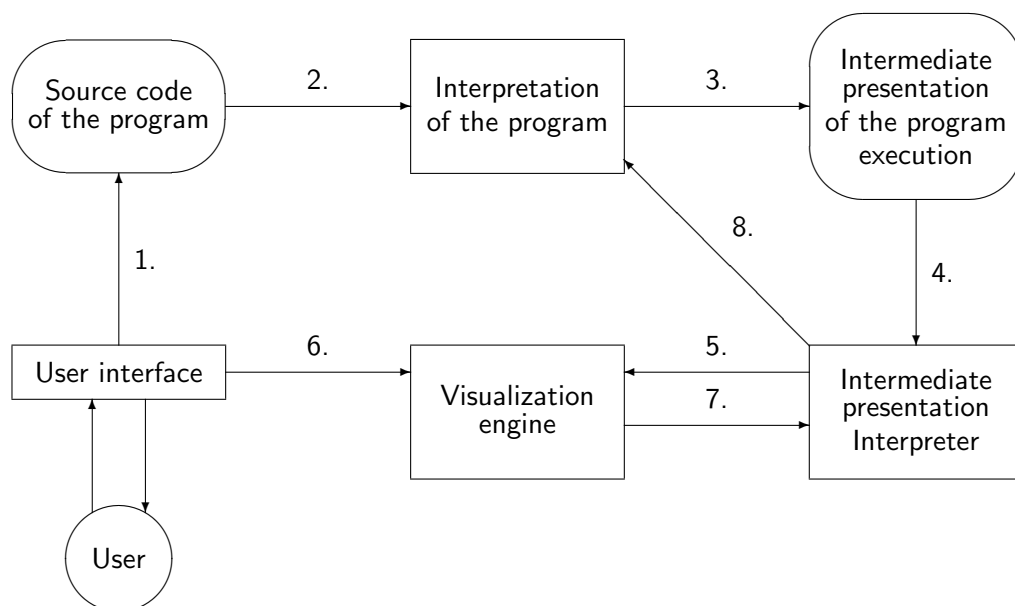
the information acquired during the interpretation process. Currently, Jeliot supports only Java but if interpreters of other programming languages are integrated to the system the question generation works for other languages as well. Problets supports multiple programming languages because of the language independent templates that are then translated to the programming language in question. Problets can be used for learning and testing as the automatic question generation in Jeliot 3.

Another related system is *WadeIn II* (Brusilovsky and Loboda, 2006) that visualizes the expression evaluation in C. The system consist of two modes: exploration and knowledge evaluation. The question generation is related to the knowledge evaluation mode in which student needs to demonstrate the understanding of the expression evaluation by simulating it.

### 3 Jeliot 3

*Jeliot 3* is a program visualization system that visualizes the execution of Java programs (Moreno et al., 2004). It has been designed to be used in introductory programming courses by teachers and students. Jeliot visualizes the data and control flow of the program. In a classroom study, it was found that especially the mid-performers benefited from the use of Jeliot while the performance of others was not harmed (Ben-Bassat Levy et al., 2003).

We describe the structure of Jeliot 3 in Figure ?? in order to explain in the next section how the automatic question generation fit into the current design. The user interacts with the user interface and creates the source code of the program (1). The source code is parsed and checks are performed before the actual interpretation by *DynamicJava*, a Java interpreter (2). During the interpretation, the intermediate code, MCode, is extracted (3). The intermediate code is interpreted and the directions are given to the visualization engine (4 and 5). The user can control the animation by playing, pausing, rewinding or playing step-by-step the animation (6). Furthermore, the user can give input to the program executed by the interpreter (6, 7 and 8).



**Figure 1:** The functional structure of Jeliot 3 (Moreno et al., 2004).



## 4 Automatic Question Generation in Jeliot 3

We used the modular design of Jeliot 3 in order to collect the necessary information for prediction question generation. We implemented a preprocessor for the intermediate presentation of the program execution that goes through the MCode and extracts the needed information for the questions. In the Figure ??, this phase would be located in the middle of the arrow 4. The information that is saved from every question contains:

- Expression identifier, that we can ensure, that there is only one time when the question is popped up.
- Concepts related to the expression that the question concerns.
- Correct answer that is the result of the interpretation.
- One to three incorrect answers when the question type is multiple choice. Currently, incorrect answers are determined randomly, but the design would allow certain heuristics to be used based on the previous steps in the execution and current values of variables.
- Expression's location in the source code.

During the interpretation of the MCode, it is checked whether there is a question for the expression that is being evaluated. If a question is found with the expression identifier, a question is popped up. We adapted the *AVInteraction* package introduced by Rößling and Häussge (2004) in order to present questions and collect users' answers. The answers can be saved into a file or a database. Thus, this feature can be used in order to adapt the program visualization as well as to summatively evaluate the students performance as part of their course work or on-line exam.

As proof of concept, we have implemented question generation only to ask the results of the assignment statements. An example of the generated question is displayed in Figure ?. The question is shown in the right together with the visualization and the related code segment is highlighted in the left.

- An actual question example in Section 3 would help readers get a better understanding of the idea. For instance, one could show a small code fragment containing just a variable assignment, and exemplify what the questions popped up would look like.

However, the question generation during program visualization allows the use of different kinds of question types related to the execution and animation of the program. We list here some possibilities:

- Predict the result of any expression evaluation.
- Ask the user to click on the variable whose value is going to be used in the expression, or into which the result of an expression evaluation is going to be assigned.
- In loops, it is possible to ask if the execution will continue for next round or not and in conditional statements, it is possible to ask if the execution will continue into the **then** or else part of the statement.
- User can be asked to click the line (or line number) that is being executed next, for instance, after a method call or in the beginning of a loop or an if-statement.
- In sorting, it would be possible to determine a swap operation and ask the user to click on those array cells that are going to be swapped.

It is not feasible to pop up questions in every possible place but there should be ways to determine when it is most appropriate and meaningful to generate a question. There is a possibility to let the user to select the variables or expression types that should generate

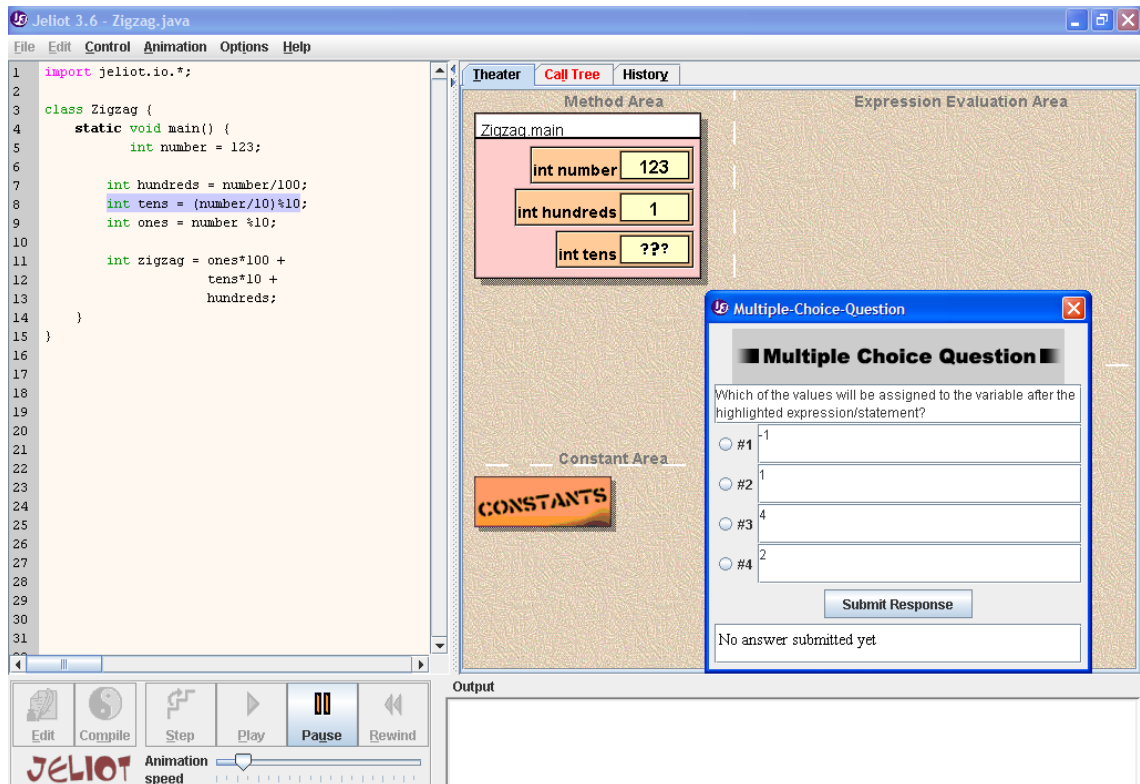


Figure 2: An example question generated by Jeliot 3.

questions and thus focus the support for learning on the selected concepts. Similarly to Proplets and WadeIn, we could use the performance data of the user as a basis for the adaptation of the question generation, visualizations and explanations as described in (Lin et al., 2006).

Moreover, there should be a possibility for a teacher to manually create questions for a certain program in order to allow the use of question generation for testing. This can be achieved with the package that we use to display and save the question information, because it provides a file format for manual question specification (Rößling and Häussge, 2004).

## 5 Conclusion and Future Work

We have presented a way to automatically generate prediction type questions during a program visualization automatically and a proof of concept implementation of it. We have also presented different types of questions that could be automatically generated with the same framework and ways to determine when those questions should be raised in order to support different ways of learning and testing.

As a future work, we plan to implement the proposed question types and test their usability. We also plan to evaluate the use of question answering both during individual as well as collaborative learning of programming concepts and programming.

## Acknowledgments

I want to thank Erkki Sutinen and Moti Ben-Ari for their helpful comments and guidance during the research work, and Andr Moreno and Roman Bednarik for fruitful discussions related to the topic of this paper.

## References

- Ronit Ben-Bassat Levy, Mordechai Ben-Ari, and Pekka A. Uronen. The Jeliot 2000 program animation system. *Computers & Education*, 40(1):1–15, 2003.
- Peter Brusilovsky and Tomasz D. Loboda. WADEIn II: A Case for Adaptive Explanatory Visualization. In *Proceedings of The Eleventh Annual Conference on Innovation and Technology in Computer Science Education*, Bologna, Italy, 2006.
- Michael D. Byrne, Richard Catrambone, and John T. Stasko. Evaluating animations as student aids in learning computer algorithms. *Computers & Education*, 33(4):253–278, 1999.
- Garrett Dancik and Amruth Kumar. A Tutor for Counter-Controlled Loop Concepts and Its Evaluation. In *Proceedings of Frontiers in Education Conference (FIE 2003)*, pages T3C–7–12, Boulder, CO, USA, 2003.
- Chris Evans and Nicola J. Gibbons. The interactivity effect in multimedia learning. Accepted to *Computers & Education*, 2006.
- Chris 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, 2002.
- Taiyu Lin, Andrés Moreno, Niko Myller, Kinshuk, and Erkki Sutinen. Inductive Reasoning and Programming Visualization, an Experiment Proposal. In *Proceedings of the Fourth International Program Visualization Workshop*, Florence, Italy, 2006.
- Andrés Moreno, Niko Myller, Erkki Sutinen, and Mordechai Ben-Ari. Visualizing Program with Jeliot 3. In *Proceedings of the International Working Conference on Advanced Visual Interfaces, AVI 2004*, pages 373–380, Gallipoli (Lecce), Italy, 2004.
- Thomas L. Naps. JHAVÉ – Addressing the Need to Support Algorithm Visualization with Tools for Active Engagement. *IEEE Computer Graphics and Applications*, 25(5):49–55, December 2005.
- Thomas L. Naps, James R. Eagan, and Laura L. Norton. JHAVÉan environment to actively engage students in Web-based algorithm visualizations. In *Proceedings of the thirty-first SIGCSE technical symposium on Computer Science Education*, pages 109–113, New York, NY, USA, 2000. ACM Press.
- 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 *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, pages 131–152, New York, NY, USA, 2002. ACM Press.
- Guido Rößling and Gina Häussge. Towards Tool-Independent Interaction Support. In Ari Korhonen, editor, *Proceedings of the Third International Program Visualization Workshop*, pages 110–117, Warwick, England, 2004.

# Program and Algorithm Visualization in Engineering and Physics

Michael P. Bruce-Lockhart, Theodore S. Norvell

*Memorial University of Newfoundland*

Yianis Cotronis

*University of Athens*

mpbl@engr.mun.ca

## Abstract

We report here on our experiences using a program animation tool, the Teaching Machine, for program and algorithm visualization for engineering and physics students at two universities. and the University of Athens, where it was adopted in 2005 to teach Physics students.

## 1 Introduction

We report here on our experiences using the Teaching Machine (TM) for program and algorithm visualization for Engineering and Physics students at two different sites: Memorial University, where it has been used since 1999 for teaching engineering students, and the University of Athens, where it was adopted in 2005 to teach Physics students.

The outline of the paper is as follows: In Section 2 we discuss the problem of what should be modelled in a program animation system. Section 3 is a short introduction to the TM. Sections 4 and 5 are respectively experience reports from Memorial University and the University of Athens. Section 6 presents a summary.

## 2 The Modeling Problem

Given a system  $T$ , Norman (1983) defined  $M(T)$  as the mental model held by particular user of that system and  $C(T)$  as a conceptual model, a “tool for the understanding or teaching of”  $T$ . Norman makes it clear  $T$  represents a physical system (in his paper a calculator). Yehezkel et al. (2004), in introducing EasyCPU, pointed out that if one replaces the actual system with a learning model  $L(T)$ , the  $M(T)$  that a student arrives at may be different from the one arrived at if they had interacted with the original system. In as much as EasyCPU represents an Intel 80x86,  $T$  is still a physical system. In teaching early programming courses, what is the  $T$  of which we desire students to develop an effective mental model?

In developing the TM (Norvell and Bruce-Lockhart, 2000, 2004), we thought about that problem a lot. Working with weaker students in the lab, we found a lot of superstitious behaviour: they would throw lines of code at a problem as if they were spells, with little understanding of what they meant. We found ourselves preaching that every line of code had a specific meaning and purpose, that it was an instruction to a machine. Developing in the students an effective mental model of that machine goes a long way toward teaching them to reason about the code they write. That machine,  $T$ , we were programming (and which we wanted the students to understand) was not really a computer, at least in the conventional sense. Consider the following simple C++ code:

```
int x = 5; int y = 12; int z; z = y/5 + 3.1;
```

In the language of programming, we say, there are four *instructions to be executed*. Instructions to what and to be executed by what?  $T$  of course, but  $T$  is certainly not the hardware. The first three “instructions” are actually to the compiler. We view them as request for allocation of memory, in the stack if they are internal declarations, in the static store if external. The fourth is a minefield. There is a truncation and two automatic type conversions. If you really

want students to understand it you, need to be able to interpret the expression and see the conversions, *but these are normally inserted by the compiler*. CPU operations include fetching the value for *y* (whether in a register or memory), carrying out the various calculations, and writing the final value back to *z*.

We define *T* to be the abstract machine to which we are giving source-level instructions. That is, *T* is largely defined by the language; it is an abstraction combining aspects of the computer, the compiler, and the memory management scheme.

The TM was then designed as a means to show the students how this machine worked, so they could write programs to control it.

### 3 The Teaching Machine

The TM has been described elsewhere (Norvell and Bruce-Lockhart, 2000, 2004) so we will review it only very briefly. The TM interprets source programs in C++ or Java while displaying visualizations of the program and the abstract machine state: the current expression under evaluation, the memory (stack and heap), the symbol table, and a console for I/O.

Before the TM we were using debuggers to good effect. We believe, as do others (Cross et al., 2002), that debuggers are a powerful tool for visualization on they're own; so we built the TM on the metaphor of a debugger. Since we expected students to continue to use debuggers in the lab, it eases the burden on them of learning two tools.

As of 2006 the TM supports C++ and Java. For each language, the model consists of a compiler, an interpreter, and a model of the abstract machine's state. The compiler translates source code to a high-level machine code that is then interpreted. Using a high-level machine code for an abstract machine allows us to preserve such information as the tree-structure of expressions, the tree structure of the data, and the data-types of each data object.

The Teaching Machine reads in standard C++ or Java files, generally prepared using conventional tools. There are some restrictions on what it can handle, as neither compiler is a complete implementation. For example, we have implemented neither threads in Java nor templates in C++. Nevertheless, the subsets that are supported are large, standards conforming, and work well in the courses in which the TM has been used.

The source files are displayed in a source window where they may be stepped using standard debugger controls. There are separate windows for each type of memory store—in C++, static store, stack, and heap. There is a window that displays expressions, with buttons to allow them to be stepped-through separately. This display, called the Expression Engine, is similar to the Expression Evaluation Area of Jeliot 3 (Moreno et al., 2004). There is a simple display for console input and output. Finally there is a Linked View, which automatically generates graphical depictions of data structures, in a manner similar to that of the LJV tool (Hamer, 2004). All views evolve dynamically as the program is executed and all execution steps can be undone and replayed.

### 4 Experience at Memorial University

At Memorial University, the TM is used in a three course stream in Electrical and Computer Engineering: Structured Programming (ENGI-2420), Advanced Programming (ENGI 3891), and Data Structures (ENGI-4892). ENGI-2420 teaches the basics of programming to all Engineering students, while 3891 emphasizes the use of classes and introduces pointers and heap allocation. ENGI-4892 emphasizes recursion and linked structures such as linked lists and trees.

Use of the TM in 3891 evolved through two transitions:

- From 1999 to 2001, we continued to use our old lecture transparencies but moved to the TM to illustrate specific points of interest. The TM was used more than the debugger, since it could show constructs the debugger could not. We spent less time at the

Cohort	ENGI-3891 2001					ENGI-3891 2002					ENGI-3891 2004				
Response	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1
2. (effectiveness vs. manual means)	31	49	12	4	2	48	26	7	2	0	59	35	4	2	0
3. (effectiveness vs. computer means)	10	65	16	2	2	33	45	10	2	0	51	37	4	0	0
4. (understanding of examples)	16	65	16	2	0	38	45	10	2	0	55	43	2	0	0
12. (understanding of examples on-line)	11	56	22	11	0	30	48	13	9	0	35	41	12	0	6

**Table 3:** Survey results. Results in percentage of respondents.

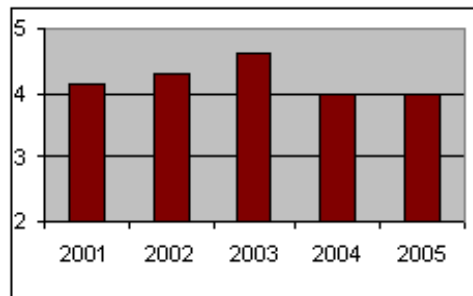
blackboard drawing and spent more time discussing examples with the students. Using the TM, however, drew us to different examples. Its use was subtly changing what we taught. We found our slides and examples rapidly getting out of sync.

- In 2002, in order to provide a fully integrated experience in the lecture and to give students access to interactive notes afterwards, we took advantage of the fact that the TM was a Java applet to integrate it directly into the notes, which were rewritten in HTML. We realized that we had to make it as easy as possible for instructors to author such interactive notes, so we created WebWriter++ (Bruce-Lockhart, 2001), a set of javascripts with a couple of additional applets. The smooth integration considerably increased the amount of lecture time spent running examples in the TM. ENGI 4892 moved to the same format the next academic year, with the Linked View being used predominantly.

Student surveys have been carried out from time to time. Table ?? shows the results for three consecutive cohorts of ENGI-3891 students. The 2001 cohort (51 surveys out of 55 students) got the TM in standalone mode with fewer examples. The 2002 cohort (42 of 63) got integration, but notes were developed on the fly. The 2003 cohort was actually surveyed at the end of 4982 in 2004, when separate surveys were given to the same group of students (26 of 44 responded of whom 23 had done 3891); because the results were so similar the two were combined. For each question, 3 represents a neutral response, with 5 and 4 being very and somewhat positive, and 1 and 2 being very and somewhat negative. The table reports percentages with missing values representing either ‘not applicable’ or no response at all. All three cohorts had either not seen the TM at all or had seen very limited use of it in their first programming course, ENGI-2420, so were able to compare its effectiveness vs. other means (questions 2 and 3). Question 4 asked about the effect of the TM on the student’s understanding of the examples it was used on, while question 12 asked about effect on understanding for those students who ran examples themselves on-line (only 9 of 51 in 2001, when few examples were available on-line; 23 of 42 in 2002 and 34 of 49 in 2003/2004). As can be seen, the results become more positive as the integration with the course notes becomes tighter.

Teacher ratings were independently surveyed. Figure ?? shows the results for 3891 through the last five years, using the same instructor. 2003 represents the mature version of the course as currently taught. Long regarded by students as one of the most difficult in Electrical and Computer Engineering, it was allotted four hours teaching hours a week, instead of three. Student satisfaction rose sufficiently high that in 2004 the Faculty of Engineering decided to cut the course back to three hours a week. Approvals dropped back to historic levels, but the course was delivered using 25% fewer lectures, while covering the same material, with no effect on outcomes. With this as evidence, the TM was (and is) regarded by the Faculty as successful in improving the delivery of 3891. A decision to move it into ENGI-2420 in 2004 was made on that basis.

The impact of the TM in 2420 is hard to measure yet due to confounding factors such as a doubling of the section size to over 200 and a change to the curriculum. In trying to make his



**Figure 1:** Quality of instructor ratings for 3891. 2=fair, 3=good, 4=very good, 5=excellent.

notes and lectures as understandable as possible in such a large venue, the instructor (Bruce-Lockhart) fell into a classic trap. The students got into the habit of letting the instructor do all the work. In the taxonomy developed at the 2002 Programming Visualization Workshop (Röbling et al., 2004) we collectively got stuck at stage 2: viewing. Although the TM had been designed to allow students to study examples on their own, and the students of 3891 and 4892 appear to be doing just that, the message hadn't been properly delivered to 2420 students. As an interim measure, the entire class was taken to the lab in sections and walked through fresh examples of pass-by-value vs. pass-by-reference. The objective was to get them to use the TM themselves and take them to the point where they could predict what the next step would produce (stage 3, responding). It is too early to tell what the real impact is yet, but numerous students approached the instructor afterwards to express approval of the technique and to suggest it be a regular feature of the course, but much earlier in the term.

## 5 Experience at Athens

At the University of Athens one of the authors has been teaching the subject of Principles of Programming Languages and Programming Techniques for the past six years, as part of the curriculum of a joint M.Sc. degree between the Departments of Physics and Informatics & Telecommunications. It is a well-known and highly respected conversion programme in which mostly Physics graduates from a number of Greek Universities enrol. The course is divided into three parts: the first and second parts discuss syntax (regular expressions, BNF) and computation aspects of programming languages, respectively; the third part discusses programming techniques (structured programming, top down program development).

The TM was introduced the fall 2005 semester augmenting teaching for the second and longest part, the computational aspects. Our teaching approach is analytical rather than empirical, aiming for exposing students to concepts recurring in programming languages, rather than teaching a specific language; students separately follow a two hour per week course on C. The teaching material for the past three years has been based on the six computation chapters of Part II of Fisher and Grodzinsky (1993), which fits well with our teaching approach. Students were given a number of small course-work exercises, which aim to practice and establish concepts.

Although students were satisfied with the course and appreciated the importance of it, they could not complete all exercises and they just “passed” the final written paper. A number of reasons contributed to this situation, the most significant being the difference between the cultures of Physics and Computer Science: most physics students consider programming “simple” when compared to laws, theories and models of the world. Consequently, as they are not easily convinced that a more complex programming world exists, they keep their misconceptions on how programs work.

We were convinced that a visualizing tool would help to clarify concepts and bring out their misconceptions, a tool which would visualize program execution above the level of a compiler. Having taught compilers for a number of years, we did not want a tool for visualizing compilers, but acting as the operational semantics for languages abstracting away compiler details. Searching the Web, we came across the TM, which proved to be the tool that we needed, at least regarding the abstraction level. The TM displays memory (static, stack, and heap), in which values of objects are depicted in decimal and binary, showing the address and space they actually use; and a stack implementation of a symbol table. TM has three levels of visualizing program execution (procedures, statements, and expressions) all with backtracking. It also has a Linked View (objects and pointers to them are actually depicted as graphs), which was really an unexpected and valuable feature.

We found that the TM did not impose at all on our teaching approach, but rather complemented it. We implemented tens of small programs demonstrating programming concepts, in the spirit of Fisher and Grodzinsky (1993). The TM could depict each point in focus. We were able to demonstrate principles, consequences, and some “not expected” behaviour. Let us mention a few notable examples. The (binary) representation of float and double was used to show that  $(1.0/n) * n$  may not be 1 (a really surprising fact for physicists), or that we could “compute”  $\sin(x)$  and get a value greater than  $10^8$ . It was demonstrated that the concept most students had for one-to-one association of variable names to objects in memory is a misconception. We relied on the expression execution of the TM to demonstrate that expressions in C++ may be undefined and return strange results. We demonstrated the differences between parameter passing by (pointer) value and by reference; a misconception found not only among students but also in a number of programming books.

Most of the programs were demonstrated in class, but, as students had a copy of the TM, they could run their own programming experiments at home and very frequently came in class with questions.

We believe that most, if not all, learning styles as proposed by the Felder and Silverman (1988) may benefit from the use of TM. Active learners (AL) tend to retain and understand information best by doing something active with it; reflective learners (RL) prefer to think about it quietly first. AL may try their own programming “experiments” while RL have the opportunity to review the material shown in class using the TM.

Sensing learners tend to like learning facts while intuitive learners (IL) often prefer discovering possibilities and relationships. The TM only helps with IL but everybody is sensing sometimes and intuitive sometimes.

Visual learners (ViL) remember best by demonstrations. Clearly the Teaching Machine is strongly oriented toward ViL. Verbal learners (VeL) get more out of words-written and spoken explanations. Although VeL may benefit less, it is accepted that everyone learns more when information is presented both visually and verbally.

Sequential learners (SL) tend to gain understanding in linear steps, with each step following logically from the previous one; global learners (GL) tend to learn in large jumps, absorbing material almost randomly without seeing connections and then suddenly “getting it.” Textbooks and classes essentially benefit SL as each teaching topic focuses on a specific aspect of programming. As the TM is the unique tool for demonstrating all programming topics, GL tend to benefit as well since the “whole picture” is there at any time.

## 6 Summary

The TM has proved to be useful in the delivery of a variety of courses. Its ability to handle both physical and abstract models has made it successful for helping engineering and physics students understand the abstract machine defined by the programming language. In one case it has helped reduce the resources required to teach a course. Its deep modeling of how a compiler and a processor interact allowed it to be used at Athens in ways the original designers



had not anticipated.

Does it help students develop effective mental models:  $M(T)$  approximating the instructors  $C(T)$ ? Anecdotal evidence supports this conclusion: one anonymous response to the Memorial surveys is telling. “*The Teaching Machine provides a great means of visualizing the internal mechanisms and operations of software. When I write code, I visualize how the Teaching Machine would translate it.*”

Finally, we have come to realize that the TM is a solid platform for new developments. It is easy to use, robust, flexible, and reasonably complete. Its data model is easily extendable to more powerful, algorithmic visualizations. By providing it with specialized input and output plugins we can provide better visualizations and better interactivity; we expect that adding such plugins will be far easier than developing specialized visualizers from scratch.

## References

- Michael P. Bruce-Lockhart. WebWriter++: A small authoring aid for programming. In *Proceedings of the Newfoundland Electrical and Computer Engineering Conference*, St. John's, Newfoundland, 2001.
- James H. Cross, T. Dean Hendrix, and Larry A. Barowski. Using the debugger as an integral part of teaching CS1. In *32nd ASEE/IEEE Frontiers in Education Conference*, pages F1G-1–F1G-6, 2002.
- Richard M. Felder and Linda K. Silverman. Learning and teaching styles in engineering education. *Engineering Education*, 78(7):674–681, 1988. With preface <http://www.ncsu.edu/felder-public/Papers/LS-1988.pdf> (2002).
- A. E. Fisher and F. S. Grodzinsky. *The Anatomy of Programming Languages*. Prentice-Hall, 1993.
- John Hamer. A lightweight visualizer for Java. In *Proceedings of the Third Program Visualization Workshop, Research Report CS-RR-407, Department of Computer Science, University of Warwick*, pages 54–61, 2004.
- Andrés Moreno, Niko Myllerand Erkki Sutinen, and Mordechai Ben-Ari. Visualizing programs with Jeliot 3. In *Advanced Visual Interfaces*, 2004.
- Danald A. Norman. Some observations on mental models. In Dedre Gentner and Albert L. Stevens, editors, *Mental Models*, chapter 1, pages 7–14. Lawrence Erlbaum Associates, 1983.
- Theodore S. Norvell and Michael P. Bruce-Lockhart. Taking the hood off the computer: Program animation with the Teaching Machine. In *Canadian Electrical and Computer Engineering Conference*, Halifax, Nova Scotia, May 2000. Available at <http://www.engr.mun.ca/~theo/Publications>.
- Theodore S. Norvell and Michael P. Bruce-Lockhart. Teaching computer programming with program animation. In *Canadian Conference on Computer and Software Engineering Education*, Calgary, Alberta, 2004. Available at <http://www.engr.mun.ca/~theo/Publications>.
- Guido Rößling, Felix Gliesche, Thomas Jajeh, and Thomas Widjaja. Enhanced expressiveness in scripting using AnimalScript 2. In *Proceedings of the Third Program Visualization Workshop, Research Report CS-RR-407, Department of Computer Science, University of Warwick*, pages 10–17, 2004.
- Cecile Yehezkel, Mordechai Ben-Ari, and Tommy Dreyfus. Inside the computer: Visualization and mental models. In *Proceedings of the Third Program Visualization Workshop, Research Report CS-RR-407, Department of Computer Science, University of Warwick*, pages 82–85, 2004.

# Integrating Algorithm Visualization Systems

Ville Karavirta

*Helsinki University of Technology*

*Department of Computer Science and Engineering*

*Helsinki, Finland*

`vkaravir@cs.hut.fi`

## 1 Introduction

Helping students to understand difficult pieces of code remains a challenge in Computer Science education. By providing a view of the code on a higher level of abstraction, Algorithm Visualization (AV) aims at making the code more understandable. However, there is still speculation about its effectiveness in learning (Hundhausen et al., 2002). Recent studies indicate that to be *educationally effective* (*i.e.* aid students' learning) algorithm visualizations need to be more than passive animations; they must require users to interact with the animation (Hundhausen et al., 2002). This interaction can, for example, require the user to respond to multiple-choice questions during the animation or to construct an algorithm animation by using a visualization.

However, algorithm visualizations have not been widely adopted in teaching. One of the main reasons is the time and effort required to find and adopt such a system (or ready-made examples) and to design, create, and integrate the visualizations (Naps et al., 2003). Thus, teachers need to have easier ways to find and produce visualizations that provide the needed interaction. For meeting these needs, several AV systems have been developed. Some of the systems that are currently being developed are ANIMAL (Röbling and Freisleben, 2002), JAWAA (Akingbade et al., 2003), and MatrixPro (Karavirta et al., 2004). These systems provide different ways to create the animations as well as different types of interaction. Thereby, integrating these systems could be beneficial by, for example, allowing the teacher to select the system based on the level of interaction it provides.

One way to achieve this integration is by defining a common language for the algorithm visualization systems. This was the topic of one of the working groups at the Conference on Innovation and Technology in Computer Science Education (ITiCSE) 2005. The working group aimed at defining XML specifications for the various aspects of AV. The working group's report (Naps et al., 2005) provides examples of these specifications as well as guidance on how to use the specifications in existing visualization systems. This paper refers to this group and its work as ITiCSE XMLWG or as working group.

The working group did a lot of good work in coming up with XML specifications for different aspects of AV. However, a lot of the work remains on the level of examples instead of concrete specifications. One idea of the working group was to continue the work in the future. The work presented in this paper builds on the specifications and examples provided by the working group and introduces more concrete specifications in the form of a language, XAAL (eXtensible Algorithm Animation Language), defined to be used in data exchange between algorithm animation systems. In addition, this paper introduces a set of tools that will hopefully benefit AV system developers. The aim of the tools is to allow data exchange between the current AV systems as well as support other useful export formats from the systems.

In the following, Section ?? briefly introduces the main features of the language. For a more specific documentation, see the XAAL website at <http://www.cs.hut.fi/Research/SVG/XAAL/>. Section ?? in turn describes the tools that support the data exchange. Finally, Section ?? discusses the usefulness of such a language and tools as well as looks into the future.

## 2 eXtensible Algorithm Animation Language

XAAL (eXtensible Algorithm Animation Language) is defined as an XML (Extensible Markup Language) language by specifying the allowed document structure. XML makes it easy for software to process data using the multitude of different tools and architectures available today. In addition, transforming XML documents to different XML formats or text is relatively simple and flexible using XSLT (Extensible Stylesheet Language Transformations).

An important aspect of defining the language has been the need of transformation between different existing algorithm animation languages. To find out the requirements for a language used in data exchange, a survey of the existing description languages was made (Karavirta, 2005). Based on this survey, a taxonomy of algorithm animation languages was defined (Karavirta, 2005). This taxonomy indicates that the features of an algorithm animation language can be roughly divided into three categories: data structures, graphical primitives, and animation. Problems arise when trying to exchange data between tools that have different approaches to AV. For example, ANIMAL (Rößling and Freisleben, 2002) describes animations using mostly graphical primitives, whereas MatrixPro (Karavirta et al., 2004) uses only data structures. These different approaches have been taken into account when defining XAAL.

In this section, we will briefly introduce the most important features of XAAL. The reader should note that this text is merely an overview of the language. For a more detailed discussion, see Karavirta (2005) and for the actual XML schemas, see the XAAL website.

### 2.1 Graphical Primitives

The basic graphical components that can be composed to represent arbitrarily complex objects (e.g., a tree data structure) are graphical primitives. The graphical primitives in XAAL are as specified by the ITiCSE XMLWG (Naps et al., 2005), where the following have been defined: point, polyline, line, polygon, arc, ellipse, circle and circle-segment, square, triangle, rectangle, and text.

Other features specified by the working group are the definition of reusable *shapes* from the graphical primitives and changing the visual appearance of the graphical primitives using *reusable styles*. Both of these are intended to aid the creation of more complex primitives. Listing ?? in Appendix ?? gives an example of a shape definition that uses some graphical primitives.

### 2.2 Data Structures

XAAL supports the usage of data structures to specify the visualizations, lowering the effort needed to produce them. The set of structures is basically the same as, for example, in JAWAA (Akingbade et al., 2003): array, graph, list, and tree. The content of these data structures is described using nodes (or indices in case of an array) and edges connecting the nodes. The structures can form arbitrarily complex hierarchies. On the other hand, the simplest kind of structure can be a string or a number. Moreover, to support the different approaches of existing algorithm animation languages, all structures support an optional graphical presentation indicating how the structure should be visualized. Listing ?? in Appendix ?? shows an example of an array definition.

### 2.3 Animation

A crucial part of the algorithm animation language is the animation functionality. In the following, we will introduce the elements available in XAAL for defining the animations. The animation operations in XAAL have been divided in three groups: graphical primitive transformations (for example, rotate), elementary data structure operations (for example, replace), and abstract data structure operations (for example, insert).

The operations for manipulating graphical primitives use the format specified by the ITiCSE XMLWG (Naps et al., 2005). These include operations such as show, hide, move, rotate, and scale. Listing ?? in Appendix ?? is an example of a rotate operation.

The elementary data structure operations available in XAAL are create, remove, replace, and swap. To be consistent with the structure definitions, every operation can have an optional part describing how the operation should be visualized using graphical primitive animation.

Abstract data structure operations are operations that depend on the semantics of the target structure. In XAAL, the available operations are insert, delete, and search. As with the elementary data structure operations, these operations can have an optional part describing how the operation should be visualized using graphical primitive animation. In addition, for systems that do not know the semantics of the abstract operation, the behavior can be optionally described using elementary data structure operations. Listing ?? in Appendix ?? shows an example of a delete operation.

## 2.4 Schema Specification

We have defined an XML Schema for XAAL. To make the language more modular, we have divided the schema into several XML Schema documents. This kind of modularity makes it possible to more easily change or reuse some parts of this language in other languages.

## 3 Implementation

Our objective was to implement XAAL in a modular way that could be useful for the other AV system developers in their aims at implementing the ITiCSE XMLWG specifications. In this section, we will introduce two different processing pipelines usable to add XAAL support into existing algorithm animation systems. In addition, we will briefly describe a prototype implementation of XAAL and several transformations.

### 3.1 Object hierarchy

The first processing solution is an architecture discussed by the ITiCSE XMLWG to implement its specifications. The basic idea is to have one XAAL *parser* that can be used by multiple algorithm animation systems. This parser generates a Java *object hierarchy*. In addition, there is a part of software that can *serialize* the object hierarchy to a XAAL document.

The existing AV systems can then implement *adapters* that convert the XAAL object hierarchy into an animation in that particular system. By implementing a *generator*, the existing systems can generate the object hierarchy and serialize it as XAAL.

This solution requires no major modifications to the existing systems, and thus the workload of implementing XAAL remains fairly low. Another advantage is that the document has to be parsed only once. There is, however, one extra step in the process compared to the direct approach of parsing the XAAL document directly into the AV system. However, implementing a XAAL parser for each system would not be sensible, and thus the extra processing is not considered a major issue.

### 3.2 XSLT Processing

Another way to integrate XAAL with existing systems is to transform it to a format of the target system using XSLT. This way provides a simple solution to import XAAL documents into existing AV systems that have an algorithm animation language. It can also be used to export different formats from a system that supports XAAL.

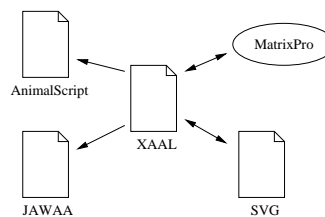
The benefit of this approach is that the XSLT stylesheets are quite simple to write for a person who knows the syntax of XAAL, the target language, and XSLT. Moreover, the target system need not be changed at all. This makes it possible to integrate XAAL with systems

that are not open-source. On the negative side, this approach requires parsing of three files: the stylesheet, the XAAL document, and the generated AV system document.

### 3.3 Prototype Implementations

We have implemented the XAAL object hierarchy with a parser and a serializer. The implementation is on a prototype level, and not all elements are fully supported.

We have also implemented various adapters and generators between XAAL and other algorithm animation languages. Thus, we already have several different formats available for the same animation. The current selection of formats is presented in Figure ?? . In addition to the formats in the figure, the systems allow some other export formats as well. For example, ANIMAL can be used to export QuickTime movies and MatrixPro exports TeXdraw illustrations. Thus, we could create, for example, a QuickTime movie from an SVG animation.



**Figure 1:** Prototype format transformations implemented. The arrows represent the direction of the transformation. The ellipse represents a AV system, whereas the other are documents.

The prototypes do not implement all the features of the new language. For example, graphical primitive animation is currently not implemented, although it can be considered very important when exchanging information between systems like ANIMAL or JAWAA. Still, the prototypes enable us to transfer data between AV systems. Although the data currently is only static visualizations, we feel that we have demonstrated that this kind of approach is suitable for the problem at hand and it would be worthwhile to complete the implementation and include other languages in the future.

## 4 Discussion

In this paper, we have introduced an XML language for describing algorithm animations. The language is based on the specifications and examples of ITiCSE XMLWG. In addition, we have described a set of tools for exchanging data between algorithm animation systems. In the following, we will discuss what use this work can have for participants of the ITiCSE XMLWG, developers of AV systems, and teachers wanting to use AV.

For the participants and the continuing work of the working group, the specifications and tools presented in this paper provide concrete tools aiding the adoption of a common language. As the XML schema of XAAL is modular, the other participants of the group can take advantage of the specifications using the working group's definitions.

Developers of AV systems can implement different import and export formats for the existing systems with a reasonable effort. This can improve the applicability of the systems and thus promote the usage of AV systems.

Teachers can benefit from this in two ways. First, since the AV systems have different approaches to the animation creation, a teacher can combine the good features of multiple systems in the process of creating animations. Second, ready-made animations can be used in multiple systems and one system can use animations created for another system, thus expanding the selection of ready-made animations. In addition, the teacher can select the system based on the level of interaction it provides.

## 4.1 Future Visions

We have numerous improvements and ideas for the future of the language, and here we will present some of the most interesting ones.

The most urgent requirement is to finish the prototype implementations. A natural continuation for this is to support new formats. These new formats could be other algorithm animation languages, graph description languages, or something more different like OASIS OpenDocument, programmable graphics canvas element of HTML, or Macromedia Flash. Furthermore, as can be seen from Figure ??, many of the transformations are only available in one direction. A future challenge is to be able to generate XAAL documents with other systems, or alternatively, parse/transform other formats into XAAL.

The language could be extended to include programming concepts and thus allow the definition of algorithms and program visualization. There is also a need for more complete metadata and semantic information to be included in the algorithm visualizations allowing more flexible searching from algorithm animation repositories. This would lower the time needed for teachers to search for suitable ready-made examples. Furthermore, we need to include and implement the interaction specification of the ITiCSE XMLWG.

For this to happen, it would be important for other educators to join in the development of common tools for the whole community. Having a group of researchers developing open source tools for AV systems could allow the development of high-quality tools easily adoptable by teachers.

## 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 SIGCSE technical symposium on Computer science education, SIGCSE'03*, pages 162–166, Reno, Nevada, USA, 2003. ACM Press. ISBN 1-58113-648-X. doi: <http://doi.acm.org/10.1145/611892.611959>.
- Christopher D. Hundhausen, Sarah A. Douglas, and John T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages & Computing*, 13(3):259–290, June 2002.
- Ville Karavirta. XAAL - extensible algorithm animation language. Master's thesis, Department of Computer Science and Engineering, Helsinki University of Technology, December 2005.
- Ville Karavirta, Ari Korhonen, Lauri Malmi, and Kimmo Stålnacke. MatrixPro - A tool for on-the-fly demonstration of data structures and algorithms. In Ari Korhonen, editor, *Proceedings of the Third Program Visualization Workshop*, Research Report CS-RR-407, pages 26–33, The University of Warwick, UK, July 2004. Department of Computer Science, University of Warwick, UK.
- 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.
- 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. ISSN 0097-8418. doi: <http://doi.acm.org/10.1145/1113847.1113891>.


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.

## A Xaal Examples

```

1 <define-shape name="cat">
2   <circle-segment>
3     <center x="10" y="5"/>
4     <radius length="4"/>
5     <angle total="310" start="295"/>
6   </circle-segment>
7   <circle-segment>
8     <center x="10" y="15"></center>
9     <radius length="7"/>
10    <angle total="255" start="105"/>
11  </circle-segment>
12  :
13  <!-- specification of ears as lines and tail as arc -->
14 </define-shape>

```



Listing 1: Example of defining a shape, in this case, a cat.

```

1 <array indexed="false" size="7" orientation="horizontal">
2   <index index="0"><key value="A"/></index>
3   <index index="1"><key value="B"/></index>
4   <index index="4"><key value="C"/></index>
5 </array>

```

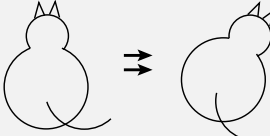
A   B   C

Listing 2: Example of an array definition.

```

1 <rotate degree="30" type="simple">
2   <object-ref id="catObj1"/>
3   <timing><delay s="2"/></timing>
4   <coordinate x="10" y="10"/>
5 </rotate>

```



Listing 3: Example of a rotation operation. In the example, it is assumed that `catObj1` is an instance of the shape defined in Listing ??.

```

1 <delete target="BST">
2   <key value="C"/>
3   <elementary>
4     <remove target="nodeC"/>
5     <remove target="edgeCA"/>
6     <replace target="edgeMC">
7       <edge from="nodeM" to="nodeA"/>
8     </replace>
9   </elementary>
10 </delete>

```



Listing 4: Example of delete operation. The figures show the structure before (on the left) and after (on the right) the deletion.

# A Framework for Generating AV Content on-the-fly

Guido Rößling, Tobias Ackermann  
Technische Universität Darmstadt  
Darmstadt, Germany

roessling@acm.org

## Abstract

A main reason for not adopting Algorithm Visualization (AV) for teaching or learning purposes is the time needed to find or generate “appropriate” content (Naps et al., 2003). This paper tries to remedy this by illustrating an easy to use and yet flexible framework for generating content on-the-fly - but tailored to the end user’s preferences.

## 1 Introduction

Most educators and students do not use AV content to improve their teaching or learning. The AV Working Group at ITiCSE 2002 (Naps et al., 2003) examined the underlying reasons for this and gave some guidance on how this could be addressed. In a pre-conference survey, the following major reasons for not adopting AV were given by the survey participants:

- 93% mentioned the “time required to search for good examples”,
- 90% mentioned the “time it takes to learn the new tools”,
- 90% complained about the “time it takes to develop visualizations”,
- 83% mentioned the “lack of effective development tools”, and
- 79% stated the time needed to adapt AV content “to teaching approach/course content”.

What can be done about this? We can try to make tools easy to use - but this may mean that the time to develop animations will increase. For example, ANIMAL (Rößling and Freisleben, 2002) and JAWAA2 (Akingbade et al., 2003) contain a visual editor for creating AV content by clicking in a GUI front-end. This makes content creation very easy, if also time-consuming. However, ensuring that the portrayed visualization accurately represents the underlying algorithm is difficult. Systems like *Jeliot* (Kannusmäki et al., 2004) require only the appropriate underlying code. However, they are often less prepared for adapting the contents of the visualization to the educator’s preferences or course materials.

In this paper, we describe our *generator framework* approach that offers a GUI front-end for user-tailored content and on-the-fly visualization generation.

## 2 The Generator Framework Approach From the End User’s Perspective

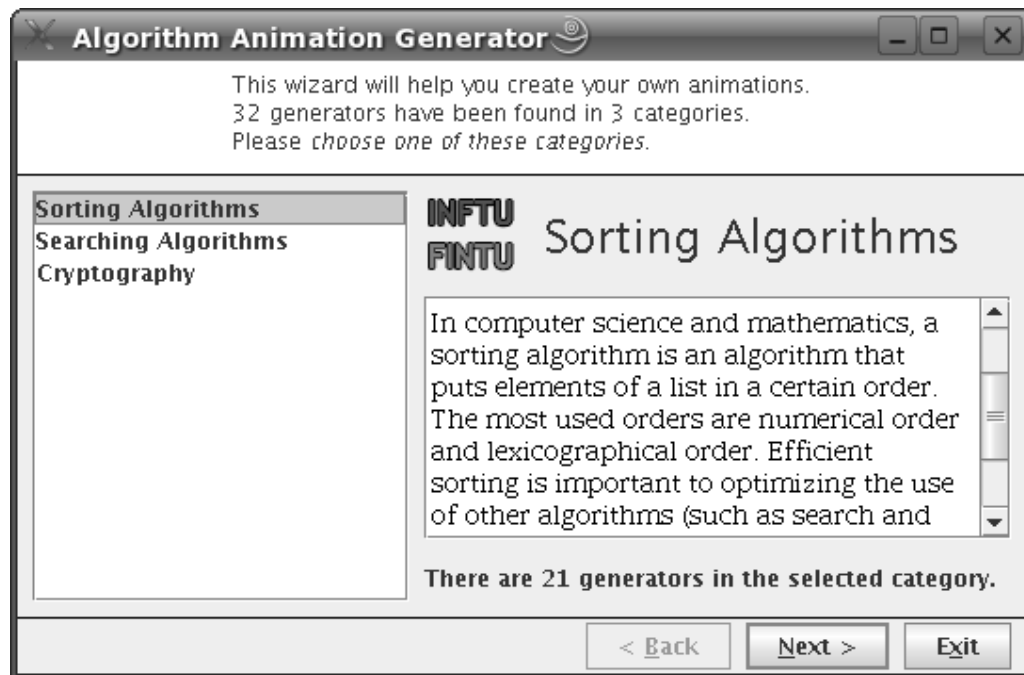
We wanted to design a component for easy content generation. The goal of this component was to address the user complaints described above: it simply takes too long to learn a new tool, develop a visualization, or adapt it to the user’s preferences. We wanted to address this by reducing most of these complaints:

- Gathering multiple generators in one place, preferably one of the established AV content repositories, makes searching for examples easier;
- The “time to learn” is reduced by using ANIMAL, a system with a relatively simple GUI;
- The “time to develop visualizations” is addressed by using prepared generators, which require just a few mouse clicks;



- As the generators shall create input based on the user’s settings, they effectively act as the wanted “effective development tool”;
- The “time to adapt content” is reduced by making input values and selected aspects of the visual layout adaptable to the user’s wishes.

How does this work? A set of *Generators* are responsible for generating AV content. Typically, each generator will generate a visualization for one concrete algorithm. For example, a *QuickSortGenerator* class may generate a visualization for Quicksort.



**Figure 1:** Generator Step 1: Selecting the topic area

The end user first starts up the tool, as shown in Figure ???. The top of the window informs the user about the tool and what steps can be taken next. The user initially sees a list of topic areas (in Figure ??, this is sorting, searching, and cryptography). Once an area is selected, the appropriate icon, title, and description are shown on the right side. At the bottom, the user is informed about the number of content generators available for this category.

After selecting a category, the user can select a concrete generator from a list (Figure ??). The text areas to the right contain a brief description of the generator, and sample output or an example of the underlying code. This is especially important for algorithms such as Quicksort, which exist in a large number of variations. However, only the variant used in the lecture may be helpful for both educators and students. Below these areas, the output format (here “asu” for ANIMAL’s ANIMALSCRIPT format) is shown.

After pressing *Next*, the user is led to the content adaptation step, as shown in Figure ??. Here, the user can edit the input values and selected visual properties. Once the user presses the *Next* button in this step, the framework will ask the user for a filename to which the content shall be stored. After this, the generator is activated and produces the appropriate output, for example, an ANIMALSCRIPT file for the ANIMAL AV system.

The user can then run the visualization system to show the newly generated visualization, or generate additional content by going back and modifying the values. We have also integrated the generator framework into the ANIMAL AV system, where it is activated by a menu item. Once the user has stored the file, it is automatically parsed by ANIMAL’s ANIMALSCRIPT parser and shown as the new visualization, as displayed in Figure ??.

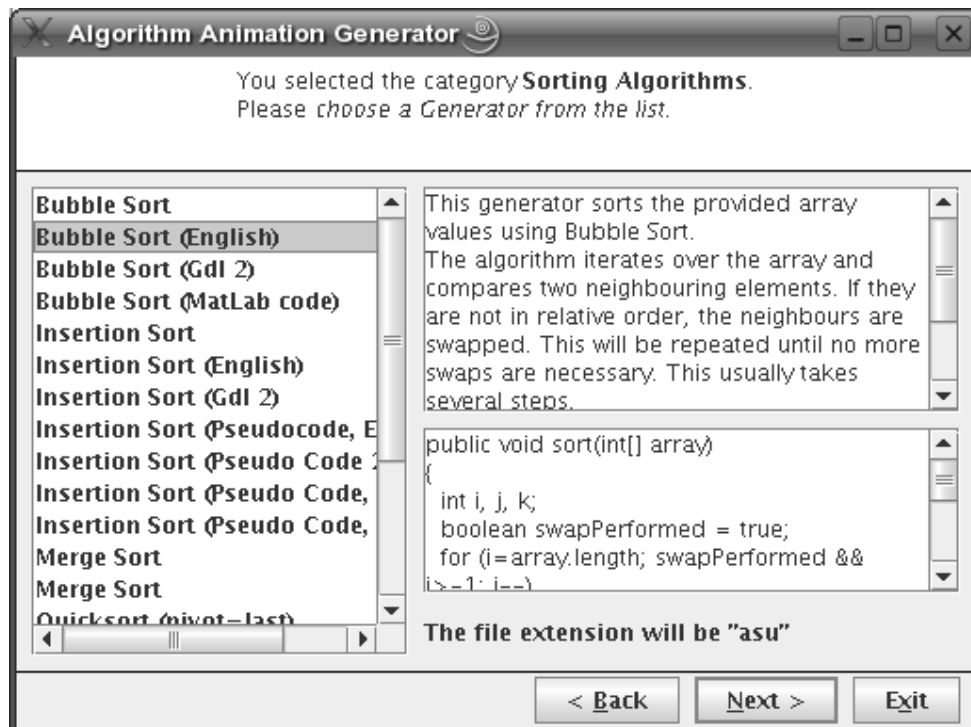


Figure 2: Generator Step 2: Selecting the algorithm

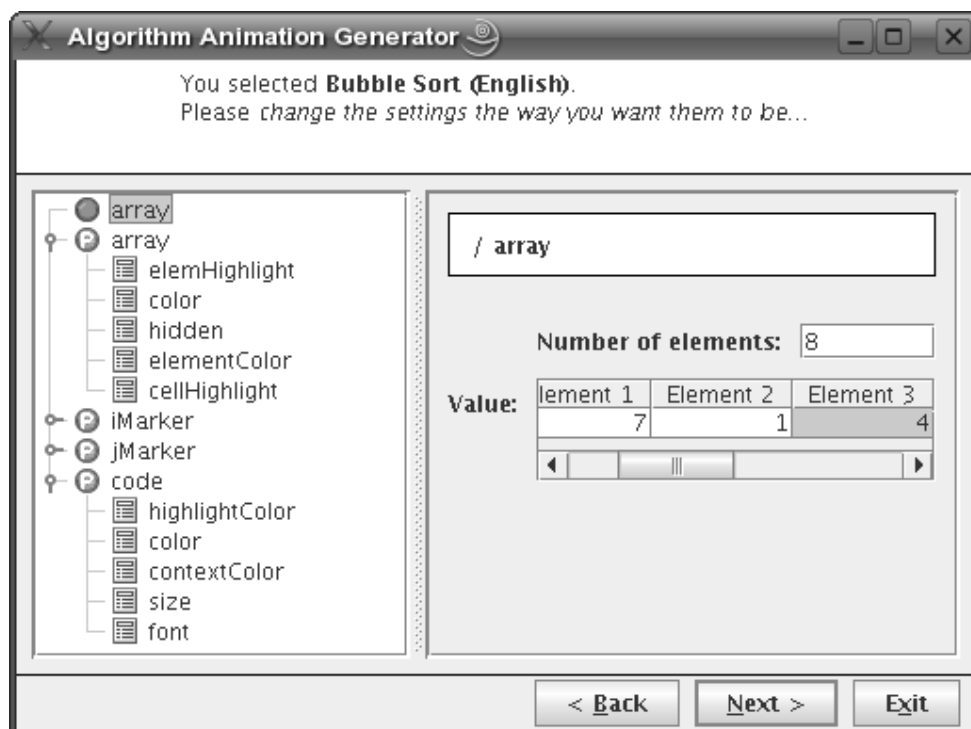


Figure 3: Generator Step 3: Adapting the input values and visual appearance

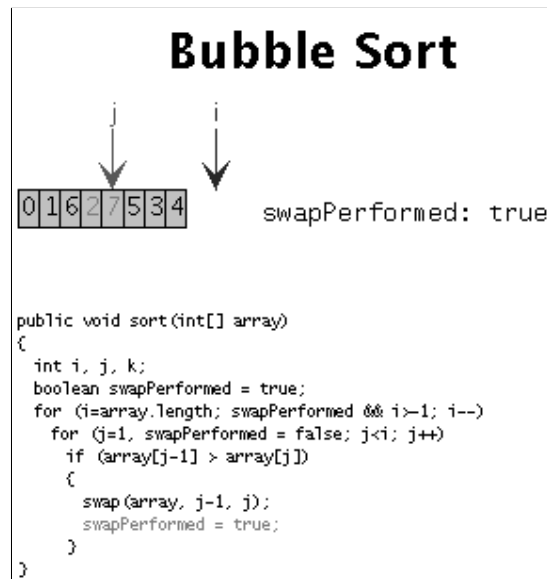


Figure 4: Generator Step 4: The resulting animation

End-users should therefore find it as easy as initially promised to generate their “personal” AV content, including input values and visual properties - assuming, of course, that appropriate content generators are available. In the next section, we will therefore examine the content generator’s point of view.

### 3 The Generator Framework from the AV Content Generator’s Perspective

AV content generator implementors have to perform the following simple steps to include a new generator in the generator framework:

1. Implement a generator class for the desired algorithm, placed in the *generatorImplementations* package. The generator has to implement the *generator.Generator* interface. This requires the implementation of the following simple *public* methods:

**GeneratorType getGeneratorType()** returns the type of algorithm generated by this algorithm. This information is used to determine the contents of the topic area list in Figure ???. The constructor of class *GeneratorType* expects a combination of the predefined constants for the different sorts of algorithms, e.g., *GeneratorType.GENERATOR\_TYPE\_SORT*;

**String getName()** returns the algorithm’s name, as shown in Figure ??;

**String getDescription()** returns the algorithm’s description (top right of Figure ??);

**String getCodeExample()** returns an example of the generator’s output (or the underlying algorithm’s code); as shown on the lower right side of Figure ??;

**String getFileExtension()** returns the file extension for the generator output, e.g., *asu* for ANIMALSCRIPT, as also shown in Figure ??;

**String generate(AnimationPropertiesContainer props, Hashtable prims)** has to generate the content and return it as a String object. The two parameters contain the set of visual properties and the set of user-defined primitive objects, e.g., *int* values or arrays.

The last method is really the only part that creates some amount of work. The other methods can typically be implemented in less than a minute each.

2. Ensure that the algorithm uses the values specified by the user, not hard-wired settings. This is done by checking that fixed values, e.g. *Color.black*, are replaced by accesses to the *AnimationPropertiesContainer* or *Hashtable* parameters of the *generate* method.

For example, the array defined by the user in Figure ?? can be accessed as

```
int[] myArray = (int[])prims.get("array");
```

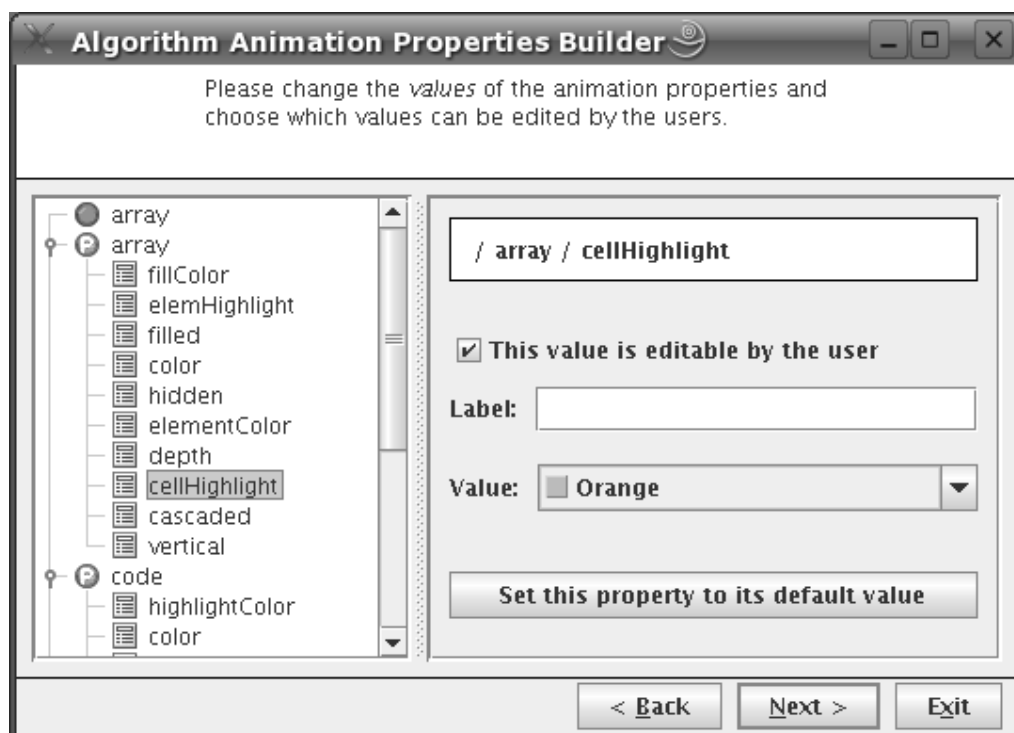
where *prims* is the name of the Hashtable parameter. Similarly, the *highlight* color of the associated code is accessed as

```
Color hlColor = (Color)props.get("code", "highlightColor");
```

where *props* is the *AnimationPropertiesContainer* parameter passed to the *generate* method. Note that “array” and “code” match the name of the array primitive and the code property set in Figure ??, respectively. “highlightColor” is the name of one of the properties for the “code” property set.

3. Specify the properties and primitives used by the algorithm. A simple GUI front-end can be used for this, as shown in Figure ?. This front-end lets the author select the set of primitives and properties used by the generator.

Each primitive and property has a default value which can be adjusted according to the author’s preferences. Additionally, the author can decide individually for each property whether this should be user-editable. Thus, the end user will be able to adapt anything between “nothing” and “everything” regarding the visual layout to his or her tastes.



**Figure 5:** The Animation Properties Builder front-end for content generation authors

Figure ?? shows the properties generation window. The first frame (not shown in the Figure) allows the content generation author to load an existing specification or create a new one. Each specification can contain a set of folders, primitive objects, and properties (indicated by the circled letter P). The editing front-end for primitives is identical to the one shown in Figure ?.

As shown in Figure ??, the author can specify whether a given entry – here, “cell-Highlight” – shall be editable by the end-user. When comparing Figures ?? and ??, it

becomes obvious that the author has deactivated user editing for a number of properties, such as *fillColor*, *filled*, *depth*, *cascaded*, and *vertical* for the “array” element.

Once the editing process is completed, the settings are stored as an XML file.

The implementation of the generator, together with the XML specification, drive the generation process shown in the first three Figures of this paper. From our experience, about 80%-95% of the time needed to implement an algorithm generator are tied to the *generate* method, and therefore to the actual visualization. This leaves only an overhead of 5-20% for embedding the generator into the framework - and in most cases, it will probably be 20% for only the first attempt, and closer to 5% for all others.

At the moment, there are more than 30 generators for the topic areas sorting, searching, and cryptography, as indicated in Figure ???. We hope to increase this number even further before PVW 2006, and plan to add other areas, such as tree algorithms and string searching.

The generator framework is fully internationalized, although it currently only supports English and German. Volunteers who want to help in translating the language resources (less than 100 lines of text for the GUI) are welcome!

## 4 Summary and Further Work

In this paper, we have presented a simple yet expressive framework that makes it significantly easier for an AV end-user to specify the input for a visualization - and adapt its visual properties to the user’s preferences or content design. Using the component is easy and straightforward for the end-user, and also easy and quick for the AV content generator. The component can be integrated easily into existing systems, as shown in the example integration into the ANIMAL system.

We hope that the generator framework will be intensively discussed during the Workshop. Although the framework has only been used in conjunction with ANIMAL so far, there is no reason why it should not be able to produce output for other AV systems, such as JAWAA2 (Akingbade et al., 2003).

Additionally, we hope that some of the other AV system authors will want to consider adopting (or adapting) this framework to suit their personal tastes - and their own system.

## 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 34<sup>th</sup> ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2003)*, Reno, Nevada, pages 162–166. ACM Press, New York, 2003.
- Osku Kannusmäki, Andrés Moreno, Niko Myller, and Erkki Sutinen. What a Novice Wants: Students Using Program Visualization in Distance Programming Courses. In *Proceedings of the Third Program Visualization Workshop, Research Report CS-RR-407*, pages 126–133. Department of Computer Science, University of Warwick, UK, 2004.
- Thomas L. Naps, Guido Röbling, 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, 2003.
- Guido Röbling and Bernd Freisleben. ANIMAL: A System for Supporting Multiple Roles in Algorithm Animation. *Journal of Visual Languages and Computing*, 13(3):341–354, 2002.

## JHAVÉ – More Visualizers (and Visualizations) Needed

Thomas L. Naps

*University of Wisconsin Oshkosh  
Oshkosh, WI, USA*

Guido Rößling

*Technische Universität Darmstadt  
Darmstadt, Germany*

naps@uwosh.edu / roessling@acm.org

### Abstract

We recap the results of an ITiCSE 2002 Working Group report (Naps et al., 2003), which set the stage for the work described here. That work has resulted in the newest release of a system called JHAVÉ, which fosters active engagement on the part of learners by providing a set of standard support tools for certain types of Algorithm Visualization (AV) systems. These *AV engines* must implement the JHAVÉ *Visualizer* interface. In return, such engines have convenient access to the engagement-based tools offered by JHAVÉ. The details of adapting one such engine, ANIMAL, are also described.

## 1 Background

The report of an ITiCSE 2002 Working Group codified an *engagement taxonomy* for the modes for allowing students to be *active participants* in exploring an algorithm with an AV system (Naps et al., 2003). The report defined four active categories of engagement:

- Responding
- Changing
- Constructing
- Presenting

Students *responding* to questions regarding the current visualization are usually asked to predict the algorithm’s behavior (“what will happen next?”), or to reflect on more conceptual aspects (“which part of the code caused this effect?”). The AV system here becomes a supporting resource for posing questions and helping the student answer them.

*Changing* the visualization usually requires students to specify input that will cause a certain behavior. For example, the system could ask for an array that will cause exactly one value to end in its correct position during each loop iteration of Bubble Sort. After specifying the input set, the AV system can show the student if the input achieved the desired goal.

*Constructing* a personal visualization clearly entails a larger time commitment on the part of the student. A common technique for constructing a visualization is to first code it and then annotate it with calls to have graphics produced at “interesting events” during the algorithm’s execution. Well-designed class libraries can make this relatively painless for the student to do. For example, a data structure object such as a graph may have a “display” method that can be called to produce an aesthetically pleasing picture of the object in the AV system without requiring the student to do much beyond a method call (Lucas et al., 2003).

It is important to note that constructing the visualization does not always entail having the student code the algorithm. Systems such ANIMAL (Rößling and Freisleben, 2002) and ALVIS (Hundhausen and Douglas, 2000) provide visualization designers with a collection of modeling tools that are particularly well-suited to algorithm visualization. In effect, they allow the user to create a movie about the algorithm working strictly from a conceptual perspective, without ever having to code the algorithm.

In the *presenting* category, the student is asked to use a visualization to help explain an algorithm to an audience for feedback and discussion. The visualization may or may not have been created by the presenter.

## 2 JHAVÉ – A Pedagogical Support Environment for AV Systems

Clearly, graphics alone are not sufficient to *effectively* deliver educational applications of algorithm visualization. Unfortunately, the extra tools that are needed for such effective deployment, namely “hooks” by which to actively engage the student with the visualization, often require more effort to produce than the graphic displays themselves.

This section of the paper describes JHAVÉ (Java-Hosted Algorithm Visualization Environment) (Naps et al., 2000), a system designed to overcome this impediment. JHAVÉ is not an AV system itself, but rather a support environment for AV systems (called *AV engines* by JHAVÉ). JHAVÉ aims to be a common platform for staging a set of different AV systems, enabling the integration of available tools into one smooth front-end, while also offering added value for each integrated AV engine.

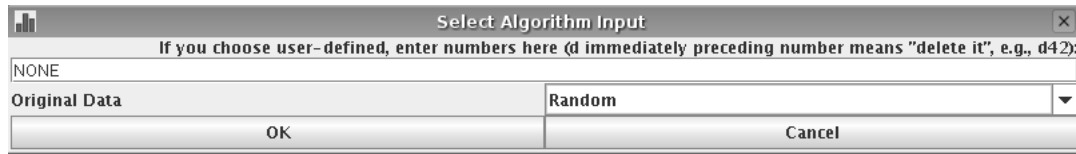
In broad terms, JHAVÉ gives such an engine a drawing context on which it can render its pictures in any way that it wants. In return, JHAVÉ provides the engine with effortless ways to synchronize its graphical displays with:

- A standard set of *VCR-like controls*. These allow students to step through the visual display of the algorithm. Hence, there is a standard “look and feel” to the GUI used by the student that is essentially independent of the AV engine being used.
- *Information and pseudo-code windows*. These are essentially HTML windows where visualization designers can author static or dynamically generated content to help explain the significance of the graphical rendering of the algorithm. The information window is used for higher-level conceptual explanations. The pseudo-code window may display a pseudo-code description of the algorithm complete with highlighting of lines that are particularly relevant for the picture currently being displayed.
- *“Stop-and-think” questions*. The visualization designer can designate questions in a variety of formats – true-false, fill-in-the-blank, multiple-choice, and multiple-selection (multiple-choice with more than one right answer) – to “pop up” at key stages of the algorithm. The question will typically ask the student to predict what they will see next, facilitating the *responding* category of engagement described in the previous section.
- *Input generators*. These objects gather input data from a student, if the visualization supports the *changing* category described in the previous section. This input data can be fed to the visualization, allowing the student to learn whether or not their data set drives the pictures in the anticipated fashion.
- Meaningful *content generation* tools. These include a variety of class libraries to help an AV designer (who may be a teacher or a student) create a visualization – both its graphical display and the interaction support tools that JHAVÉ offers.

The AV engine must produce the visualizations of an algorithm by parsing a textual visualization script and then rendering its pictures based on the script. Each AV engine is free to define its own scripting language – indeed, this is what allows a great deal of variation among the engines. The algorithm to be visualized has to generate an appropriate script file, which is then parsed and rendered by the AV engine.

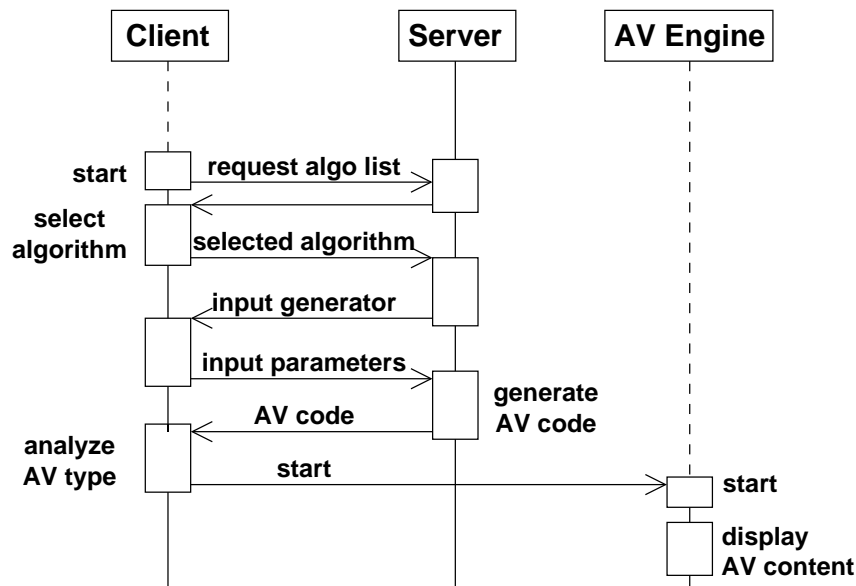
The reasons for this approach are tied to the client-server architecture of JHAVÉ. The server application manages the available algorithms and generates the visualization scripts that the client can display. In a standard session, a student first launches an instance of the client application, which displays a listing of available algorithms, tailored to the particular subject that the viewer is studying. The AV engines are included in the client distribution.

When the user selects an algorithm from that list, the client sends a request to the server, which will run a program that generates the script for that algorithm and sends the URL of



**Figure 1:** Input generator gets data from user to direct the visualization

the script file to the client. If the algorithm requires input from the user, the server sends an input generator object to the client, which opens a simple frame with appropriate input areas (see Figure ??). Once the user fills out these areas, the client returns the user's input to the server as a data set to use when running the algorithm. After the script is generated on the server, the client downloads, parses, and renders it. The steps in this process are illustrated in Figure ?. The rendering is staged using JHAVÉ's VCR-like viewing controls, "stop-and-think" questions, and information/pseudo-code windows (Figure ??). The script generation program is usually written by a visualizer or educator, not by the end-user.

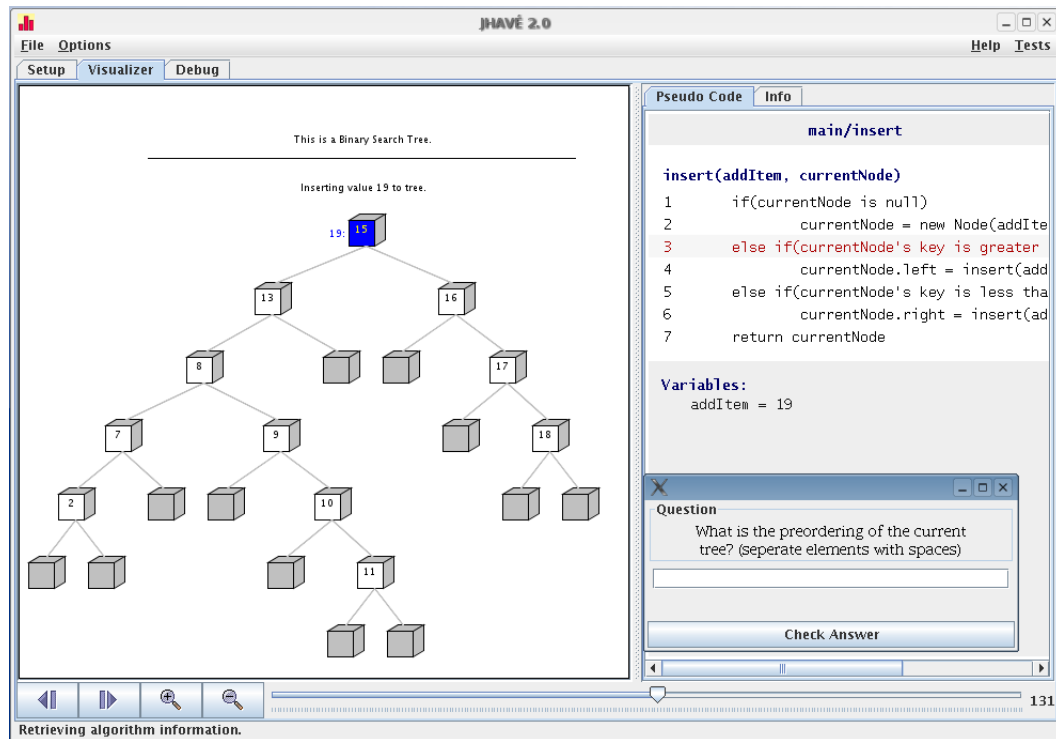


**Figure 2:** Sequence of steps used to generate and display AV content

AV engines that presently “plug into” the newest version of JHAVÉ as clients in the fashion described above are Gaigs-ML (an XML-based data structure description language described in another PVW 2006 submission by McNally and Naps) and, more recently, ANIMAL and ANIMALSCRIPT (Rößling and Freisleben, 2002). To become a JHAVÉ AV engine, a renderer must declare that it is a subclass of JHAVÉ’s abstract *Visualizer* class. As a consequence of this, it then must:

- Inform JHAVÉ of its capabilities by calling the *setCapabilities* method. JHAVÉ allows a Visualizer to support a subset of the following capabilities:
  - Stepping forward to the *next picture* in the visualization,
  - Stepping backward to a *prior picture* in the animation,
  - Going directly to a *particular frame* in the animation, skipping the rendering of all frames between that frame and the current frame,
  - “*Zooming*” in or out on the picture in the current visualization frame,





**Figure 3:** AV engine renders picture in leftmost pane, with support from the JHAVÉ environment in the form of VCR-like controls, pseudo-code window, and stop-and-think question

- *Animation* mode. A Visualizer object that supports this mode is capable of delivering its visualization as a smoothly animated motion picture instead of as a slide show consisting of discrete snapshots. Slide show mode is the default unless a visualizer declares otherwise. Gaigs-ML operates in slide show mode, whereas ANIMAL and ANIMALSCRIPT offer richer animation capabilities.

JHAVÉ uses this information about the Visualizer’s capabilities to display the appropriate subset of VCR controls in its GUI for that Visualizer. For example, Figure ?? shows that the controls for the Gaigs-ML Visualizer allow stepping forward, stepping backward, zooming, and direct access to a particular frame.

- Implement a constructor that receives an input stream as its only parameter. When JHAVÉ instantiates a particular Visualizer object, it calls this constructor, passing in the animation script, which exists as a URL on the server, for the input stream parameter.
- Implement three additional abstract methods from the base Visualizer class: *getCurrentFrame* returns the index number of the current frame in the visualization, *getFrameCount* returns the total number of frames in the visualization, and *getRenderPane* returns the Java component in which the engine renders the visualization.
- For each capability that it declares itself capable of supporting, it must override a method in the base Visualizer class to ensure the appropriate action is taken when the viewer uses the JHAVÉ GUI to trigger the event associated with that capability.

If an AV engine encounters a tag for a question or a pseudo-code/documentation window during parsing, it can invoke a *fireQuestionEvent* or *fireDocumentationEvent* method in the Visualizer base class. These methods handle everything connected with processing that event and hence free the particular AV engine from concerning itself about the details of parsing and

displaying the information associated with them. Instead that is all handled by the Visualizer base class. The AV engine itself can thus focus on rendering the visualization in its pane.

### 3 Integrating ANIMAL and ANIMALSCRIPT into JHAVÉ

ANIMAL with its scripting notation ANIMALSCRIPT (Rößling and Freisleben, 2002) was an almost obvious candidate for integrating an “external” AV engine into JHAVÉ. Based on past cooperations and the fact that ANIMAL already supported most controls, integrating ANIMAL into JHAVÉ was expected to be easy.

The adaptation was spread over two classes. One class is responsible for handling ANIMAL’s internal format (compressed or uncompressed), while the other handles compressed or uncompressed ANIMALSCRIPT. The code is mostly identical where JHAVÉ is concerned. In fact, most methods for mapping JHAVÉ’s calls to the underlying ANIMAL visualization engine contain between one and four lines of code.

While most of the work was done quickly, the main part of the work lay in preparing ANIMAL for cooperation with JHAVÉ. ANIMAL’s animation window used to be a stand-alone frame which incorporates a rendering canvas and two extensive toolbars for controlling the animation progress. As JHAVÉ has its own set of controls, only the canvas was to be used. Due to interconnections between canvas and controls, the underlying ANIMAL system had to be adapted somewhat. The net effect of this was that if an ANIMAL or ANIMALSCRIPT visualization is required, the tool will now start ANIMAL, initialize all components, and return the animation canvas as the rendering area for JHAVÉ.

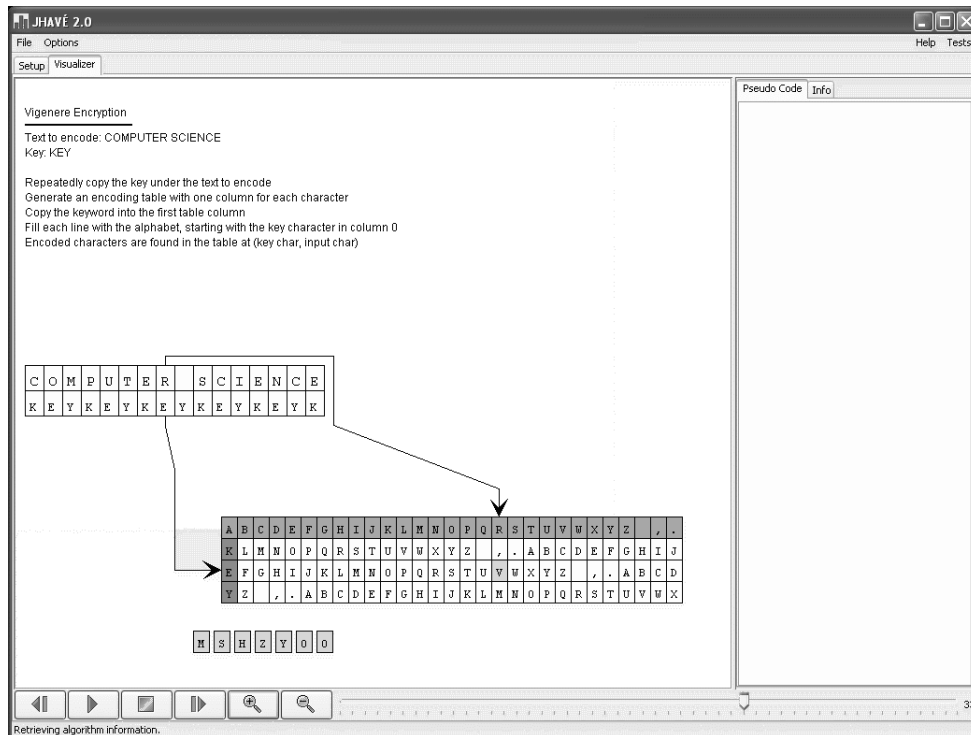
Another aspect which is probably specific to ANIMAL concerned the numbering of steps. The JHAVÉ GUI lets users jump directly to a step with a given number, assuming that steps are numbered consecutively and without gaps. When animations are generated in ANIMAL’s GUI front-end, each animation step receives a unique number. While these numbers are provided sequentially, the animation steps may be placed in an arbitrary order. Therefore, animations do not always start with step 1 – in fact, there may not be a step with the number 1! Additionally, the largest step number does not necessarily belong to the last step, and may also not be connected to the actual number of steps. Therefore, some additional code was written to ensure that when the user requests frame 3, he or she will see the content of the third step from the animation’s start - no matter what the ANIMAL-internal step number of this step may be.

Once these adaptations were put into place, both ANIMAL and ANIMALSCRIPT could be integrated into JHAVÉ as new visualizers. We expect that adding other systems to JHAVÉ should be similarly easy.

Figure ?? shows an example animation window of JHAVÉ running ANIMAL as the visualization engine. At first glance, there is not much difference between this Figure and Figure ?. ANIMAL adds an animation and pause button, compared to the visualization engine presented in Figure ?, as ANIMAL supports smooth animations. The pseudo code / information area to the right is currently empty, as we are still working on getting content for this area embedded into the existing ANIMALSCRIPT animation files.

### 4 Conclusion and Future Directions

It is our sincere hope that the work presented here will encourage other AV developers who use a scripting language in their work to consider providing a version of their system as a JHAVÉ AV engine satisfying the Visualizer interface that we have described. By doing so, they will help make available to students an increasing base of interesting visualizations that are conveniently instrumented with the learning devices we have described here.



**Figure 4:** An ANIMAL animation running in JHAVÉ

## References

- Christopher D. Hundhausen and Sarah Douglas. SALSA and ALVIS: A Language and System for Constructing and Presenting Low Fidelity Algorithm Visualizations. *IEEE Symposium on Visual Languages, Los Alamitos, California*, pages 67–68, 2000.
- Jeff Lucas, Thomas L. Naps, and Guido Rößling. VisualGraph: a Graph Class Designed for both Undergraduate Students and Educators. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, pages 167–171. ACM Press, 2003. ISBN 1-58113-648-X. doi: <http://doi.acm.org/10.1145/611892.611960>.
- Thomas Naps, James Eagan, and Laura Norton. JHAVÉ: An Environment to Actively Engage Students in Web-based Algorithm Visualizations. *31<sup>st</sup> ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2000), Austin, Texas*, pages 109–113, March 2000.
- 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, 2003. ISSN 0097-8418. doi: <http://doi.acm.org/10.1145/782941.782998>.
- 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.

**List of Authors**

Ackermann, T., 106  
Ahoniemi, T., 34, 54

Barowski, L., 76  
Bruce-Lockhart, M. P., 94  
Brusilovsky, P., 11  
Byckling, P., 6

Cotronis, Y., 94  
Cross, J. H., 76

Frengov, A., 11

Gerdt, P., 6  
Grissom, S., 1

Hamilton-Taylor, A., 44  
Hendrix, D., 76

Jain, J., 76  
Joy, M. S., 48

Karavirta, V., 100  
Kasabov, S., 17  
Kinshuk, 83  
Korhonen, A., 60  
Kraemer, E. T., 44  
Kumar, A. N., 11, 17, 67

Lahtinen, E., 34, 54  
Lin, T., 83  
Loboda, T. D., 11

McNally, M., 1  
Moreno, A., 48, 83  
Morth, T., 39  
Murray, K., 72  
Myller, N., 83, 89

Naps, T., 1, 112  
Nikander, J., 60  
Norvell, T. S., 94

Oechsle, R., 39

Piskuliyski, F. T., 67

Reed, B., 44  
Rhodes, P., 44  
Rößling, G., 23, 106, 112

Sajaniemi, J., 6  
Schneider, S., 23  
Sutinen, E., 83

Urquiza-Fuentes, J., 29

Valanto, E., 60  
Velázquez-Iturbide, J. Á., 29  
Virrantaus, K., 60

Wei, X., 72