ISSN 0105-8517

Proceedings of the Second Program Visualization Workshop

HornstrupCentret, Denmark, 27-28 June 2002

Mordechai Ben-Ari, Editor

DAIMI PB - 567
December 2002

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF AARHUS
Ny Munkegade, Bldg. 540
DK-8000 Aarhus C, Denmark

Foreword

The Program Visualization Workshops aim to bring together researchers who design and construct program visualizations, and, above all, educators who use and evaluate visualizations in their teaching. The first workshop took place in July 2000 at Porvoo, Finland. The Second Program Visualization Workshop was held in cooperation with ACM SIGCSE and took place at HornstrupCentret, Denmark, on 27-28 June 2002, immediately following the ITiCSE 2002 conference in Aarhus, Denmark.

Twenty-six participants from ten countries attended the workshop, which was held in the splendid isolation of HornstrupCentret. Two days of intensive presentations and discussions enabled the participants to obtain an overview of the research being done in this field and to develop personal relationships for future collaborative work.

This proceedings contains the revised versions of the papers that were presented at the workshop. Some participants have since published their work elsewhere, and in these cases we include only an abstract. The authors of the papers retain their respective copyrights.

Michael Kölling of the University of Southern Denmark, was responsible for the local arrangements. We would like to thank Michael Caspersen of the University of Aarhus, the co-chair of ITiCSE 2002, for assisting in the arrangements and in the production of this proceedings.

Moti Ben-Ari

Program committee

Moti Ben-Ari (chair)	Weizmann Institute of Science, Israel
Thomas Naps	University of Wisconsin - Oshkosh, USA
Marta Patiño-Martínez	Universidad Politecnica de Madrid, Spain
Guido Rößling	Darmstadt University of Technology, Germany
Jorma Tarhio	Helsinki University of Technology, Finland
Ángel Velázquez-Iturbide	Universidad Rey Juan Carlos, Spain

Contents

Mental Imagery, Visualisation Tools and Team Work

M. Petre
Faculty of Mathematics and Computing, The Open University, Milton Keynes, UK

`m.petre@open.ac.uk`

1 Introduction: Cognitive questions in software visualisation

In a 1998 paper on software visualisation (Petre et al, 1998), we asked a collection of "cognitive questions" about software visualisation, among them:

- What is visualisation suitable for? (Are all aspects of software amenable to visualisation?)
- Is SV a way 'into the expert mind' or a way out of our usual world view?
- Why are experts often resistant to other people's visualisations?
- Are visualisations trying to provide a representation that is more abstract, or more concrete?
- What kind of tasks are we supporting?

This paper describes empirical investigations that followed from such questions, into the relationship between expert reasoning about software and software visualisation, and into what experts want to use visualisation for. It is organised around four topics:

- expert programmers' mental imagery
- the externalisation of mental imagery
- why experts tend not to use other people's tools
- what visualisation tools experts build for themselves

Each topic has been investigated empirically, and the findings are summarised in sections 4-7 below. The empirical reports are preceded by discussions of the research context and some relevant observations in the literature, and are followed by the usual summary discussion.

2 Context: Which experts, which software, which overall tasks?

It is important to note that this work is based on studies in a specific context, one determined pragmatically – by which companies were willing to allow access to their expert software engineers.

2.1 The experts

The experts, from both industry and academia, and from several countries in Europe and North America, share the same general background: all have ten or more years of programming experience; all have experience with large-scale, real-world, real-time, data- and computation-intensive problems; and all are acknowledged by their peers as expert. All are proficient with programming languages in more than one paradigm. The coding language used was not of particular interest in these investigations, but, for the record, a variety of styles was exercised in the examples, using languages including APL, C, C++, Hypercard, Java, common LISP, macro-assembler, Miranda, Prolog, and SQL. Their preferred language was typically C or C++, because of the control it afforded, but the preference did not exclude routine verbal abuse of the language.

2.2 The companies and teams

All were small teams of 3 to 12 members, all included at least one expert programmer of the calibre of 'super designer' (Curtis et al., 1988), and all were in companies where the generation of intellectual property and the anticipation of new markets characterised the company's commercial success. All were high-performance teams: effective intellectual-property-producing teams that tend to produce appropriate products on time, on budget, and running first time. The companies were small, not more than 200-300 employees, although some were autonomous subsidiaries of much larger companies.

2.3 The eomains

Most were in large, long-term (1- to 2-year) projects. Often the software was one component of a multi-disciplinary project including computer hardware and other technology. Industries included computer systems, engineering consultancy, professional audio and video, graphics, embedded systems, satellite and aerospace—as well as insurance and telecommunications. Programmers generate between 5 and 10,000 lines of code per compile unit, typically around 200 lines per compile unit, with on the order of 3,000 files per major project.

It is important to note that these experts work in relatively small companies or groups that typically produce their own software rather than working with legacy systems. The software they produce is 'engineering software' rather than, for example, information systems, although products may include massive data handling and database elements. Goel (1995) argues, in the context of external representation, that there is a principled distinction to be made between design and non-design problems. That distinction is pertinent here, and the results presented may not generalise beyond this variety of design and this style of working.

2.4 Limitations

Experts are well-known for rationalising their practice 'on-the-fly'. As reported by Schooler, Ohlsson & Brooks (1993), there is evidence that solving insight problems relies on essentially non-reportable processes, even that verbalisation interferes with some important thought processes. On the other hand, although subjective tests may be suspect, they have in some cases been shown to be reliably consistent, and to produce results just as good as those from more objective tests (Katz, 1983) There is some evidence that self-ratings do correlate with demonstrated ability (Ernest, 1977) and are stable in cases where they do. These studies relied on subjects whose reports of activity in earlier studies corresponded well to other evidence of their activity, such as notes and observed actions, i.e., it relied on subjects who appeared to be 'good self-reporters'.

Given this context, the focus of this paper must be restated more precisely: the focus is on the relationship between expert mental imagery and software visualisation—in the context of *software design and generation*, rather than legacy software comprehension.

3 Background: relevant observations from the literature

There is widespread anecdotal evidence (e.g., Lammers's interviews of well-known programmers, 1986) that programmers make use of visual mental images and mental simulations when they are designing programs. There are broad, well-established literatures on mental imagery, expert problem-solving, memory and schema that, although they do not specifically address software design, can contribute to our thinking about imagery and representation in this context.

3.1 Literature on mental imagery

The psychology literature on mental imagery, informed recently by neurological and experimental evidence that imagery involves different systems (visual, spatial, verbal, temporal, propositional/semantic) which are usually handled in different parts of the brain, gives reason to consider that we maintain multiple mental representations and that imagery is multiply-encoded (in different modalities and systems) (e.g., Kieras, 1978; Mani & Johnson-Laird, 1982; Payne, 1993). Many of the hypotheses about sources of insight are based on interactions between encodings. For example, Anderson and Helstrup (1993) argue that mental imagery is a source of discovery and synthesis. Bartlett (1927) wrote that imagery leads into bypaths of discovery. Logie (1989) described an economy of images in memory, through which access to previously unrelated bits of information might be achieved: many informative elements are integrated together in a structural whole, increasing the available amount of information in working memory. Lindsay (1988) claimed that images allow inferences that are not based on proof procedures. The most pertinent shortcoming of the imagery literature is the tasks involved. Most of the studies deal with particular, usually concrete or real world images and simple tasks. Hence their conclusions might not generalise to a realm in which the imagery concerns complex, abstract, imagined images.

3.2 Literature on expert problem solving

Although there is apparently little empirical research on programmers' mental imagery, there is a well-established literature on expert problem solving which, taken together, suggests a pattern of mental structure building preliminary to efficient externalisation of solutions. The literature on expertise (and on expert-novice differences) consistently finds the same features across domains (for reviews, see Kaplan et al., 1986, Allwood, 1986), among them that:

- Expert problem solvers differ from novices in both their breadth and organisation of knowledge; experts store information in larger chunks organised in terms of underlying abstractions.

- When categorising problems, experts sort in terms of underlying principles or abstract features (whereas novices tend to rely on surface features) (e.g., Chi et al., 1981, Weiser and Shertz, 1983).

- Experts remember large numbers of examples—indeed, the literature suggests that experiencing large numbers of examples is a prerequisite to expertise (e.g., Chi et al., 1988). Experts' memories of previous examples include considerable episodic memory of single cases, particularly where these are striking. Implicit learning requires large numbers of instances with rapid feedback about which category the instance fits into (Seger, 1994).

- In many cases, experts seem to perform almost automatically or intuitively. The knowledge in these cases may be difficult to verbalise, or even to introspect about, because it has been 'compiled' or chunked (Anderson, 1982).

- Experts form detailed conceptual models incorporating abstract entities rather than concrete objects specific to the problem statement (Larkin, 1983). Their models accommodate multiple levels and are rich enough to support mental simulations (Jeffries et al., 1981, Adelson and Soloway, 1985).

- Experts tend to spend more time than novices planning and evaluating. Experts are better able to form overviews, but thereafter they take longer to develop their understanding and representations, and they consider interactions among functions or components of a system more fully (Adelson et al., 1984).

- Experts often engage in systematic exploration, whereas novices are less likely to engage in exploratory interactions (Petre, 1995).

The recurrent theme of overview and abstraction in expert reasoning has relevance for tool-building, for which it raises the issue: how can a tool show the essence rather than the superficial?

3.3 Literature on memory

There is a well established literature on memory, dating back over a century to pioneers such as Ebbinghaus (1913). Relevant findings from this literature include:

- The distinction between different memory stores and types, e.g., short term memory versus long-term memory (Miller, 1956), episodic versus semantic memory, (Tulving, 1983), and so on.
- Primacy and recency effects: e.g., list order affects likelihood of recall.
- Recall is weaker than recognition.
- More vivid images tend to be better remembered.
- There is little or no correlation between the vividness and the accuracy of a memory (Loftus and Palmer, (1974).
- Memory is an active encoding process, not a passive recording process, and is subject to distortions and bias at the encoding and retrieval stages. The encoding typically involves schemata, and this has many implications for the field of visualisation research (Bartlett, 1932, and others).
- Things which are easily visualised can be held in memory more easily – there is an entire literature on mnemonics, for instance (e.g., Luria, 1968).

3.4 Literature on schema

It was Bartlett (1932) who first suggested that memory takes the form of schema which provide a mental framework for understanding and remembering. In general, the term 'schema' is used to indicate a form of mental template which organises cognitive activity such as memory, reasoning, or behaviour. Key characteristics or elements in the schema have 'slots' in the mental template: the slots represent the range of values acceptable for those key characteristics. Schema may be of varying levels of complexity and abstraction; their importance is in providing structure and economy. Chi *et al.* (1988) suggest that the nature of expertise is due largely to the possession of schemas that guide perception and problem solving—i.e., experts have more and better schemas than novices. Simon (1973) observes that, when a task is ill-defined, users resort to pre-existing concepts: stereotypes, schemata, or other knowledge. Cole and Kuhlthau (2000) see the use of schemata as fundamental to sense-making at the outset of problem solving: the problem-solver invokes a schema or model of the problem in order to create a frame of reference and hence to identify the initial problem state. On one hand, the use of existing schemata enables the user to take some action in unfamiliar or ill-defined tasks. On the other hand, the use of existing schemata can lead to misconception, mis-action, or fixedness. (Tourangeau and Sternberg, 1982)

4 Programmers' mental imagery

So what evidence is there about the nature of programmers' mental imagery? A previous paper (Petre and Blackwell, 1997) describes a study into the mental imagery of ten individual expert programmers, who were questioned directly regarding the nature of their mental representations while they were engaged in a design task. This study consisted of structured observations and interviews attempting to elicit introspective reports of mental imagery, *not* a controlled laboratory experiment. The experts, all familiar informants whose reports of activity in earlier studies corresponded well to other evidence of their activity, demonstrated a readiness to describe the form and content of their thinking. The main images are as follows (see Petre and Blackwell, 1997, for a more complete summary):

- *dancing symbols* ("text with animation")
- *mental descriptions or discussion* (mental verbalisations)

- *auditory images* (auditory presentations of solution characteristics, with auditory qualities like loudness or tone reflecting some aspect of the solution)
- *visual imagery*
- *machines in their minds* (dynamic mental simulations, of three sorts: abstract machines, pictures of implementations, and mechanical analogies)
- *surfaces* (a strongly spatial, mathematically-oriented imagery of 'solution surfaces')
- *landscapes* (a strongly spatial imagery, a surface or landscape of solution components over which they could 'fly')
- *presences* (a sort of imagery that was not verbal, visual, or physical; an imagery of presence (or knowledge) and relationship)

There were some common characteristics of the imagery, *viz:*

- *stoppably dynamic*: All of the images were described as dynamic, but subject to control, so that the rate could be varied, or the image could be frozen.
- *variable selection:* The 'resolution' of the imagery was not uniform; the experts chose what to bring into and out of focus.
- *provisionality:* All of the imagery could accommodate incompleteness and provisionality, which were usually signalled in the imagery in some way, e.g., absence, fuzziness, shading, distance, change of tone.
- *many dimensions:* All of the experts reported using more than four dimensions. The extra dimensions were usually associated with additional information, different views, or strategic alternatives.
- *multiplicity:* All of the experts described simultaneous, multiple imagery. Some alternatives existed as different regions, some as overlaid or superimposed images, some as different, unconnected mental planes.
- *naming:* Although some of the imagery was largely non-verbal, the experts all talked about the ready ability to label entities in the imagery.

5 Externalisation of mental imagery

A key question in this area is whether personal mental imagery *ever* becomes public. A follow-on question is whether personal mental imagery would be of any use if it does become public. Some images and imagery, for instance, may be extremely useful to the individual, but by their nature may be very difficult to describe verbally and to use as a shared metaphor, because they are not well suited to reification and shared physical representations (such as diagrams, gestures, physical analogies, etc).

It seems intuitively obvious that there are times when imagery *does* become externalised and when the externalisation is useful. Yet, at the time of the 1998 paper, we found no published evidence of effective, direct externalisation of personal mental imagery in software development, apart from introspective justifications for software tool design. This section reports a form of externalisation which has been observed to occur naturally in high-performance development teams: when an individual's mental image becomes focal to team design activity and reasoning.

The evidence discussed here is a 'by-product' of other studies: initially, of the mental imagery study summarised above; subsequently, of a number of other *in situ* observational studies of early design activity. Those studies had other issues as their focus, for example design representations and processes used by multi-disciplinary concurrent engineering teams, representations (including ephemeral ones) used in early ideas capture, group discussions and processes in very early conceptual design, the generation and use of software visualisations. Thus, the core evidence was accumulated from five different software development teams and ten different projects in three different companies over a period of some five years. Only one example of a focal image is given below, but each group manifested at least one example, and the observations reported are representative of all examples.

One typical example arose in the context of the mental imagery study described above. The expert was thinking about a problem from his own work and articulated an image: "...the way I've organised the fields, the data forms a barrier between two sets of functions...It's kind of like the data forming a wall between them. The concept that I'm visualising is you buy special things that go through a wall, little ways of conducting electrical signals from one side of a wall to another, and you put all your dirty equipment on one side of a wall full of these connectors, and on the other side you have your protentially explosive atmosphere. You can sort of colour these areas...there's a natural progression of the colours. This reinforces the position queues...There's all sorts of other really complex data interlinkings that stop awful things happening, but they're just infinitely complex knitting in the data. (Of course it's not pure data...most of the stuff called data is functions that access that data.) The other key thing...is this temporal business we're relying on...the program is a single-threaded program that we restrict to only operate on the left or on the right...a hah!...the point is that the connections to the data are only on one side or the other. The way I organise the data is...a vertical structure, and the interlinkings between data are vertical things...vertical interlinkings between the data tell me the consistency between the data, so I might end up, say, drawing between the vertically stacked data little operator diagrams..." After he described the image fully, he excused himself and went down the corridor to another team member, to whom he repeated the description, finishing "And that's how we solve it." "The Wall" as it became known, became a focal image for the group.

5.1 How they occurred

In the observed examples, the mental imagery used by a key team member in constructing an abstract solution to a design problem was externalised and adopted by the rest of the team as a focal image. The images were used both to convey the proposed solution and to co-ordinate subsequent design discussions. The examples all occurred in the context of design, and the images concerned all or a substantial part of the proposed abstract solution.

5.2 The nature of the images

The images tend to be some form of analogy or metaphor, depicting key structural abstractions. But they can also be 'perspective' images: 'if we look at it like this, from this angle, it fits together like this' — a visualisation of priorities, of key information flows or of key entities in relationship. The image is a *conceptual* configuration which may or may not have any direct correlation to eventual system configuration.

5.3 The process of assimilation

In all of the examples observed, the image was initially described to other members of the team by the originator. Members of the team discussed the image, with rounds of 'is it like this' in order to establish and check their understanding. Although initial questions about the image were inevitably answered by the originator, the locus did shift, with later questions being answered by various members of the team as they assimilated the image. The image was 'interrogated', for example establishing its boundaries with questions about 'how is it different from this'; considering consequences with questions like 'if it's like this, does it mean it also does that?'; assessing its adequacy with questions about how it solved key problems; and seeking its power with questions about what insights it could offer about particular issues. In the course of the discussion and interrogation, the image might be embellished – or abandoned.

5.4 They are sketched

Sketching is a typical part of the process of assimilation, embodying the transition from ‘mental image’ to ‘external representation’. The sketches may be various, with more than one sketch per image, but a characteristic of a successful focal image is that the ‘mature’ sketches of it are useful and meaningful to all members of the group. This fits well with the literature about the importance of good external representations in design reasoning (e.g., Flor and Hutchins, 1991; Schon, 1988; and others).

5.5 Continuing role reflected in team language

If the image is adopted by the team, it becomes a focal point of design discussions, and key terms or phrases relating to it become common. Short-hand references to the image are incorporated into the team’s jargon to stand for the whole concept. But the image is ‘team-private’; it typically does not get passed outside the team and typically does not reach the documentation.

5.6 Imagery as a coordination mechanism

The images discussed and interrogated by the team provide a co-ordination mechanism. Effective co-ordination will by definition require the use of images which are meaningful to the members of the group. The literature on schema provides explanation here (e.g., Bartlett, 1932). Co-ordination—meaningful discourse – requires shared referents. If there is a shared, socially-agreed schema or entity, this can be named and called into play. But what happens when the discourse concerns an invention, an innovation, something for which there is no existing terminology, no pre-existing schema? A preverbal image in the head of one participant, if it cannot be articulated or named, is not available to the group for inspection and discussion. The use of extended metaphor, with properties in several different facets, provides a way of establishing a new schema. The borrower chooses what is salient in the properties of interest. In describing the image, the borrower is establishing common reference points, co-ordinating with the rest of the team a shared semantics (cf. Shadbolt’s research (1984) on people’s use of maps and the establishment of a common semantics). The discussion of the metaphor allows the team to establish whether they understand the same thing as each other. The establishment of a richly visualised, shared image (and the adoption of economical short-hand references) facilitate keeping the solution in working memory (e.g. Logie, as earlier).

It is interesting to note that this co-ordination issue has been taken on board by recent software development methodologies, which often try to address it by creating an immersive environment of discourse and artefacts which is intended to promote regular re-calibration with the other team members and with artefacts of the project. For example, ‘contextual design’ (Beyer and Holtzblatt, 1998) describes ‘living inside’ displays of the external representations in order to internalise the model, referring to the displayed artefacts as “public memory and conscience”. In another example, ‘extreme programming’ (Beck, 1999) emphasises the importance of metaphor, requiring the whole team to subscribe to a metaphor in order to know that they are all working on the same thing. In that case, the metaphor is carried into the code, for example through naming.

So, individual imagery does sometimes enter external interaction. The mental imagery used by a key team member in constructing an abstract solution to a design problem can in some cases be externalised and adopted by the rest of the team as a focal image. Discussing, sketching and ‘interrogating’ the image helps the team to co-ordinate their design models so that they are all working on the same problem—which is fundamental to the effective operation of the team.

6 Why these programmers don’t use available visualisation tools

Given the naturally-occurring use of image, abstraction, and sketches and other external design representations, why don’t these programmers use available visualisation tools to help them? This section reports on informal, opportunistic interviews with experts about their use of (or reluctance to use) available software visualisation tools. The interviews were conducted in the shadows of other studies, during ‘slack time’ (lunch, or coffee breaks, or pauses while systems were rebooted, or other time not filled with work activities), with key team members – those likely to make decisions on models and solutions as well as decisions on tools. Interviews were augmented with email queries to additional informants. Overall, some dozen experts were consulted.

The experts talk about software visualisation with respect to three major activities: comprehension (particularly comprehension of inherited code), debugging, and design reasoning. These experts showed no reluctance to investigate potential tools, and they described trials of tools as diverse as Burr-Brown DSP development package, Cantata, MatLab, Metrowerks Code Warrier IDE, Mind Manager, MS Project, MS-Select, Rational Rose, Software through Pictures (STP), and Visio (among others). However, take-up was extremely low. So what makes tools into shelf-ware? (Please note that ‘Not invented here’ was never offered as a reason not to use a tool.)

- *reliability:* Packages that crashed or mis-behaved on first exposure didn’t usually get a second chance.

- *overheads:* The cost of take-up was perceived as too high. Usually the cost was associated with taking on the philosophies, models or methodologies embodied in and enveloping the visualisation elements. Where there were major discrepancies of process between the package and existing practice, or where there was incompatibility between the package and other tools currently in use, the package was typically discarded as likely to fail. It must be considered that these are high-performance teams, with well-established methodologies and work practices. They continually seek tools and methods that augment or extend their practice, but they are reluctant to change work practices (particularly work style) without necessity.

- *lack of insight:* Tools that simply re-present available information (e.g., simplistic diagram generation from program text) don’t provide any insight. Experts seek facilities that contribute to insight, e.g., useful abstractions, ready juxtapositions, information about otherwise obscure transformations, informed selection of key information, etc.

- *lack of selectivity:* Many packages produce output that’s too big, too complicated, or undifferentiated. For example, all processes or all data are handled in the same way, and everything is included. Experts want ways of reasoning about artefacts that are ‘too big to fit in one head’; simply repackaging massive textual information into a massive graphical representation is not helpful. They seek tools that can provide a useful focus on things that are ‘too big’, that can make appropriate and meaningful selections for visualisation.

- *lack of domain knowledge:* Most tools are generic and hence are too low-level. Tools that work from the code, or from the code and some data it operates on, are unlikely to provide selection, abstraction, or insight useful at a design level, because the information most crucial to the programmer – what the program represents, rather than the computer representation of it – is not in the code. At best, the programmer’s intentions might be captured in the comments. As the level of abstraction rises, the tools needed are more specific, they must contain more knowledge of the application domain. Experts want to see software visualised in context – not just what the code does, but what it means.

- *assimilation:* Sometimes tools that do provide useful insights are set aside after a period of use, because the need for the tool becomes ‘extinct’ when the function (or model or method) the tool embodies is internalised by the programmer.

What the experts seek in visualisation tools (selectivity, meaningful chunking or abstractions)—especially in tools for design reasoning—appears to relate directly to their own mental imagery, which emphasises selectivity, focus, chunking and abstraction. It relates as well to what the literature has to tell us about ways in which human beings deal with massive amounts of information, e.g.: chunking, selection, schema – and to the nature of the differences between experts and novices. They want (and as the next section will indicate, they build) tools that have domain knowledge and can organise visualisations according to *conceptual structure*, rather than physical or programme structure – but that can maintain access to the mapping between the two. For example, the experts talk about tracking variables, but at a *conceptual* rather than a code level. A ‘conceptual variable’ might encompass many elements in the software, perhaps a number of data streams or a number of buffers composing one thing, with the potential for megabytes of data being referred to as one object. Experts distinguish between ‘debugging the software’ (i.e., debugging what is written) and ‘debugging the application’ (i.e., debugging the design, what is intended).

7 The visualisations they build for themselves

So, what sorts of visualisation tools do experts build for themselves, and what relationship do they have to experts’ mental imagery? In association with other studies (as described earlier), we asked experts and teams to show us the visualisation tools they built for themselves. In some cases, the demonstrations were volunteered (for example, in the mental imagery study, when one expert in describing a form of imagery said, ‘here I can show you’; or for example in the informal survey on package use, when some experts included their own tools in the review of tools they’d tried).

The experts own visualisations tended to be designed for a specific context, rather than generic. In one expert’s characterisation of what distinguished his team’s own tool from other packages they had tried: “the home-built tool is closer to the domain and contains domain knowledge”. The tools appeared to fall into two categories, corresponding to the distinction the experts made between ‘debugging the software’ and ‘debugging the application’. Each category is discussed in turn.

7.1 Low-level aspect visualisation

A typical visualisation tool in this class is one team’s ‘schematic browser’. This program highlighted objects and signal flows with colour. It allowed the user to trace signals across the whole design (i.e., across multiple pages), to move up and down the hierarchy of objects, and to find and relate connections. It moved through the levels of abstraction, relating connections at higher levels to connections lower down through any number of levels and through any number of name changes, for example following one conceptual variable from the top to the bottom of a dozen-deep hierarchy and automatically tracking the name changes at different levels. It allowed the user to examine the value of something on a connection, and to manipulate values, for example altering a value on one connection while monitoring others and hence identifying the connective relationship between different parts. The program embodied domain information, for example having cognizance of the use of given structures, in effect having cognizance of what they represented, of the conceptual objects into which schematic elements were composed.

It appears that these tools reflect some aspects of what the imagery presents, but they don’t ‘look like’ what the engineers ‘see’ in their minds. There are a number of such tools, especially ones that highlight aspects of circuits or code (e.g., signal flows, variables) or tools for data visualisation, as well as tools that represent aspects of complexity or usage patterns. In effect, they visualise things engineers need to take into account in their reasoning, or things they need in order to form correct mental models, rather than depicting particular mental images.

7.2 Conceptual visualisation

A typical visualisation tool in this class is one team’s ‘rubber sheet’. This program is a visualisation of a function used to build digital electronic filters for signals, which normally have a very complex and non-intuitive relationship between the values in the equations and the effect they have on the frequency response of the resulting filter. The tool allows the user to design the frequency response visually by determining the profile of a single line across the ‘rubber sheet’. The user moves peaks and dips in a conforming surface, rather than changing otherwise random-looking values in equations. The altitude one encounters on a walk through the resulting terrain along a particular path determines the final frequency response. The insight comes from the ‘terrain’ context. If one is just moving the points where the peaks and dips are, and can only sees the values along the line, it's hard to see how the values along the line are varied by moving the peaks and dips. However, if one can see the whole terrain, it's easy to comprehend why the peaks and dips have moved the fixed path up and down. Designers don't need to know all this terrain information in order to know what the filter does, but it provides the link between what the filter does and what the designer can control.

It appears that these tools can be close to what engineers 'see' in their minds. (Indeed, this example was demonstrated as a depiction of one programmer's personal mental imagery.) As in the example, they often bear strong resemblance to mathematical visualisations or illustrations.

The two categories of tool differ not just in their relationship to experts' mental imagery, but also in how they are used. The low-level aspect visualisations tend to be used to debug the artefact. They pre-suppose that the expert's understanding of the artefact is correct, and the examine the artefact in order to investigate its behaviour. The conceptual visualisations tend to be used to debug the concept or process – to reason about the design.

8 Implications and discussion

The features observed in expert practitioner behaviour in this domain are consistent with findings in a range of related literatures. What are the implications, and what practical advice can be offered as a result of these literatures?

One implication is that generic tools are not selective. Because they don't contain domain knowledge, they cannot depict what the programmers actually reason about when they reason about design. Automatic generation from code is inherently unlikely to produce conceptual visualisations because the code does not contain information about intentions and principles. The extent to which domain knowledge can be encoded is a suitable topic for further research.

The distinction between low-level aspect visualisation and conceptual level visualisation (in the self-built tools) is also important. At feature level the visualisation contributes to the mental imagery rather than reflecting it. At conceptual level, by contrast, it appears that there can be a more direct relationship between the mental imagery and the software visualisation. More work is also needed on design visualisations (as opposed to software visualisations) and on the interaction between the two – to what extent does understanding design visualisation contribute to solving problems in the domain of program visualisation?

It is important to remember that there are differences between design visualisation and program visualisation. Design visualisation is a divergent thinking problem in the early stages at least, which requires creativity and readiness to think 'outside the box'. Schon (1988) talks about a design as a 'holding environment' for a set of ideas. The importance of fluidity, selectivity, and abstract structure are emphasised by both the experts' own mental imagery and by their stated requirements for visualisation tools. It is little surprise that, in this context, experts conclude that "NOBO [whiteboard]...The best of all tools until the pens dry out. No question." and "Nothing is as good—and as quick—as pencil and paper." Program visualisation, in contrast, often involves dealing with existing legacy systems, where an important part of the task is reconstructing the design reasoning of previous programmers which led to the system under investigation—this paper contributes little to legacy system comprehension.

9 Conclusion

It appears that, in the context of the design and generation of 'engineering software', there is sometimes a fairly direct relationship between mental imagery and software visualisation—but that it is more often the case that visualisations contribute to rather than reflect mental imagery. It also appears that the externalisation of expert mental imagery can play an important role in the design reasoning of high-performance teams, both through co-ordination of team design discussions, and through embodiment in custom visualisation tools. Experts tend not to use available visualisation tools, because they don't contribute sufficiently to design reasoning. Their custom visualisation tools differ from others in their embodiment of domain knowledge, facilitating investigations at a conceptual level.

10 Acknowledgements

The author is profoundly grateful to the expert programmers, without whom the paper would not be possible, and to their companies which permitted access. Thanks are due to colleagues who provided essential commentary, including Alan Blackwell, Peter Eastty, Marc Eisenstadt, Henrik Gedenryd, Simon Holland, William Kentish, Jennifer Rode, and Helen Sharp. Special thanks are due to Gordon Rugg, who was instrumental in writing the paper. Some of the observations were conducted under EPSRC grant GR/J48689 (Facilitating Communication across Domains of Engineering). Others were conducted under an EPSRC Advanced Research Fellowship AF/98/0597.

11 References

Adelson, B., and Soloway, E. (1985) The role of domain experience in software design. *IEEE Transactions on Software Engineering*, **SE-11** (11), 1351-1360.

Adelson, B., Littman, D., Ehrlich, K., Blakc, J., and Soloway, E. (1984) Novice-expert differences in software design. In: *Interact '84: First IFIP Conference on Human-Computer Interaction*. Elsevier.

Allwood, C.M. (1986) Novices on the computer: a review of the literature. *International Journal of Man-Machine Studies*, **25**, 633-658.

Anderson, J.R. (1982) Acquisition of Cognitive Skill. *Psychological Review,* **89**, 369-406.

Anderson, R.E., and Helstrup, T. (1993) Visual discovery in mind and on paper. *Memory and Cognition*, **21** (3), 283-293.

Bartlett, F.C. (1927) The relevance of visual imagery to thinking. *British Journal of Psychology*, **18** (1), 23-29.

Bartlett, F.C. (1932) *Remembering: An Experimental and Social Study*. Cambridge University Press.

Beck, K. (1999) *Extreme Programming Explained: Embrace Change*. Addison-Wesley.

Beyer, H., and Holtzblatt, K. (1998) *Contextual Design: Defining Customer-Centered Systems*. Morgan Kaufmann.

Chi, M.T.H., Feltovich, P.J., and Glaser, R. (1981) Categorization and representation of physics problems by experts and novices. *Cognitive Science*, **5**, 121-152.

Chi, M.T.H., Glaser, R., and Farr, M.J. (Eds) (1988) *The Nature of Expertise.* Lawrence Erlbaum.

Cole, C., and Kuhlthau, C.C. (2000) Information and information seeking of novice versus expert lawyers: how experts add value. *The New Review of Information Behaviour Research 2000.* 103 – 115.

Cooke, Nancy J. (1994) Varieties of Knowledge Elicitation Techniques. *International Journal of Human-Computer Studies*, **41**, 801-849.

Curtis, B., Krasner, H., and Iscoe, N. (1988) A field study of the design process for large systems. *Communications of the ACM*, **31** (11), 1268-1287.

Ebbinghaus, H. (1913) *Memory: A Contribution to Experimental Psychology*. (Henry A. Ruger and Clara E. Bussenius, trans.) New York Teachers College, Columbia University.

Ernest, C.H. (1977) Imagery ability and cognition: a critical review. *Journal of Mental Imagery*, **1** (2), 181-216.

Flor, N.V., and Hutchins, E.L. (1991) Analysing distributed cognition in software teams: a case study of team programming during perfective software maintenance. In: J. Koenemann-Belliveau, T.G. Moher and S.P. Robertson (Eds), *Empirical Studies of Programmers: Fourth Workshop*. Ablex.

Goel, V. (1995) *Sketches of Thought*. MIT Press.

Jeffries, R., Turner, A.A., Polson, P.G., and Atwood, M.E. (1981) The processes involved in designing software. In: J.R. Anderson (Ed) *Cognitive Skills and Their Acquisition*. Lawrence Erlbaum. 255-283.

Kahneman, D., Slovic, P., & Tversky, A. (Eds.) (1982) *Judgment under Uncertainty: Heuristics and Biases*. Cambridge University Press.

Kaplan, S., Gruppen, L., Leventhal, L.M., and Board, F. (1986) *The Components of Expertise: A Cross-Disciplinary Review*. The University of Michigan.

Katz, A.N. (1983) What does it mean to be a high imager? In: J.C. Yuille (ed), *Imagery, Memory and Cognition: Essays in Honor of Allan Paivio*. Erlbaum.

Kieras, D. (1978) Beyond pictures and words: alternative information-processing models for imagery effects in verbal memory. *Psychological Bulletin*, **85** (3), 532-554.

Lammers, S. (1986) *Programmers at Work*. Microsoft Press.

Larkin, J.H. (1983) The role of problem representation in physics. In: D. Gentner and A.L. Stevens (Eds), *Mental Models*. Lawrence Erlbaum.

Lindsay, R.K. (1988) Images and inference. *Cognition*, **29** (3), 229-250.

Loftus, E.F., and Palmer, J.C. (1974) Reconstruction of automobile destruction: an example of the interaction between language and memory. *Journal of Verbal Learning and Verbal Behaviour*, **13**, 585-589.

Logie, R.H. (1989) characteristics of visual short-term memory. *European Journal of Cognitive Psychology*, **1**, 275-284.
Luria, A.R. (1968) *The Mind of a Mnemonist: A Little Book about a Vast Memory*. (Lynn Solotaroff, trans.) Basic Books.
Mani, K., and Johnson-Laird, P.N. (1982) The mental representations of spatial descriptions. *Memory and Cognition*, **10** (2), 181-187.
Miller, G.A. (1956) The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological Review*, **63**, 81-97.
Payne, Stephen. J. (1987*)* Complex problem spaces: modelling the knowledge needed to use interface devices. *HCI/Interact '87*.
Payne, S.J. (1993) Memory for mental models of spatial descriptions: an episodic-construction-trace hypothesis. *Memory and Cognition*, **21** (5), 591-603.
Petre, M., and Blackwell, A. (1997) A glimpse of programmers' mental imagery. In: S. Wiedenbeck and J. Scholtz (Eds.), *Empirical Studies of Programmers: Seventh Workshop*. ACM Press. 109-123.
Petre, M., Blackwell, A., and Green, T.R.G. (1998) Cognitive questions in software visualisation. In: J. Stasko J. Domingue, M. Brown and B. Price (Eds), *Software Visualization: Programming as a Multimedia Experience*. MIT Press. 453-480.
Petre, M. (1991) What experts want from programming languages. *Ergonomics*, **34** (8), 1113-1127.
Petre, M. (1995) Why looking isn't always seeing: readership skills and graphical programming. *Communications of the ACM*, **38** (6), 33-44.
Schon, D. (1988) Design rules, types and worlds. *Design Studies*, 9 (3), 181-190.
Schooler, J.W., Ohlsson, S., and Brooks, K. (1993) Thoughts beyond words: when language overshadows insight. *Journal of Experimental Psychology: General*, **122** (2), 166-183.
Seger, C.A. (1994) Implicit learning. *Psychological Bulletin*, **115** (2), 163-196.
Shadbolt, N.R. (1984) Constituting Reference in Natural Language: The Problem of Referential Opacity. PhD Thesis, University of Edinburgh.
Simon, H.A. (1973) The structure of ill-structured problems. *Artificial Intelligence*, **4**, 181-202.
Tourangeau, R., and Sternberg, R. (1982) Understanding and appreciating metaphors. *Cognition*, **11**, 203-244.
Tulving, E. (1983) *Elements of episodic memory*. Oxford University Press.
Weiser, M., and Shertz, J. (1983) Programming problem representation in novice and expert programmers. *International Journal of Man-Machine Studies*, **19**, 391-398.

The Algorithm Animation Repository

Pierluigi Crescenzi
Facoltà di Scienze Matematiche, Fisiche e Naturali, Università degli Studi di Firenze, Via C. Lombroso 6/17, 50134 Firenze, Italy

Nils Faltin
Learning Lab Lower Saxony, Expo Plaza 1, 30539 Hannover, Germany

Rudolf Fleischer
Department of Computer Science, HKUST, Clear Water Bay, Kowloon, Hong Kong
`rudolf@cs.ust.hk`

Christopher Hundhausen
Information and Computer Sciences Department, University of Hawai'I at Manoa, 1680 East-West Road, POST303D, Honolulu, HI 96822, USA

Stefan Näher
Fachbereich IV Informatik, Universität Trier, 54286 Trier, Germany

Guido Rößling
Department of Electrical Engineering and Computer Science, University of Siegen, Hölderlinstr. 3, 57068 Siegen, Germany

John Stasko
College of Computing/GVU Center, Georgia Institute of Technology, Atlanta, GA 30332-0280, USA

Erkki Sutinen
Department of Computer Science, University of Joensuu, P.O. Box 111, 80101 Joensuu, Finland

1 Introduction

As researchers in theoretical or practical computer science, we are used to publishing our results in form of research papers that appear in conference proceedings or journals. Journals are normally considered more prestigious than conference proceedings because their more rigorous refereeing standards presumably guarantee a higher quality of the published research papers. This well-established practice of publishing research results puts real practical researchers whose main interest is to write software at a certain disadvantage. There is no established way to 'publish' software (except for describing the software in a companion paper that may be considered publishable) unless you want to go the long way of commercializing your system. But this usually only makes sense for certain large systems. Therefore, all the effort that goes into the development of smaller programs is usually not rewarded by the academic community because there is no way to make these little programs known in a way that other people can actually use them as they can use published research papers (a research paper is 'used' by reading it, a piece of software is 'used' by running it).

This is in particular the case for programs that visualize or animate algorithms. Often, these animations are written either by instructors who need them for their teaching, or by people developing algorithm animation tools who use them to demonstrate the strengths of their new system. Since writing good animations can be a very difficult task that requires lots of effort and experience, it is a pity that all these nice programs are to a great extent unavailable for the general public (because they will never know about them) and that the authors of the pograms are not rewarded for their efforts.

Last year, at the (Dagstuhl Seminar on Software Visualization), this problem was recognized and it was decided to build an Algorithm Animation Repository (AAR). Eight participants of the seminar formed the Editorial Board of the AAR, and the chairman of the Board, Rudolf Fleischer, was given the task to build the repository at his home university HKUST. With additional funding from his university, the implementation of the AAR made good progress and we expect to launch it in 2002 (try `http://www.algoanim.net` or `http://algoanim.cs.ust.hk`).

2 Goals of the Repository

Currently, instructors that want to use algorithm animations in the classroom but do not have the time (or expertise) to write their own animations can only try to search the web to find some animations. This is tedious because there are always lots of unsuitable links, and often it is also frustrating because many of the animations found do not meet minimal standards for a good teaching animation. The main purpose of the AAR is to make this task easier. The AAR will collect programs of animated algorithms (these can be applets, executables, GIF animations, movies, program packages for download and installation, etc.) and 'publish' them, and thus make them accessible to the general public. We expect that mainly instructors would use the AAR to find good animations, but also students that want to understand better what they learned in the classroom.

All entries in the AAR will be refereed and ranked according a certain evaluation scheme (that is why we have the Board of Editors). We hope that this kind of refereed publication of programs will raise the level of acceptance for software development in the community. Besides of the editorial ranking, users of the AAR will also have the possibility to comment on the published software, similar to the reader book ratings at Amazon. We hope that this will not only improve the quality of the rankings (so that other users have a better guidance to find 'good' programs quickly), but will also provide a valuable feedback mechanism for the authors of the animations.

The AAR will only provide links to the software on their owners' homepages, so the owners of a piece of software will keep full control (and copyrights) of their work; in particular, they are free to withdraw it at any time from the AAR if they are not satisfied with the usage of their programs.

The AAR will provide a convenient search engine for its entries, so that it will be possible to restrict the search to various special needs (platforms, languages, etc.). Such search restrictions are usually impossible in general web searches.

The AAR will not only collect programs of animated algorithms, but also systems for writing algorithm animations (like XTANGO, MOCHA, etc.), animated hyperbooks, online courses, and anything else related to algorithm animation.

The AAR will also contain unrefereed material like a collection of links to other useful animation web pages, links to the home pages of researchers in the field (of course, only if they give their consent to be added), and a bibliography of algorithm animation publications. In particular, we would be very interested in collecting (or doing ourselves) studies on the effectiveness of algorithm animations in teaching environments (classroom, online courses, etc.).

3 Conclusions

The Algorithm Animation Repository (AAR) is currently being built as a joint effort of an Editorial Board selected among the participants of last year's (Dagstuhl Seminar on Software Visualization). The goal is to provide a place for the community where they can publish their animated programs, animation systems, or other related material. The programs will be refereed and ranked to make it easy for other users to find 'good' programs quickly.

Our project has some overlap with the Computer Science Teaching Center project (CSTC). Unfortunately, their collection of animations is not very large (but growing) and many entries are not refereed (contrary to their initial intentions). Thus, the CSTC is at the moment not really helpful for quickly finding good animations.

There are also similarities to the Complete Collection of Algorithm Animations (CCAA). However, the CCAA only provides a sorted collection of algorithms with very terse descriptions. While the animations are classified according to the type of algorithm, there is no review process by either a fixed set of reviewers or a single central reviewer.

A nice collection of animations for teaching mathematics can be found at the (Math Archives).

Of course, the AAR faces the problem of finding enough material to publish. It can only be a success if it becomes widely accepted in the community as *the* medium to publish good animation software. The support we got at the Dagstuhl meeting was encouraging enough to start the project, but only time will tell whether it can live up to its expectations.

References

CCAA. The Complete Collection of Algorithm Animations, 2001. `http://www.cs.hope.edu/~algoanim/ccaa`.

CSTC. Computer Science Teaching Center, 1999. `http://www.cstc.org`.

Dagstuhl Seminar on Software Visualization. Seminar No. 01211, May 20-25, 2001. `http://www.dagstuhl.de/01211`.

Math Archives, 1992. `http://archives.math.utk.edu/index.html`.

Visuals First Approach in an Introductory Web-based Programming Course

Anja Kareinen
Joensuun normaalikoulu, P.O. Box 111, FIN-80101 Joensuu, Finland

Niko Myller, Jarkko Suhonen and Erkki Sutinen
Department of Computer Science, University of Joensuu, P.O. Box 111, FIN-80101 Joensuu, Finland

`Anja.Kareinen@jnor.joensuu.fi`
`{Niko.Myller, Jarkko.Suhonen, Erkki.Sutinen}@cs.joensuu.fi`

1 Introduction

The Virtual Approbatur is an ongoing distance education project at the Department of Computer Science, University Joensuu. One of the concrete goals in the project is to develop ways to teach introductory programming over the web (Haataja et al., 2001). Because the Virtual Approbatur studies are aimed at high schools students, we decided to use visual objects to teach programming structures. In this way we could construct visually appealing game-like examples and assignments already at the early stages of introductory programming courses. However, since the courses were supposed to be at regular university level we did not compromise with the content but with teaching approaches.

Algorithm visualization can be used to attract student's attention during lectures, explain concepts in visual terms, automate examples and demos and present mental models to enhance the learning process. The visualization techniques offer students tools for experimentation and active processing of problems. It seems obvious that visualization tools help the students to learn the subject at hand. However, it is not totally certain whether these visualization systems actually help the students to learn the subject or not (Anderson and Naps, 2000). An obvious drawback of visualization tools in general is that they often provide only few ways to implement the visualization. Hence, there is no room for student's own mental and visual models.

In this paper we present a way to teach programming by starting with objects. Our approach is based on the fact that we have a different kind of user population than in traditional university-level circumstances. To make the learning materials and teaching methods as close to student's view of life as possible is in our context one of the crucial aspects. If the methods and materials are too far away from students' own "world", then there is a great risk that students cannot really relate to the domain at hand. This leads to a situation where the motivation for learning often comes from outside the student. We believe that visual objects are a good starting point to make the materials more appealing especially for young students.

2 Visuals First Approach

2.1 Course Contents in Virtual Approbatur

The programming part of the Virtual Approbatur project consists of three courses, all given in the Java programming language:

- P1: Basics of Programming (2 cu);
- P2: Introduction to Object-Oriented Programming (2 cu); and
- P3: Laboratory Project of Introductory Programming (2 cu).

In the Basics of Programming (P1) course the main goal was to introduce students the core concepts of programming. Hence, the key subjects at the course were: programming techniques, basic concepts of the programming, variables, loops, arrays, methods, etc.

After the basic concepts of P1, the second programming course (P2) consists of an introduction to object-oriented programming, event-driven programming and a choice of data structures, like stacks and queues. In the P1 course the learning material consists mainly of stand-alone applications, while the P2 course is founded heavily on applets.

Not seldom, the exercises of web-based learning environment are boring and underestimate the students' skills. We use exercises, which we hope will spark the students' interest in programming. The learning material in the web consists mainly of applet-based exercises and examples. The students practice the theoretical aspects of programming by making animations with visual objects and utilizing the event-driven approaches in the form of keyboard and mouse programming. In this way the students are able to learn programming by doing interactive, interesting and object-oriented exercises.

In the current state the graphical objects are introduced and utilized at the end of the P1 and throughout the P2 course. Moreover, the same approach is used in the laboratory project as well. In laboratory project a large number of the students will program a game or other visual program that suits students interests. Our intention is to expand the use of graphical objects to the teaching of all the basic structures of programming. Our purpose is not to use complicated and conceptually difficult graphical examples, but instead concentrate on building the examples from simple objects.

2.2 Spark the Students' Motivation

The general idea of web-based learning materials in programming courses is to attract or motivate students to program graphical objects and even animations from the early stages of studies. In this way, the students are able to learn programming by doing exercises that probably interest them more than traditional text-based or arithmetically oriented assignments, while still learning the basic structures of programming.

Particularly in the context of high school students graphical and interactive methods are almost essential for presenting the programming concepts. Visual objects combined with game-like theory, examples and exercises offers us a way to present the potentiality of programming in a meaningful way. The student can use her own imagination and intuition during the process of constructing visual programs. We believe that the method of using visual objects allures the students to concentrate also to more abstract and complex issues of computer science and programming. The main idea is that after we have inflamed the students' interest to programming and more general in computer science, the student gets an internal motivation to continue her studies further on.

Another aspect of using visual objects is to make the student really investigate the algorithm behind graphical events and animations. One could say that the students are encouraged to discover themselves the essential features of programming structures and methods. In our perspective visual objects and interactive applications gives tools to create active and experimental learning experiences. In our opinion this will inspire students to work hard during distance learning courses which requires more from a student than a regular course.

2.3 Examples

We give some examples about how visual objects have been used in programming courses. In Figure 1 there is an example of a programming assignment utilized in course P2. Exercise in Figure 1 is used to direct students to discover the possibilities of array structure or vector. At the same time the students get an opportunity to study the keyboard programming and event-handling. Students can construct various solutions and use different structures according to the knowledge level of Java. Beginners will probably choose the approach given by the material, but students with deeper knowledge of Java structures have an opportunity to use complicated methods. The example can work as a kind of basis for open modification of students.

Another example of applets used in P2 programming course can be found at Figure 2. The purpose of the assignment could be to make a similar looking game as illustrated in Figure 2. The purpose of the game is that the player should try to click the "button" that has a different color than any other. When the player hits the button he will get one point. The assignment includes also some tips for students on how to proceed with the solution. For example there can be suggestion for students to inherit their own class from the Button class for the game and utilize the array of instances of self-made class to make the object handling easier.

The purpose of the example shown in Figure 2 is to illustrate the concepts of objects and object arrays. Furthermore exercise includes some event programming and object-handling.

2.4 Constructing Examples and Exercises

When developing web-based learning material the question of needed resources always arises. In case of visual objects the need of resources is not greater than in normal web-based course design. The examples and exercises need to be done anyway but in this approach they are just compiled differently. The most time consuming individual process of course development is to make the examples and exercises meaningful for the students. We have found that a good way to accomplish this is to use the ideas and feedback collected from the students. Furthermore, some high school students have been working during the summer time to design and develop the examples and exercises from their own perspective. In this way we can use exercises and examples which have been created by peer learners. Hence, the examples and exercises are more closely related to high school students' own thinking.

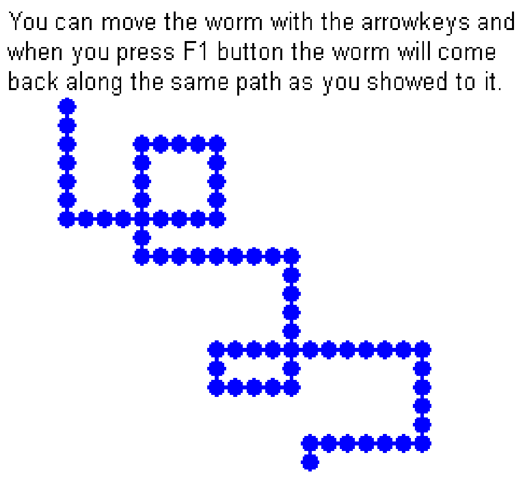

You can move the worm with the arrowkeys and
when you press F1 button the worm will come
back along the same path as you showed to it.

Figure 1: Example of graphical assignment.

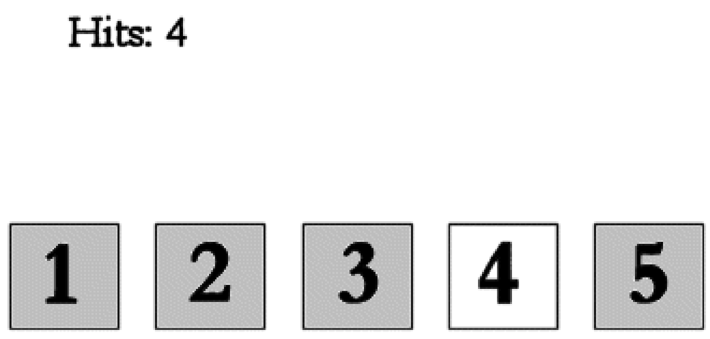


Figure 2: Example of objects and classes.

2.5 Pros and Cons of Using Graphics in Java

Our choice of using the graphics and event-driven approach has approved to be a correct one. The students at the programming course found that graphical programming was motivating. The students have been enthusiastic to complete the given assignments and they have given positive feedback about the content of the learning materials and examples.

However using graphical components unnecessarily add complexity. The programming assignments were designed to instruct students on specific concepts, and programming graphical components hampers inexperienced students from completing the required tasks. Some argue that the applets and graphical components should not be used to teach basics of programming. There are simply too many complex and confusing concepts to be mastered before they can be efficiently exploited during the courses. On the other hand there exists reports that describe the possible problems for using graphics in Java (Roberts and Picard, 1998). For example, despite Java's claims of full-fledged system independence, often one can not be sure how threaded or windowed graphics programs will act on different platforms.

Naturally these factors have more weight at the distance learning situation. If the learning material consists of many confusing factors it can hinder the learning or even make it impossible. On the other hand simplified examples or use of predefined graphical components can ease the complexity, but at the same time we narrow the possibilities of visual objects and graphics and, thus frustrate an eager student.

Hence, finding the balance between complexity and other factors is crucial. In our opinion for example the theory-first approach is a little bit boring and heavy for novice programmers (see Section 3). Surely, solid theory and properly defined concepts help learner to understand features of Java language. On the other hand this can lead to situation where the students have to master various concepts and facts before they can start constructing meaningful applications. It is not certain that all students have motivation to do this. In our opinion the design and implementation of interactive graphical programs motivates students to use objects and write methods while constructing appealing pieces of software.

3 Visuals First Approach Compared to Other Methods

Because of the early stage of the Computer Science Education field, it is hard to find an appropriate taxonomy of different approaches to teach introductory programming. However, at least the following pedagogical starting points can be easily identified:

- *Theory-driven approach.* A programming course builds on a certain programming paradigm. Examples and exercises tend to be rather academic and serve a specific topic from the contents (Budd, 2000).

- *Language-specific approach.* A course introduces the students to programming by presenting a language feature by feature (Kernighan and Ritchie, 1979).

- *Case-based approach.* The expressive power of a language is described by starting from real-world cases which correspond to the major structures of the language (Ben-Ari, 1998).

- *Problem-based approach.* Teaching starts by closed or open problems which a student solves by learning specific program concepts and applying them in an implementation (Morelli, 2002).

- *Tool-assisted approach.* A course has been designed around a tool which the student is encouraged to use throughout her learning process. The tool might be an entire programming environment, a debugger, or an animation tool. For example the software packages like BlueJ could be seen as en example of this approach (Barnes and Kölling, 2003).

It is important to note that a student can probably reach the same "level" of expertise by following any of the approaches above. However, individual differences related to factors like learning style, age, or prior programming skills might make one approach more appealing than others. Moreover, a given approach is not restricted to a specific educational theory, like behaviorism.

From the program visualization point of view, all the presented approaches would probably regard various visual or animation tools as opportunities to elaborate a code under study. In this respect, the introduced Visuals first approach is of a different origin. Here, the visual task is elaborated by a program, not vice versa. A student learns by using a programming language in a functional way: as a tool to create visuals he cannot resist to see on the screen. From the educational point of view, therefore, the presented approach considers learning to program a side-effect of solving an inspiring visual assignment—thus emphasizing attitudinal issues like joy of learning, internal motivation, and solving cognitive dissonance.

Naturally the learning materials in programming courses have also some features that can be found basically in all the approaches above. For example we have used to some extent Jeliot 2000 program in order to receive its potentiality in distance education context (Haajanen et al., 1997). There were no direct instructions on how and when to use Jeliot. Students simply had an opportunity to make use of it, if they thought it would be useful for them. On the other hand we did not control the use of the program, there were only some indication at the discussion forum that some of the students had done some experiments with it. Furthermore, some of the materials are based on theory-driven approach, but the visual objects are presented at early stages of programming.

4 Conclusion

Visual objects-first approach presented in this paper is one solution to start building web-based programming courses. We saw in Chapter 3 that there exist various approaches that take another perspective. We do not suggest that our way is the silver bullet that solves all the problems involved with teaching basics of programming. Nevertheless, we believe that visual and interactive examples and exercises play an important role in learning programming, especially in the distance learning context. Visual objects and interactive programs have features that seem to motivate students to struggle with their studies. Of course we can use also sound or other medias in similar way, which expands the possible interest area of students. In an ideal situation the learning material could consist of examples and assignments that excite and motivate as wide population of students as possible.

5 Acknowledgements

Virtual Approbatur project is a part of Finnish Virtual University and it is co-funded by Ministry of Education.

References

J.M. Anderson and T.L. Naps. A context for the assessment of algorithm visualization systems as pedagogical tools. In Sutinen E., editor, *the Proceedings of the First Program Visualization Workshop*, International Proceedings Series 1, pages 121–130, Porvoo, Finland, 2000.

D.J. Barnes and M. Kölling. *Objects First with Java - A Practical Introduction Using BlueJ*. Pearson Education Limited, New Jersey, USA, 2003.

M. Ben-Ari. *Ada for Software Engineers*. John Wiley & Sons, New York, USA, 1998.

T Budd. *Understanding Object Oriented Programming with Java*. Addison Wesley, Boston, USA, 2000. 3rd Edition.

J. Haajanen, M. Pesonius, E. Sutinen, J. Tarhio, T. Teräsvirta, and P. Vanninen. Animation of user algorithms in web. In *the Proceedings of VL'97 IEEE Symposium on Visual Languages*, pages 360–367, Isle of Capri, Italy, 1997.

A. Haataja, J. Suhonen, E. Sutinen, and S. Torvinen. High school students learning computer science over the web. *Interactive Multimedia Electronic Journal of Computer-Enhanced Learning (IMEJ)*, 3, October 2001. Available at: http://imej.wfu.edu/articles/2001/2/04/index.asp.

B.W. Kernighan and D. Ritchie. *The C Programming Language*. Prentice-Hall, New Jersey, USA, 1979.

R. Morelli. *Java, Java, Java - Object Oriented Problem Solving*. Prentice-Hall, New Jersey, USA, 2002.

E. Roberts and A. Picard. Designing a java graphics library for cs1. In *ITiCSE'98*, pages 215–218, Dublin, Ireland, 1998.

A New Approach to Variable Visualization: Roles as Visualization Objects

R. Wilhelm
Informatik, Universität des Saarlandes, 66123 Saarbrücken, Germany

`wilhelm@cs.uni-sb.de`

T. Müldner
Jodrey School of Computer Science, Acadia University, Wolfville B0P 1X0 N.S. Canada

`tomasz.muldner@acadiau.ca`

1 Introduction

Our report proposes a new approach to the field of *explaining* algorithms. Traditionally, this field borrows from the Software Visualization and Program Animation, and aims at explaining the inner working of the algorithm by using computer graphics and animation. The most common approach is to execute the algorithm step-by-step (forwards or backwards). This execution may be interleaved with showing various important events, for example a visualization of swapping two elements during sorting.

Since this kind of information is by its nature dynamic, the standard way of implementing the visualization is to actually run the program that represents the algorithm, using concrete input data. Therefore, the understanding of the algorithm requires careful and repetitive selections of representative sets of input data.

We propose a different approach. Instead of *running* a program on *single* input data, we suggest to *abstractly excute* it on *all* possible input data. This is possible by statically analyzing the program to retrieve information that is sufficiently rich to create invariants. It is our belief that *invariants* play the most crucial role in understanding an algorithm, and therefore the algorithm explanation should concentrate on visualization of the invariants.

To help the reader understand the above point, consider the insertion into a binary search tree. Standard algorithm visualization does not focus on the relevant part of the data structure; specifically it shows the entire tree, even though the insertion takes place only in one of the subtrees. Also, it does not show any invariants and thus forces the viewer to deduce them. In our approach, we show an essential invariant, stating that for each inner node, the elements in its left subtree have data components smaller than that of the node and elements in its right subtree have data components larger than that of the node, and we focus on the part of the tree that is being modified.

Let us recall from Wilhelm et al. (2002) the phases in the construction and use of an algorithm explanation system:

1. The *designer* selects a data structure and an algorithm to be explained. He then selects a set of properties characterizing the data structure, i.e., those properties that would be used to formulate invariants for the algorithm.

2. He runs a *shape analysis*, as implemented in the TVLA (Three-Valued-Logic Analyzer) system on the program using properties mentioned above. The shape analysis will annotate each program point with a set of *shape graphs* describing all heap contents that may exist when program execution reaches this program point. Together, all these shape graphs form an invariant for this program point.

3. He defines a visual representation for the shape graphs computed by the shape analysis.

4. Now, comes the *user's* time! What does he get to see and how does he get to see it? The decisions made in point 1 above determine *what* and the decisions made in point 3 above determine *how*. The user may want to select one particular shape graph at the entry to one particular

program statement and then execute the statement and observe the effect on the chosen shape graph. He may also want to perform a *visualized abstract execution* of the algorithm. This is a depth-first traversal of the program's control flow graph together with the sets of shape graphs at the program points. Each step in the visual execution corresponds to the execution of one statement as described above.

Shape analysis is described in Sagiv et al. (1999, 2002); Lev-Ami et al. (2000). A system, TVLA, implementing it is available, (see http://www.math.tau.ac.il/∼rumster/TVLA/). A full version of this paper has appeared in Wilhelm et al. (2002). Below, we explain our methodology using a simple example.

2 Algorithm Explanation: An Annotated Example

We explain our approach with the example of insertion sort using a singly linked list. (This paper deals only with algorithms on totally ordered domains.) For our considerations, we use a singly linked list of double values defined as is Figure 1(a). The code of insertion sort is provided in Figure 1(b) (in this code, we show two labels that will be used in discussions below).

The basic idea of this algorithm, see Aho et al. (1983), is to split the sequence of data to be sorted into two consecutive sub-sequences; a *sorted prefix*, in Fig. 1 pointed to by x, and an *unsorted suffix*, in Fig. 1 pointed to by r, which follows the sorted prefix. Initially, the prefix is empty, and the suffix consists of all elements. The algorithm loops until the suffix is empty (and so the sorted prefix contains all the elements). In each step of the loop, the algorithm inserts into the prefix the element, pointed to by r that immediately follows the prefix, thereby keeping the prefix sorted. Therefore, the basic invariant of this algorithm is: "*the prefix is sorted*".
The two figures provided on the next page show how the above invariant can be visualized while making a single step in the algorithm (because of space limitation, in this paper we do not explain how the invariant can be derived using the shape analysis; for details, see Wilhelm et al. (2002).

In our figures, horizontal arrows are implicitly labelled *n*; where *n* is the successor in the singly-linked list. First, we explain Fig. 2. Here, the inner loop has been executed several times and the current execution point is P1. This figure shows the *abstract heap*; a summarization of the concrete heap accomplished using shape analysis. The abstract heap describes an infinite set of singly linked lists with a set of pointers pointing into this list and some conditions on the minimal lengths of sublists. In the abstract heap shown in the two figures above, there are three kinds of nodes. *Unique nodes* represent single concrete elements of the heap, and are shown as rectangles. *Summary nodes* represent sets of nodes that are not distinguishable by the selected observation properties, and are shown as cubes. Finally, non-empty lists of nodes are shown as cubes with an additional rectangle at the back. Dashed self-loops show that we deal with summarized sublists, i.e. sets of heap cells connected by *n*-components. Finally, the placement on the x-axis indicates the relative value of the represented concrete elements (using the "<" order; here called *dle*; i.e. nodes are placed left-to-right according to the *dle*-order). In summary, the graph in Fig. 2 represents all lists for which the prefix of the list, pointed to by x, up to the element pointed to by pr, is already sorted (the *sorted* prefix). The next element in the list is pointed to by pointer variable r; it is the one to be inserted in the sorted prefix; the suffix of the list following this element is represented by t3.

The graph shown in Figure 2 seems to show sortedness and comparability properties of the represented list elements and sublists by placing nodes along the axis. Unfortunately not all elements shown in this figure are comparable with respect to *dle*; for example, the element pointed to by r is incomparable with any element in the suffix of the compared prefix starting at the element pointed to by l, since it has not been compared with them yet. Therefore, we need a graphical representation of *partial order* on nodes, and introduce *ascending chains* (for a definition of a chain, see Wilhelm et al. (2002). Chains can be visualized in a variety of ways; here we use patterns; the combination of two or more patterns indicates that a node belongs to more than one chain.

The left-to-right placement showing comparability is then only relevant for nodes that are members of the same chain. In Figure 2, we have two chains:

```
/* list.h */
typedef struct elem {
  struct elem *n;
  double data;
} *List;
```

(a)

```
/* insertion.c */
#include ''list.h''
List insert_sort(List x) {
  /* sort list x, return sorted list */
  List r, pr, l, pl;
  r = x;
  pr = NULL;
  while (r != NULL) {
  /*while the suffix is not empty */
    l = x;
    pl = NULL;
    while (l != r) {
      /*P1:  */ if (l->data > r->data) {
        /* P2:  after positive outcome of
        comparison, move r before l */
        pr->n = r->n;
        r->n = l;
        if(pl == NULL)
          x = r;
        else
          pl->n = r;
        r = pr;
        break;
      }
      pl = l;
      l = l->n;
    }
    pr = r;
    r = r->n;
  }
  return x;
}
```

(b)

Figure 1: (a) Declaration of a linked-list data type in C. (b) A C function that performs insertion sort on a list pointed to by parameter x.

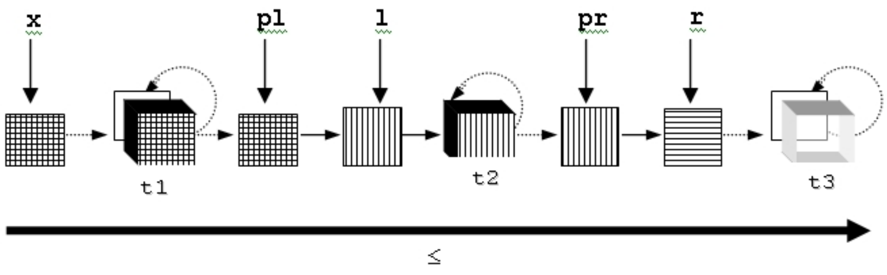


Figure 2: A shape graph at point P1.

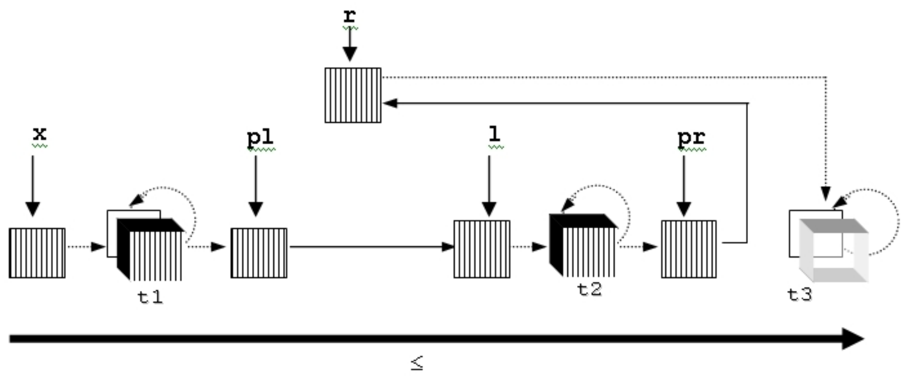


Figure 3: A shape graph at point P2 (i.e. after the comparison with positive outcome.

x, t1, pl, l, t2, pr
x, t1, pl, r
We will now describe what happens in the transition corresponding to the comparison "l->data > r->data", assuming it is true. Fig. 3 shows the next state of the algorithm, when the current program point is P2 (see Fig. 1). At this point, we have the information, that "l->data>r->data". This transition increases the sortedness of the list as now the element to be inserted has found its place between the pl- and the l-elements. Therefore, the element pointed to by r appears now between these elements. There is only one chain, representing the sorted prefix, and the transition, from the state shown in Fig. 2 to the state shown in Fig. 3 explains the preservation of the invariant.
Shape analysis applied to this sorting program starts with a description of all possible inputs and has to compute invariants at all program points, in particular show that the postcondition holds, namely that upon termination the elements are sorted. Invariants do not refer to actual data, but only mention relative sizes. For example, an invariant for a sorted sublist could be that the data-component of each but the last element of the list is less than or equal to the data-component of its successor. Shape analysis abstracts from the values of data-components. Data structure elements are thus incomparable to start with, but "gain comparability" by abstractly executing conditions. Mathematically, we would say that only a *partial order* on the elements of the data structure is known to the shape analysis. Abstractly executing comparisons may insert elements into chains, i.e. totally ordered subsets of the partial order.
Recall that in our approach to algorithm explanation, we preprocess an algorithm by running a shape analysis on this algorithm and then *visualize an abstract execution*. Instead of executing the algorithm with concrete input data and on concrete states, it is executed with a *representation of all input data and on abstract states*. Therefore, there is no need to determine representative input data, and repeat experiments using various inputs.

3 Conclusion

We have presented a new approach to algorithm explanation based on compile-time techniques used to automatically generate and visualize invariants. The example described in this paper is simple but representative for many similar programs operating on linked lists. Our approach involves several steps and only some of them are fully researched; others require additional work. The invariants are pre-computed by shape analysis, implemented in the TVLA system. This system provides a graphical output format using Graphviz, and supports a traversal strategy designed to visually execute the result of the analysis. There are two main area of the future research. Current analysis produces too many graphs for the users to traverse. Some of the graphs are redundant for the abstract execution and can be eliminated. In addition, we need to find appropriate visualizations for various kinds of nodes in shape graphs. Nevertheless, we believe that our research is very promising because it is the only known approach which offers the following facilities:

- Automatic generation of invariants; based on observation properties
- Abstract execution that runs on *all* input data
- Focusing on the relevant part of the data structure

References

A.V. Aho, J.E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *Intl. Symp. on Software Testing and Analysis*, pages 26–38, 2000.

M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape-analysis via 3-valued logic. In *Proc. of 26th ACM SIGACT-SIGPLAN Symposium on Priciples of Programming Languages*, San Antonio, Texas, 1999.

M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217 – 298, 2002.

R. Wilhelm, T. Müldner, and R. Seidel. Algorithm explanation: Visualizing abstract states and invariants. In S. Diehl, editor, *Software Visualization*, LNCS, pages 381 – 394. Springer Verlag, 2002.

Algorithm Simulation—A Novel Way to Specify Algorithm Animations

Ari Korhonen, Lauri Malmi, Jussi Nikander and Panu Silvasti
Helsinki University of Technology, Laboratory of Information Processing Science, PL 5400, 02015 TKK, Finland

{archie,lma,jtn,psilvast}@cs.hut.fi

1 Introduction

Software visualization is a broad research area covering many issues such as static visualization of program structures, algorithm animation, code animation and visual debugging. Intensive research on this field has been carried out over 20 years producing a substantial number of SV tools. Two basic problems have been addressed in these works. First, how the information for building the visualization is gathered from the code? Second, how the gathered information is shown to the user in a most informative way? In this paper, we concentrate on the first problem in the context of algorithm animation. We discuss techniques, such as code annotation, probing and interface typing which all fall under the category *Method* in the principled taxonomy of software visualization presented by Price et al. (1993). These *connection techniques* build the link between the actual implemented code and the visualization. Moreover, a closely related issue is the *visualization specification style*. In the taxonomy, the following basic styles are mentioned: hand-coding, using libraries of ready-made visualizations and automatic analysis of the program. In practice many systems allow the visualizer to use more than one style or a combination of them. We consider this topic in more detail later in this text.

Connection techniques can be classified as *invasive* or *non-invasive*, *i.e.* they either modify the original code or not. Invasive methods include *instrumenting* or *annotating* the code. When instrumenting a code the user adds statements to print visualization commands at interesting events. An example system based on instrumentation is BALSA (Brown and Sedgewick, 1984). More recent systems utilizing this approach include Samba (Stasko, 1997) and its Java version Lambada (Astrachan et al., 1996). These systems read graphical animation code produced by user-added print commands during the execution of the animated program. Code annotation (Price et al., 1993) is a procedure where the user can mark interesting points in the code so that the original source code remains unchanged but the executable code is changed. Annotation is used, for example, by TANGO (Stasko, 1990). A more advanced version of that is automatic annotation, such as implemented in Jeliot (Haajanen et al., 1997), in which the user provides the source for the system which automatically annotates it and creates the animation.

Non-invasive data gathering methods typically require the programmer to use some existing code that supports visualization when implementing the algorithm. Early works such as Transparent Prolog Machine TPM (Eisenstadt and Brayshaw, 1988) or Pavane (Roman et al., 1992) were based on attaching non-invasive *probes* to data structures or code in a declarative manner. Then the structures could be monitored without affecting the actual source code at all. A closely related method is using ready-made *library components* for building the application algorithm (LaFollette et al., 2000). The JDSL Visualizer tool (Baker et al., 1999) uses a third related method, the *interface typing*, where the user modules must implement a number of interfaces. These enable the system to analyze the working of the code and produce the visualization.

In some cases manual intervention with the code is intentional, because it supports educational purposes to understand the working of the code. This approach has been successfully applied by Stasko (1997). Hundhausen and Douglas (2000) have also reported the effectiveness of this approach in education. However, there are many cases, where the goal is quite the opposite: to create the visualization as effortlessly as possible. The automatic annotation method used by Jeliot is one big step forward in this direction. Rasala (1999) uses templates and operator overloading available in C++ to provide automatic visualization for array algorithms.

In this paper we discuss the Matrix system (Korhonen and Malmi, 2001, 2002), in which several different connection techniques have been incorporated. Moreover, we describe how these techniques have been applied to allow *algorithm simulation* where the user can create algorithm animations by directly manipulating graphical objects on the screen. Both primitive operations and operations based on abstract data types implemented in the system are available. Essentially Matrix provides tools for direct interaction with implemented data structures and algorithms through a graphical user interface. Algorithm simulation can be regarded as a novel visualization specification style. Hence our approach provides powerful tools for exploring the working of pre-existing or user-coded new algorithms.

In the following Section 2 we present the Matrix system more closely. Section 3 includes case examples how various connections techniques are applied by the user. In the final Section 4 we reconsider the classification of connection techniques and compare them to each other. We also present our vision of their role in different fields of applications, and how we aim towards more effortless visualization of algorithms.

2 Matrix

Matrix is a system for visualizing data structures and algorithms with specific emphasis on their interactive manipulation. It is based on a general-purpose framework written in Java which is capable of representing data structures and their visualizations to arbitrary depth and complexity. Matrix includes a graphical user interface (GUI) for interactive operations. The system can be used as a stand-alone application to explore the ready made components available, but it can also be used to visualize and exploration of user's own algorithms. In this section we explain the basic functionalities of the system.

2.1 Algorithm Simulation

Algorithm simulation is one of the key features of Matrix. We therefore define the term more closely and relate it to algorithm animation. See Figure 1.

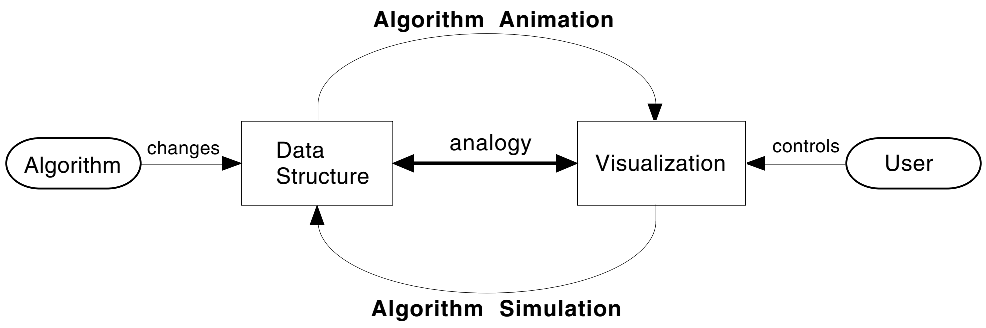


Figure 1: Algorithm animation and simulation

Algorithm animation is a process where an algorithm manipulates data structures and the resulting changes in the structures are visualized for the user. In general, algorithm animation includes dynamic visualizations based on observing the working of an algorithm.

Algorithm simulation is a similar process, where the user, instead of an executed implementation of an algorithm, directly manipulates the data structures in the memory by using tools available in the user interface. The user can apply similar operations as an algorithm would do. He or she can either change the contents of data types or connections between them, thus working on a level of primitive assignment operations. The user can also work with abstract data types, like dictionaries, and perform operations which activate the implemented ADT operations in the system. Examples of these operations will follow later.

The data types involved in the simulation could be Matrix library components or user made structures. The latter ones can be dynamically loaded into Matrix, which also in this case provides the user the full simulation functionality to explore the behavior of the algorithm code. In the following, we discuss the implementation issues of these features more closely.

2.2 Visualization methods in Matrix

The system supports several different connection techniques. All of the major connection techniques, aiming at effortless algorithm animation, are supported: interface typing, probes and annotation. The most fundamental technique is the interface typing, and the probes are based on interfaces. Annotation is only used for code animation purposes. Moreover, it is only a brief experiment and not fully tested or integrated into the framework.

The visualizer may specify his or her own visualization in several different ways. The easiest way is the simulation in which the user just picks up a ready-made library component (probe) and simulates the algorithm by directly manipulating its visualization. However, user-made code can also be visualized and simulated by extending one of the library components. If this is not possible, an appropriate concept interface can be implemented directly. Implementing concept interfaces in a class provides the system the access to the actual data structures. Moreover, any structure can be interpreted and visualized simultaneously using several different concepts, such as arrays, lists, trees or graphs, if the corresponding concept interface has been implemented. The system lets the user to choose the concept used for visualization, validates that the actual structure conforms to the concept, and finally lays out the concept.

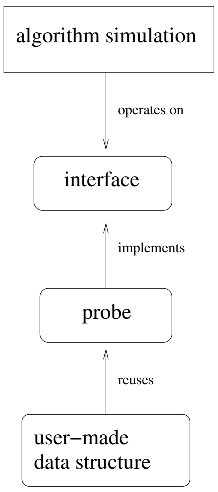


Figure 2: Hierarchy of visualization specification styles and connection methods in Matrix.

2.3 Specifying Visualization by Simulation

Algorithm simulation is a method where the user specifies the visualization solely using the Matrix GUI features. Multiple windows and menu commands are used to ease the process. A new structure can be inserted either into a new window or into a window with some existing structure. The data structures to be used can be selected from the menus. The simulation functionality consists of a sequence of *drag&drop operations* where the user drags some source object to a new position and activates some operation. The source object can be a key, a node, an edge or some complex structure. The user can create, modify and delete connections between objects. On this level the data structures have no semantic meaning, except what the user has on his mind. Keys, nodes and structures can also be dragged into the title bar of some other structure. In this case the corresponding insert routine for the structure is invoked. On this level the semantics for the structure and for the insert operation should have been defined.

Consider, for example, a case in which the user chooses to insert a new binary search tree from the menu. As a result a new binary tree with an empty root node is displayed. Thereafter the user can drag keys into the desired position in the binary tree. Alternatively the keys can be dropped into the title bar to invoke the binary search tree insert operation. Similarly, the user can insert a whole array of keys into the search tree, and the system interprets this operation as a sequence of insertions. Each key in the array is then separately inserted into the search tree, as illustrated in Figure 4. Of course, such operations, in which different data structures are combined interactively, are possible only if meaningful semantic interpretations have been defined for them.

Whichever method is used, the result will be an algorithm animation, which in Matrix consists of a sequence of states of the data structure. These states can be shown step by step, backward and forward. The sequence can also be rewinded to the beginning. In any state, it is possible to invoke new operations on the structure to explore its behavior further.

The GUI provides also some additional helpful features. The user can choose methods to be invoked and keys to be inserted from the menus. Moreover, it is possible to change the layout of the concepts which are displayed on the screen or display the same structure with multiple concepts in the same window. In the latter case, the simulation operations can be performed on any displayed version of the structure and the changes will appear immediately on all of them. In addition, the corresponding parts of the structure are highlighted whenever the cursor is positioned on them.

As a whole, the versatile functionality provided by the primitive operations, the ADT operations and the GUI commands allows the user to explore the behavior of a data structure in an easy and interesting way.

2.4 Connections techniques

A connection technique describes how the connection is made between the visualization and the actual software being visualized. Several techniques are currently supported. The set of concept interfaces and the probes derived from these are used to produce the conceptual displays of data structures in algorithm animation. The annotation technique is solely used to produce code animation.

2.4.1 Concept interfaces

The concept interfaces enable visualizations of structures and interaction between the user and the underlying structures. All visualizations are based on the information gathered through the corresponding concept interfaces. In addition, several independent decorative interfaces can be implemented to customize the layout. Moreover, the concept interfaces allow primitive simulation without any additional coding. However, more advanced simulation operations require additional interfaces. In the following, fundamental data types (FDT) and abstract data types (ADT) correspond to the concept interfaces and the additional interfaces, respectively. On the FDT level, the visualized data structures have no semantics. The user can change and swap keys, move substructures, manipulate pointers between nodes, etc. On the ADT level, the semantics for the abstraction should be defined and implemented. In general, the ADT interface allows the user to manipulate the structure efficiently by invoking more abstract operations without, for example, any pointer manipulation. As an example, the user may insert a key into a search tree and the key is automatically placed to the correct position in the tree.

2.4.2 Probes

The other connection techniques expand the functionality enabled by the interfaces. For example, probes enable storing algorithm animations and reversing their temporal direction. The probes included in Matrix implement a number of interfaces for visualization and interaction purposes. However, they also store their instance variables in special memory classes provided by Matrix, which correspond to primitive types of Java. The amount of functionality gained for a user made animation depends on how the data structure is implemented. By implementing a number of Matrix interfaces, it is possible to produce visualizations, algorithm animations and simulations. By extending a Matrix probe, or by using the special memory classes provided by Matrix, it is also possible to reverse the temporal direction of the visualization and store it for later use.

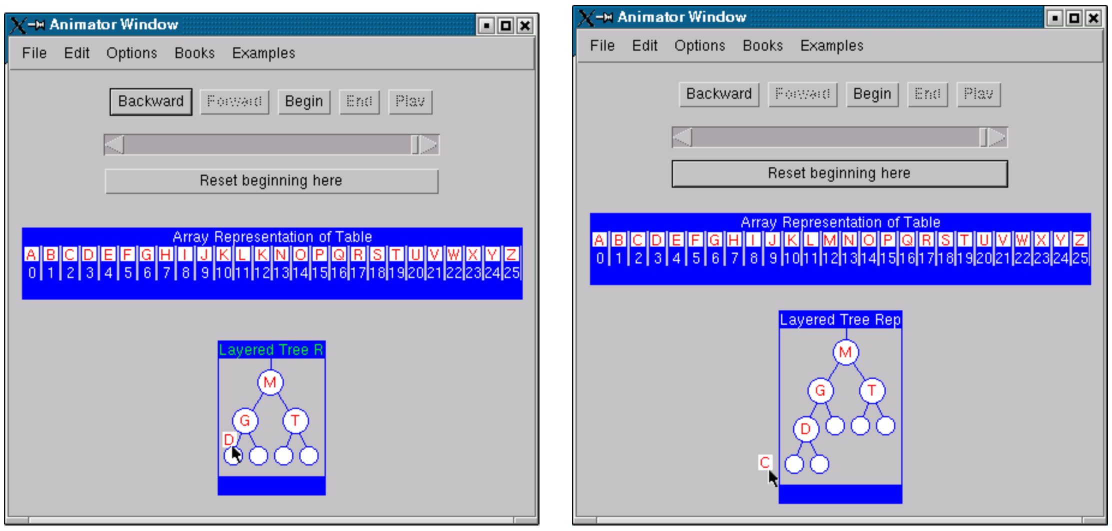


Figure 3: Primitive simulation. On the left the user drags the key D into an empty leaf position. On the right he proceeds with the key C.

2.4.3 Annotation

The annotation technique is currently at the prototype stage in Matrix. A few abstractions have been implemented to allow primitive code animation. Annotation is not used as much for the data structure visualization, as it is used to visualize the execution of the algorithm code. In Matrix, annotation technique has currently been tested only in one string-matching problem (Korhonen et al., 2002), in which the Boyer-Moore-Horspool algorithm was demonstrated, thus we do not discuss it here any further. However, it should be noted that this example shows how all the techniques can be mixed to produce new applications rapidly.

3 Example Cases

As mentioned in the previous section, Matrix supports many different techniques to produce algorithm visualizations. In this section, we consider a number of use cases more closely to illustrate the techniques in use.

Let us assume that we would like to use data structure visualizations as a learning aid in a lecture situation. We would require a tool which could visualize and animate data structures. It would be helpful if the tool is interactive so that we can use it to answer to the questions asked by students. It would also be useful if we could work on many different levels of abstraction with the same tool. Then, we could demonstrate the basic behavior of a data structure by manually simulating its basic operations step by step (*e.g.* the details of the binary search tree delete routine). On the other hand, some special cases could be explained better by invoking ADT operations. In the following, we assume that we would like to demonstrate different kinds of binary search trees in a lecture.

3.1 Simulating Library Structures

First, we can work on the fundamental level by directly manipulating the nodes of the data structures in terms of algorithm simulation. On this level, we are treating the data structure as a fundamental data type and the structure has no semantics except what we simulate on it. We can, for example, explain how the different binary search tree algorithms (*i.e.* insertion, deletion) work in detail. Figure 3 illustrates how this can be achieved in Matrix by drag & dropping arbitrary keys from the upper array into the correct positions at the lower binary tree.

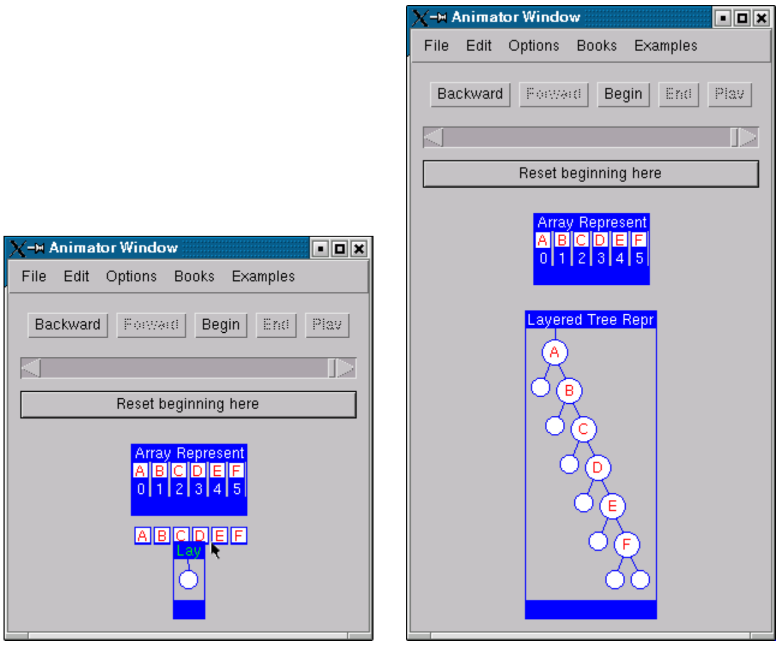


Figure 4: Abstract operation invocation: degenerated binary search tree before and after inserting a sorted array of keys.

The task can be accomplished in two ways. We can either take (*i.e.* select from the menu) a real FDT, or we can use an abstract data type as we would use the FDT. The ADT has the advantage that we can easily raise the level of abstraction later and use the same structure while demonstrating some more complex behavior. Of course, we should also have a way to expose the underlying structure of the ADT before we can directly manipulate it at the FDT level. This may sometimes cause a problem because information hiding may have been used to deny the access to the underlying structure, as explained below in the example of visualizing JDK data structures.

The working on more abstract level with concepts requires the use of ADTs. Figure 4 demonstrates how to apply Matrix to rapid creation of an algorithm animation in terms of algorithm simulation. By using the basic ADT operations, *e.g.* the insertion to a binary search tree, we can start to explore the behavior of a binary search tree in different special cases. For example, it is rather clumsy to manually place each key into a leaf node, if we are showing how a binary search tree degenerates to a list. However, Matrix allows the user to drag & drop a whole set of keys into an initially empty tree at once to demonstrate the case.

3.2 Implementing Interface

It is often useful to visualize an existing data structure in order to observe or demonstrate its behavior. Since Matrix uses interfaces for visualization, the observed structure must be modified so that it conforms to the desired interface(s). Usually, this is done by modifying the source code. If the source is unavailable, it might be possible to use inheritance to implement the required interfaces. One such a case is the following experiment in which `java.util.Vector` from the Java Software Development Kit is visualized by using the array concept.

Vector works as a dynamic array with some added functionality. A Vector grows and shrinks depending on the number of keys it stores. By extending Vector and implementing few Matrix interfaces we can visualize and examine its behavior. The Matrix interface Array allows us to visualize Vector in Matrix, and the interface ADT provides some additional Vector methods to be accessible through the Matrix GUI. However, since we visualize Vector by only implementing the interfaces above, the Matrix cannot save the states of the data structure. Hence moving back and forth in the animation sequence is not possible.

After the desired interfaces have been implemented, the data structure can be loaded into Matrix by using the dynamic class loader provided by the framework. Now we can start to explore the behavior of java.util.Vector as illustrated in Figure 5.

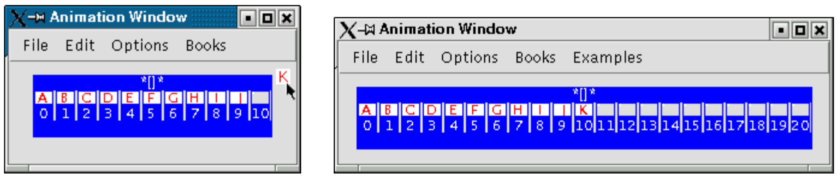


Figure 5: Visualization of an existing structure: dynamic behavior of java.util.Vector is visualized using the array concept. The length of the visual array illustrates the capacity of the Vector.

Unfortunately, even this kind of inheritance cannot always be used in order to visualize a data structure. The information hiding techniques can be used to hide the information needed to visualize an abstract data type properly. The concept interface cannot be implemented because the subclass inherited from the ADT cannot access the underlying structure. Even though one could implement the concept interface in order to produce a visualization, the non-primitive simulation methods are beyond the scope.

3.3 ADT Extends Probe

Finally, let us assume that we do not have an implementation for AVL-tree, but we would like to create one for our own purposes. This might be the case, for example, if we would like to animate an AVL tree of our own. The easiest way to implement and visualize an AVL-tree is to reuse the binary search tree probe and to change its behavior properly. For example, one can inherit the existing binary search tree and override its insert and delete methods to keep the tree balanced.

4 Discussion

Algorithm simulation provides lecturers as well as students a valuable aid for demonstrating and surveying the working of basic algorithms because no user written code is needed. Moreover, the instruments that are capable of visualization and simulation of user's own code greatly extend the learning possibilities.

Even though educational software is the main application area for Matrix, it is obvious that similar tools can be applied also to other areas. These instruments can be used for testing purposes or visual debugging in software engineering. However, the more application areas there are involved the more flexibility is needed to meet the overall requirements. Our strategy, in this sense, is to support as many techniques as possible. Moreover, our research aims at new techniques and specification styles, such as algorithm simulation.

Even though some techniques are not fully supported at the moment, the Matrix framework is at least capable of adopting them with little or no effort. Thus, it is flexible enough to be extended further in terms of fast prototyping. We believe that this is the way we can keep the framework live enough also for further research. Of course, real applications for special areas are better implemented from the scratch, but at that point it should be clear which techniques to use.

From this point of view, it is obvious that there is not just one technique that could dominate the others. The invasive methods have their applications in education as stated by Stasko (1997). However, they cannot be applied to software engineering, if effortless techniques are required. The interface typing can produce arbitrary complex structural visualizations, but lacks the ability to store the animation or reverse its temporal direction. Probing can fix this problem, but requires inheritance. Annotation can be applied to automatic code animation, but it has its weaknesses as it is an invasive method. Thus, by supporting multiple techniques and specification styles the system can allow the visualizer to choose the technique used, so that it would be the most suitable for the specified case.

4.1 Future

Obviously, the range of different techniques can be extended even further. At the moment, the research is aiming at even more effortless visualizations. One possible area of development is the intelligent systems that are capable of concept visualizations of arbitrary data structures. The idea is not completely new, but the Java language provides new powerful tools to implement the technique. From Matrix point of view this could mean a process where the conceptual displays are generated without the need to implement any of the interfaces.

References

O. Astrachan, T. Selby, and J. Unger. An object-oriented, apprenticeship approach to data structures using simulation. In *Proceedings of Frontiers in Education*, pages 130–134, 1996.

Ryan S. Baker, Michael Boilen, Michael T. Goodrich, Roberto Tamassia, and B. Aaron Stibel. Testers and visualizers for teaching data structures. In *Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education*, pages 261–265, New Orleans, LA, USA, 1999. ACM.

M.H. Brown and R. Sedgewick. A system for algorithm animation. In *Proceedings of SIGGRAPH'84*, pages 177–186. ACM, 1984.

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

J. Haajanen, M. Pesonius, E. Sutinen, J. Tarhio, T. Teräsvirta, and P. Vanninen. Animation of user algorithms on the web. In *Proceedings of Symposium on Visual Languages*, pages 360–367, Isle of Capri, ITALY, 1997. IEEE.

Chris D. Hundhausen and Sarah A. Douglas. Using visualizations to learn algorithms: Should students construct their own, or view an expert's? In *IEEE Symposium on Visual Languages*, pages 21–28, Los Alamitos, CA, September 2000. IEEE Computer Society Press.

Ari Korhonen and Lauri Malmi. Proposed design pattern for object structure visualization. In *The proceedings of the First Program Visualization Workshop – PVW 2000*, pages 89–100, Porvoo, Finland, 2001. University of Joensuu.

Ari Korhonen and Lauri Malmi. Matrix – concept animation and algorithm simulation system. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, pages 109–114, Trento, Italy, May 2002. ACM.

Ari Korhonen, Erkki Sutinen, and Jorma Tarhio. Understanding algorithms by means of visualized path testing. In *Software Visualization: International Seminar*, pages 256–268, Dagstuhl, Germany, 2002. Springer.

P. LaFollette, J. Korsh, and R. Sangwan. A visual interface for effortless animation of c/c++ programs. *Journal of Visual Languages and Computing*, 11(1):27–48, 2000.

B.A. Price, R.M. Baecker, and I.S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.

R. Rasala. Automatic array algorithm animation in c++. In *The Proceedings of the 30th SIGCSE Technical Symposium on Computer science education*, pages 257–260, New Orleans, LA, USA, 1999. ACM.

G.C. Roman, K.C. Cox, C.D. Wilcox, and J.Y. Plun. Pavane: A system for declarative visualization of concurrent computations. *Journal of Visual Languages and Computing*, 3:161–193, 1992.

J.T. Stasko. TANGO: a framework and system for algorithm animation. *IEEE Computer*, 23(9): 27–39, 1990.

J.T. Stasko. Using student-built algorithm animations as learning aids. In *The Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education*, pages 25–29, San Jose, CA, USA, 1997. ACM.

Model-Based Animation for Program Visualization

Amruth N. Kumar
Ramapo College of New Jersey, 505, Ramapo Valley Road, Mahwah, NJ 07430-1680

amruth@ramapo.edu

1 Introduction

We have been developing tutors to help students learn programming language constructs by solving problems. These tutors are capable of generating problems, grading the student's solution, providing immediate and detailed feedback about the correct answer, and logging the student's performance. A typical problem generated by our tutors includes a code segment, and the answer is qualitative, rather than quantitative.

Our tutors are designed to promote active learning. They target application in Bloom's taxonomy of educational objectives (Bloom and Krathwohl, 1956), and are expected to supplement the traditional programming projects assigned in a course, which emphasize synthesis. Education research indicates that focused practice such as that provided by the tutors is just as important for learning as contextualized and expansive projects (Mann et al., 1992), so we expect the use of tutors to complement the traditional programming projects. The tutors can be used to practice solving problems in class, for assignments after class and for testing in the course. The tutors are delivered over the Web, so they can be accessed any time, anywhere.

To date, we have developed and tested tutors on several topics, including expression evaluation in C++ (Krishna and Kumar, 2001), pointers for indirect addressing in C++ (Kumar, 2001), static scope and referencing environment in Pascal (Kumar, 2000), nested selection statements in C++ (Singhal and Kumar, 2000) and parameter passing in programming languages (Shah and Kumar, 2002).

The effectiveness of using such tutors has been well documented in literature:

- The field of Intelligent Tutoring Systems, which is the basis for our work, has documented an improvement of one standard deviation through the use of tutors (Anderson et al., 1995).
- The use of similar tutors has been shown to increase student performance by 10% in Physics (Kashy et al., 1997), largely due to increased time spent on the task.
- Our own work in building and testing tutors for Computer Science has shown that the average performance in a class improves by 100% after using the tutor (Krishna and Kumar, 2001), (Kumar, 2001), and that the improvement is systematic (Shah and Kumar, 2002).

2 Program Animation

Our tutors are designed to be user-driven: at each step, the user must interact with the tutor to generate a new problem, solve it, and obtain feedback. Therefore, our tutors promote active learning. However, since the problem (code segment), solution and feedback are all text, our tutors favor verbal learners over visual learners in the Felder-Silverman Learning Style model (Felder, 1993). We would like to address this shortcoming by incorporating graphic program animation into our tutors. Since execution of program code is sequential and appeals to sequential learners, we hope that providing graphic animation might also appeal to global learners in the Felder-Silverman Learning Style model.

Research indicates that novice programmers do not have an effective model of a computer, and that the model must be explicitly taught (Ben-Ari, 1998). The difference between the "notional" machine mentally constructed by novice programmers and the correct model can contribute to the difficulties encountered by novice programmers (Brusilovsky et al., 1997)(Boulay, 1989). Moreover, the details of program execution remain hidden in typical programming environments - this obscures the semantics of programming languages and contributes to the learning curve of novice programmers (Brusilovsky et al., 1997)(Boulay, 1989).

Program animation addresses both of these concerns: by presenting the "ideal" visualization, it helps the learner construct the correct model of a computer, or resolve his/her model with the correct model. By animating the effect of each program statement on this visualization, it explicates the hidden semantics of a programming language.

Researchers have concluded that textual explanations should accompany visualization for the visualization to be pedagogically effective (Naps et al., 2000)(Stasko et al., 1993). In our work, we are approaching this issue from the other end: our tutors already provide textual explanations, and we would like to supplement them with graphic animations. 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). We hope to apply this observation to visualize programs (rather than algorithms) that are dynamically generated by our tutors as part of the problems they present to the user.

Several systems have been built to animate programs. Some of the systems animate the execution of a program primarily in terms of the transfer of control in the program text - the animation consists of highlighting lines of code (e.g., DYNALAB (Boroni et al., 1996), THETIS (Freund and Roberts, 1996), AnimPascal (Satratzemi et al., 2001), EORSI (George, 2002)). Others animate the execution of a program in terms of the data objects affected by the control flow, and use graphic representation of data for the purpose (e.g,. TeaCup (Linington and Dixon, 2001), AnimalScript (Rossling and Freislebel, 2001), PSVE (Naps and Stenglein, 1996), JELIOT (Levy et al., 2000)). These two approaches may be suited for different purposes: control flow animation for programs that involve loops and selection statements, and data object animation for programs that involve parameter passing, linked list manipulation, pointers, etc. We plan to provide **control flow** animation in our tutors in the code text displayed as part of each problem. However, our primary focus in this work is to animate **data flow** in programs in terms of graphic objects, so that we can better address the needs of visual learners.

Animation tools can be categorized into two types: those that enable learners to construct their own animations and those that present animations pre-designed by the instructor for specific programs. Our objective is to present animations that are dynamically constructed by the system rather than predesigned by the instructor or painstakingly assembled by the learner. Such a system would be free to animate any arbitrary program. But, it would present animations at a pre-determined depth of detail, and with pre-defined notions of "ideal" graphic representation of objects and their animation.

In order to build an animation system that dynamically generates the animation for any arbitrary program, we plan to use model-based reasoning. Currently, we use model-based reasoning in our tutors to dynamically generate problems and program segments, solve them, and provide feedback. So, our proposed work would involve extending this technology to, in addition, animate the programs it generates.

3 Model-Based Animation

In Model-Based Reasoning (Davis, 1984), a model of the domain is first constructed, consisting of the structure and behavior of the domain. In our case, this would be a model of the programming language, consisting of objects in the language and their mechanisms of interaction. This model is used to simulate the correct behavior of an artifact in the domain. In our case, the model is used to simulate the expected behavior of some particular program or language construct, e.g., pointers. The correct behavior is compared with the behavior predicted by the learner for that artifact. The discrepancies between these two behaviors are used to hypothesize structural discrepancies in the (mental model of the student for that) artifact. In our case, the behavioral discrepancies are used to generate feedback to tutor the student. Figure 1 illustrates the architecture of a tutor that uses model-based reasoning for domain modeling (figure is adapted from (deKleer and Williams, 1987)).

Currently, we have modeled parts of the C++ programming language in our tutor as follows: we have modeled the objects in the language separately, e.g., variables, symbolic constants and functions. We have organized these objects in a hierarchy of composition, i.e., smaller objects are composed to create larger, aggregate objects, such as functions. The behavior of aggregate objects is derived by composing the behaviors of component objects. We plan to extend this model for program animation as follows:

- Each object will be in charge of its own visualization, especially local issues such as form and content.
- Aggregate objects will derive their visualization by composing the visualizations of component objects. Aggregate objects will be in charge of global issues for the visualization of component objects, such as layout.
- During simulation of the domain model, each object will render itself on to the visualization panel when it is first created in the program. After this, every event, i.e., action performed on the object will be animated through a corresponding change in its visualization.

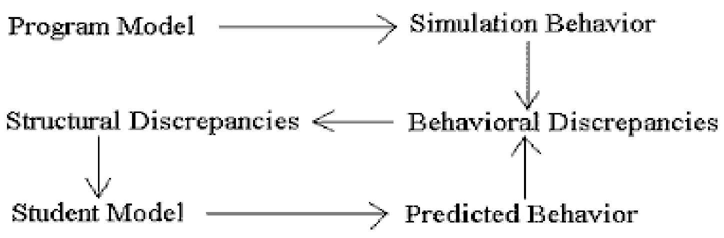


Figure 1: The Architecture of Model-Based Reasoning

3.1 Proposed Animation - An Example

We have been building a tutor to discuss dangling pointers and lost objects in C++ programs that involve pointers and dynamic memory allocation (Kumar, 2002). In this section, we will describe our proposed animation for this tutor. The following is a typical program presented by the tutor:

```
void compute()
{
   int * pointer;              // Pointer Declaration
   pointer = new int(25);      // Allocation
   {
      int number = 50;
      cout << *pointer;        // De-referencing
      pointer = &number;       // Assignment
      cout << *pointer;        // De-referencing
   }
   cout << *pointer;           // De-referencing
}
```

In the tutor, we would like to clearly illustrate the semantics of execution of the above program using animation. In particular, we would like to graphically illustrate the creation of a lost object during assignment to the pointer; and that the pointer is a dangling pointer when it is last de-referenced.

In our implementation, pointer, anonymous variable, and block objects are modeled separately. Each would be in charge of its own form, content and animation. For instance, the variable would remember that it is rendered as a rectangle, with the name to the left and the value inside. We will step through the execution of the function compute() to illustrate model based animation:

- In the beginning, the animation panel contains only the panel for the program of which `compute()` is a part. The program panel displays the data model of the program, consisting of the global space, the stack and the heap.
- When the function `compute()` is called, it creates a new panel with its name. It passes the handle to this panel up to the program. The program displays the function's panel at the top of the stack in its panel.
- When the pointer is declared, the pointer creates a rectangular graphics object, and passes the object's handle to the function. The function displays the object on its panel.
- When the pointer is allocated space, two events occur: an anonymous variable is created, and the pointer is assigned to point to the anonymous variable.

  The anonymous variable creates a rectangular graphics object, and passes the object's handle to the function, which in turn passes it onwards to the program, because the anonymous object is not bounded by function scope. The program displays the anonymous variable's graphic representation in the heap in its panel.

  The pointer must now point to the anonymous object. In order to facilitate the animation of assignment, the handles of the pointer and the anonymous variable are passed to the function, which passes them in turn to the program. The program draws an arrow from the pointer to the anonymous variable.
- When the program execution enters the block, the block creates a panel, and passes its handle to the function. The function displays the panel on top of its own panel, so as to enclose it, but not be obscured by it.
- When the variable is declared, the variable creates a rectangular graphics object, and passes the object's handle to the block. The block displays the object on its panel.
- When the pointer is de-referenced, its value, i.e., the value of the anonymous variable to which it points, is printed on a panel designated for output.
- When the pointer is assigned to point to the variable number, once again, the handles of the pointer and the variable are passed to the function, which passes them in turn to the program. The program draws an arrow from the pointer to the variable.

  The anonymous variable keeps track of the pointers pointing to it. When no more pointers are pointing to it, it changes the background color of its graphic representation to indicate that it is a lost object.
- When the pointer is de-referenced, its value, i.e., the value of the variable number is printed on the panel designated for output.
- When the block is exited, the background color of the block's panel is changed to indicate that the block is no longer accessible. Semantically, the panel should be disposed of, but this is less conducive to producing a mental model of a program trace than to simply disable the panel.
- When the pointer is de-referenced the third time, its value, i.e., the value of the now de-allocated variable number is printed on the panel designated for output, but in a color that indicates that the printed value is invalid.
- Finally, when the function `compute()` is exited, the background color of its panel is changed to indicate that it is no longer accessible. However, for accuracy, especially if the program contains multiple functions, we may want to dispose of the function's panel, i.e., pop it from the stack of the program panel.

During the execution of the program, we plan to animate data flow as described above. In addition, we plan to animate control flow by highlighting the code statements in the order in which they are executed. An additional option for control flow animation is to animate the function call graph and control flow graph of the program (Logiscope, http://www.telelogic.com).

Research comparing textual and graphical notations (Green et al., 1991)(Moher et al., 1991) shows that graphical notations are not a panacea, even though they may be easier to read, give an overview of program structure and supply more information. The question is to what extent the graphical (or for that matter, any) notation 1) makes information accessible, and 2) whether the user has adequate experience with that notation.

The most accurate notation for program animation is arguably the one that is true to the underlying implementation of the language. However, such a notation may not necessarily make the information clear; and it may not necessarily be easy for novices to understand. The "appropriate notation" for program animation appears to be subjective, and is influenced by all the following:

- **Time:** Once a function is exited, should its panel be left on the screen or removed? If it is left on the screen, it may help the learner understand the program by serving as a referent to past program objects and events. However, the resulting model would not be an accurate model of the underlying implementation of the language.

- **Space:** When a function is called, the activation record of the function that is pushed on the stack already contains designated spaces for all the local variables in the function. However, for clarity, we may want to add variables to the function's panel only after they are declared in the program. While this animation is clearer, it is not true to the underlying implementation of the language.

- **Learner's Knowledge Level:** In the two examples given above for time and space, the notational choices we make may be different if the user is advanced rather than a novice. We may choose to delete the panel of a function once it is exited, and we may choose to display all the local variables in the activation record of a function when the function is called, although we may choose to gray the variables until execution reaches their declaration in the program.

3.2 Advantages & Disadvantages

There are several advantages in using model-based reasoning for program animation:

- There is no need to separately script the animation. Since animation is entrusted to the model of the programming language, it is realized implicitly through message passing among the domain objects rather than explicitly through calls in the script.

- Since there is no need to write scripts to animate a program, the system can handle any program, without having to first preprocess it to generate the animation script. The completeness of the animation depends on the comprehensiveness of the model of the domain built into the system.

- Since the animation is assembled dynamically, this approach is more cost-effective than annotating programs individually.

For program animation, the actions performed during the simulation of the programs serve as adequate cues for model-based animation. On the other hand, for algorithm animation, the individual steps in an algorithm may not be at a sufficiently low level of abstraction to be directly translatable into cues for model-based animation.

Finally, a model-based system presents the "ideal" animation, i.e., the instructor's idea of what the animation should look like. It does not provide for learner control over issues of animation such as the level of detail and the choice of highlights.

3.3 Implementation

We are in the process of extending the models in our tutors to animate programs in addition to simulating them and providing feedback. Currently, our models have been organized in a hierarchy of inheritance and composition in the tradition of object-oriented programming. As per the Model-View-Controller pattern (Krasner and Pope, 1988), we have implemented the content, i.e,. program logic in the model, and form, i.e., presentation in the view.

Currently, program objects are presented in the form of text, i.e., various program statements that affect it. We plan to implement alternative views of objects to present their graphic visualizations. During simulation of the model to explain the behavior of the program, we plan to cue the animation of the program using the messages passed between the model and the view of each object.

We have proposed this scheme to generate program animation based on our experience generating explanations for the behavior of programs using model-based reasoning. Some issues that we will have to resolve during our implementation include coordinating the visualizations of multiple objects, and rewinding animations.

4 Evaluation

We have been conducting controlled tests of our tutors to evaluate their effectiveness. The protocol of our tests consists of a pretest, followed by a practice session with the tutor, and finally, a post-test. During the practice session, the control group and test group are given different treatments of the tutor to evaluate its effectiveness, e.g., tutor with no feedback versus tutor with detailed feedback.

We plan to use the above protocol to evaluate the effectiveness of using animation with our tutors - does it help students learn better? Is there a correlation between the improvement (if any) and the learning style of the student? We also expect to formatively evaluate the design of our animation.

Acknowledgments

Partial support for this work was provided by the National Science Foundation's Course, Curriculum and Laboratory Improvement Program under grant DUE-0088864.

References

J.R. Anderson, A.T. Corbett, K.R. Koedinger, and R. Pelletier. *Cognitive Tutors: Lessons Learned*, volume 4(2). Lawrence Erlbaum Associates, Inc., 1995.

M. Ben-Ari. Constructivism in computer science education. In *Proceedings of the 29th SIGCSE Technical Symposium*, pages 257–261, Atlanta, 1998.

B.S. Bloom and D.R. Krathwohl. *Taxonomy of Educational Objectives: The Classification of Educational Goals, by a committee of college and university examiners, Handbook I: Cognitive Domain*. Longmans, Green, New York, 1956.

C.M. Boroni, T.J. Eneboe, F.W. Goosey, J.A. Ross, and R.J. Ross. Dancing with dynalab: Endearing the science of computing to students. In *Proceedings of the 27th SIGCSE Technical Symposium*, pages 135–139, Philadelphia, 1996.

B. Du Boulay. *Some Difficulties of Learning to Program. Studying the Novice programmer*. Lawrence Erlbaum Associates, 1989.

P. Brusilovsky, E. Calabrese, J. Hvorecky, A. Kouchnirenko, and P. Miller. *Mini-languages: A Way to Learn Programming Principles*, volume 2 (1). 1997.

R. Davis. *Diagnostic Reasoning Based on Structure and Behavior*, volume 24. 1984.

J. deKleer and B.C. Williams. *Diagnosing Multiple Faults*, volume 32. 1987.

R. Felder. *Reaching the Second Tier: Learning and Teaching Styles in College Science Education*, volume 23(5). 1993.

S.N. Freund and E.S. Roberts. Thetis: An ansi c programming environment designed for introductory use. In *Proceedings of the 27th SIGCSE Technical Symposium*, pages 300–304, Philadelphia, 1996.

C. George. Using visualization to aid program construction tasks. In *Proceedings of the 33rd SIGCSE Technical Symposium*, pages 191–195, Northern Kentucky, 2002.

T.R.G. Green, M. Petre, and R.K.E. Bellamy. Comprehensibility of visual and textual programs: a test of superlativism against the 'match-mismatch' conjecture. In *Empirical Studies of Programmers: Fourth Workshop*, pages 121–146, Norwood, NJ, 1991.

E. Kashy, M. Thoennessen, Y. Tsai, N.E. Davis, and S.L. Wolfe. Using networked tools to enhance student success rates in large classes. In *Proceedings of FIE '97*, Pittsburgh, PA, 1997.

G.E. Krasner and S.T. Pope. *A Cookbook for Using the Model View Controller User Interface Paradigm in Smalltalk-80*, volume 1(3). 1988.

A. Krishna and A. Kumar. *A Problem Generator to Learn Expression Evaluation in CS I and its Effectiveness*, volume 16(4). 2001.

A. Kumar. Dynamically generating problems on static scope. In *Proceedings of The Fifth Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2000)*, pages 9–12, Helsinki, Finland, 2000.

A. Kumar. Learning the interaction between pointers and scope in c++. In *Proceedings of The Sixth Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2001)*, pages 45–48, Canterbury, UK, 2001.

A. Kumar. A tutor for using dynamic memory in c++. In *Proceedings of FIE 2000*, Boston, MA, 2002.

R.B. Levy, M. Ben-Ari, and P.A. Uronen. An extended experiment with jeliot 2000. In *Notes of the Program Visualization Workshop*, University of Joensuu, 2000.

J. Linington and M. Dixon. Picturing program execution. In *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2001)*, page 175, Canterbury, England, 2001.

P. Mann, P. Suiter, and R. McClung. *A Guide for Educating Mainstream Students*. Allyn and Bacon, 1992.

T.G. Moher, D.C. Mak, B. Blumenthal, and L.M. Leventhal. Comparing the comprehensibility of textual and graphical programs: The case for petri nets. In *Empirical Studies of Programmers*, pages 80–98, Norwood, NJ, 1991.

T.L. Naps, J.R. Eagan, and L.L. Norton. Jhave - an environment to actively enhage students in web-based algorithm visualizations. In *Proceedings of the 31st SIGCSE Technical Symposium*, pages 109–113, Austin, TX, 2000.

T.L. Naps and J. Stenglein. Tools for visual exploration of scope and parameter passing in a programming language course. In *Proceedings of the 27th SIGCSE Technical Symposium,*, pages 305–309, Philadelphia, 1996.

G. Rossling and B. Freislebel. Animalscipt: An extensible scripting language for algorithm animation. In *Proceedings of the 32nd SIGCSE Technical Symposium*, pages 70–74, Charlotte, NC, 2001.

M. Satratzemi, V. Dagdilelis, and G. Evangelidis. A system for program visualization and problem-solving path assessment of novice programmers. In *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2001)*, pages 137–140, Canterbury, England, 2001.

H. Shah and A. Kumar. A tutoring system for parameter passing in programming languages. In *Proceedings of The Seventh Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2002)*, Aarhus, Denmark, 2002.

N. Singhal and A. Kumar. Facilitating problem-solving on nested selection statements in c/c++. In *Proceedings of FIE 2000*, Kansas City, MO, 2000.

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, 1993.

On Visualization of Recursion with Excel

J. Eskola
Department of Computer Science, University of Helsinki, P.O. Box 26 (Teollisuuskatu 23), FIN-00014 University of Helsinki, Finland

J. Tarhio
Department of Computer Science and Engineering, Helsinki University of Technology, P.O. Box 5400, FIN-02015 HUT, Finland

Jukka.Eskola@cs.helsinki.fi

1 Introduction

Recursion is one of the fundamental concepts of Computer Science. Because novices have troubles to understand it (Ben-Ari, 1997), several visualizations and animations (Dann et al., 2001; Dershem et al., 1999; Jiménez-Peris and Patiño-Martínez, 2001; Terada, 2001; Tung et al., 2001) have been developed to demonstrate it.

Spreadsheet programs like Microsoft Excel offer powerful facilities to visualize, illustrate, and describe different phenomena and experiment with these. Rautama et al. (1997) present how simple algorithms can be animated with Excel. The expressive power (Dybdahl et al., 1998) and user-friendliness make Excel a fascinating tool for educational visualizations.

In this paper we consider ways to apply Excel to visualize recursion. Earlier we have implemented an iconic programming environment (Eskola and Tarhio, 2001) called Flopex (FLOwchart Programming with EXcel) running on Excel. Currently we are working on animation of data flow languages. Because as a part of this work we need a visualization of recursion, we decided to enhance our Flopex system with recursion, for it is likely that we will also implement our data flow animations with Excel.

The rest of this paper is organized as follows. In Chapter 2 we review some earlier visualizations of recursion. Then we describe the main features of the original Flopex, which we now on call Flopex 1, and introduce the new features of Flopex 2 in Chapter 3. Besides Flopex we will consider other ways to demonstrate recursion with Excel. In Chapter 4 we consider visualizations of call trees. Before the concluding remarks, visualization of inefficient recursion is discussed in Chapter 5.

2 Visualizations of recursion

Several program visualization systems have been developed for visualizing and explaining the concept of recursion. As our purpose is to demonstrate recursion, we will shortly review some of the earlier works.

Alice98 (Dann et al., 2001) uses 3-D animations for program visualization offering an approach to introducing fundamental concepts such as recursion to novice programmers. Alice serves as a programming language environment where students can immediately see their programs in action. Alice has been constructed on the top of the Python language and takes advantage of many Python's features. Multiple recursion, such as the Towers of Hanoi can also be visualized by Alice. When visualizing recursion in this way students become aware of that the time needed for the animation is straightly related to the number of moves needed to transfer $n-1$ rings from the original peg to the spare peg. In essence recursive calls for the animated actions in Alice do not get stacked on the top of one another which might evoke that Alice animation do not provide true recursion.

Terada's source code animation system (Terada, 2001) called "Program Paper-Slide-Show" uses overlapping slides in the window and their smooth motions to teach recursive programs to novice users. Function calls (activations) are performed by a slide displaying the callee's source code, moved smoothly from the right side of the window onto the previous slides. The rightmost slide represents the current function and nested calls can be seen as overlapping slides, in which the current control point is shown by a highlighted color belt. The idea of Terada's system, the "paper-slide-show" is based to the traditional form of Japanese children's entertainment.

Vismod (Jiménez-Peris and Patiño-Martínez, 2001) is an interpreter system for imperative languages to visualize recursion and dynamic memory. Activations of recursive programs in Vismod are shown in separate windows, and in each window there is represented the code of the subprogram, highlighting a statement which is going to execute next. The local variables and arguments are represented in an extra pane which allows also the associated data of each activation to be seen. However, when a recursive program makes multiple calls, the content of the calls can be lost. For example the computation of Factorial is visualized properly, but there is a problem in visualizing of Fibonacci numbers. One solution of this problem could be the visualization of the call tree.

The Java Function Visualizer (Dershem et al., 1999) allows the visualization of function calls for Java functions to enhance understanding of recursion. When the visualization starts, the code lines are highlighted as they are executed and when another function is called a new window appears on the top of the previous one constructing a visualization of the function call stack. If the recursion is performed in that way it may distort the idea of recursion, because the user can assume that the program copies a new source code on the top previous window. The same problem is present in Terada's system (Terada, 2001).

RainbowScheme (Tung et al., 2001) is a program visualization system designed to help CS students to understand and analyze recursive programs. RainbowScheme presents the run-time state of a Scheme program using icons, colored environment trees, and colored code. The program can be executed step-by-step by using a set of visual transformation rules called visualcode. Being able to visualize intermediate steps of a running program makes each individual step and the entire execution process easier to comprehend.

3 Flopex 1

Flopex 1 (Eskola and Tarhio, 2001) is an iconic programming environment running on Microsoft Excel. Flopex 1 visualizes the execution of a flowchart, which is constructed with ready-made flowchart icons.

The development of Flopex 1 had three aims. The most obvious purpose was to help novices in learning programming. The second goal was to provide a lightweight animation system, which is easy to adapt to various purposes. The third aim was to demonstrate the capabilities of Excel as a platform for implementation of visual languages and making experiments with them.

Flopex 1 is an Excel application that allows the user to draw an algorithm as a flowchart and to follow the visual execution of a flowchart. At first the user constructs the flowchart by combining icons which represent control blocks and flow lines. The icons have conventional shapes and distinct colors for easy recognition.

When the flowchart has been completed, the user initializes the system and starts the interpreter, which has been written in *Visual Basic for Applications* (VBA), the macro language of Excel. The flowchart is the input of the interpreter. The control starts from the start icon of the flowchart and traverses the flowchart according to the actions connected with each icon. The user can apply two execution modes: continuous or step-by-step. The control of execution proceeds according to the direction of flow lines and the values of conditional expressions. The active cell is highlighted during execution. The continuous execution stops when the end icon is entered or an error is detected.

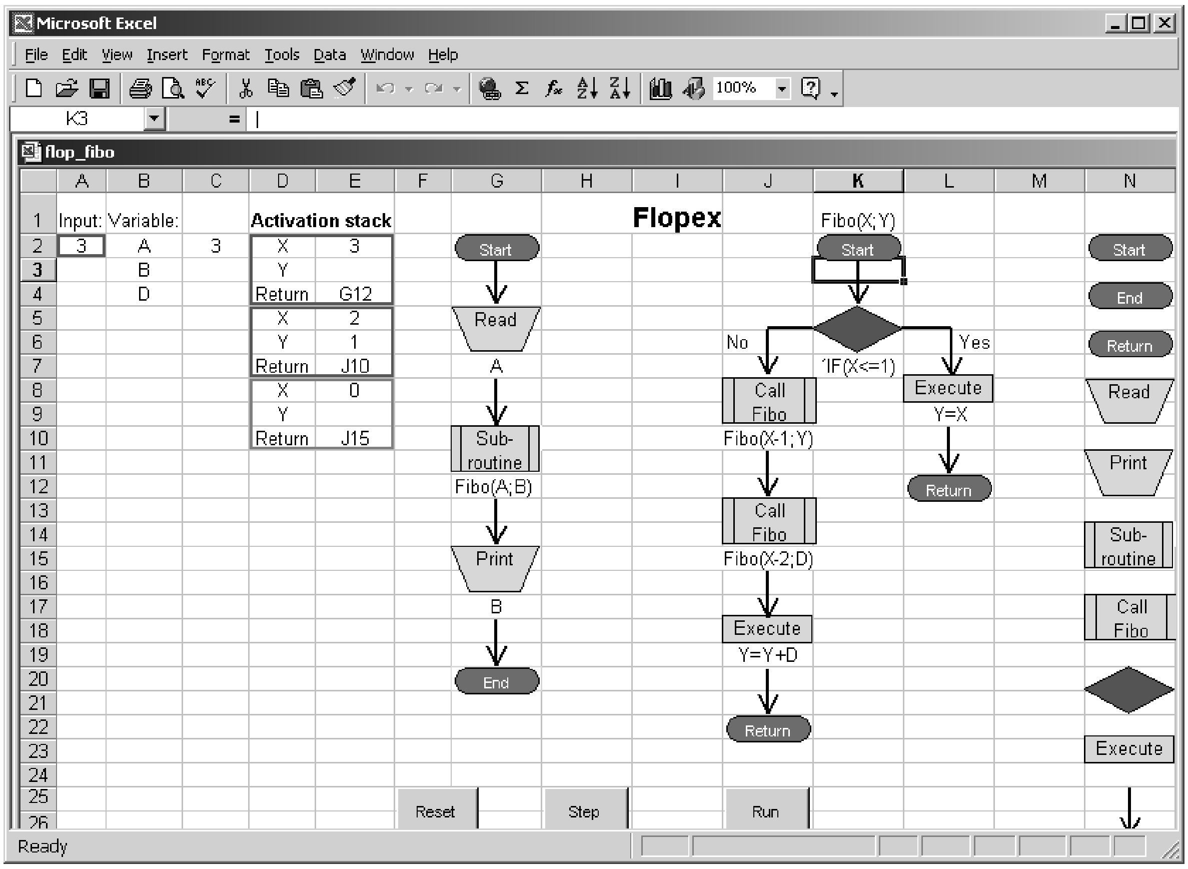
The variables of an algorithm are represented as cells of the sheet. Thus the user sees the value of a variable on the screen all the time during execution. This makes also possible to animate user data structures automatically in a graphical form during the visual execution of a flowchart.

4 Recursion in Flopex 2

Subroutines and recursion are new features of Flopex 2. Otherwise Flopex 2 works as Flopex 1. The topmost icon of the flowchart of a subroutine is the start icon as the main flowchart, but all the paths end with the return icon. Above the start icon of a subroutine is the heading of the subroutine as text:

identifier(X; Y), where X is the input parameter and Y the output parameter. (For clarity we consider the case, where there is only one input parameter and one output parameter.) In addition to being the output parameter, Y serves as a local variable of the subroutine. A subroutine call is denoted with the subroutine icon. Under the subroutine icon is the actual subroutine call as text. The form of the call is identifier(A; B), where A is an expression for X and B is the variable where the final value of Y is assigned to.

During execution the contents of the activation stack is shown in the columns D and E of the sheet. When the subroutine is recursive, the user sees the active history of recursion. Each item of the stack is a record containing a return address and the values of X and Y. When a call happens, a new activation record is created at first. Then the return address is stored to the record and the value of A is copied to the field corresponding to X. When a return happens, the value of Y is assigned to the variable B, the return address is fetched, and the record is popped.



	A	B	C	D	E
1	Input:	Variable:		Activation stack	
2	3	A	3	X	3
3		B		Y	
4		D		Return	G12
5				X	2
6				Y	1
7				Return	J10
8				X	0
9				Y	
10				Return	J15

Figure 1: A recursive flowchart for computing Fibonacci numbers.

In Fig. 1 there is an example of a recursive flowchart for computing Fibonacci number 3. The figure represents the situation, when subroutine Fibo has been called four times. In activation stack (E2:E10) there are now three activation records from which we show local variables X and Y, and a return address 'Return'. Table 1 shows phases of recursion in the activation stack.

5 Tree of subroutine calls

The tree of subroutine calls demonstrates several aspects of a recursive program. We consider how to visualize such a tree with Excel. There are two versions of the tree: a history of calls and pending calls. We start with the former. We study a recursive subroutine fibo computing Fibonacci numbers.

```
procedure fibo(a,b);
```

Table 1: Phases of the activation stack.

X	3	3	3	3
Y				
Return	G12	G12	G12	G12
X		2	2	2
Y				1
Return		J10	J10	J10
X			1	0
Y			1	
Return			J10	J15

```
var c1,c2:integer;
if a<2 then b:=a
else {
   fibo(a-1,c1);
   fibo(a-2,c2);
   b:=c1+c2;}
```

We augment the subroutine with calls taking care of drawing the tree. The range (x1, x2) is reserved for the x coordinates of a subtree. The root of a subtree is placed at x3 = (x1+x2)/2, i.e. in the middle of the range.

```
procedure fibo1(a,x1,x2,y,b);
var c1,c2:integer; x3:real;
x3:=(x1+x2)/2;
draw_node(a,x3,y);
if a<2 then b:=a
else {
   draw_edge(x3,y,(x1+x3)/2,y-1)
   fibo1(a-1,x1,x3,y-1,c1);
   draw_edge(x3,y,(x3+x2)/2,y-1)
   fibo1(a-2,x3,x2,y-1,c2);
   b:=c1+c2;}
```

The routines implementing `draw_node` and `draw_edge` of fibo1 are present in common graphic packages, but the scaling of the drawing area needs some additional code.

Although we described here the drawing calls explicitly, it is straightforward to implement such feature in an automatic animation system like Jeliot (Haajanen et al., 1997), because Jeliot just annotates the user code with animation calls, from which the user selects which are shown.

The form fibo2 of the subroutine has been made for Excel.

```
Sub fibo2(a As Integer,x1 As Double,x2 As Double,b As Integer);
    Dim x3 As Double
    Dim c1 As Integer
    Dim c2 As Integer
    x3=(x1+x2)/2
    Call insert_node(x3,a)
    If a<2 then
       b=a
    Else
        Call fibo2(a-1,x1,x3,c1)
```

```
            Call insert_node(x3,a)
            Call fibo2(a-2,x3,x2,c2)
            Call insert_node(x3,a)
            b=c1+c2
        End If
    End Sub
```

When we have the coordinates of points on an Excel sheet, the XY chart shows a break line that connects the points. Any graph can be drawn as a break line which travers a part of edges several times if necessary. We apply this trick as follows. The call `insert_node` adds the coordinates of a

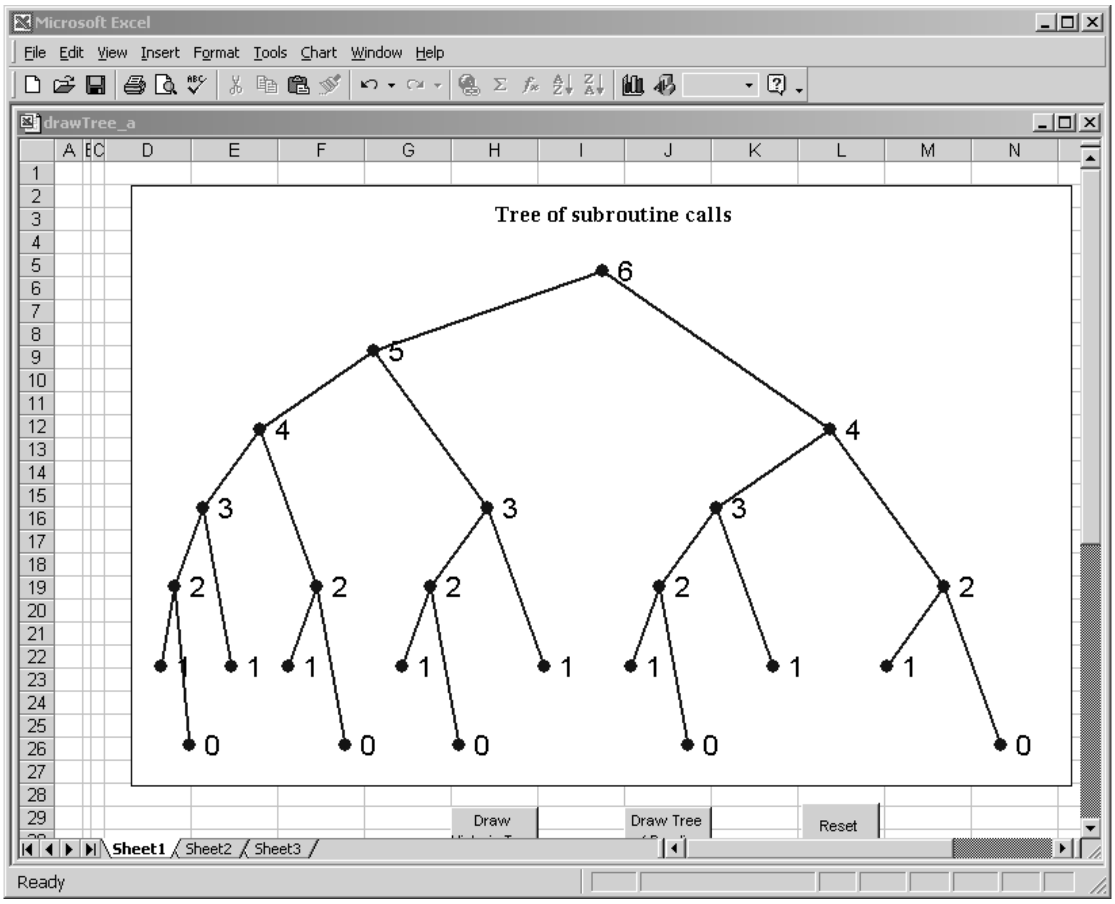


Figure 2: A history tree for the Fibonacci(6).

node to a table on an Excel sheet. The tree of subroutine calls is seen, when this table is shown as a XY chart such that y values are shown as data labels (see Fig. 2). Excel takes care of scaling the tree. Note the value of a is used explicitly as the y coordinate of a node.

The insertion of a node for three times simulates depth-first search of a graph. Each edge of the tree is drawn twice and every inner node is drawn for three times.

The tree of pending calls (see Fig. 3) is got with a small modification to fibo2. In the end of the subroutine, three last nodes should be removed from the table. Note that the necessary modification is simpler than in the case of fibo1.

We have plans to integrate these visualizations with Flopex. We also consider to implement a general Excel application which shows the call tree based on a call trace (given as input) of an arbitrary program.

6 Visualization of inefficient recursion

There are several algorithmic problems which have a straightforward divide-and-conquer algorithm with exponential time complexity. However, in many cases there is only a polynomial number of subproblems. Then the problem can be solved in polynomial time with dynamic programming (Aho et al., 1982). Computing a Fibonacci number is such a problem, other examples include matrix multiplication, triangulation of a polygon, and the win probability in a play-off series.

In the inefficient solutions of these problems, recursion is the cause of exponential time. A recursive subroutine is repeatedly called with the same set of parameters. A straightforward correction goes as follows. The subroutine stores the values of input parameters and the result in a global array. The subroutine checks in the beginning, whether the same set of input parameters has appeared before. If it has appeared, the stored result is returned instead of computing it once more.

This process is particularly easy to visualize with Excel. The algorithm may be a VBA program or a flowchart program in Flopex. At first the original inefficient version is run while showing the history tree of subroutine calls. Then the algorithm is run so that each set of input parameters is stored in a table on the sheet and the stored value of the function is always used when available. Note that it is straightforward to automate the process of showing these two runs.

This approach demonstrates in a nice way the consequences of undesired way of using recursion and gives hints to a student how to improve the algorithm under study.

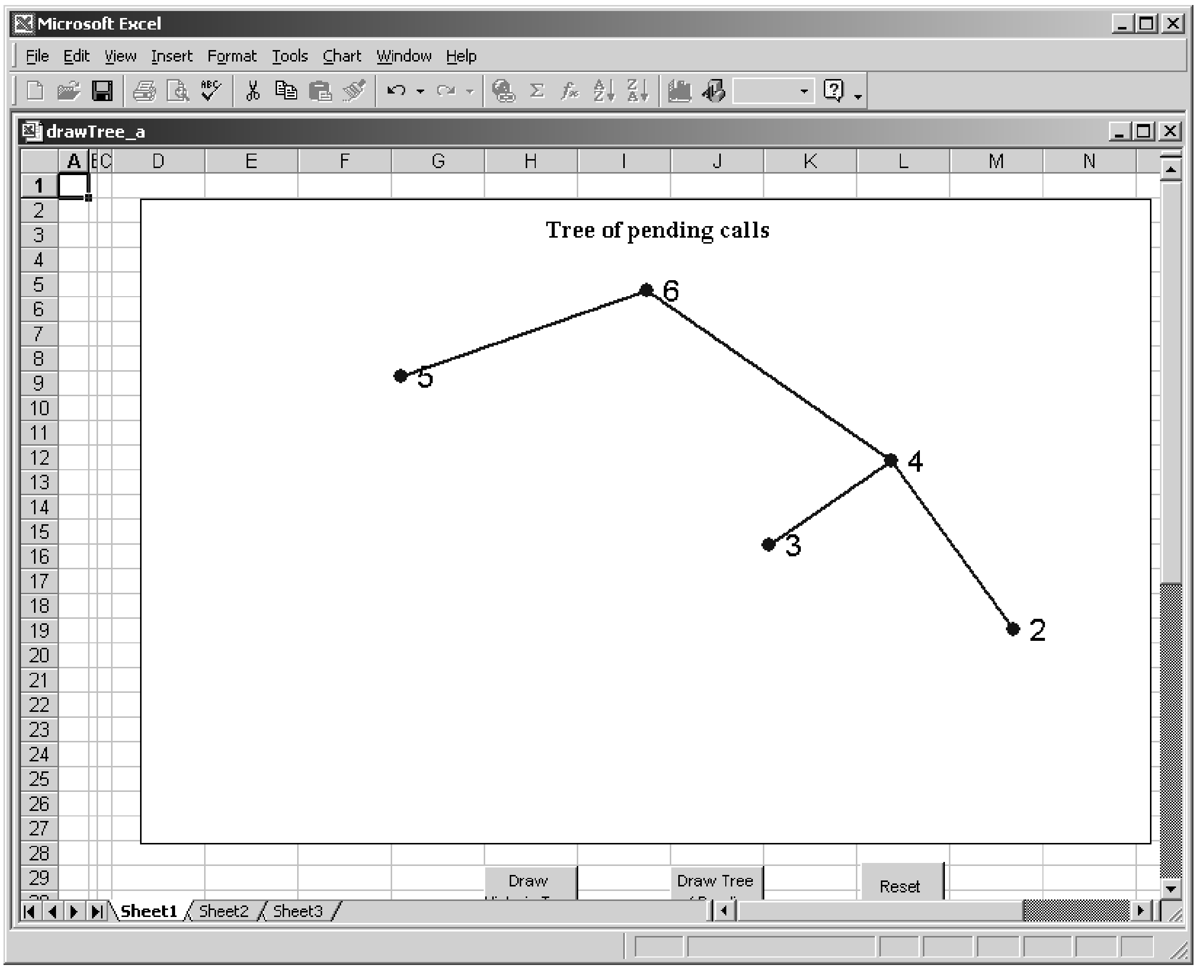


Figure 3: A tree of pending calls, generated as a snapshot of Fibonacci(6), when Fibonacci(2) has just been evaluated.

7 Concluding remarks

We examined ways to use Microsoft Excel to visualize recursion. We presented how we enhanced Flopex, our flowchart programming environment with subroutines and recursion. Because the interpreter of Flopex is written in VBA, the code of the interpreter is available to any user. So Flopex provides an open environment where a student can also examine or alter the implementation of recursion in addition to the visualization.

We also presented how to draw recursive call trees with Excel. Although this was rather easy, Excel did not offer much additional value when compared with other visualization packages.

Finally we considered how to apply Excel to reveal inefficiency in divide-and-conquer algorithms caused by recursion. This task is easier to implement with Excel than in some other way, because one has all three aspects at hand: visualization, interpreted language, and a visual table.

As a conclusion, we can say that Excel is an excellent tool for light educational visualizations. Integration of a programming environment with spreadsheet graphics has implicit expressive power, which makes Excel ideal for visual experiments and fast prototyping.

References

Alfred V. Aho, John E. Hopcroft, and Jeffrey Ullman. *Data Structures and Algorithms*. Addison Wesley Publishing Company Reading, Massachusetts, 1982.

Mordechai Ben-Ari. Recursion: From drama to program. *Journal of Computer Science Education*, 11 (3):9–12, 1997.

Wanda Dann, Stephen Cooper, and Randy Pausch. Using visualization to teach novices recursion. In *Proc. of the 6th Annual SIGCSE/SIGCUE Conference on ITiCSE'2001, Canterbury, UK, 2001*, pages 109–112. ACM, 2001.

Herbert L. Dershem, Daisy Erin Parker, and Rebecca Weinhold. A java function visualizer. *Journal of Computing in Small Colleges*, 15(1):220–230, 1999.

Arne Dybdahl, Erkki Sutinen, and Jorma Tarhio. On animation features of excel. In *Proc. of the 3rd Annual SIGCSE/SIGCUE Conference on ITiCSE'1998*, pages 77–80. ACM, 1998.

Jukka Eskola and Jorma Tarhio. Animation of flowcharts with excel. In Erkki Sutinen, editor, *Proc. of the First Program Visualization Workshop*, International Proceedings Series 1, pages 59–68, University of Joensuu, Department of Computer Science, 2001.

J. Haajanen, M. Pesonius, E. Sutinen, J. Tarhio, T. Teräsvirta, and P. Vanninen. Animation of user algorithms on the web. In *Proc. VL '97, IEEE Symposium on Visual Languages, Capri, Italy, September, 1997*, pages 360–367. IEEE, 1997.

Ricardo Jiménez-Peris and Marta Patiño-Martínez. Visualizing recursion and dynamic memory. In Erkki Sutinen, editor, *Proc. of the First Program Visualization Workshop*, International Proceedings Series 1, pages 111–120, University of Joensuu, Department of Computer Science, 2001.

Erkki Rautama, Erkki Sutinen, and Jorma Tarhio. Excel as an algorithm animation environment. In *Proc. of the 2nd Annual SIGCSE/SIGCUE Conference on ITiCSE'1997, Uppsala, Sweden, 1-5 June, 1997*, pages 24–26. ACM, 1997.

Minoru Terada. Animating c programs in paper-slide-show. In Erkki Sutinen, editor, *Proc. of the First Program Visualization Workshop*, International Proceedings Series 1, pages 79–88, University of Joensuu, Department of Computer Science, 2001.

Sho-Huan Tung, Ching-Tao Chang, Wing-Kwong Wong, and Jihn-Chang Jehng. Visual representations for recursion. *International Journal of Human-Computer Studies*, 54(3):285–300, 2001.

ANIMAL-FARM: An Extensible Algorithm Visualization Framework

Guido Rößling
Department of Computer Science, Darmstadt University of Technology, D-64283 Darmstadt, Germany

roessling@acm.org

1 Introduction

One central challenge in computer science lies in understanding the dynamics of algorithms and data structures. This is of particular importance in two areas: computer science education and programming, especially concerning debugging. Understanding both how and why an algorithm works is challenging for students and novice programmers. Detecting and fixing bugs can be difficult even for advanced programmers.

Software teaching novices how to program and debug their code is increasingly used in computer science education. Algorithm Visualization (AV) systems are a special application of software visualization, focusing on visualizing the dynamic behavior of software. The interest in AV from computer science education has steadily increased over the last years, as evidenced by the number of publications on the topic.

A large number of AV systems are already available, mainly from educational institutions. They differ in a large number of ways, including the degree of interactivity, display strategy and the type of supported input. They also range from non-interactive slide show-like displays (Gosling et al., 2001) to full-fledged debuggers that let the user specify attribute values and invoke methods (Stasko, 1998; Faltin, 1999). A large variety of different display control elements exist between the two extremes. Finally, the input data used by AV systems ranges from actual source code in a specific programming language (Ben-Bassat Levy et al., 2001) to interactive graphical editing using direct manipulation (Stasko, 1998).

The large variety of available systems presents a formidable challenge for users interested in employing AV. The strengths, weaknesses and restrictions of a given system are usually not obvious at first glance. One drawback common to most current AV systems is the lack of support for easy extension and customization. It is highly unlikely that a given fixed system can meet all demands of future applications (Khuri, 2001). However, most systems cannot easily be extended with additional features.

In this paper, we present an AV framework that is comparatively easy to implement. The framework is geared for flexible general-purpose AV systems that can in principle animate any type of algorithm. Several framework components can be added or removed on the fly. The input type is not fixed, allowing for a wide range of different animation generation approaches. Advanced controls including the reverse playing of steps are prepared in the framework and do not require much work by developers encoding AV systems based on the framework.

The paper is structured as follows. In Section 2, we present some requirements for AV frameworks. Section 3 presents basic design issues of the framework, followed by the framework components in section 4. Section 5 concludes the paper and outlines areas of future research.

2 AV Framework Requirements

A framework consists of a set of cooperating classes that together represent a reusable design for a specific class of software (Gamma et al., 1995). In this chapter, we focus on motivating a framework for algorithm visualization. The framework defines the basic architecture of all AV systems built using it. In general, it emphasizes the ability to reuse the basic design rather than individual classes or pieces of code.

Frameworks are built for reuse. Typically, a system developer will have to extend certain framework classes and follow implementation guidelines to build a concrete framework-based application.

Frameworks usually contain fixed and ready-to-use components as well as abstract classes and interfaces that have to be filled with content by the developers. All AV systems based on the framework will therefore have the same core components and a similar structure of classes, making them easier to maintain and more consistent.

The downside of framework development is that specifying a good framework structure is very difficult. Significant amounts of work have to go into creating a framework structure that can be reused easily on the basis of its merits regarding conciseness, understandability and easy applicability. The benefit of the framework is that it can make developing new concrete implementations much easier.

One issue in specifying an AV framework is making it applicable for the large variety of different understandings of what constitutes AV. For example, there are large conceptual differences between topic-specific systems that illustrate a single algorithm or a limited topic area, and code interpretation-based systems that illustrate the actual behavior of the underlying code. Therefore, the framework should not fix the animation generation approach.

Additionally, it is highly unlikely that any given AV system will be able to address all expectations of the target audience (Khuri, 2001). The "standard" process in many areas of software development is to start developing a new system once a (foreign) system is found to be inadequate in some areas. Instead, it would be preferable if one could simply extend the other system to address the required functionality. The key word here is *simply*: usually, even if the source code of the system is available, adding extensions is only possible after a detailed study of the implementation. Additionally, the developer will often have to modify parts of the (foreign) source code. In the sense of object-orientation, we therefore postulate that the framework should be easily extensible, based on a documentation of its structure – without requiring any deep system knowledge, and if possible also without modifying existing code.

There are a variety of additional requirements that the framework can already address, thus providing the functionality to all implementing concrete systems. First of all, a clean specification of the participating parties in animation helps developers implement the concrete systems, as well as extensions. If the main animation component classes are fixed, the framework can already provide a complete default implementation of the animation display front-end. This front-end has to cover advanced control functions, such as setting the magnification and speed of the animation display. A dynamic and static view of the individual animation steps shall also be incorporated.

3 Framework Design Issues

As discussed, the two central design goals of the ANIMAL-FARM AV framework are extensibility and configurability. Thus, it shall be comparatively easy to implement new functionality for a framework-based system, ideally without having to touch the actual source code. Achieving this ambitious goal is not easy, regardless of whether we examine the issue from an AV angle or from software engineering.

We have isolated four target areas that need addressing in a dynamically extensible framework:

- Internationalization issues,
- Representation of object state,
- Dynamic component loading and configuration,
- Component decoupling.

Note that these requirements are not specific for AV usage, but apply to all conceivable extensible frameworks. Internationalization touches on the required support for translating the given system's GUI front-end on the fly to the selected target language. It is a powerful feature that can instantly make any system more attractive to prospective users, provided that their mother tongue is supported with good translations. However, while translating the user interface may be helpful, it is rendered ineffective if the actual content remains in a different language. Therefore, the framework shall also be able to address content translation. For this end, we have implemented a dedicated Java package that allows the easy creation of translatable Swing-based GUI elements. In fact, the generation of Swing elements is easier using this package than coding them "normally", without internationalization support!

Both the translation of the front-end and the AV content relies on the presence of external files containing the translated content. Depending on the presence of language resource files, the translation into an arbitrary number of languages is possible on the fly. Note that content translation is only possible if the AV components are laid out dynamically using relative coordinates, as the width of translated texts varies widely between languages. For GUI elements, the internationalization support translates all possibly affected entries for each translatable component. Thus, changing the language will typically affect the label, tool tip text and hot key of a button or menu item, and can also impact the optional icon.

The representation of object state presents an important consideration for extensibility. Typically, object state is represented by attributes. The advantages of this approach are well-known: fast access and type checking. However, adding new functionality to the system may in some cases also require the introduction of one or more new attributes, and thus the modification of existing code. In general, we want to avoid the need for code modifications, as any modification may introduce bugs and make reading the code more difficult for the next developer.

Therefore, we have decided to represent the object state by properties. In Java, properties are instances of the *java.util.Properties* class, a specialized hash table that stores String key and value pairs. The framework also benefits from the default values supported by properties. We only had to implement a customized conversion class that converts various typical attribute types to and from a String representation, for example colors, points or font information. Using properties allows the developer to introduce new property values by simply inserting them into the properties hash at run-time, without requiring any code modification. The runtime differences between property access and direct attribute access are minimal, and are certainly acceptable due to the additional expressiveness we gain.

To make the framework as dynamic as possible, we have further decided to load the individual components dynamically based on a set of configuration files. These files specify the actual components to use, and are located based on the directory where the system was started. Therefore, users can in principle have as many different configurations as there are directories in the file system. Note that this also means that multiple different configurations can share the same code base, for example an end-user and a developer who is currently testing experimental features. In this case, the end-user will be completely unaware of the experimental code.

We maintain the loaded components in a hash structure. The hash table entries act as prototypes for new instances and are cloned to generate the actual components. The framework is therefore able to add or remove individual components at run-time. Note that the removable components are restricted to "non-core" components to ensure that a meaningful system remains after reconfiguration.

Finally, to enhance the development of extensions, we have followed a strict component decoupling. Instead of letting two connected components communicate directly with each other, we introduce a *handler* class as an intermediate agent. The handler is responsible for satisfying queries regarding the offered functionality of a given entity, also based on its current state. When a given functionality is requested for performance, the handler maps the functionality into the appropriate set of operations. As handlers are tied to classes rather than concrete instances, implementing these operations is usually easy. Additionally, the developer can add or remove handler extensions at run-time, which for all purposes act as if their code were specified inside the original handler code. However, they make touching the handler code for new functionality unnecessary. The strict separation of concerns resulting from this approach makes implementing the components connected by a handler much easier.

4 Framework Components

The framework we propose consists of five main segments: an import and export layer, a display and editing front-end, and the internal animation representation. Figure 1 shows the conceptual components of the framework. Note that the editing front-end is system-specific and therefore only prepared in the framework, but not directly provided.

The main participating parties in an animation are graphical *primitives*, *animation effects*, and *handlers*. *Primitives* present a simple representation of reusable graphical objects, for example texts, arcs, polygons and points. In our framework, primitives are mainly responsible for two operations: encoding their current state with appropriate methods for modifying the state, and painting themselves using the standard Java *paint(java.awt.Graphics)* method. Typically, the state can be queried or modified using a pair of *getXX / setXX* methods, where *XX* stands for the name of the property or attribute to retrieve or modify.

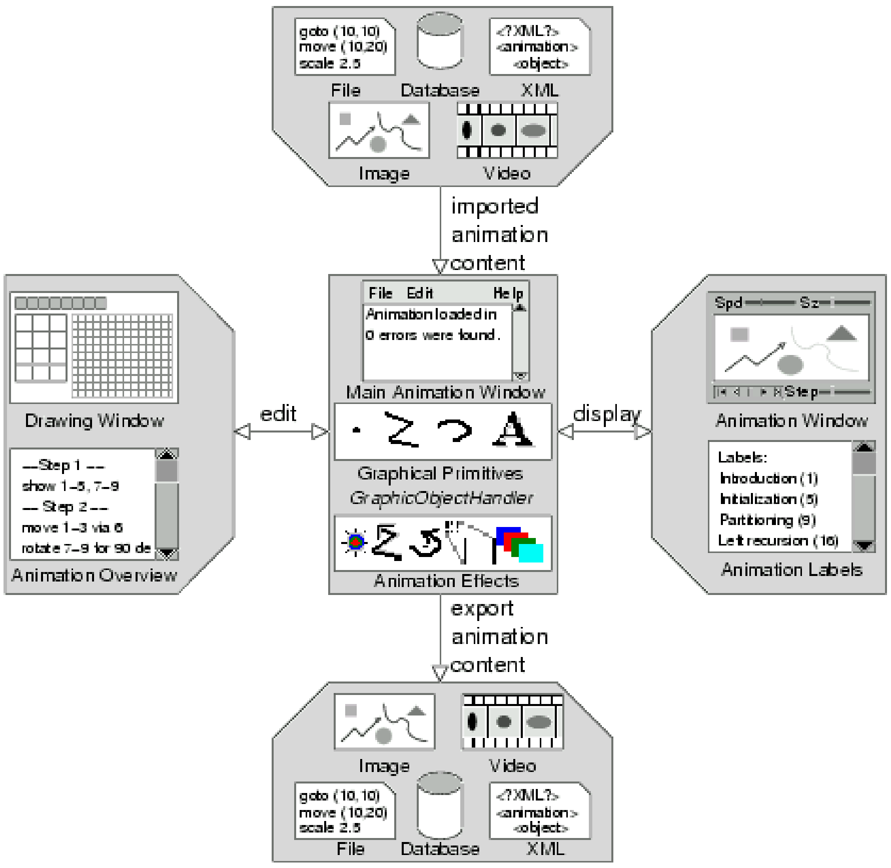


Figure 1: Conceptual Framework Component

Animation effects represent a generic instance of a family of transformation types. For example, all conceivable ways of moving a set of primitives can be encoded in one animation effect, typically called *Move*. The responsibility of the animation effect is twofold. First, it has to determine the concrete animation effect subtype to perform on the selected primitives, for example *"move the second node"*. Second, it has to resolve the timing specification of the effect and determine an appropriate intermediate value.

Handlers in our framework act as negotiating agents between animation effects and primitives. They determine the set of possible transformation types applicable to a primitive and provide this to the animation effect. The animation effect can then build an intersection of all such sets returned by the individual selected primitives and offer this list to the user building the visualization. When a given animation effect has to be performed, the animation effect determines the abstract target state to assume for the current point in time. Abstract target state here means that the state to assume does not depend on the underlying primitives, but only on the timing specification of the animation effect. For example, the target state to assume after 80% of a given move effect have passed is always the same point on the object along which all primitives are moved. The location of the primitives after the operation is highly likely to vary, but their relative movement is still identical. The effect then forwards this information along with the animation effect name to the handler of each selected primitive. The handler is then responsible for mapping the animation effect into a set of method invocations on the primitive that causes the appropriate visual effect.

The handler class offers several key advantages. It offers a very clean separation of concerns for primitives and animation effects. Primitives only take care of painting themselves according to their current state. Animation effects only resolve abstract animation operations without having or needing contextual awareness.

Due to the strict separation of concerns encouraged by the handler, developing new primitives or animation effects becomes much easier. When implementing a primitive, the programmer does not have to reflect on how the primitive could be animated. In fact, the implementation can be just the same as for any "static" drawing program! Implementing an animation effect also requires no knowledge about the primitives to animate, but only the skill to interpolate the intermediate states for each point in execution. Implementing a new handler is usually also very easy to accomplish, as the handler is tied to exactly one primitive class type and therefore fully aware of its usage context.

Together, these three classes provide good support for reuse and extensibility. However, the framework goes further than that in two respects. First, the representation of the internal state of primitives and animation effects is not based on fixed attributes, but rather on dynamic lookup structures (*java.util.Properties*, to be precise). Thus, developers can also add or remove "attributes" for representing the instance at run-time.

Second, the components are not statically assembled based on import statements in the source code. Rather, they are gathered from a configuration file that specifies the names of the classes to load for both primitives and animation effects. Note that this is not possible for basic handlers, as they are fixed to the primitive they handle. However, handler extensions can be loaded and integrated dynamically. The configured components are loaded dynamically and gathered as Prototypes (Gamma et al., 1995). Whenever a new instance is required, the prototype is retrieved from the hash table and cloned. Note that the delay caused by this look-up process and cloning is usually negligible even when large amounts of objects are used in the system.

The full structure of the animation is separated in two aspects. The "static" view of the animation, consisting of all primitives, animation effects and links connecting the animation steps, is gathered in a Singleton (Gamma et al., 1995) object called *Animation*. The dynamic state of the animation, for example representing "now" during the display of the animation, is gathered in an *AnimationState* object.

When a given animation state is to be executed, the *AnimationState* clones all animated primitives and causes them to assume the state at the start of the current animation state by quickly executing all previous animation steps. Again, the delay caused by this operation is usually negligible. All animation effects are then initialized based on the current point in time. During a looped execution of the animation step, each animation effect checks whether it has to work (that is, the offset before its execution has passed and the effect has not yet finished). If so, the percentage state of execution to assume is calculated from the start time of the animation step, the effect's offset and duration as well as the current time.

The animation effect determines its target state based on this information. This state is then forwarded to the individual handler Singletons, along with the previous state, the effect name and the current animated primitive. The handler Singleton examines the effect name and both old and new state to determine appropriate method invocations on the primitive. As the net effect, the primitive assumes the correct state as defined by the animation effect.

To allow for better reuse and extensibility, we also support the dynamic integration of *handler extensions* which provide additional animation types for a given primitive. These are also configured and loaded dynamically at run-time and maintained in a hash table.

The import and export layer of the framework provides a simple yet expressive abstract filter class. The individual filters can be added or removed at run-time, just as all other components. The correct filter to use for a file is determined by the user within a specially adapted file dialog. The selection is performed on the basis of the file's MIME type and extension.

The graphical front-end for animation playing supports the setting of the display speed and magnification. It offers a full-fledged video player control bar, including smooth forward and reverse playing, as well as forward and reverse "slide show" modes. A text field for entering the target animation step to assume and a slider for adjusting the percentage of the animation currently shown are also included.

5 Conclusions and Further Work

The framework as outlined here offers sophisticated support for extensibility and reuse. To prove the flexibility of the framework, we have developed an animation system (Rößling and Freisleben, 2001, 2002) based on the framework which exhibits the functionality. Developing a system based on the framework presented here is relatively easy, but offers great benefits for both users and animation generators. Note that many key decisions have not been fixed in the framework, such as the type of input needed for generating an animation. Our prototypical system combines three different generation approaches: manual generation within a GUI, generation by scripting and by API invocations. The latter actually generates scripting code.

We hope that other persons interested in AV system development will adopt the framework and thus improve the chances of animation reuse in the field of AV. Of course, much further work is left. We have to formalize the framework specification and provide a careful evaluation of the framework features, both strengths and liabilities. Other interesting approaches for generating animation content, such as code interpretation, can in principle be incorporated into systems built on top of the framework. However, we need to provide a "proof of concept" system that illustrates this. For this purpose, we would be very grateful if other AV system developers could share Java code for parsing and interpreting code, as done for example in the JELIOT 2000 (Ben-Bassat Levy et al., 2001) system.

References

Ronit Ben-Bassat Levy, Mordechai Ben-Ari, and Pekka A. Uronen. An Extended Experiment with Jeliot 2000. *First International Program Visualization Workshop, Porvoo, Finland. University of Joensuu Press*, pages 131–140, July 2001.

Nils Faltin. Designing Courseware on Algorithms for Active Learning with Virtual Board Games. *4th Annual ACM SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education (ITiCSE '99), Cracow, Poland*, pages 135–138, June 1999.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

James Gosling, Jason Harrison, and Jim Boritz. Sorting Algorithms Demo. WWW: `http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html`, 2001.

Sami Khuri. A User-Centered Approach for Designing Algorithm Visualizations. *Informatik / Informatique, Special Issue on Visualization of Software*, pages 12–16, April 2001.

Guido Rößling and Bernd Freisleben. Software Visualization Generation Using ANIMALSCRIPT. *Informatik / Informatique, Special Issue on Visualization of Software*, 8(2):35–40, April 2001.

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.

John Stasko. Building Software Visualizations through Direct Manipulation and Demonstration. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization*, chapter 14, pages 187–203. MIT Press, 1998.

Redesigning the Animation Capabilities of a Functional Programming Environment under an Educational Framework

F. Naharro-Berrocal
Escuela Universitaria de Informática Universidad Politécnica de Madrid, Spain

C. Pareja-Flores
Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Spain

J. Urquiza-Fuentes, J. Á. Velázquez-Iturbide, and F. Gortázar-Bellas
Escuela Superior de CC. Experimentales y Tecnología, Universidad Rey Juan Carlos, Spain

a.velazquez@escet.urjc.es

1 Introduction

There has been much research on algorithm animation systems since the seventies. Some of these systems have been used in the industry, but most of them have been used in programming or algorithm courses. However, these research efforts have been mostly driven by research interests, without much concern on their educational effectiveness.

During the nineties, there was increasing interest on testing their educational effectiveness. Most of these efforts were directed to evaluate animation systems by means of controlled experiments (Ben-Ari, 2001; Stasko and Lawrence, 1998). Recently, some approaches have been proposed for the systematic design or analysis of these systems, e.g. to capture their requirements (e.g. Lattu et al. (2001) for using these systems for demonstration in the classroom), to propose a user-centered approach to their design Khuri (2001), or to measure their adequacy to education by means of a scale (Anderson and Naps, 2001).

During the nineties, we developed an integrated programming environment for functional programming, called WinHIPE (Naharro-Berrocal et al., 2001a). It provides graphical visualization/animation of the evaluation of functional expressions. Its design was driven by the goal of simplifying the production of visualizations/animations as much as possible; ideally, they would be produced automatically. To this end, we used the metaphor of office documents, where the user easily develops and maintains documents by means of typing and menus. Notice that the integration of a tool for visualization/animation production into a programming environment guarantees their universal applicability to any algorithm.

In this paper, we reexamine and redesign WinHIPE on a systematic basis. In the second section, we explain the three main components of our approach: a model of program execution, a model of program visualization/animation, and support to build and maintain animations. In the third section, we use Anderson and Naps' framework (Anderson and Naps, 2001) to assess and refine the animation capabilities of the system for education. Finally, we summarize our conclusions.

2 Basic elements of WinHIPE

Our approach to animation construction and maintenance is based on three elements: a model of program execution, a model of program visualization/animation, and a process to build and maintain animations. We review them systematically, identifying current and future features of WinHIPE.

2.1 Model of program execution

WinHIPE is an integrated programming environment for the functional language Hope. It includes facilities for editing, compiling, executing and debugging, as well as program bookkeeping.

Execution of functional programs is easily understood with the calculator metaphor. The programming environment allows evaluating any expression composed of predefined values and operators, as in a typical calculator. Besides, the programmer can define data types and functions in a program to be compiled. Since, the programmer can evaluate any expression composed of elements, either predefined in the programming language or defined in his/her program.

For example, consider the typical factorial function: dec fact: num -> num;
— fact n <=
if n=0 then 1 else n*fact(n-1); This definition can be used to evaluate functional expressions involving the factorial function. Recall that programs are not executed monolithically, but any expression can be written for evaluation. An excerpt of the visualization of a step-by-step evaluation follows:

$\boxed{\texttt{fact 4}}$
$\downarrow$
$\texttt{if } \boxed{\texttt{4=0}} \texttt{ then 1 else 4*fact(4-1)}$
$\downarrow$
$\boxed{\texttt{if false then 1 else 4*fact(4-1)}}$
$\downarrow$
$\texttt{4*fact(}\boxed{\texttt{4-1}}\texttt{)}$
$\downarrow$
$\texttt{4*}\boxed{\texttt{fact(3)}}$
$\downarrow$
...
$\downarrow$
$\texttt{4*(3*(2*(1*}\boxed{\texttt{fact 0}}\texttt{)))}$
$\downarrow$
...
$\downarrow$
`24 :  num`

As the example illustrates, the execution of functional expressions is displayed by WinHIPE at a high level of abstraction. WinHIPE models the evaluation of expressions as a rewriting process, where the initial expression is rewritten into intermediate expressions until a final expression, i.e. the value, is obtained. Every intermediate expression obtained is shown to the programmer. The programmer controls the pace of evaluation with five options: to evaluate either one rewriting step, n steps, the redex, the complete expression, or until a break-point is reached. (The redex or “reducible expression" is the next subexpression to be rewritten or reduced. Examples included in the paper highlight redexes by framing them or writing them in red color.)

Another view of the same evaluation follows, where an arrow labeled with ‘r’ means advance to a breaking point, labeled with ‘x’ means evaluation of the redex, and labeled with ‘*’ means complete evaluation:

$\boxed{\texttt{fact 4}}$
$\downarrow$r
$\texttt{4*}\boxed{\texttt{fact 3}}$
$\downarrow$r
$\texttt{4*(3*}\boxed{\texttt{fact 2}}\texttt{)}$
$\downarrow$r
$\texttt{4*(3*(2*}\boxed{\texttt{fact 1}}\texttt{))}$
$\downarrow$r
$\texttt{4*(3*(2*(1*}\boxed{\texttt{fact 0}}\texttt{)))}$
$\downarrow$x
`4*(3*(2*(1*1)))`
$\downarrow$*
`24 :  num`

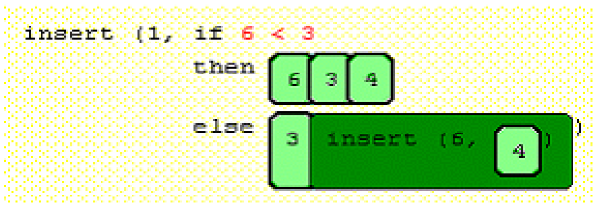
2.2 Model of program visualization and animation

As described in the previous subsection, WinHIPE displays the intermediate expressions requested by the programmer. Let us see in detail both their visualization and their animation.

2.2.1 Model of program visualization

An important design decision for the environment is that it always shows the whole expression, so that the programmer can know the current state of the evaluation. Each expression is automatically visualized according to the formatting and typographic choices made by the programmer.

Typically, intermediate expressions should be displayed in textual format, according to the definition of the language. WinHIPE also provides graphical representations for lists and binary trees (Jiménez-Peris et al., 1996) because expressions denoting data structures are frequently unreadable. The result is a mixed display, where lists and binary trees are displayed graphically and the other kinds of expressions are displayed textually. For instance, the mixed display of an expression resulting while sorting by insertion the list `[1,6,4,3]` is shown in Figure 1.



```
insert (1, if 6 < 3
           then 6 3 4
           else 3 insert (6, 4) )
```

Figure 1: A visualization containing lists.

Notice that expression visualizations in the example contain both data and control. The particular features of the functional programming paradigm makes it difficult to catalogue their visualizations according to existing taxonomies (e.g. Brown (1998)). An interesting issue is whether the environment generates program or algorithm visualization. Strictly speaking, we produce expression visualizations, a particular case of program visualizations, but the high level of abstraction of functional languages and graphical visualizations make them very close to algorithm animations.

2.2.2 Model of program animation

Animations are displayed in WinHIPE as a discrete sequence of visualizations. An animation can be watched by means of either one or two (consecutive) visualizations at a time. In the case of two consecutive visualizations, an arrow is displayed in between (as in the previous textual examples) in order to explain briefly the transition from the first to the second visualization.

When the user initiates an animation, it either automatically starts running or the first (one or two) snapshots are statically displayed. In either case, the user uses an VCR-like interface at the bottom of the window: advance or backtrack one snapshot, play at the specified speed, pause, and go to the first or the last snapshot. The number of the current snapshot relative to the total number of snapshots, and the animation speed are also displayed. Figure 2 shows the animation window with two visualizations at a time, for an expression computing the mirror of a given binary tree.

2.3 Support to build and maintain animations

Most animations systems require expertise of the user in using either a programming or a scripting language. This poses a burden on the user and discourages many potential users from building animations. Our approach tries to overcome these difficulties by simplifying the user interaction a much as possible. This also requires automating as many tasks as possible, even at the risk of obtaining less attractive animations than manually (Brown, 1998).

We have designed a simple process to build animations based on the metaphor of office applications. Metaphors are a well-known way of explaining a concept by means of different concepts. They are often used in computer science, being the desktop metaphor the most widely known. Other authors have proposed different categories of metaphors (Madsen, 1994). Our use of metaphors is pragmatic: we emphasize their use as a particular kind of seeing as governed by previous situations and examples rather than by rules and fixed categories. We also acknowledge that our metaphor is probably incomplete, but claim that it is useful to explain our approach. Metaphors are often used in animations in order to simplify either their production or their play. Thus, the video player metaphor is typically used in most systems (Gloor, 1998) to control the direction and pace of animations. It consists of a set of standard buttons (play, pause, rewind, etc.) and a speed control, often complemented with algorithm-specific controls. Another stimulating metaphor for playing animations is comic strips (Biermann and Cole, 1999).

Metaphors have also been used for modeling the different parts of a complex animation system, from construction to play. A theater metaphor has been used to explain Jeliot facilities (Haajanen et al., 1997). Thus, an animation can be considered a theatrical performance, an algorithm to be visualized can be considered a script, etc. Electronic books is also a common metaphor that helps in understanding the structuring of hypermedia elements to form a lesson or a set of lessons on algorithms (Brown and Najork, 1996; Brown and Raisamo, 1997).

2.3.1 The office metaphor

We use the metaphor of office applications in order to understand the construction and maintenance of animations. There are many kinds of office products: documents, spreadsheets, slide presentations, web pages, etc. However, they share some common properties. For instance, they are usually produced by typing and selecting menu operations. Documents are easily modified by customizing their typography (fonts, sizes etc.). Depending on the kind of document, they also are composed of different, standard parts: text, graphics, scripting code, etc. Finally, documents are produced and stored; afterwards, they can be retrieved, modified and stored again.

Following the office metaphor, visualizations and animations are products delivered with an application (here, a programming environment) and they should be easily constructed, customized and amenable to be stored, retrieved and modified.

An important feature of office applications is their facility to produce automatically documents, without the need to know complex languages. There are documental languages (e.g. LaTeX) which allow generating sophisticated documents, but office applications prevent the user from learning them.

WinHIPE gives support to (semi)automatically generate visualizations and animations. Both textual visualizations and graphical visualization of lists are produced automatically, since they are a

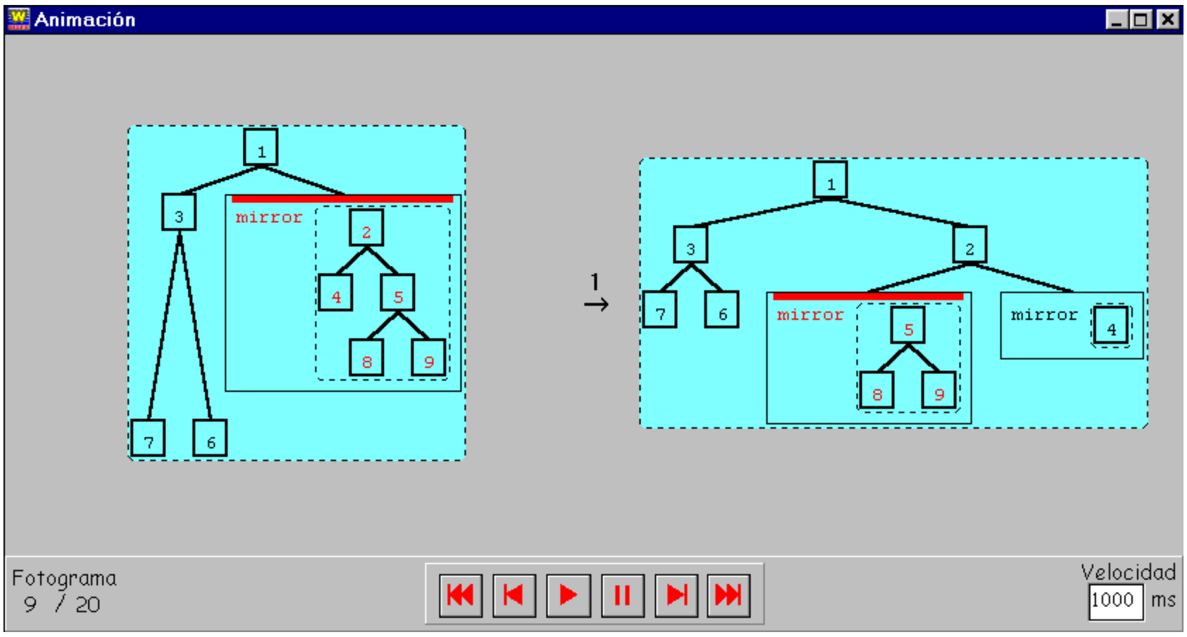


Figure 2: Interface of an animation.

predefined elements of the language. On the other hand, the graphical visualization of trees requires binding their graphical format to a binary tree data type by specifying the identifiers of tree constructors. A configuration file saves the user from explicitly making this binding, but he/she can redefine it wherever he/she wants.

Animations are created on the fly by playing visualizations with the “animations window" shown in Figure 2. The raw material for building an animation is the set of visualizations of expressions generated during an evaluation. By default, an animation consists of all of the expressions, but the user may exclude irrelevant ones (according to his/her subjective criterion) by marking a check box within the “visualizations window".

This way of defining animations requires a global view of all the visualizations. An additional “reductions window" provides a summary view of all the visualizations in the same way as PowerPoint does for slide shows. Showing reduced views of visualizations in a meaningful way is a technically difficult problem (Naharro-Berrocal et al., 2002) that has not been definitively solved.

Figure 3 shows all the visualizations obtained evaluating step-by-step the reversal of a given binary tree.

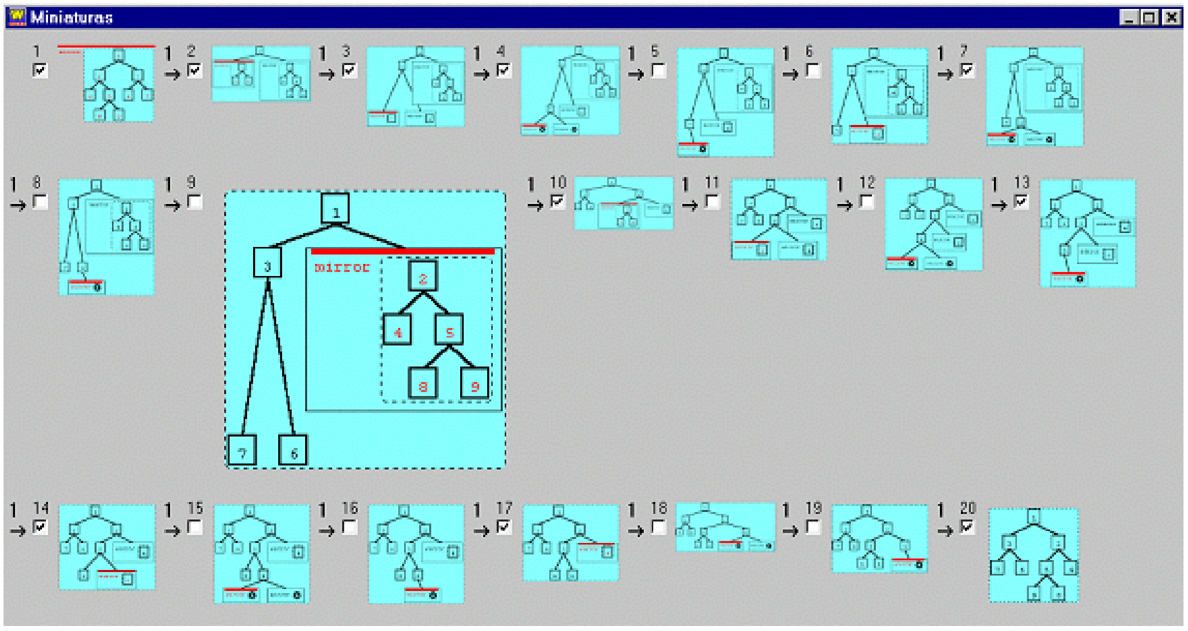


Figure 3: An improved grid of reduced visualizations.

2.3.2 Customization

WinHIPE allows the programmer to customize, by means of simple dialogs, the visualization of intermediate expressions in several ways (Velázquez-Iturbide and Vázquez-Presa, 1999):

- The programmer can select either pure textual or mixed text-graphics visualization of expressions.
- The typography of visualizations can be customized either by demonstration (pretty-printing formats) or using Word-like dialogs (graphical elements).
- Long expressions can be automatically abbreviated by a “fish-eye" technique (Velázquez-Iturbide, 1998), only showing the parts most relevant to understand the current evaluation state.

Figure 4 shows three different displays of an expression obtained while reversing a given binary tree. The second image is similar to the first one, with different typographic properties and without highlighting the redex; the third image is an abbreviation of the first one.

Customization does not only affect to a single visualization, but to all the visualizations or to the animation. In order to keep coherence, a customization of any of the previous three kinds propagates to any of the generated visualizations.

Sometimes, not only typography changes, but also the expression to evaluate, e.g. input data to an algorithm, such as the value of an element in a data structure. A facility is provided to rebuild such an animation given new input data, in an easy and consistent way. This facility is automatic when modifications in input data do not alter execution behavior, i.e. the nature and number of rewriting steps remains constant, but it can also deal with changes in the rewriting process.

Figure 5 shows the “rebuilding animation" dialog. The left side displays a vertical list of all the rewriting operations performed during an evaluation (represented with the same letters as in Section 2.1). This list of operations can be transferred to another list, at its right, representing the new list of operations from which to rebuild the animation. Here, operations can be included or deleted if necessary when input data changed. Finally, a button rebuilds the whole set of visualizations by applying this list of rewriting operations.

2.3.3 Storage and retrieval

Following the office metaphor, an animation can be stored as any other document. We allow two storage formats for animations. The simplest one only requires a directory to store the visualizations and a definition file. Such a file specifies whether by default the animation is to be played automatically or to be controlled manually; in the former case, the speed of transition between snapshots must also be specified. In a later session, the user can retrieve an animation in order to play or modify it.

Animations can also be stored as Web pages (Naharro-Berrocal et al., 2001b). Web pages gener-

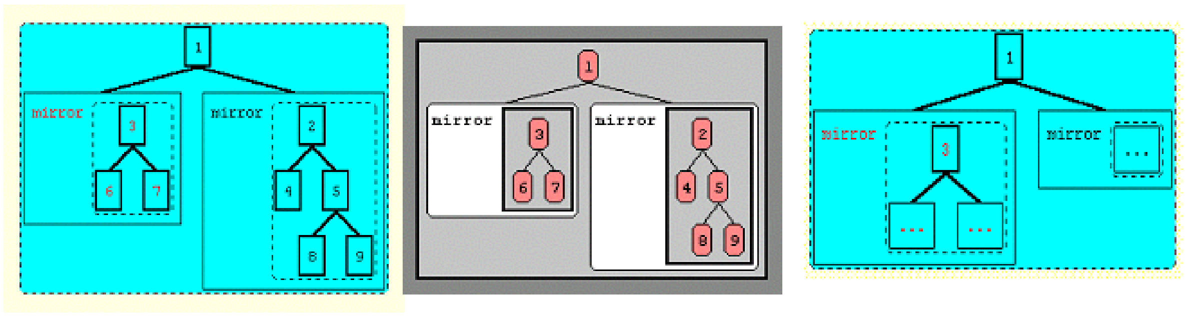


Figure 4: Three visualizations of the same expression.

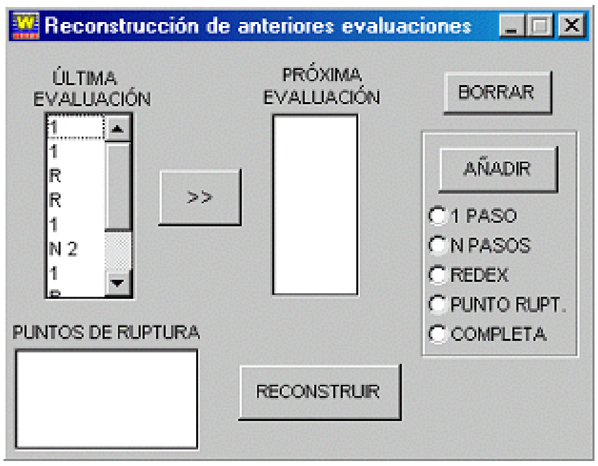


Figure 5: Dialog to rebuild a given animation.

ated by WinHIPE have a structure appropriate for lessons on algorithmic problem solving, composed of four parts: problem statement, algorithm description, program, and animation. This format allows integrating explanations and animations for educational use. A preliminary set of Web pages containing visualizations generated with WinHIPE can be found at `http://dalila.sip.ucm.es/~cpareja/winhipe/de`

Animations stored in either format can be retrieved and modified. In animations stored with the simplest format, individual visualizations and the animation style are retrieved. In animations stored as Web pages, the program and explanations are also retrieved. In particular, retrieval of the program gives the possibility of going on with a programming session.

Both kinds of animations make sense on their own. The most typical usage is using the first format for animations of programs in progress, and the second one for animations of definitive programs, so that they are complemented with code and user explanations. The latter case will be used by teachers to prepare lessons, or by students to document assignments or to reinforce self-study.

3 Validation of animation capabilities for education

In this section, we use Anderson and Naps' framework (Anderson and Naps, 2001) to assess algorithm animation systems as educational tools. They propose an "instructional design continuum" to measure the degree and fashion by which an animation system supports student interaction. It consists of eight levels, from simple to complex support. Assessing any system (in particular, WinHIPE) with the instructional design continuum has two consequences:

- We can know the strengths and weaknesses of the system for student interaction.
- We can identify opportunities for improvement.

Let us review the eight levels.

3.1 Level I: Animation

Any animation system satisfies at least this level, since it provides animations (obvious!). In our case, an animation is a sequence of discrete visualizations. However, some dynamism could still be provided in order to highlight the transition from a visualization to the following one. For instance, consider the animation interface in Figure 5. If the user presses the "advance one step" button, the second visualization could move smoothly from the right position to the left one, and then the new second visualization could be displayed.

3.2 Level II: Explanatory Textual Materials

We allow generating Web pages containing explanations, code and animations. We only consider at the moment static explanations about the problem and the algorithm. In general, it is impossible to automatically generate explanations.

However, some complementary information can be derived automatically. In our case, it is possible to explain how one visualization leads to the following one, by recording the advance actions selected by the user on evaluating an expression. The environment provides this information in the visualizations and animation windows as labeled arrows, to facilitate the understanding of the transition from a visualization to the following one.

Other syntactic information could also be automatically derived from the evaluation process, such as the kind of rewriting steps performed from a visualization to the following one (function application, conditional expression, etc.). Explanations about behavior in specific points could also be obtained from comments written by the programmer in certain places (i.e. assertions).

3.3 Level III: User input data

As our system is integrated in a programming environment, the user can input any data for execution.

3.4 Level IV: Stop-and-think questions

This is a facility the system does not provide by the moment. A simple version of this facility should not be very difficult to implement. The user would be able to write annotations associated to given visualizations or to given events (e.g. specific function application). Storage of animations would also keep these annotations; in particular, Web pages would open these questions (for instance, in a new, small browser window) when arriving at their corresponding points.

3.5 Level V: Adjust the amount of information

This facility is related to the underlying model of program visualization. Our system allows the user to simplify visualizations by means of the fish-eye technique. It also allows adjusting the size of visualizations by means of the “reductions window", in order to have a global view of them.

More control over the amount of information can be given only if the model for visualization is enriched.

3.6 Level VI: Backtrack in the algorithm execution

Our animation interface considers advancing forward and backward, both one single step and to the last visualization in each direction. Thus, the user has a high degree of control over the animation. In addition, the programming environment allows watching all the visualizations in two windows (one with images at natural size and other with images at reduced size), thus allowing direct access to any visualization.

The backtracking facility could also be included, in the programming environment, as another kind of advance for expression evaluation. It simply requires to keep a list of the intermediate expressions generated during evaluation and allowing to backtrack to a previous expression, deleting visualizations positioned forward. This facility is not currently implemented, but we plan to make it immediately.

3.7 Level VII: User design of the animation

Our approach tries to facilitate the construction and maintenance of animations by following the office metaphor. Thus, this level is at the heart of our system.

3.8 Level VIII: Tight linking between the animation system and lectures

Web pages generated by the system loosely link WinHIPE and lectures by means of embedded explanations. A tighter integration could be achieved by transforming our programming environment into a dynamic book on programming, containing lessons and exercises, and allowing the user to solve programming problems with a language interpreter and visualization/animation tool. This is a direction where promising results can be anticipated (Martínez-Unanue et al., 2002), but by the moment we do not plan to adopt it for WinHIPE.

4 Conclusions and discussion

We have redesigned the animation capabilities of the programming environment WinHIPE so that it is more adequate for education and its design is more coherent. To this end, we have followed a two-fold process. First, we have isolated three main components of our approach: a model of program execution, a model of program visualization/animation, and support to build and maintain animations. The last component is based on a metaphor of animations as office documents. Second, we have used Anderson and Naps' framework to assess and refine the animation capabilities of the system for education.

After this redesign, the main features of WinHIPE remain constant, but we hope that the usability of WinHIPE for educators and students has been enhanced in a systematic way. Currently, we have reimplemented the visualization and animation windows, we have implemented storage and retrieval facilities, and are designing and implementing the improvements derived from using Anderson and Naps' scale. We also expect to measure the usability of the new capabilities.

There still are possibilities for further improvement in any of these elements. Our separation of concerns has the advantage of facilitating the designer to focus independently on any dimension. Thus, the designer can focus on enhancing the execution model, e.g. by providing finer control over expression evaluation. Alternatively, he/she can concentrate on the model of visualization/animation, e.g. to provide multiple views.

Other improvements can be obtained at the dimension of building/maintenance. By taking more literally the office metaphor, we lack an unambiguous definition of an animation as a document. A more powerful definition would include the original program and the initial expression. As this approach was adopted for animations in Web pages, it would also have the advantage of being homogeneous. Additional potential enhancements can be given to customize graphics. For instance, the user can be assisted in selecting colors for different components of a visualization by showing (within the dialog) a sample with the effects of his/her selection, or the environment could even check them internally for their visual compatibility.

5 Acknowledgments

This work was supported by projects no. TIC2000-1413 of the Spanish Research Agency CICYT and PIGE-02-03 of the Universidad Rey Juan Carlos.

References

J. M. Anderson and T. L. Naps. A context for the assessment of algorithm animation systems as pedagogical tools. In *First Program Visualization Workshop*, pages 121–130, University of Joensuu, Porvoo, Finland, 2001.

M. Ben-Ari. Program visualization in theory and practice. *Informatik/Informatique*, 2:8–11, April 2001.

H. Biermann and R. Cole. Comic strips for algorithm visualization. 1999-778. Technical report, New York University, 1999.

M. H. Brown. A taxonomy of algorithm animation displays. In J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price, editors, *Software Visualization*, pages 35–42. MIT Press, 1998.

M. H. Brown and M. Najork. Collaborative active textbooks: A Web-based algorithm animation system for an electronic classroom. In *1996 IEEE Symposium on Visual Languages*, pages 266–275, Boulder, Colorado, USA, 1996. IEEE Computer Society Press.

M. H. Brown and R. Raisamo. JCAT: Collaborative active textbooks using java. *Computer Networks and ISDN Systems*, 29:1577–1586, 1997.

P. A. Gloor. User interface issues for algorithm animation. In J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price, editors, *Software Visualization*, pages 145–152. MIT Press, 1998.

J. Haajanen, M. Pesonius, E. Sutinen, J. Tarhio, T. Teräsvirta, and P. Vanninen. Animation of user algorithms on the web. In *1997 IEEE Symposium on Visual Languages*, Boulder, Colorado, USA, 1997. IEEE Computer Society Press.

R. Jiménez-Peris, M. Patiño-Martínez, C. Pareja-Flores, and J. Á. Velázquez-Iturbide. Graphical visualization of the evaluation of functional programs. In *SIGCSE/SIGCUE Conference on Integrating Technology into Computer Science Education*, pages 36–38, Barcelona, Spain, 1996. ACM Press.

S. Khuri. A user-centered approach to designing algorithm visualizations. *Informatik/Informatique*, 2:12–16, April 2001.

M. Lattu, V. Meisalo, and J. Tarhio. On using a visualization tool as a demonstration tool. In *First Program Visualization Workshop*, pages 141–162, University of Joensuu, Porvoo, Finland, 2001.

H. K. Madsen. A guide to metaphorical design. *Communications of the ACM*, 37(12):57–62, December 1994.

R. Martínez-Unanue, M. Paredes-Velasco, C. Pareja-Flores, J. Urquiza-Fuentes, and J. Á. Velázquez-Iturbide. Electronic books for programming education: A review and future prospects. In *7th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education*, pages 34–38, Aarhus, Denmark, 2002. ACM Press.

F. Naharro-Berrocal, C. Pareja-Flores, J.Urquiza-Fuentes, and J. Á. Velázquez-Iturbide. Approaches to comprehension-preserving graphical reduction of program visualizations. In *ACM Symposium on Applied Computing*, pages 771–777, Madrid, Spain, 2002. ACM Press.

F. Naharro-Berrocal, C. Pareja-Flores, and J. Á. Velázquez-Iturbide. Foundations for the automatic construction of animations and their application to functional programs. In *First Program Visualization Workshop*, pages 29–40, University of Joensuu, Porvoo, Finland, 2001a.

F. Naharro-Berrocal, C. Pareja-Flores, J. Á. Velázquez-Iturbide, and Martínez-Santamarta. Automatic Web publishing of algorithm animations. *Informatik/Informatique*, 2:41–45, April 2001b.

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, 1998.

J. Á. Velázquez-Iturbide. Automatic simplification of the visualization of functional expressions by means of fisheye views. In *Joint Conference on Declarative Programming*, pages 101–112, Universidad de La Coruña, La Coruña, Spain, 1998. ACM Press.

J. Á. Velázquez-Iturbide and A. Vázquez-Presa. Customization of visualizations in a functional programming environment. In *29th ASEE/IEEE Frontiers in Education Conference*, pages 22–28, San Juan, Puerto Rico, 1999. Stipes Publishing. Session 12b3.

IPPE—How To Visualize Programming with robots

I. Jormanainen, O. Kannusmäki and E. Sutinen
University of Joensuu, Department of Computer Science, P.O. Box 111, FIN-80101 Joensuu, Finland

{ijorma, okannus, sutinen}@cs.joensuu.fi

1 Introduction

Observations have shown that children and novice programmers may have difficulties to follow the execution of a program because the lack of a mental model in order to comprehend the concept of programming (Ben-Ari, 1998). One way to visualize programming is to use robotics as a concretizing tool in order to make programs closer to the programmer. Cognitive theories by Jean Piaget (Inhelder and Piaget, 1966), (evolved by Seymort Papert (Papert, 1980)) brings forward the idea of learning processes where the active role of the learner is crucial. Learner constructs knowledge by the manipulation and construction of real objects (Papert, 1980), for example robots (Miglino et al., 1999).

Lego has produced cheap models of robots which are very useful when learning basic skills of programming. With these robots it is possible to see a program in a very concrete way because the programs and algorithms can be seen, heard and even touched. Lego robots and teaching of programming are widely researched. However, the art of programming can be divided into different abstraction levels. Usually basics of programming have been taught at a high abstraction level (e.g. the programming environment of Lego Mindstorms). In this case the programmer does not necessarily understand how the program (a sequence of commands) is constructed. On the other hand lower level of abstraction forces the programmer to follow exactly defined syntax. The step to the lower level of abstraction is usually very high for a novice. Our approach with IPPE takes place in between these two abstraction levels and narrows the gap between them.

2 IPPE

There are many programming environments for Lego robots. For example Lego robots can be controlled using NQC (Not Quite C) or a fully graphical programming environment made by the LEGO Group. NQC is a programming language with a C like syntax (Baum) and this may cause some problems for novice programmers. Syntax sensitive languages also make programming difficult and this might frustrate students. Lego's own programming tool is a highly simplified way to control robots (LEG, 1999).

The main idea of project is to produce an environment for learning programming in the lower abstraction level than for example Lego's own environment is. Algorithms on this level are presented as pseudo-code. Programs made with IPPE are quite similar to programmers' native language. In this way, foreign language is not a constraint on the learning process. However, our goal has been to design GUI so that it helps to make the programming visual and concrete.

During the first phase of our project we have constructed a test version of our software and it has been used by small group of children for evaluation. We have learned that the idea can be used and it is worth of further studying.

2.1 Usage of IPPE

A program is constructed in two consecutive stages. In the first step a set of behaviors is created. In the second phase these parts are used to create a full program which can be transferred to the robot. This method is based on constructivism in computer science education. This means that knowledge is combined with existing to create new cognitive structures recursively (Ben-Ari, 1998). This approach of programming also prevents syntax errors and the programmer can totally focus on the semantics of the program.

2.1.1 Creating behaviors

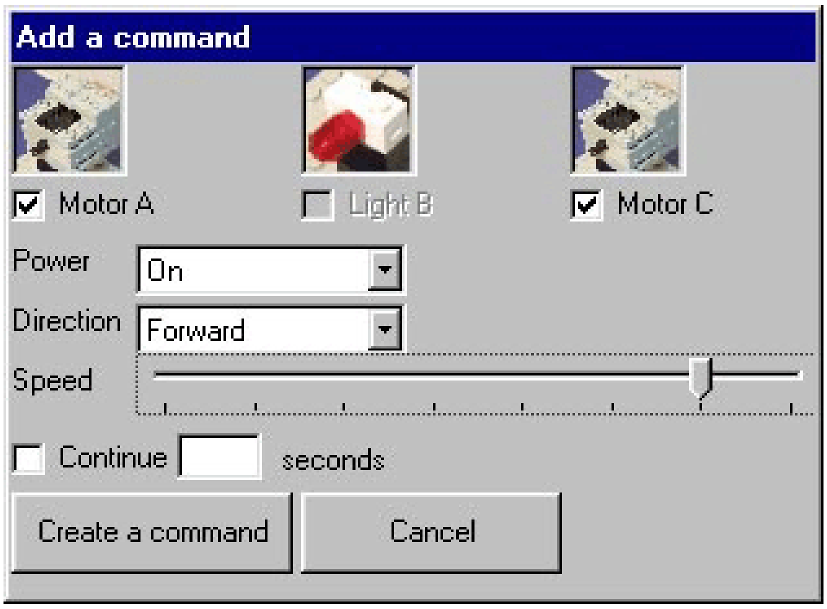
The programmer creates a set of behaviors in order to construct the program. The behavior is a small and simple task which carries out one function of the robot or is a control flow statement. An example of the behavior in natural language could be "Turn on motors A and C with speed 6". This is presented as environments own pseudo code as follows:

```
AC: On Forward.  Speed 6.
```

The programmer creates these behaviors with a dialog (Figure 1) and they can be added to a task list:

```
AC: On Forward.  Speed 6.
IF TS1=1
   END IF
IF LS2<70 OR LS3<70
   END IF
REPEAT FOREVER
   END REPEAT
PAUSE 2 sec.
IF (TS1=1) OR LS2>=41
   ELSE
   END IF
```

It is possible to create as many behaviors as needed and these can be used freely in any order during the design process of the final program.


Add a command

[✓] Motor A   [ ] Light B   [✓] Motor C

Power: On

Direction: Forward

Speed

[ ] Continue seconds

Figure 1: Behavior dialog. Set on motors A and B to forward direction with speed 6.

2.1.2 Constructing a program

When a sufficiently large set of basic behaviors is created, the programmer can start to construct a program to solve the task. The programmer constructs the program piece by piece using previously created behaviors. If a new behavior is needed, it can be added at any time.

The actual program code will appear in a code window:

```
AC: On Forward.  Speed 6.
REPEAT FOREVER
   IF LS2<41 OR LS3<41
      AC: Chance the direction
   END IF
END REPEAT
```

It is automatically indented by IPPE as in common programming style. In this way it is possible for the programmer to figure out different blocks and flows of the program.

2.1.3 Uploading, saving and printing

When the program is ready, the programmer can upload it to the RCX-unit in order to test if it works correctly. It is also possible to save and print out the program code for further use.

3 Field tests

The current version of the program has been used in Kids' Club which is a science club for children between 9 to 13 years old arranged by the Department of Computer Science at University of Joensuu. In these club sessions two groups of children have built their own robots and programmed them with the IPPE in order to complete tasks which instructors have been giving them.

3.1 Test case

In the Kids' Club session children were asked to perform the following task:

"On the floor is a track (see Figure 2). Design a program and build the robot so that it runs through the track and stops when it encounters the colored line on the floor."

Figure 5.

Figure 2: A track for a robot in Kids' Club session.

3.2 Observations

Children had about 2.5 hours to construct their robots and make the program. Children were already quite familiar with the environment and they had started to construct their robots beforehand. Both groups solved the problem within 2 hours. Figure 3 presents the program code made by the other group.

The most difficult part of the assignment was to use the infinite loop for checking the sensors of the robot continuously. Instructors had to give them hints how the sensors can be read continuously. Children used trial and error method when they tested their programs. First they built the program so that the robot ran trough the track. After that they solved the stopping problem. It was interesting to notice that the children learned very quickly how to build even complex if-then-else clauses.

This is only one example of assignments children had (and are still been given) in the Kids' Club sessions. At the beginning of year 2002 other Kids' Club group has been established and we are planning to arrange a test which would show how children's programming skills are improved when they use IPPE. We believe that if the children would have been programming the robots without IPPE they would not have had so much experience of procedural programming. After they have used IPPE they seem to adopt the basic concepts of procedural paradigm so that they could easily start to use lower abstraction level languages for controlling the robots.

4 Current implementation

For the time being IPPE (with a draft name "Lego-Compiler") is a pilot version running in Windows (Anttila et al., 2001). The current version has been implemented with Visual Basic 6.0. Communication between program and robot is handled by ActiveX-control called Spirit.ocx which includes in Lego's Mindstorms SDK. The current version of the program includes multilingual support. Nowadays English and Finnish are supported but it is easy to import new languages. The language can be selected with a command-line parameter. This version has been used in field tests. The knowledge of the tests will be used to design and develop fully working, functional and portable application.

5 Further development

The first part of the project is almost finished and the results of evaluation period are being processed. The project will continue with developing and implementing a new software. This environment will be written in Java. Therefore it will be portable to any appropriate platform and operating system.

The main goals with the new version are scalability and better usability. This means that the core of the program will be constructed in a way that makes it possible to extend the environment for example with new sensors or completely different hardware (e.g. Empirica Control (Lavonen et al., 2000)). Users on the different levels should be considered for example by developing possibilities to adjust an interface and the process of the programming with IPPE.

IPPEs GUI will be redesigned for the future version by carefully studying the material about learning environments and common standards of user interface design. Visualization of the programming will be increased in order to concretize algorithms in different ways. This can be achieved with various visual presentations of the program, e.g. flow charts.

6 Discussion

An interesting observation during the test period was that more experienced programmers asked for possibilities to write actual code themselves. This has to be considered when the next version is designed.

Besides learning of programming, IPPE can be used as a tool for concretising and visualizing logic. There has been some unofficial discussions if IPPE could be used as a tool to teach logic at

```
REPEAT FOREVER
    IF LS3>45
        AC: Off.
    ELSE
        AC: On Forward. Speed 7.
    END IF
    IF TS1=1
        AC: On Forward. Speed 0.
        PAUSE 1 sec.
        AC: Off
        A: On Forward. Speed 2.
        C: On Reverse. Speed 4.
        Pause 1 sec.
        AC: Off
    END IF
END REPEAT
```

Figure 3: Group's code to solve the problem. LS = Light sensor, TS = Touch sensor.

the university level. In this case, IPPE would be used in the level between mathematical software and paper presentations to demonstrate power of logic in a very efficient way.

7 Acknowledgements

We would like to thank Iiro Anttila and Juha Lehtonen for their work with previous version of the program.

References

Iiro Anttila, Ilkka Jormanainen, Osku Kannusmaki, and Juha Lehtonen. Lego-compiler. In *Proceedings of the First Annual Finnish / Baltic Conference on Computer Science Education*, University of Joensuu, Department of Computer Science. Report Series A, pages 9–12, Joensuu, Finland, 2001.

David Baum. *NQC - Not Quite C*. URL `http://www.enteract.com/~dbaum/nqc/index.html`. (Accessed 2002-02-14).

Mordechai Ben-Ari. Constructivism in computer science education. *SIGCSE Bulletin*, 30(1):257–261, 1998.

Barderl Inhelder and Jean Piaget. *La psychologie de L'enfant*. P.U.F, Paris, France, 1966.

Jari Lavonen, Veijo Meisalo, and Matti Lattu. Visual programming: basic structures made easy. In *Proceedings of the First Program Visalization Workshop*, University of Joensuu, Department of Computer Science, International Proceedings Series, pages 163–177, Joensuu, Finland, 2000.

Operation Manual for Lego Mindstorms. LEGO Group, 1999.

Orazio Miglino, Henrik Hautop Lund, and Maurizio Cardaci. Robotics as an educational tool. *Journal of Interactive Learning Research*, 10(1):25–47, 1999.

Seymort Papert. *Children, Computers and Powerful Ideas*. Basic Books Inc. Publishers, New York NY, USA, 1980.

A New Approach to Variable Visualization: Roles as Visualization Objects

J. Sajaniemi
University of Joensuu, Department of Computer Science, P.O.Box 111, FIN-80101 Joensuu, Finland

Jorma.Sajaniemi@Joensuu.Fi

1 Introduction

Learning to write computer programs is a hard task for many students. One obvious reason is that programs deal with abstract entities – formal looping constructs, pointers going through arrays etc. – that have little in common with everyday issues. These entities concern either the programming language in general or the way programming language constructs are combined to produce meaningful combinations of actions in individual programs.

Visualizations may be used to make both programming language constructs and program constructs more comprehensible (Stasko et al., 1998). For the programmer, a programming language is just a tool to build the more important artefacts, programs, and hence visualization of higher-level program constructs is more important than visualization of language-level constructs. For example, Petre and Blackwell (1999) note that visualizations should not work in the programming language level because within-paradigm visualizations, i.e. those dealing with programming language constructs, are uninformative.

```
program doubles;
var input, count, value: integer;
begin
    repeat
        write('Enter count: '); readln(input)
    until input > 0;
    count := input;
    while count > 0 do begin
        write('Enter value: '); readln(value);
        writeln('Two times ', value, ' is ',
                 2*value);
        count := count - 1
    end
end.
```

Figure 1: A short Pascal program.

As an example, consider the Pascal program in Figure 1. In this program the comparison "`some_variable > 0`" occurs twice – first to test whether the input value is valid and then to test whether there are still new values to be processed. In *programming language terms*, these two comparisons are equal: they both yield `true` if the value of the variable is greater than zero; otherwise they both yield `false`. In *program terms*, the meanings of these two comparisons are radically different: the first is a guard that looks at whatever the user has given as input and either accepts or rejects it, while the second is looking at a descending series of values and finds the moment when the series reaches its bottom. If these two tests are visualized equally, these visualizations do not provide students information about the program but about the programming language.

The dissimilarity of the tests cannot be observed by looking at the tests themselves or by looking at their locations within the syntactic constructs of the program. The distinctive issue is the nature of the values involved which requires a deeper analysis of the program. In the first test, the variable is an input value holder while in the second test the variable is a descending counter. In order to describe the true meaning of the tests, we must take into account this nature. In the following, we use the term *role* for the dynamic character of a variable embodied by the sequence of its successive values as related to other variables. The role characterizes the higher-level program information of variables.

Current visualization systems pay practically no attention to the roles of variables. Semi-automatic visualization systems (e.g., DYNALAB (Birch et al., 1995), Eliot (Lahtinen et al., 1998), Jeliot (Haajanen et al., 1997)) provide a set of ready-made representations for variables. These representations are based on text and simple geometric forms, and operate basically on the programming language level which does little to help students to understand how variables are used to build up meaningful constructs. Hand-crafted visualization systems (e.g., BALSA-II (Brown, 1988), LogoMedia (DiGiano et al., 1993), Pavane (Roman et al., 1992), POLKA (Stasko and Kraemer, 1993), TANGO (Stasko, 1990), Zeus (Brown, 1991)) give more freedom for variable visualization but they are used to demonstrate algorithms (e.g., various sorting methods) to intermediate students. Such visualizations are too difficult to novices trying to learn basic programming skills. In this paper we propose a mechanism to visualize variable roles to novice programmers and show how it is implemented in a prototype system we are currently developing.

The rest of this paper is organized as follows: Section 2 discusses the roles of variables and presents a set of roles with their characteristic behaviors. Section 3 is an overview of variable visualization in current systems. Section 4 describes how roles can be visualized, and discusses a new animation system that uses this approach. Finally, Section 5 contains the conclusions.

2 Roles of Variables

The role of a variable characterizes the dynamic nature of the variable, e.g., a *counter* starts from zero and is continuously increased by 1. A role is not supposed to be a unique task in some specific program (e.g, the counter that counts lines in a text processing program) but a more general concept characterizing many variables in various programs.

To find an appropriate set of roles we started with roles listed by Green and Cornah (1985) in their proposal for a tool, Programmer's Torch, intended to clarify the mental processes of maintenance programmers. Among other features, the tool is supposed to reveal roles of variables listed tentatively as follows:

- Constant: a variable with an unchanged value
- Counter: a variable stepping through a series of values each depending on its own old values and possibly some other variables
- Loop counter: a counter used to control the execution of a loop
- "Most-recent" holder: a variable holding the last value encountered in going through a series of values (with no restriction on the nature of the series)
- "Best-of" holder: a variable holding the best value encountered so far (with no restriction on how to measure the goodness of a value)
- Control variable: a variable controlling the execution path based on a condition evaluated earlier in the program
- Subroutine variable: a variable used for communication between the caller and the called function; parameter, result, `var` parameter, or local variable

In our classification, we will not consider the way a variable is used but we will look at the succession of values it will encounter and how this depends on other variables. A role has, of course, an effect on the way the variable is used, but this can be taken care of by employing *role-sensitive visualizations of operations*, i.e., the visualization of an operation depends on the roles of the variables participating in that operation. We started with the above role list and analyzed all programs in two Pascal programming textbooks that have had continuous reprints, and in a more advanced textbook intended to be used in a second course on programming (Foley, 1991), (Jones, 1982), (Sajaniemi and Karjalainen, 1985). All the books contain numerous entire programs making a total set of 109 programs that was analyzed. By analyzing the behavior of each variable we created the following classification of variable roles.

- *Constant:* a variable whose value does not change after initialization (e.g., an input value stored in a variable that is not changed later) possibly done in several alternative assignment statements (e.g., a variable that is set to `true` if the program is executed during a leap year, and `false` otherwise) and possibly corrected immediately after initialization (e.g., an input value that is replaced by zero if it is negative)

- *Stepper:* a variable going through a series of values not depending on values of other non-*constant* variables (e.g., a counter of input values, a variable that doubles its value every time it is updated, or a variable that alternates between two values) even though the selection of possibly alternative update assignments may depend on other variables (e.g., the search index in binary search)

- *Follower:* a variable going through a series of values depending on the values of a *stepper* or another *follower* but not on other non-*constant* variables (e.g., the “previous” pointer when going through a linked list, or the “low” index in a binary search)

- *Most-recent holder:* a variable holding the latest value encountered in going through a series of values (e.g., the latest input read, or a copy of an array element last referenced using a *stepper*) and possibly corrected immediately after obtaining a new value (e.g., to scale into internal data representation format)

- *Most-wanted holder:* a variable holding the best value encountered so far in going through a series of values with no restriction on how to measure the goodness of a value (e.g., largest input seen so far, or an index to the smallest array element processed so far)

- *Gatherer:* a variable accumulating the effect of individual values in going through a series of values (e.g., a running total, or the total number of cards in hand when the player may draw several cards at a time)

- *One-way flag:* a Boolean variable that will not get its initial value after its value has once been changed (e.g., a variable stating whether the end of input has been encountered, or a variable registering the occurrence of error(s) during a long succession of operations)

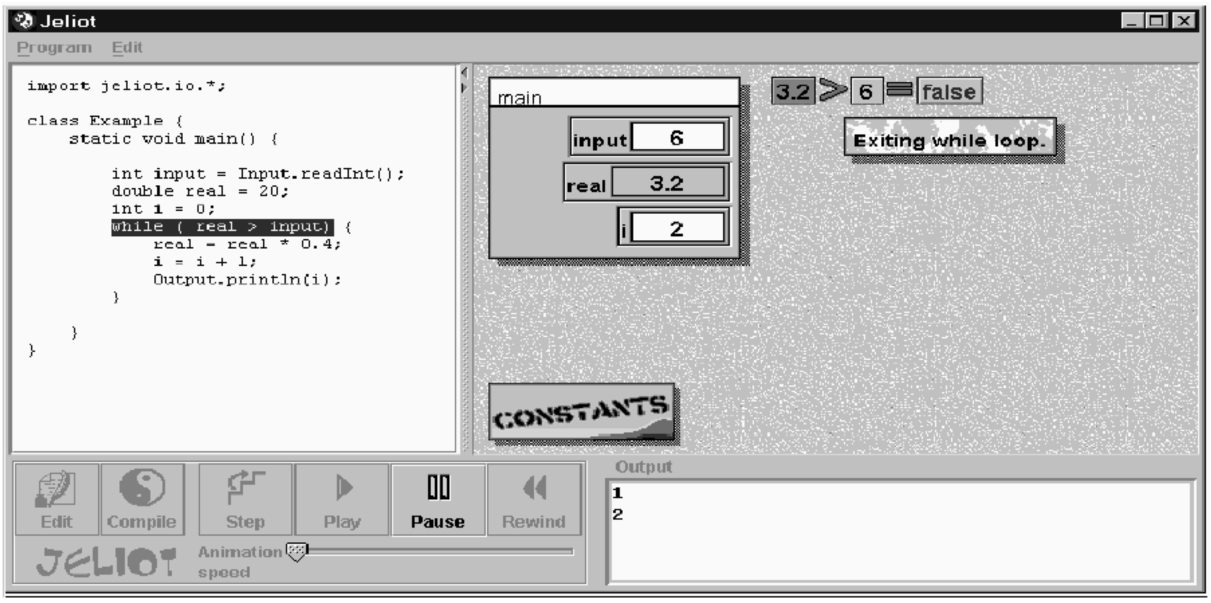
- *Temporary:* a variable holding some value for a very short time only (e.g., in a swap operation)

- *Organizer:* an array which is only used for rearranging its elements after initialization (e.g., an array used for sorting of input values); otherwise an array is classified based on the role of its elements

- *Other:* all other variables

For example, in the program of Figure 1 the variables `input` and `value` are *most-recent holders*, and the variable `count` is a *stepper*.

The three most frequent roles, *constant*, *stepper*, and *most-recent holder* cover 84.0 % of all the roles in the above three books. Only 1.1 % of variables were classified as *other* and all of them



```
import jeliot.io.*;

class Example {
    static void main() {

        int input = Input.readInt();
        double real = 20;
        int i = 0;
        while ( real > input) {
            real = real * 0.4;
            i = i + 1;
            Output.println(i);
        }

    }
}
```

Exiting while loop.

```
1
2
```

Figure 2: User interface of Jeliot 2000.

occurred in the more advanced book. We may deduce that the above set of roles is sufficient to characterize variables in novice programming.

3 Traditional Visualization of Variables

Current program visualization systems can be divided into two categories depending on their approach to visualization of variables. *Semi-automatic* visualization systems (e.g., DYNALAB (Birch et al., 1995), Eliot (Lahtinen et al., 1998), Jeliot (Haajanen et al., 1997)) provide a set of ready-made representations for variables from which users can choose whatever representation they like. On the other hand, *hand-crafted* visualization systems (e.g., BALSA-II (Brown, 1988), LogoMedia (DiGiano et al., 1993), Pavane (Roman et al., 1992), POLKA (Stasko and Kraemer, 1993), TANGO (Stasko, 1990), Zeus (Brown, 1991)) give more freedom for variable visualization.

Semi-automatic visualization systems have a fixed set of optional visualizations for variables, usually with a larger set of alternatives for simple variables than for arrays. Visualizations may be purely textual or involve a simple graphical form, like a rectangle, with the name and value of the variable in textual form. Operations are visualized automatically using smooth transition of values from one variable to another or to the side of a comparison operator. In order to capture users' attention the operands may change color or size just before the movement starts. Figure 2 (Ben-Bassat Levy et al., 2000) showing the user interface of Jeliot 2000 is a representative example of this approach.

All these visualization techniques operate solely on the programming language level. Some aspects of variables can be stressed by a carefully considered selection of graphical forms, but it is questionable how much information about the nature of various variables is brought to the novice user by, e.g., the fact that the corners of the rectangle containing the value are rounded.

Hand-crafted visualization systems provide more opportunities for the visualization of variables. The value of a variable may be reflected in the size, position or color of its visualization. The form is usually some simple 2-dimensional graphical form but may also be 3-dimensional. Operations are visualized in the same way as in semi-automatic systems but sounds can also be attached to some actions. These systems are used mostly for the visualization of algorithms, e.g. different ways to sort arrays. Each visualization is hand-crafted to clarify essential features of the algorithm, and no advice for the visualization of variables is given.

Figure 3 shows a view of Quicksort in the POLKA system when the array to be sorted contains 20 integers. From the algorithm's point of view the most important issues concern the current order

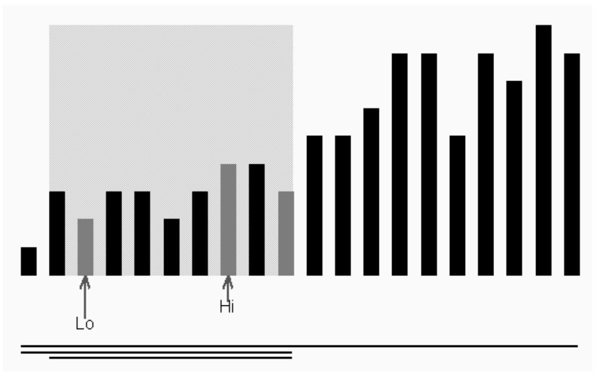


Figure 3: Visualizations of variables in POLKA.

of the elements, what part of the array is currently being sorted, and how the knowledge of the current elements is maintained. As a consequence, the array elements are visualized with rectangles whose height depend on the value of the element, and the variables denoting the current elements (`Lo` and `Hi`) are visualized with short arrows pointing at the elements. The visualization of a pointing variable reveals its relation to the array but does not tell how the variable itself behaves, i.e., is it a constant, does it go through some pre-defined set of values, or is its values directed by other variables? A user can deduce this behavior from the successive values of the variable and to an intermediate programmer this should become apparent through the visualization.

For a novice, however, it would be very difficult to decipher the roles and how the variables contribute to the program as a whole. As the visualization is not intended for novice programmers, this is not a problem. Indeed, hand-crafted visualizations are directed to intermediate programmers. Novices need visualizations that make it clear that there are different roles and that these roles occur in programs again and again.

4 Visualization of Roles

The visualization of a role must be both general in the sense that the same form will be used for all variables of that role, and specific in the sense that it should make clear how the successive values relate to each other and to other variables. For example, a *constant* should give the impression of a value that is not easy to change, and a *stepper* should make it clear that the next value is usually known in advance and most often in close vicinity of the previous value.

It should be noted that graphic visualizations are to some extent related to cultural environment. For example, a variable that is always increased by 1 could be visualized by a concrete tally counter, a small hand-held device used for counting purposes, but such a device is not widely known in all countries. For example in Finland, practically nobody knows the existence of such a device. As a consequence, role visualizations must be tailored to the cultural background of users, and the following sketches may not appear natural to all readers.

Figure 4 gives examples of role visualizations. A *constant* is shown as a stone that is not so easy to change. A *stepper* shows its current value and two lists of some values it has had or may have in the future, together with an arrow giving the direction that the stepper is currently going. A *follower* is attached to the variable it follows. Both a *most-recent holder* and *most-wanted holder* show the current and previous values, a *most-recent holder* depicted by squares in a neutral color and a *most-wanted holder* by flowers of different colors: green for the current value, i.e. the best found so far, and yellow for the previous, i.e., the next best.

A *gatherer* is depicted as a box holding the current and the previous value with new values coming into it from above through its lid. A *one-way flag* is a light bulb which will break when the flag goes off. Finally, a *temporary* is shown as a flashlight that is on just as long as the value is used. When the value has no meaning any more, the flashlight goes off and the value itself turns into grey.

As the deep meaning of operations is different for different roles, their visualizations must also

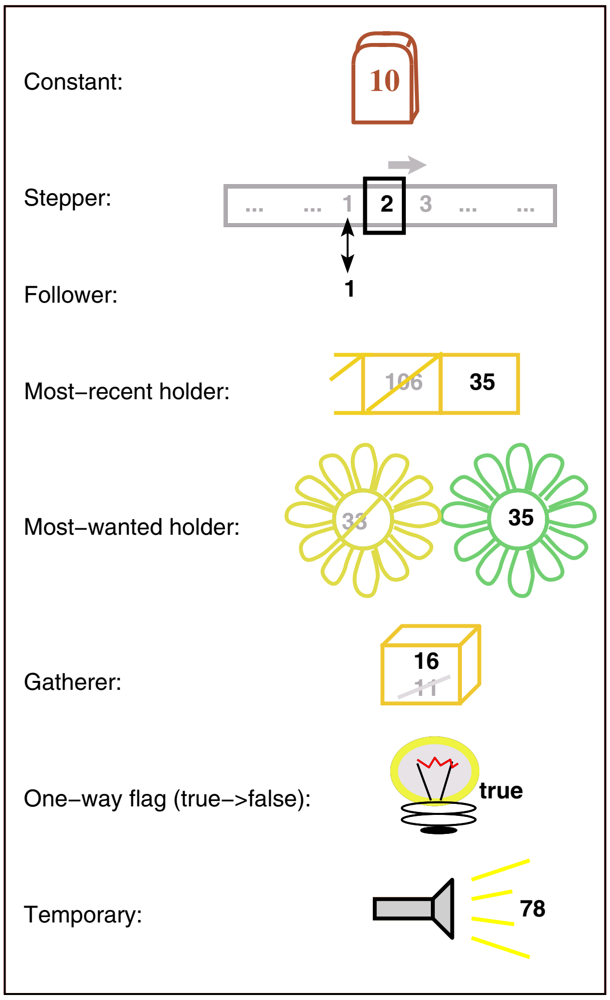


Figure 4: Example visualizations for various roles.

be different. For example, Figure 5 gives visualizations for the two syntactically similar comparisons "`some_variable > 0`" of the program in Figure 1. In case (a), the variable is a *most-recent holder* and the comparison just checks whether the value is in the allowed range. In the visualization, the set of possible values emerges, with allowed values colored in green and disallowed values in red. The arrow that points to that part of the values where the variable lays, appears as green or red depending on the values it points to. The arrow flashes to indicate the result of the comparison.

In Figure 5 (b) the variable is a *stepper* and, again, the allowed and disallowed values are colored. However, these ranges are now part of the variable visualization and no new values do appear. The current value of the variable flashes and the user can see the result by its color.

In both visualizations, if the border value used in the comparison is an expression (as opposed to a single constant), the expression is shown next to the value. If two variables are involved in a comparison, say "`a > b`", the visualization could be built around either variable, but the roles can be used to determine which one should be used. For example, if one variable is a *stepper* and the other is a *most-recent holder*, then it is reasonable to assume that the *stepper* is the central variable that is compared to a *most-recent holder* border.

Figure 6 shows the same comparison as animated by Jeliot 2000 in Figure 2. The roles of the variables are now clear, and it is easy to see that the variable `real` is going downwards and the colors (red up to 6, and green from there on) show clearly that the variable has gone below its limit. It is interesting to note that even though the program does not make any sensible computation, the visualization in Figure 6 gives a good overview of how it works.

As a final example, Figure 7 depicts variables in a program that finds the average and maximum

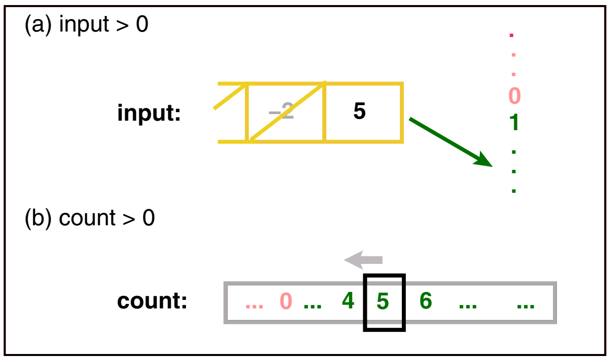


Figure 5: Visualizations of the same operation for different roles.

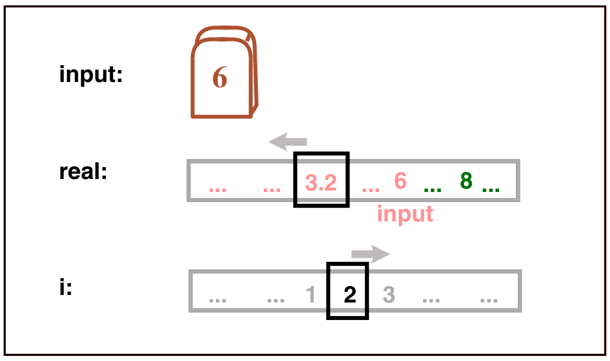


Figure 6: Visualization of the comparison in the Jeliot 2000 example.

of its inputs. In spite of the nonsense names given to the variables the meaning of each variable can be immediately recognized by their roles. Even the method used to detect the end of input is easy to deduce from the visualization of the comparison. In real use, the variable names should of course be meaningful, and program code should be available to users. This example just shows that the visualization of variable roles contains more information than provided by a visualization based on mere values of variables.

The role of a variable may change during the execution of a program and this happens usually somewhere between two loops. For example, in the program of Figure 1, the two variables `input` and `count` could be combined to a single variable, say `count` (making the assignment "`count := input;`" unnecessary). The role of this variable would first be a *most-recent holder* and then, in the second loop, a *stepper*. This can be animated by a smooth deformation of the visualization.

We are currently building a prototype animation system (called PlanAni) that animates semi-automatically small programs using the role-based approach. The user interface consists of three panes: one for the program text, one for the variables, and one for input/output operations. The current statement of the program is enhanced with color background and it is connected with an arrow to the corresponding variable. To avoid unnecessary details PlanAni does not animate the evaluation of expressions: only the resulting value – accompanied by the expression itself – is shown and its effect in a comparison or assignment is animated (see, e.g., the expression "`input`" in the visualization of the variable `real` in Figure 6).

Because it is impossible for humans to divide their visual attention to two or more targets – especially if the task is demanding for the person in question (Lavie, 1995) – it is hard for novices to follow what happens in various parts of the user interface. In order to minimize users' needs to jump back and forth between the program code and the variables, we use a speech synthesizer that explains what is going on in the program. This includes variable creation (e.g., "*creating a gatherer called sum*"), role changes ("*the variable count acts henceforth as a stepper*"), operations ("*comparing count with zero*"), and control constructs ("*entering a loop*").

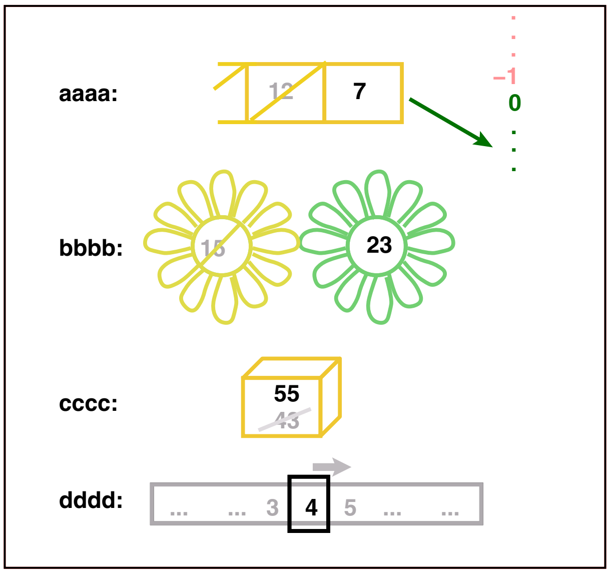


Figure 7: Variables for finding the average and maximum of inputs.

5 Conclusions

Learning to write computer programs is a hard task for many students and research into techniques that could be used to help them is needed. One such issue is the various roles of variables that do occur in programs again and again. We have presented an exhaustive classification of variable roles and showed how the visualization of variables can be based on the roles. Moreover, we have presented a prototype program animation system that uses this approach.

Empirical studies have shown that visualization is problematic for very early novices (Mulholland and Eisenstadt, 1998) implying that learners at different skill levels need different visualizations. The roles of variables represent very basic programming skills and, in accordance with this, variable role visualization is intended for novice use. More experienced programmers may find these visualizations too space-demanding.

We do not expect that (perhaps any) visualizations used for roles would be immediately understandable to all novices. Therefore, animations should be preceded by an explanation of the various roles, how they are visualized, and why the visualizations look the way they look. Other features of the animation are more obvious.

6 Acknowledgments

The author would like to thank Raija Sarajärvi for acting as the second judge in classifying variable roles in the analyzed programs and for programming a demonstration version of the PlanAni system.

References

R. Ben-Bassat Levy, M. Ben-Ari, and P. A. Uronen. An extended experiment with Jeliot 2000. In *Program Visualization Workshop*, Porvoo, Finland, 2000.

M. R. Birch, C. M. Boroni, F. W. Goosey, S. D. Patton, D. K. Poole, C. M. Pratt, and R. J. Ross. Dynalab a dynamic computer science laboratory infrastructure featuring program animation. *ACM SIGCSE Bulletin*, 27(1):29–33, 1995.

M. H. Brown. Exploring algorithms using balsa-ii. *IEEE Computer*, 21(5):14–36, 1988.

M. H. Brown. Zeus: A system for algorithm animation and multi-view editing. In *Proceedings of the 1991 IEEE Workshop on Visual Languages*, pages 4–9. IEEE Computer Society Press, 1991.

C. J. DiGiano, R. M. Baecker, and R. N. Owen. Logomedia: A sound-enhanced programming environment for monitoring program behaviour. In *INTERCHI'93*, pages 301–302. ACM, 1993.

R. W. Foley. *Introduction to Programming Principles Using Turbo Pascal*. Chapman&Hall, 1991.

T. R. G. Green and A. J. Cornah. The programmer's torch. In *Human-Computer Interaction - INTERACT'84*, pages 397–402. IFIP, Elsevier Science Publishers (North-Holland), 1985.

J. Haajanen, M. Pesonius, E. Sutinen, J. Tarhio, T. Teräsvirta, and P. Vanninen. Animation of user algorithms on the web. In *Proceedings of VL' 97 IEEE Symposium on Visual Languages*, pages 360–367, 1997.

W. B. Jones. *Programming Concepts – A Second Course*. Prentice-Hall, 1982.

S.-P. Lahtinen, E. Sutinen, and J. Tarhio. Automated animation of algorithms with eliot. *Journal of Visual Languages and Computing*, 9:337–349, 1998.

N. Lavie. Perceptual load as a necessary condition for selective attention. *Journal of Experimental Psychology: Human Perception and Performance*, 21:451–468, 1995.

P. Mulholland and M. Eisenstadt. Using software to teach programming: Past, present and future. In *Software Visualization: Programming as a Multimedia Experience*, pages 399–408, 1998.

M. Petre and A. F. Blackwell. Mental imagery in program design and visual programming. *International Journal of Human-Computer Studies*, 51(1):7–30, 1999.

G.-C. Roman, K. Cox, C. Wilcox, and J. Plun. Pavane: A system for declarative visualization of concurrent computations. *Journal of Visual Languages and Computing*, 3(1):161–193, 1992.

J. Sajaniemi and M. Karjalainen. *Suppea johdatus Pascal-ohjelmointiin (A Brief Introduction to Programming in Pascal)*. Epsilon ry, Joensuu, Finland, 1985.

J. Stasko, J. Domingue, M. H. Brown, and Blaine A. Price, editors. *Software Visualization: Programming as a Multimedia Experience*. MIT Press, Cambridge, MA, 1998.

J. T. Stasko. Tango: A framework and system for algorithm animation. *IEEE Computer*, 23(9):27–39, 1990.

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

A Vision of Visualization in Teaching Object-Oriented Programming

M. Ben-Ari, N. Ragonis, R. Ben-Bassat Levy
Department of Science Teaching, Weizmann Institute of Science, Rehovot 76100 Israel

moti.ben-ari@weizmann.ac.il

1 Introduction

In this paper we wish to consider the visualization of programs in the context of teaching an introductory course in object-oriented programming (OOP). We are investigating an objects-first approach, which is based on the premise that students should study OOP from the beginning so as to avoid a shift of the programming paradigm that occurs in an objects-late approach. However, objects-first is also problematical because the students are required to simultaneously master: (a) general concepts relating to computation, such as source code, compilation, execution, (b) paradigm-independent aspects of programming, such as assignment, procedure and function declaration and invocation, parameter passing, as well as (c) OOP-specific concepts, such as class, object, constructor, accessor, mutator.

Our perspective is that of teachers of elementary computer science, so we will attempt to specify what we think an ideal visualization environment should look like, without attempting to investigate the feasibility of any particular aspect of the specification. Nevertheless, we are not working in a vacuum. Our objects-first teaching has made extensive use of the popular and successful BlueJ (Kölling, 1999) educational integrated development environment for OOP programs in Java. It will be convenient to present our proposal as suggested modifications and extensions of BlueJ. Our proposal was also influenced by our use of the Jeliot (Haajanen et al., 1997; Ben-Bassat Levy et al., 2002) program visualization system; in particular, we have adopted the principle that program animation should be complete and continuous (Ben-Bassat Levy et al., 2002).

Section 2 will set the background: why the complexity of the execution of OO programs seems to require visualization. In Section 3, we discuss some of the difficulties and misconceptions that arose when teaching an objects-first course using BlueJ. Then in Section 4, we will specify our vision of a visualization system. Section 5 discusses the complications caused by fields whose values are themselves objects.

2 Problems with visualizing OOP

The dynamics of the execution of an object-oriented program are quite complex when spelled out in complete detail, but these details are essential for a proper understanding of OOP by a student. Here are the two main scenarios that occur:

- Object creation: The class declaration is consulted to determine the fields so that the correct amount of memory can be allocated to the object. After performing default or explicit initializations of the fields, the constructor is executed. The instructions of constructor are again obtained from the class declaration. Since constructors are most frequently used to initialize fields, the flow of data is from actual parameters to formal parameters and then by the assignment statement to the fields of this object.

- Invocation of a method on an object: The class declaration is consulted to obtain the instruction sequence of the method. The actual parameters are evaluated and passed to the formal parameters. The instructions are then executed, where references to identifiers are to identifiers that are declared in the class, while the values themselves are in the fields of the object.

These scenarios are central to any understanding of OOP and must be understood by the student, whether or not the teacher explicitly teaches these details! In the terminology of constructivism, a mental model must be constructed. We believe that the details of the scenarios must be explicitly taught, because otherwise there is too much maneuver room and the students are likely to construct non-viable models. Of particular importance is the interplay between the static declarations of fields and methods in the class, and the dynamic access and modification of values in each object of the class. (See Corradi and Leonardi (2000) for a further discussion of the relation between static and dynamic concepts in OOP.)

Some of the misconceptions that we have found in teaching objects-first arise from misunderstandings related to the following elements of OOP:

1. The difference between a class and an object of the class.
2. The execution of a constructor as part of object creation.
3. Operations can only be invoked on objects.
4. The state of an object (i.e. the values of its fields) and the fact that an operation will usually change the state.
5. The connection between the execution of an operation and its source code.
6. Actual vs. formal parameters.
7. The "uses" relation between classes, in particular, the value of a field of an object can itself be an object.

3 Teaching OOP with BlueJ

BlueJ (Kölling, 1999) is an integrated development environment for teaching OOP in Java. It includes a multi-window text editor, an interface to the Java compiler and interpreter, and a debugger. The environment itself displays a class diagram of a program which shows the "uses" and inheritance relationships among the classes. A separate panel contains icons for each object that has been directly created. Schematically, the BlueJ display is shown in Figure 1.

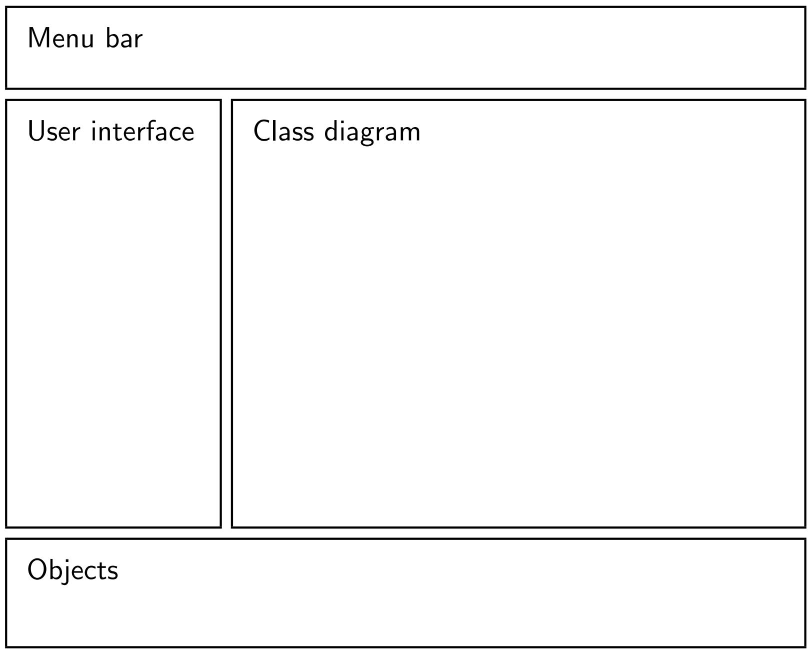


Figure 1: BlueJ user interface

The special attraction of BlueJ for teaching is that objects can be interactively created from a class icon, and methods can be interactively invoked from an object icon. This enables you to teach elementary concepts of OOP incrementally without getting bogged down in writing test programs. In practice, we encountered misconceptions that can be attributed to the use of BlueJ. This may be since BlueJ emphasizes visualization of OO concepts and implicitly assumes a knowledge of general computing concepts and elementary programming concepts, such as those mentioned in the first paragraph of the paper. Here are some difficulties we encountered:

1. In graphics programs (such as the example Shapes supplied in the BlueJ distribution), students found it difficult to differentiate between the object (the icon in the environment) and the window that the object happens to display. Similar problems can be found differentiating the name of an object and the title of a window.

2. Object icons are labeled only with their names and types. While the list of methods is easily seen in a popup menu adjacent to the icon, the fields and their values can only be seen by popping up a separate window. The object state is not highlighted and therefore students do not become familiar with the concept.

3. Actual parameters of the invocation of methods and constructors are entered interactively; students learn that they are literal values (or actual objects) and then have difficulty understanding that an actual parameter can be any expression.

4. Paradoxically, interactive invocation is the source of some misunderstandings, because it is difficult to make the transition to the invocation of a method by another method. For example, if a method returns a value, BlueJ pops up a window displaying that value; this behavior is quite reasonable, but does not help the student understand that in a real program, something must be done with the value. Students became confused, because they were not able to integrate these individual actions into a coherent entity ("a program"), especially those students with previous exposure to writing or executing programs. Their favorite complaint was a sports metaphor: "I can't figure out who is playing whom."

The strong point of BlueJ is its visualization of the static structure in the class diagram, but there is, in effect, no dynamic visualization of the execution of a program. The type of coherent, dynamic animation that we are looking for is demonstrated in Jeliot (Ben-Bassat Levy et al., 2002). The next section presents our vision for a Jeliot-like animation within the framework of BlueJ. Petre and Green (1993) noted that experts and novices have different requirements for visualization. Our proposal is for a system for total novices learning objects-first. If the suggestions are implemented within a framework such as BlueJ, there would have to be the option of reducing the detail of the visualization as students become more experienced.

One final note: the authors of this paper have not been able to come to a consensus as to whether visualization for total novices should be implemented as a special mode of a more advanced system like BlueJ or whether it would be better to have a separate visualization tool for the first few weeks or months. Similarly, we do not agree whether the general computing and programming principles should be visualization within a OOP tool like BlueJ or perhaps initially studied and visualized out of context, like parameter passing in PSVE (Naps and Stenglein, 1996). We are looking forward to discussing these issues at the workshop.

4 A vision of OOP visualization

In this section, we will propose a visualization system for teaching OOP to novices, and we will describe two scenarios, one for constructing objects and the other for invoking a method on an object. Figure 2 shows the proposed screen layout.

The environment panel will be toggled between the class diagram and the code of a selected class. The line-of-code panel will display the source code of the current line being visualized. The execution animation is performed on the stage (a term taken from Jeliot). The objects panel will have an icon for each object that has been created. Unlike BlueJ which has unadorned icons, each object icon will directly show the fields and their current values. If there is not enough room, a combo-box or scroll bar will be used, but in initial examples, a typical object will contain only 2-3 fields and these can all be shown (Figure 3). (See the Appendix for the source code of the examples.)

Now for the description of the creation of an object. Since some of the actions are complex, it may be advisable to pause the animation at intermediate points, but we leave this consideration to the detailed design of the system. Some of the actions are the same as those in BlueJ, so we are assuming familiarity with that system.

1. After loading the project, the class diagram will be displayed in the environment window.

2. A right-click on a class icon will bring up a menu with a list of the constructors; a constructor can be chosen and invoked with actual parameters entered interactively. The line-of-code panel will show a constructor invocation, for example:
   `Song song1 = new Song("Waterloo", "Abba", 197);`

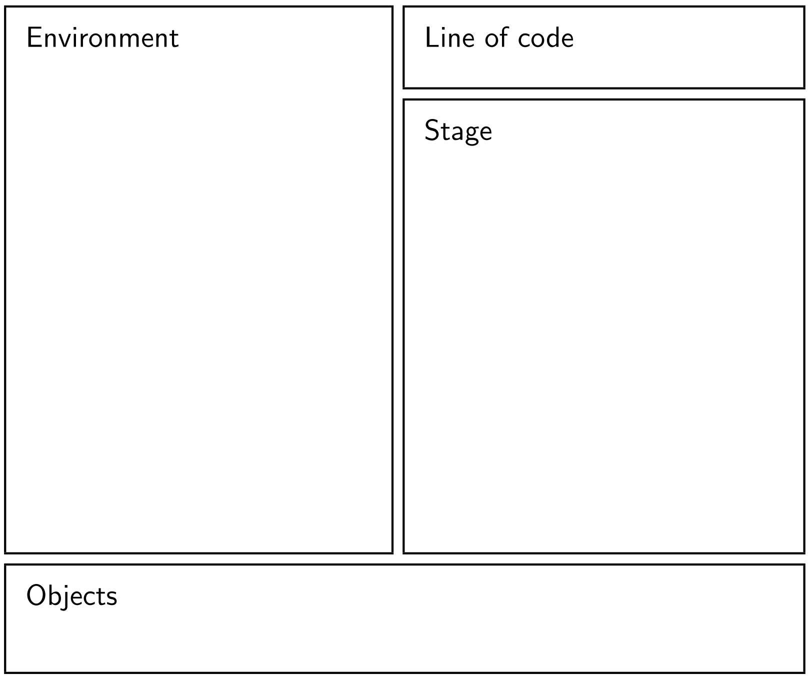


Figure 2: Proposed user interface

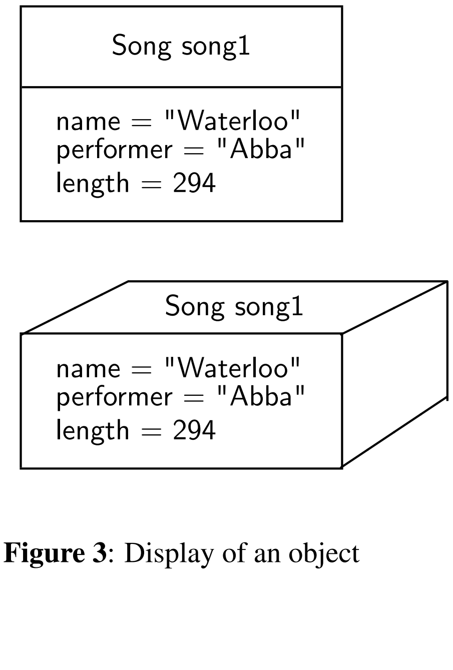


Figure 3: Display of an object

3. The object declaration `Song song1` will be highlighted, and an icon will appear, labeled with name and class of the object. In the environment panel, the source code for the class will appear, and it will scroll until the list of fields is shown. (Obviously, there will have to be conventions dictating that all fields are declared together, perhaps at the start of the class.) Each field declaration will be animated and it will appear with a default initial value.

4. The constructor invocation `Song("Waterloo", "Abba", 197)` will be highlighted and the source code will scroll until the code of the constructor is displayed. On the stage, temporary cells containing the actual arguments will appear, and as in Jeliot, they will be animated as moving into formal parameters and then into the fields in the object icon.

5. Following the execution of the constructor, the stage and line-of-code panels will be cleared, and the environment panel restored to display the class diagram.

Let us now turn to the invocation of a method on an object.

1. As in BlueJ, the invocation will commence with right-clicking on the object icon, selecting a method and entering the actual parameters. The resulting invocation will be displayed in the line-of-code panel:
   `song1.setLength(231);`

2. The source code of the object's class will be displayed in the environment window and will scroll until the code of the method appears. The matching of the actual and formal parameters will be animated, followed by the animation of the execution of the statements of the method, similarly to the animation of Jeliot. If there is a return value, it will be displayed on the stage.

3. Following execution, the stage and line-of-code panels will be cleared, and the environment panel restored to display the class diagram.

5 Fields with object values

One of the most difficult OOP concepts to teach is that fields can be of a reference type and therefore the value of the field will be an object itself. In BlueJ, this relationship is not visualized. To create an object of class `Disc` (see Appendix), you would create two objects of class `Song` and then make them actual parameters to the constructor of `Disc`. (Conveniently, it is possible to click on a formal parameter field and then on the icon of an object in order to interactively indicate that the object is the actual parameter.) All three of the objects will appear as icons, but the visual connection between the objects is lost. If you double-click on the `Disc` object icon, you receive a textual list of its field values, including the notation `<object reference>`. Double clicking again will then list the field values of this object. But it is left to the student to make the association between the state containing object values (not names!) and the object icons.

We propose that this relationship be visualized, either by arrows (Figure 4), or perhaps by color-coding, where the field is of the same color as the object icon. The difficult task is to design a visu-

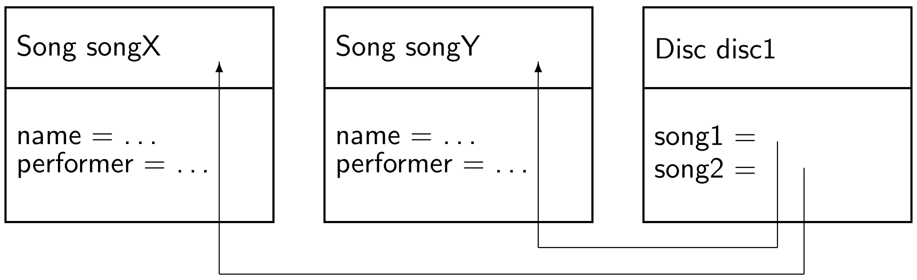


Figure 4: Compound objects

alization for the invocation of a method on the contained objects, for example, invoking the method `getTotalLength` on the object disc1 will in turn invoke the method `getLength` on the two objects of class `Song`. We propose that a method invocation cause a new window to be popped up, with the code of invocation displayed in the line-of-code panel, and the source code of the class of the method in the environment panel. There would be no need for an additional object panel. The invocation is animated as before, but after returning from the invocation, the cascaded window would be removed. This would help visualize the stack execution mechanism, though we would not explicitly mention it in an introductory course.

6 Conclusion

The visualization proposed here tries to achieve several objectives:

- Visualize the relationship between dynamic execution and the static source code.
- Display a continuous and complete animation of the execution of object creation and method invocation, so that the student can follow the entire data flow.
- Suggest a way to display as much information as possible in a single window, rather than scattering it over overlapping windows which are difficult for novices to work with.

We wish to iterate that while the description of our vision has been based on BlueJ and Jeliot, we do not necessarily advocate modifying either of these systems to implement our suggestions. It is quite possible that the best solution may turn out to be a special-purpose visualization for use only in the first month or so of teaching OOP. Once the concepts have been mastered, the student may be able to work freely with a more advanced system.

References

Ronit Ben-Bassat Levy, Mordechai Ben-Ari, and Pekka A. Uronen. The Jeliot 2000 program animation system. *Computers & Education*, 2002. (in press).

A. Corradi and L. Leonardi. Static vs. dynamic issues in object oriented programming languages. *Journal of Object-Oriented Programming*, 13(6):11–24, October 2000. http://www.adtmag.com/joop/article.asp?id=4388&mon=10&yr=2000.

J. Haajanen, M. Pesonius, E. Sutinen, J. Tarhio, T. Teräsvirta, and P. Vanninen. Animation of user algorithms on the web. In *Proceedings of VL' 97 IEEE Symposium on Visual Languages*, pages 360–367, 1997.

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

Thomas L. Naps and Jeremy Stenglein. Tools for exploration of scope and parameter passing in a programming languages course. *SIGCSE Bulletin*, 28(1):305–309, 1996.

Marian Petre and Thomas R. G. Green. Learning to read graphics: Some evidence that 'seeing' an information display is an acquired skill. *Journal of Visual Languages and Computing*, 4:55–70, 1993.

Appendix - Example program

```
class Song {

  String name;
  String performer;
  int length;

  Song(String n, String p, int l) {
    name = n; performer = p; length = l;
  }

  void setLength(int newLength) {
    length = newLength;
  }

  int getLength() {
    return length;
  }
}

class Disc {

  Song song1;
  Song song2;
  Disc(Song s1, Song s2) {
    song1 = s1; song2 = s2;

  }

  int totalLength() {
    return song1.getLength() + song2.getLength();
  }
}
```

Using 3-D Interactive Animation To Provide Program Visualization As A Gentle Introduction To Programming

Wanda Dann [1]
Computer Science Department, Ithaca College, Ithaca, NY 14850, USA

Stephen Cooper[1]
Computer Science Department, Saint Joseph's University, Philadelphia, PA 19131, USA

Randy Pausch
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, USA

`wpdann@ithaca.edu`

1 Background and Motivation

Attrition has been a longstanding problem for computer science programs in academia. Of course attrition in computer science majors occurs throughout the four years, but the largest percentage of dropouts occur during the first year of study (Seymour and Hewitt (1997)). High attrition rates over the last decade have occurred during the same period of time as most computer science departments have made the switch from imperative to object-oriented (OO) languages. Not only do OO languages require the teaching and learning of all the traditional fundamental concepts, but OO languages add the more demanding concepts of class, object, information hiding, inheritance, and polymorphism. We conjecture that a correlation exists between high attrition rates of the last decade and the switch to OO languages. Data supporting this conjecture comes from a contrast of studies from early researchers with more recent researchers. Early researchers, using imperative languages, (such as Butcher and Muth (1985); Stephens et al. (1985); Oman (1986)) showed little to no relationship between prior programming experience and success in the curriculum. In contrast with those studies, later researchers Davy et al. (2000) and Hagan and Markham (2000), using object-oriented languages, indicate a strong relationship between prior programming experience and success in the computer science major.

Our motivation is to reduce attrition in the first year of the computer science program. We ask: "Which students are at most risk of dropping out of the major?" and "What can be done to decrease attrition?" Using historical enrollment and dropout data at our own institutions, we conclude that students who have little or no previous programming experience and who are not yet ready for calculus have an extremely high dropout rate (approximately 90%). In an effort to decrease attrition in the computer science major in the first year, we have developed a course for introducing fundamentals of object-oriented programming to these "high-risk" students as a preparation for a rigorous CS1 course (Cooper et al. (2003)). This pre-CS1 course is thoroughly based on the use of program visualization.

2 Using Program Visualization

The program visualization tool we choose to use is the 3D interactive animation environment, Alice (freely available at www.alice.org). This tool follows in the footsteps of Logo, Karel the Robot, and Karel++ (Papert (1980), Pattis (1981), Bergin et al. (1997)). Other tools that might be used include Samba (Stasko (1997)) and JAWAA (Rodger (2002)).

The total focus of our pre-CS1 course is the use of program visualization to support the pedagogical goal: teaching fundamental concepts of object-oriented programming and problem solving. Students are assigned problem-solving tasks where the animation visualizes a solution to the problem. Importantly, the virtual world is one created entirely by the student and the program code is the student's own code. The program visualization can be incrementally developed and executed again and again, line by line.

[1]This work is supported in part by a grant from the National Science Foundation, DUE- 0126833.

By executing program code for animating objects in a virtual world, students view the progression of changes of state in the virtual world (Dann et al. (2000)). The smooth animations in 3D allow the student to connect individual lines of code to the animated actions of the objects. For example, a smoothly animated "move" command that evidences a bug where the object moves too far and runs off the screen is more easily interpreted than an instantaneous move, where the object "teleports" and therefore is misperceived as a "delete" operation. During the course, students develop and test a series of problem solving projects, illustrating progressively more demanding programming concepts.

The programming language concepts used in building 3D animations in Alice include:

- Action commands (such as move, turn, roll, and resize) cause state transformations in the students' virtual worlds. These commands are similar to assignment statements in more traditional programming languages. Mutable variables can be avoided, as objects maintain their own properties.
- Behavior methods allow students to group action commands into one named instruction, to define their own commands. The concept of a behavioral method is similar to the procedure concept in many other programming languages (or a function that just performs side effects, not returning anything).
- Functions (implemented as questions), decisions, expressions, collections, and repetitions are all similar to any other programming language. But, in a 3D environment students can easily see the consequences of their execution.
- Recursion is used for repeated animated actions.
- Object-oriented programming concepts are integral to the 3D program visualization. All 3D objects are obviously objects and have behaviors, state, and identity. These objects (e.g., animals and vehicles) populate the world as seen in the onscreen visualization. By writing simple code, object appearance and behavior is controlled. Likewise, events and interaction are a natural part of the visualization. Students write programs to make object actions respond to mouse and keyboard input. For example, students build simple flight simulators and space invader kinds of games.

A great advantage of using 3D animation to create program visualizations in the Alice environment is that concurrency is trivial to understand. Some actions must occur simultaneously while others must occur in sequence. An ice skater that turns and moves simultaneously can skate around a hole in the ice, while the same actions in sequence cause the skater to fall into the icy water.

3 Results

Ithaca College (IC) and Saint Joseph's University (SJU) have conducted this course over two semesters under different experimental conditions. At IC, the course was offered concurrently with the CS1 course. At SJU, the course was a stand-alone class that preceded CS1. A study of the effectiveness of this course in its different manifestations is currently underway. Early results of this study are based on a comparison of students in CS1 who have taken the course versus those students (with similar math and computing backgrounds) who did not take the course. Results include an evaluation of whether this course provides sufficient "prior programming experience" as discussed by Davy et al. (2000). Particular aspects include student performance in CS1 and student level of confidence in CS1, along with an analysis of retention. Early results, not ready for publication at this writing, will be summarized in presentation at this workshop.

References

J. Bergin, M. Stehlik, and R. Pattis. *Karel++, A gentle introduction to the art of object-oriented programming*. Wiley, New York, 1997.

D. Butcher and W. Muth. Predicting the success of freshmen in a computer science major. *Communications of the ACM*, 28(3):263–268, 1985.

S. Cooper, W. Dann, and R. Pausch. Using animated 3d graphics to prepare novices for cs1. *Computer Science Education Journal*, 2003.

W. Dann, S. Cooper, and R. Pausch. Making the connection: programming with animated small worlds. In *Proceedings of the 5th Annual Conference on Innovation and Technology in Computer Science Education*, pages 41–44, Helsinki, Finland, 2000.

J.D. Davy, K. Audin, M. Barkham, and C. Joyner. Student well-being in a computing department. In *Proceedings of the 5th Annual Conference on Innovation and Technology in Computer Science Education*, pages 136–139, Helsinki, Finland, 2000.

D. Hagan and S. Markham. Does it help to have some programming experience before beginning a computing degree program? In *Proceedings of the 5th Annual Conference on Innovation and Technology in Computer Science Education*, pages 25–28, Helsinki, Finland, 2000.

P. Oman. Identifying student characteristics influencing success in introductory computer science courses. *AEDS*, 19(2-3):226–233, 1986.

S. Papert. *Mindstorms: children, computers, and powerful ideas*. Basic Books, New York, 1980.

R. Pattis. *Karel the robot*. Wiley, New York, 1981.

S. Rodger. Introducing computer science through animation and virtual worlds. In *Proceedings of the 33rd SIGCSE Technical Symposium*, pages 186–190, Cincinatti, February 2002.

E. Seymour and N. Hewitt. *Talking about leaving*. Westview Press, Boulder, 1997.

J.T. Stasko. Using student-built algorithm animations as learning aids. In *Proceedings of the 28th SIGCSE Technical Symposium*, pages 25–29, San Jose, 1997.

L. Stephens, J. Konvalina, and E. Teodoro. Procedures for improving student placement in computer science. *Journal of Computers in Mathematics and Science Teaching*, 4(3):46–49, 1985.

Hands on Algorithms: an Experience with Algorithm Animation in Advanced Computer Science Classes

Irene Finocchi
Dipartimento di Informatica, Sistemi e Produzione, Università degli Studi di Roma "Tor Vergata", Via di Tor Vergata 110, 00133 Roma, Italy

Rossella Petreschi
Dipartimento di Informatica, Università degli Studi di Roma "La Sapienza", Via Salaria 113, 00198 Roma, Italy

finocchi@disp.uniroma2.it

1 Introduction

Each time a speaker has to introduce and to explain an algorithm to an audience, he naturally translates into a visual form his mental image of the algorithm itself. Pressley (1977) observed that this is mainly useful when the audience consists of young people, since their ability of generating their own internal images of problems is not yet completely evolved. Usually, teachers employ transparencies or chalk and blackboard to accomplish this task, though more impressive tools may better capture the students' imagination. In particular, program visualization systems have recently gained increasing attention as aids for teaching algorithms. According to a standard definition, algorithm animation is concerned with the static and dynamic use of text, images, and sound to point out semantic properties of algorithms and to make their data and control flow directly perceivable to human senses (Stasko et al., 1997). Algorithm animation systems offer a double advantage in educational settings. Indeed, the assistance of a visual image helps people to strengthen, and sometimes to correct, the mental image of a phenomenon that they already have. Moreover, animations of algorithms make it easier to build and to refine a mental image when it has not been completely elaborated. For this reason several visualization systems have been implemented (Brown and Najork, 1996; Cattaneo et al., 1998; Crescenzi et al., 2000; Domingue and Mulholland, 1997; Roman et al., 1992; Stasko, 1990) and their pedagogical utility has been analyzed in different settings (Ben-Bassat Levy et al., 2000; Kehoe and Stasko, 1996; Lawrence, 1993; Lawrence et al., 1994; Stasko, 1997; Stasko et al., 1993).

The debate on the value of algorithm animations for teaching and learning is still open and more or less favorable papers can be found in literature (Lawrence et al., 1994; Mayer, 1989; Palmiter and Elkerton, 1991; Rieber et al., 1990; Stasko et al., 1993). It is common opinion that animations can not be considered as stand-alone teaching tools, but should be accompanied by more standard methods. One of the fundamental questions is which is better between a *passive* and an *active* usage of a system, i.e., if it is better to allow students only to observe the animations realized by the teacher or to have a deeper interaction with the system. With respect to the first approach, young people appear to be the most malleable subjects: the visualization speeds up their comprehension of problems, even if it seems not to deepen it. On the other side, more experienced students appear not to benefit of a passive approach and prefer having the possibility of interacting with the system in order to create their own visualizations. From these considerations, it should be clear that the theory of *how* animations can be used to help students needs further investigation. In our opinion, one of the main factors that has limited the diffusion of algorithm animation tools for teaching and learning is the difficulty of using them: this may be due either to a non-correct and easy design of the user interfaces or, going a step beyond, to the fact that preparing animations to be illustrated in a classroom setting is usually a long and boring task even for teachers.

This paper reports on a preliminary experience with algorithm animation in the fourth year of the Degree in Computer Science at the University of Rome "La Sapienza". Undergraduates of an advanced course on parallel algorithms were involved: these students, yet experienced in sequential algorithms, were novice in parallel computing. The experiment is based on the awareness that parallel algorithms do not generally resemble the reasoning of the human mind, thus being more difficult to understand:

it is our feeling that visualization can help in making the mapping between parallel and sequential computations more intuitive. Once the fundamentals of parallel computing were assimilated by the students, we have analyzed how they were able to translate into a graphical form their own idea of parallelism, creating personal visual representations of parallel algorithms.

The topic used was the prefix sum computation, that is fundamental in parallelism because it is at the base of many problems like numbering, counting, computing elementary functions on trees. During theoretical lessons, algorithms for building prefix sums were described in detail. Three different models of parallel machines were analyzed: a shared memory model (P-RAM) and two interconnection networks (mesh and complete binary tree) (Jájá, 1992). Then, the students were asked to simulate the presented algorithms in a sequential setting, implementing them in the ANSI C language, and to visualize their behaviour trying to highlight the parallel steps. The algorithm animation system Leonardo (Crescenzi et al., 2000) has been used in the experiment. No student had ever used Leonardo before neither for observing visualizations available from its program repository nor for realizing his own animations.

Overall, our experience suggests that algorithm animation can have a positive impact on students' learning. Indeed, the involved students were able not only to realize accurate and faithful animations, but also to benefit from their own visualizations for tuning the code, i.e., for identifying less efficient implementations and consequently for improving them. Moreover, the different visualizations show different mental images of some fundamental concepts of parallel computing as perceived by each group of students. A few animations can be viewed at the URL
`http://www.dsi.uniroma1.it/~finocchi/parallel-course.html`.

The rest of this paper is organized as follows. After a brief description of the main features of Leonardo and of its approach to algorithm animation (Section 2), in Section 3 we review the algorithm studied in our experiment, we describe visualizations realized by different groups of students, and we point out some differences between them. Finally, Section 4 details how one of the visualizations has been realized.

2 Learning algorithms via special software

In this section we review the basic features of Leonardo, the algorithm animation system used in the experiment, and we briefly describe how it has been presented to the class.

2.1 The algorithm animation system

Leonardo (Crescenzi et al., 2000) is an integrated environment for the execution and visualization of C programs that introduces two main novelties w.r.t. a traditional algorithm animation system. First, it features *reversible execution* of programs, i.e., the possibility of "undoing" previous computational steps during the execution for debugging and visualization purposes. Second, the animations are realized according to a novel *logic-based approach*: they are conceived as reflecting the formal properties of algorithms and can be realized by declaring how the algorithm's data structures have to be depicted on the graphical scene. Visualization declarations are specified in a simple logic-based language called ALPHA. Smoothly changing images are also supported, so as to make program operations and data modifications more easily perceivable by the user (Demetrescu and Finocchi, 2001).

The logic-based approach makes it possible to automatically detect at-run-time events interesting for the visualization, thus simplifying the task of the algorithm visualizer, who has not to explicitly identify such events in the source code. The main task of the algorithm visualizer consists of specifying the graphical scene, letting its content depend on the program's data structures. It is therefore simple both showing data stored in main memory (e.g., a graph represented by means of an adjacency matrix) and data not explicitly stored, but deducible from the execution state (e.g., the dual or the transitive closure of a graph).

Leonardo is currently available on Macintosh platform and can be downloaded over the Web at the URL
`http://www.dis.uniroma1.it/~demetres/Leonardo/`.

2.2 Presentation to the class

We spent approximately two hours for introducing Leonardo to the class. Before our explanation no student had ever used the system neither for observing visualizations available from its program repository nor for realizing his own animations. After the presentation, we provided the students with further documentation, including a preliminary user manual (Demetrescu and Finocchi, 2000).

After a brief introduction to algorithm animation, we focused the students' attention on Leonardo's logic-based approach to visualization, stressing that visualizing an algorithm requires first to know the data structures used in its implementation and then to specify a visual presentation for them: the second step can be done using a declarative visualization language called ALPHA. We introduced ALPHA in an intuitive way, making use of many simple examples. In particular, we gave an overview to the graphical possibilities offered by Leonardo (a basic 2D graphical vocabulary as well as higher-level graphical primitives, such as graphs, matrices, histograms) and during this description we introduced the signatures of some ALPHA predicates for creating graphical objects and for setting their visual attributes (e.g., color, size, thickness). We mostly gave examples concerned with high-level primitives related to compound objects: for instance, we discussed some predicates for the visualization of graphs and we showed how such predicates can make use of the underlying C data structures (e.g., an adjacency matrix), thus establishing a connection with the source code.

We finally focused on the animation of parallel algorithms, presenting some examples of animations. For instance, we considered the problem of finding, for each node of a forest $\mathcal{F}$, the root of the tree of the forest that the node belongs to. We showed an implementation and a visualization related to the use of the pointer jumping technique for solving this problem and we discussed the source code to give a feeling of how the animation had been realized.

3 Visualizing parallel algorithms

Asking a student to implement an algorithm and to design a visualization of it is especially useful for testing his level of comprehension of the algorithm and of its relationship to a real implementation. The students involved in our experiment were attending a course on advanced algorithms and data structures during the fourth year of the Computer Science Degree at the University of Rome "La Sapienza". Though experienced in sequential algorithms, they were novice in parallel computing and algorithm animation. The students were divided in groups of two or three persons and required to prepare three animations in a period of two weeks.

The topic used in the study was the prefix sum problem. During theoretical lessons, students were presented three different algorithms for computing prefix sums on three different models of parallel machines: a shared memory model, P-RAM, and two interconnection networks, mesh and complete binary tree (Jájá, 1992; Petreschi). Students were then required to simulate these algorithms on a sequential machine and to visualize their behaviour trying to highlight the parallel steps. The aim of the experimentation was to analyze how each student was able to translate into a graphical form his own concept of parallelism.

We remark that in the process of designing an animation of an algorithm an important step is concerned with linking the visual events in the animation with the code segments that trigger them. Our experiment offered very interesting issues in this sense, because a typical parallel algorithm was implemented in a sequential setting, but the visualization was required to highlight which operations could be done at the same time on a specific parallel model.

Due to the lack of space, in the following we will only present the outcome of our experiment w.r.t. the mesh model. For completeness, we first briefly remind the parallel algorithm for computing prefix sums on this model, referring the interested reader to the literature for further details.

3.1 Computing prefix sums in parallel

The input of the algorithm consists of n different values, $x_1 \dots x_n$, while the output returns n sums, $s_1 \dots s_n$, where $s_i = \sum_{k=1}^{i} x_k$. We denote with s the side of the mesh, assuming that $n = s^2$. The

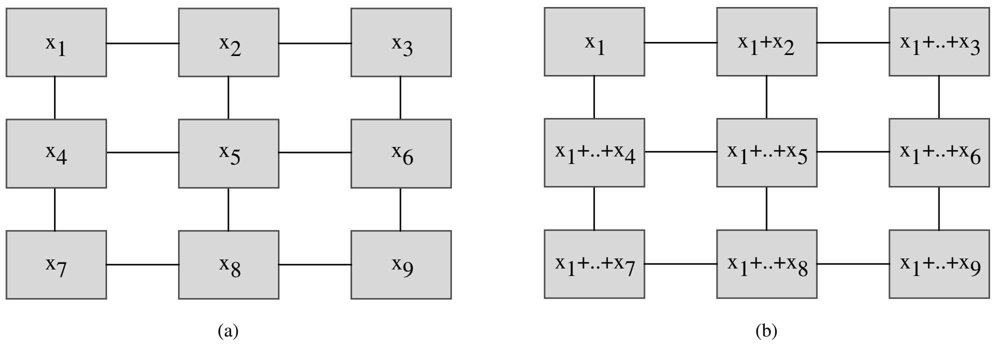


Figure 1: (a) Input and (b) output of the prefix sum algorithm on the mesh interconnection network.

algorithm works in three phases. Each processor $P[i, j]$ stores in its local memory two different values, named $u[i, j]$ and $v[i, j]$, for $1 \leq i, j \leq s$. The values $x_1 \ldots x_n$ are initially loaded in $u[i, j]$ as shown in Figure 1(a). The algorithm outputs the prefix sums in $u[i, j]$ as shown in Figure 1(b).

1. The first phase, for each $j \in [2, s]$, sequentially updates the values $u[i, j]$ increasing them by $u[i, j-1]$, for each i. The update is performed in parallel by all the processors $P[\_, j]$, i.e., all the processors in the same column j.

2. The second phase involves only processors on the last column, i.e., $P[\_, s]$. In $s-1$ sequential steps the two following operations are executed: $u[i, s] \leftarrow u[i, s] + u[i-1, s]$ and $v[i, s] \leftarrow u[i-1, s]$. After this step, all $u[1, \_]$ and $u[\_, s]$ store the output values.

3. In $s-1$ sequential steps the result is finally computed: $v[i, j+1]$ is added to $u[i, j]$ and copied in $v[i, j]$. This update is done in parallel by all the processors $P[\_, j]$ in the same column j, with j sequentially decreasing from $s-1$ down to 1.

3.2 Visualizations and discussion

Some snapshots of the visualizations of two different groups of students are shown in Figure 2, one column per group. Selected animations are available at the URL `http://www.dsi. uniroma1.it/~finocchi/paral`
Notice that both visualizations show two items per processor: the upper items correspond to the values $u[i, j]$, while the lower ones to the auxiliary values $v[i, j]$.

The images reported in Figure 2 are not sufficient to grasp the behavior of the algorithms, as they are selected from the animation in non-consecutive moments. However, they are useful to highlight some differences in the visualizations even for this very simple algorithm. The first and most important point is concerned with the way in which the current step of the algorithm has been highlighted. Color and thickness have been recognized by all the students as fundamental tools for communicating this information, but they have been differently used. The analysis of the figures shows that students of the first group (left column) were most interested in the numeric values that processors manage, and therefore on the correctness of the result of the parallel computation, while students of the second group (right column) were struck by the active processors and their interconnections, proving a deeper comprehension of the instructor's request.

As far as the algorithm has been implemented, it is worth remarking that a few groups of students, guided by the observation of their own animations, were able to change the pseudo-code trying to realize implementations with a smaller communication load. This is the reason why the v values maintained by each processor look different in the left and right column of Figure 2, even if the final result is correct in both cases. The diversity in the visualization depends on the fact that the algorithm has been differently implemented by the two groups. The automatic consistency between images and computation state, and therefore the unnecessity of identifying the interesting events, turned out

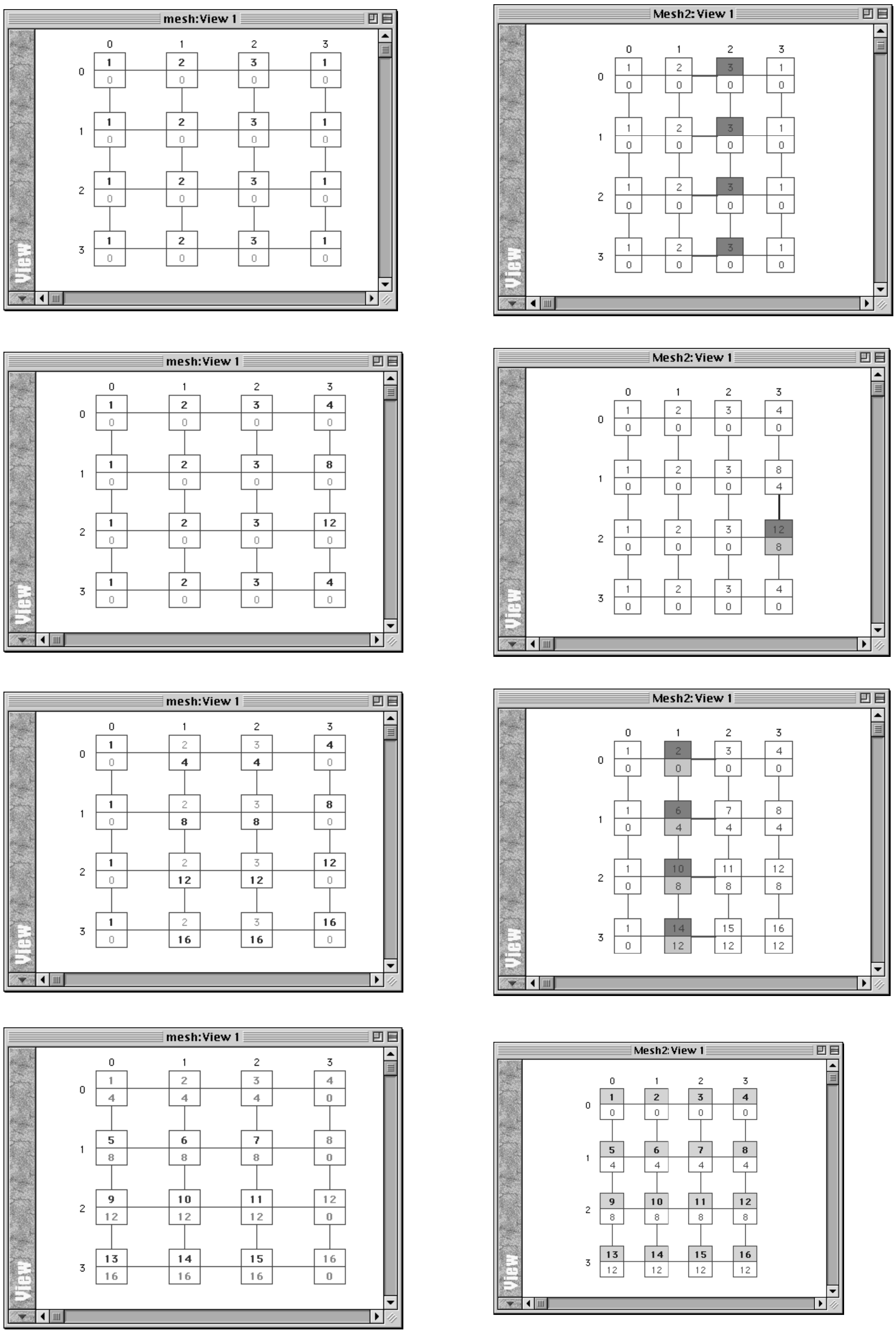


Figure 2: Prefix sums on the mesh model: snapshots taken from two different animations.

```
Mesh(Out P,Out M)
MeshProcessorItems(Out K,M)
MeshItemValue(I,J,K,Out Type,Out Val,M)
MeshItemColor(I,J,K,Out Color,M)
MeshItemSize(I,J,K,Out Size,M)
MeshItemFace(I,J,K,Out Face,M)
MeshLineColor(L,Out Color,M)
MeshLineThickness(L,Out Thick,M)
MeshLineStyle(L,Out Style,M)
```

Table 1: Signatures of ALPHA predicates for mesh visualization.

to be very useful to highlight these differences. In other words, the visualization has been used by the students to debug not only the source code, but, at a higher level of abstraction, the underlying algorithm.

4 Prefix sums on the mesh model: the code

In this section we briefly discuss the implementation and animation of the mesh algorithm as realized by a group of students. The students followed the pseudo-code presented in the theoretical lesson, without attempting at improving it (see Section 3.1 for a detailed description). Some steps of their animation are illustrated in the rigth column of Figure 2.

4.1 The mesh library

As stated in Section 2, C programs can be animated in Leonardo by embedding in the source code special visualization declarations written in a simple logic-based language. Table 1 presents the signatures of a few ALPHA predicates useful for visualizing meshes. These predicates are available in a visualization library that comes with Leonardo and have been used by the students to realize their own animations.

We limit here to give a flavor of their usage. Predicate `Mesh` declares a mesh with `P` processors per row/column and with identification number `M`. The number `K` of items managed by each processor that must be displayed in the visualization is declared by predicate `MeshProcessorItems`. For instance, in both the animations in Figure 2 we have `K`=2.

Using predicate `MeshItemValue`, the type (e.g., Int, Float, String) and the value of the `K`-th item managed by processor (`I`,`J`) can be declared. Similar predicates allow it to declare visual features of processors and items, such as color, size, and face.

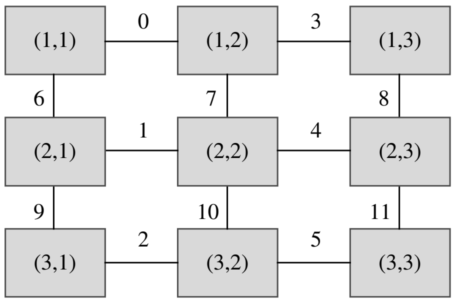
The last three predicates are related to the visualization of communication links. Differently from processors, which are numbered by means of pairs (`I`,`J`), the numbering of communication links is a little bit more tricky, as reported in Figure 3. The predicates for link manipulation take the link number `L` and the mesh identification number `M` as input, and return color, thickness, and style of the link, respectively.

4.2 Data structures and implementation of the algorithm

The implementation code of the prefix sum algorithm consists only of the `main` function, at the beginning of which the following variables are declared:

```
int i,j,side,*data,*aux;
```

Variables `i` and `j` are used for indexing mesh processors. Variable `side` stores the size of the mesh side, i.e., the square root of the number of processors. Recall that each processor $P[i, j]$ has to maintain in its local memory two values, $u[i, j]$ and $v[i, j]$. In this implementation array `data` stores the u values: at the beginning of the algorithm execution these values coincide with the input numbers; then, `data`



(c)

Figure 3: Processor and link numbering in the mesh interconnection network as used by visualization predicates.

is progressively manipulated and at the end it contains the prefix sums. Similarly, the v values are maintained by array aux.

The size of both arrays data and aux is equal to $side^2$. Actually, the mesh is allocated using arrays, instead of a matrix. Indexing these arrays is nevertheless easy: in particular, $u[i,j] = data[i \cdot side + j]$ and $v[i,j] = aux[i \cdot side + j]$. In the source code index(i,j) is used as a shortcut for the expression $\mathtt{i} \cdot \mathtt{side} + \mathtt{j}$.

4.3 A basic visualization

The visualization declarations are inserted in the code as special comments. In particular, the following three predicates are used to create the mesh and to specify some basic attributes:

```
/**
    Mesh(Out P, Out 1) Assign P = side;
    MeshProcessorItems(Out 2,1);
    MeshItemValue(I,J,K,Out Int,Out V,1)
        If K==1 Assign V = data[index(I,J)];
        Moreover If K==2 Assign V = aux[index(I,J)];
**/
```

The first predicate declares a mesh with identification number 1 whose side P is equal to the content of the C variable side. Each processor (I,J) in the mesh maintains two local values, as declared by MeshProcessorItems. Both items store integer values, equal to data[index(I,J)] and to aux[index(I,J)], respectively. These declarations are sufficient to create a basic visualization: each time the content of any of the variables side, data, or aux is changed by the underlying program computation, the visualization of the mesh is updated accordingly.

4.4 A more refined visualization

The students were not satisfied by such a simple visualization, because it does not clearly convey which processors are operating at each instant, which ones are sending information, and which ones are receiving it. The use of colors and of other graphical features was identified by the students as a proper mean to obtain the desired effect.

The refined visualization has required introducing an auxiliary integer variable, named step. This variable maintains the number of the current step of the algorithm, and can be either 1 or 2 or 3 (see the algorithm explanation in Section 3.1). The following additional declarations use variable step to produce the more refined visualization:

```
/**
      MeshItemColor(I,J,1,Out Cyan,1) If (step==1 && J==j);
      MeshItemColor(I,J,K,Out Col,1)
            If (step==2 && I==i && J==side-1) || (step==3 && J==j)
            Assign Col = (K==1) ?  Cyan :  LightGrey;

      MeshLineColor(L, Out Red ,1)
            If step==1 && j<side && (j-1)*side<=L && L<j*side;
      MeshLineColor(L, Out Blue,1)
            If step==2 && i<side && L==side*(side-1)+i*side-1;
      MeshLineColor(L, Out Red ,1)
            If step==3 && j<side && j*side<=L && L<(j+1)*side;

      MeshLineThickness(L,Out Thick,1)
            For Color:  MeshLineColor(L,Color,1)
            If Color==Red || Color==Blue;
**/
```

The first two predicates are used to color the active processors. In particular, during the first step, the underlying program propagates information left to right in the *u* values, letting the column index `j` increase from `0` to `side-1`. Hence, the first definition of predicate `MeshItemColor` assigns `Cyan` color to the *u* value managed by processor `(I,J)` if `step==1` and the processor is in the currently active column `j` (i.e., `J=j`). Since no test is performed on parameter `I`, the predicate holds for each value of `I`. Similar conditions are imposed to color processors during the other steps.

The remaining predicates are used for highlighting the active communication links at any step. Here we need to recall that communication links among processors are indexed as in Figure 1(c): numbers from 0 to $side*(side-1)-1$ are devoted to horizontal links and numbers from $side*(side-1)$ to $2*side*(side-1)-1$ to vertical ones. Based on this numbering, during the first step, as long as the C variable `j` is smaller than the mesh side, a link `L` has `Red` color if `(j-1)*side≤L<j*side`. It is easy to see that the previous condition identifies all the horizontal links between processors in columns $j-1$ and j, respectively: for instance, in the mesh with side 3 given in Figure 1(c), when `j=2=side-1` links 3, 4, and 5 are highlighted. Similar considerations hold for coloring links in the second and third steps. Moreover, all red and blue colored links have thick size, as asserted by predicate `MeshLineThickness`.

We finally need to show how visual events are linked to the source code. Due to the lack of space we consider only the first step of the algorithm, the code for the other steps being similar. We recall that in Leonardo visual events are automatically detected by the system each time the content of a data structure changes. Hence, in order to highlight correctly the parallel steps, it is crucial to identify points of the source code where the visualization must be *turned off*: in this way, when the visualization is turned on again, all the changes that took place meanwhile are visualized at the same time, even if data structures have been progressively modified by the algorithm in consecutive sequential steps. The visualization can be turned on and off by means of predicate `ScreenUpdateOn`:

```
step=1;
for (j=1; j<side; j++) {
      /** Not ScreenUpdateOn; **/
      for (i=0; i<side; i++) data[index(i,j)] += data[index(i,j-1)];
      /** ScreenUpdateOn; **/
}
```

Thanks to the use of predicate `ScreenUpdateOn`, the inner `for` cycle in the piece of code above will be visualized as a single parallel step, producing the effect that all the items in a fixed column `j` have been changed simultaneously.

Acknowledgments

We would like to thank all the students who took part in the experimentation for their enthusiasm and their active participation.

References

R. Ben-Bassat Levy, M. Ben-Ari, and P.A. Uronen. An extended experiment with JEliot 2000. In *Proceedings of the 1st Program Visualization Workshop (PVW'00)*, pages 131–140. University of Joensuu, 2000.

M.H. Brown and M. Najork. Collaborative Active Textbooks: a Web-Based Algorithm Animation System for an Electronic Classroom. In *Proceedings of the 12th IEEE Symposium on Visual Languages (VL'96)*, pages 266–275, 1996.

G. Cattaneo, U. Ferraro, G.F. Italiano, and V. Scarano. Cooperative Algorithm and Data Types Animation over the Net. In *Proc. XV IFIP World Computer Congress, Invited Lecture*, pages 63–80, 1998. System Home Page: `http://isis.dia.unisa.it/catai/`.

P. Crescenzi, C. Demetrescu, I. Finocchi, and R. Petreschi. Reversible execution and visualization of programs with Leonardo. *Journal of Visual Languages and Computing*, 11(2):125–150, 2000. System home page: `http://www.dis.uniroma1.it/~demetres/Leonardo/`.

C. Demetrescu and I. Finocchi. *The Leonardo user manual*, 2000. Available at `http://www.dis.uniroma1. it/~demetres/Leonardo/Documentation.html`.

C. Demetrescu and I. Finocchi. Smooth animation of algorithms in a declarative framework. *Journal of Visual Languages and Computing*, 12(3):253–281, 2001. Special Issue devoted to selected papers from the 15th IEEE Symposium on Visual Languages (VL'99).

J. Domingue and P. Mulholland. Teaching Programming at a Distance: The Internet Software Visualization Laboratory. *Journal of Interactive Media in Education*, 7, 1997. Available at `http://www-jime.open. ac.uk/97/1/`.

J. Jájá. *An Introduction to Parallel Algorithms*. Addison Wesley, 1992.

C.M. Kehoe and J.T. Stasko. Using animations to learn about algorithms: An ethnographic case study. Technical report, Georgia Institute of Technology, Atlanta, Tech. Rep. GIT-GVU-96-20, 1996.

A.W. Lawrence. *Empirical Studies of the Value of Algorithm Animation in Algorithm Understanding*. PhD thesis, Georgia Institute of Technology, Atlanta, 1993.

A.W. Lawrence, A.N. Badre, and J.T. Stasko. Empirically Evaluating the Use of Animations to Teach Algorithms. In *Proceedings of the 10th IEEE International Symposium on Visual Languages (VL'94)*, pages 48–54, 1994.

R.E. Mayer. Systematic thinking fostered by illustrations in scientific text. *Journal of Educational Psychology*, 81(2):240–246, 1989.

S. Palmiter and J. Elkerton. An evaluation of animated demonstrations for learning computer-based tasks. In *Proceedings of the ACM SIGCHI'91 Conference on Human Factors in Computing Systems*, pages 257–263, 1991.

R. Petreschi. *Notes from the course on parallel algorithms (in Italian)*. Available at the URL `http:// cesare.dsi.uniroma1.it/~asd/asd.html`.

M. Pressley. Imagery and children's learning: Putting the picture in developmental perspective. *Review of Educational Research*, 47(4):585–622, 1977.

L.P. Rieber, M.J. Boyce, and C. Assad. The effects of computer animation on adult learning and retrieval tasks. *Journal of Computer Based Instruction*, 17(2):46–52, 1990.

G.C. Roman, K.C. Cox, C.D. Wilcox, and J.Y Plun. PAVANE: a System for Declarative Visualization of Concurrent Computations. *Journal of Visual Languages and Computing*, 3:161–193, 1992.

J.T. Stasko. TANGO: A Framework and System for Algorithm Animation. *Computer*, 23:27–39, 1990.

J.T. Stasko. Using Student-Built Algorithm Animation as Learning Aids. In *Proceedings of the ACM SIGCSE Conference*, pages 25–29, 1997.

J.T. Stasko, A.N. 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*, pages 61–66, 1993.

J.T. Stasko, J. Domingue, M.H. Brown, and B.A. Price. *Software Visualization: Programming as a Multimedia Experience*. MIT Press, Cambridge, MA, 1997.

Using Hands-On Visualizations to Teach Computer Science from Beginning Courses to Advanced Courses

Susan H. Rodger
Computer Science Department, Duke University, Durham, NC 27708-0129

rodger@cs.duke.edu

1 Introduction

Students learn best by seeing several representations of a concept, including textual, visual, and animated. Textbooks can provide the textual and some visual representations. Software tools provide a visual and animated view. However, observing is not enough. In order to check their understanding of a concept, students must be able to interact with the concept in some way, receiving feedback.

There have been many attempts at creating visualizations of computer science concepts in textbooks. These textbooks take care of the textual representation, and possibly some part of the visual representation. For example, algorithms textbooks illustrate via snapshots the execution of sorting algorithms. Sedgewick (Sedgewick, 1988) represents numbers as columns of different heights, and shows snap shots of the movement of the columns starting in random order until the columns are sorted by heights. As another view, he shows a 2-d matrix where the *i*th column represents the number with value *i* which is shown as a black pixel somewhere in the column. All numbers are sorted when the *i* value in column *i* appears in row *i* for all *i*, forming a diagonal line on the main diagonal. These snapshots allow the student to study the animation of one example, but only that example.

As another example, automata theory textbooks visualize automata by showing transition diagrams. However it is tedious to trace an input string through an automaton, especially if the automaton is nondeterministic.

There are many animations of computer science concepts available on the web. For example, the sorting animations in Sedgewick's textbook described above appear as Java applets on several web pages, mostly the results of student projects. However, many such animations do not allow the user to interact with the animation. Only one data set is used, and the same animation plays over and over again.

Software tools are needed to view the animated representation, and to further experiment with the concepts. We have been experimenting with visual and interactive tools (Rodger, 2002a) in several levels of computer science courses from the non-major course, to the introductory programming courses (CS 1 and CS 2), to the upper-level undergraduate course in automata theory and formal languages. We describe two such tools and their use in computer science courses, JAWAA for easy creation of animations and JFLAP for experimenting with concepts in automata theory.

In Section 2 we describe our requirements in creating tools. We describe the tool JAWAA in Section 3 and its use in Section 4. We describe the tool JFLAP in Section 5 and its use in Section 6. In Section 7 we describe additional tools for animation and for use along with JFLAP. Finally we conclude with future work in Section 8.

2 Successful Interactive Visualization

For a successful interactive visualization, students should be in control of the visualization and should receive feedback immediately. Control includes changing the input, changing the speed, allowing movement forward and backward and the ability to reproduce the result. Students should be able to create their own input and receive feedback on their input immediately.

[*]The work of this author is supported by the National Science Foundation through grant NSF DUE-9752583.

3 JAWAA for Animation

JAWAA (Pierson and Rodger, 1998) is a scripting language that can be the output of a program. It is designed to easily run over the web with no installation. One can enter JAWAA commands in a file (either by hand or as output from a program), modify a templated HTML file, and easily generate an animation that runs on a web page.

With JAWAA one can create simple primitives such as circles, rectangles, polygons, lines and text, and move them around singly or in a group. With advanced features of JAWAA, one can create arrays, stacks, queues, trees and graphs easily. The animation has a speed control bar and the ability to pause or restart the animation.

4 Using JAWAA in Courses

We describe how we have used JAWAA in several computer science courses at Duke University.

4.1 JAWAA in Non-Majors Freshmen Seminar

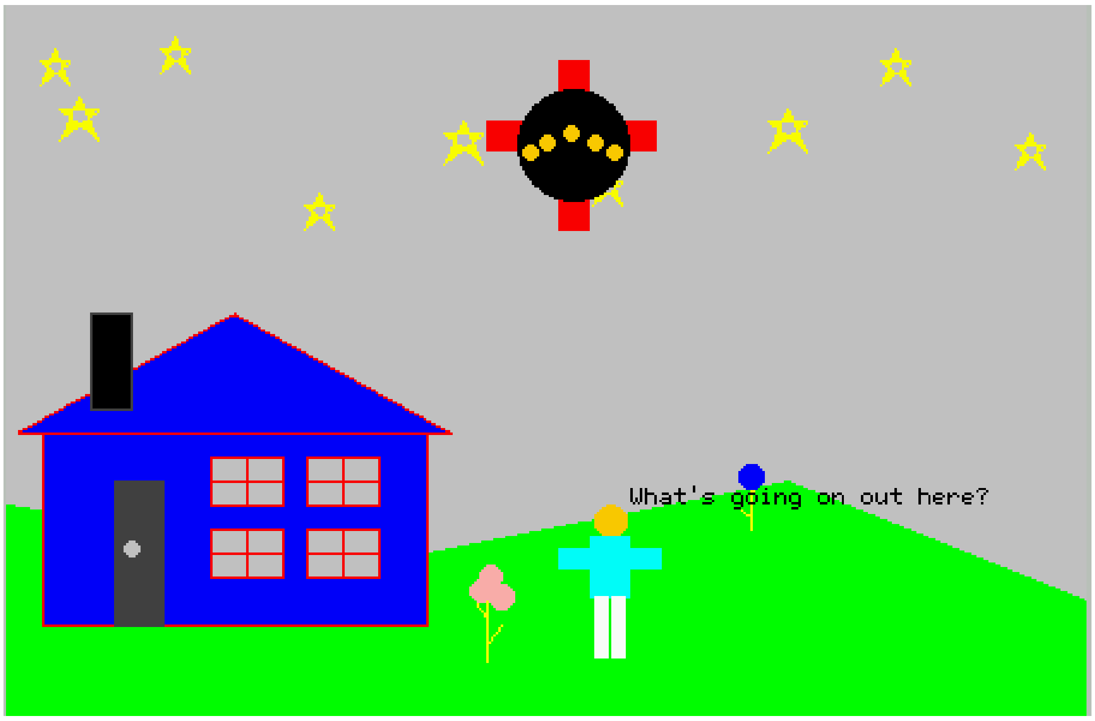


Figure 1: JAWAA Outer Space

The non-majors course CPS 49S (Rodger, 2002b), Animation and Virtual Worlds, is a seminar for freshmen to learn animation and computer science concepts. Several tools are used in this course to create 2-D and 3-D worlds, and such tools must be easy to use by nonprogrammers, as programming experience is not a prerequisite.

Creating animations with JAWAA is one of the units in this course, lasting one week and a half. Students create a JAWAA animation by typing a JAWAA script into a file. Creating an animation in this way is easy for them to do, but a bit tedious. Furthermore, students use only the simple primitives in JAWAA.

Students begin work together in pairs by following a tutorial that guides them in creating a simple animation. Next they work on specific projects in pairs. One project is a traffic intersection with vehicles and stoplights. Another project is a sorting algorithm animation. They are given columns of different heights in random order and must sort them using an algorithm they create. Finally they work individually on a JAWAA animation for homework. Figure 1 shows a student JAWAA project. In this animation the space ship flies in, pulls the bush up and takes it, and a person walks out of the house into the yard.

4.2 JAWAA in CS 1

Duke's CS 1 course, CPS 6, teaches programming using C++. In this course JAWAA is used by hiding the details of JAWAA in classes. Students use a C++ class that has already been written for them and JAWAA is automatically output into a file that students can load onto a web page.

For example, students learn about a hot air balloon class. They can control hot air balloons, making them ascend, cruise, and descend. In an early assignment when they are learning about classes, they can create the balloon and JAWAA output is automatically written to a file when they run their program. They can put this file on a web page and see the balloon move. Later when they learn about functions returning values, loops and randomness, they can create a hot air balloon race. The race has several balloons that randomly move one to four steps each. Students can continually check the distances of each balloon to decide which balloon wins the race.

4.3 JAWAA in CS 2

Duke's CS 2 course, CPS 100, teaches data structures using C++. In this course students write JAWAA commands as output in their programs. With JAWAA students can create both the simple primitives and data structures easily. All the projects in the course have data structures, so students can pick out a particular data structure to animate.

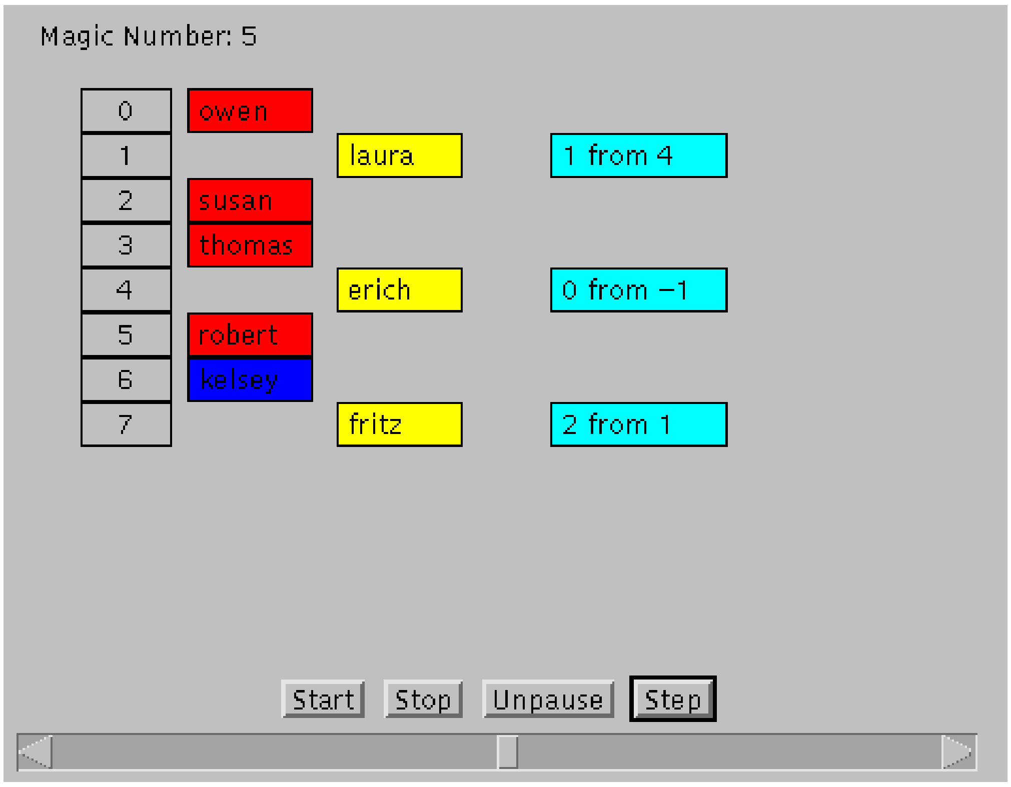


Figure 2: JAWAA Josephus Problem

Figure 2 shows a CPS 100 project illustrating how arrays are shown in JAWAA. This figure shows the Josephus problem after three passes. In the Josephus problem, the magic number chosen is 5. Starting from 0, every fifth person is pulled out of the array. The first person pulled out is erich, then laura, then fritz.

To implement the Josephus problem, there are three arrays created, one for numbers labeled 0 to 7 to show the indices of the arrays, one array for players' names and one array for tracking the order the players were pulled out of the array. For example, laura is in slot 1 and was pulled out after erich who is in slot 4.

The declaration of the array for players' names is shown below. The array's leftmost top corner is at location (80,30), there are 8 positions in the array, the array is vertical, the inside color of the array is red, and the outline color of the array is black.

```
    array Players 80 30 8 owen laura susan thomas
erich robert kelsey fritz vert black red
```

After declaring an array, the individual positions can be referenced. Thus, stepping through the array can be animated easily by changing the inside color of a position to a different color when examining it, and back to its original color when finished. The first 10 lines of the code below illustrates stepping through 5 positions in the array, changing the color to blue when it examines a particular position in the array. Another feature of JAWAA is that array positions (rectangle) can be moved. In the code below the last two lines move the position with erich in it to the right and change the background color to yellow.

```
    changeParam Players[0] bkgrd blue
changeParam Players[0] bkgrd red
changeParam Players[1] bkgrd blue
changeParam Players[1] bkgrd red
changeParam Players[2] bkgrd blue
changeParam Players[2] bkgrd red
changeParam Players[3] bkgrd blue
changeParam Players[3] bkgrd red
changeParam Players[4] bkgrd blue
changeParam Players[4] bkgrd red
moveRelative Players[4] 70 0 true
changeParam Players[4] bkgrd yellow
```

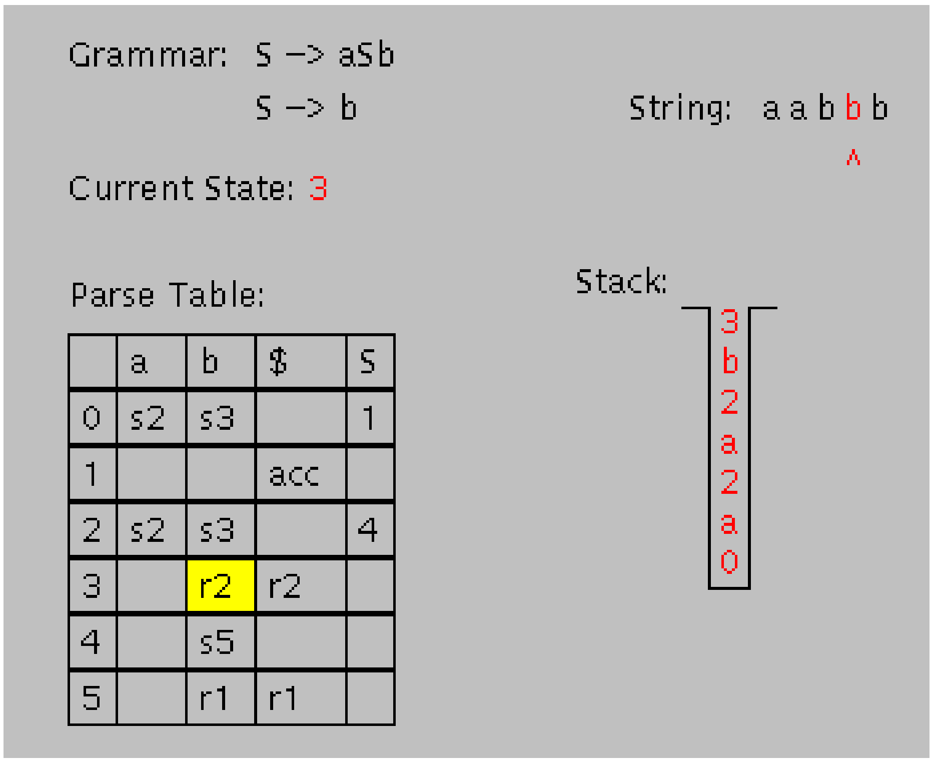
With JAWAA's features of data structures, students can generate an animation involving data structures quickly. For example, in the Josephus problem above the student used one line to declare a filled array. If the animation was created using only simple primitives, over 15 lines of JAWAA code would have been needed. In Figure 3 a stack is shown on the right. One JAWAA command creates an empty stack, one JAWAA command animates the push of an item onto the stack and one JAWAA command animates the pop of an item from the stack. Alternatively, many lines of code would be needed to animate the stack from simple primitives.

4.4 JAWAA in Other CS courses

Other courses can use JAWAA in two ways. One to create animations of concepts to study, and two to animate a part of their program. We show examples of both ways below for CPS 140, the automata theory course at Duke.

As an example of an animation to study a concept, Figure 3 shows an animation of an LR parse table. The animation shows the parsing of the string aabbb. As the parsing proceeds, the current state is listed, the current position in the table is highlighted in yellow, and the stack contains the current symbols and state numbers. Currently the parsing is in state 3 and a reduce by rule 2 is about to happen, meaning items will be popped from the stack.

For an example of using JAWAA with a programming project, in CPS 140 students study a new simple language and write an interpretor for it, studying in detail the phases of the interpretor and how they relate to automata theory. Figure 4 shows a student's animation of a CAMO program. CAMO is a simple language in which you can declare cat, mice and mouse holes and move the cat and mice around. The figure shows one cat in yellow (the lightest color) being chased by a large number of mice that all came out of the mouse hole (the darkest color). With JAWAA students can see their program execute.



Grammar: S -> aSb

S -> b

String: a a b b b

^

Current State: 3

Parse Table:

	a	b	$	S
0	s2	s3		1
1			acc	
2	s2	s3		4
3		r2	r2	
4		s5		
5		r1	r1	

Stack:

3
b
2
a
2
a
0

Figure 3: JAWAA LR Parser

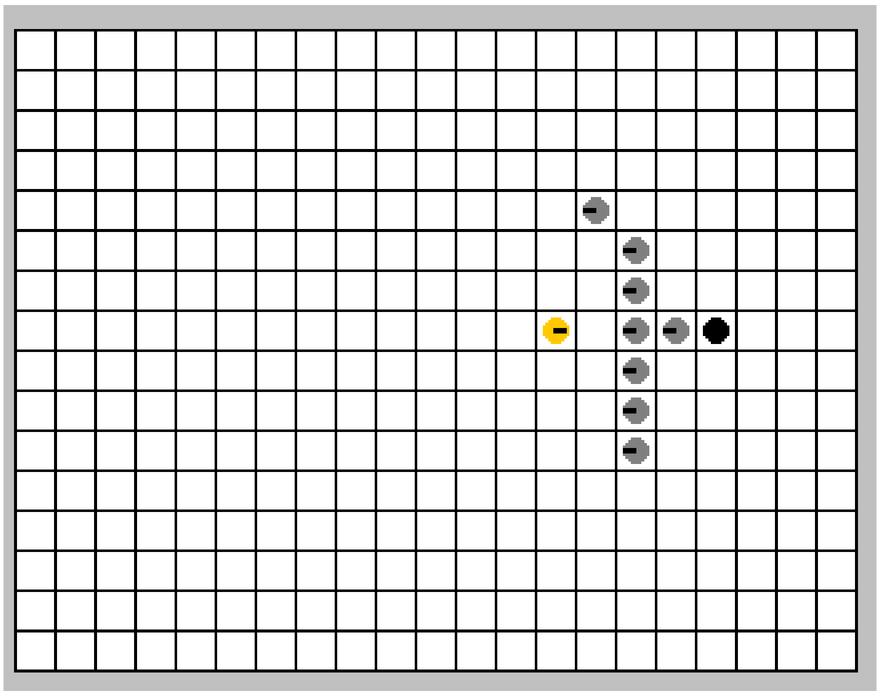


Figure 4: JAWAA Cat Vs Mouse

5 JFLAP for Automata Theory

JFLAP (Hung and Rodger, 2000; Gramond and Rodger, 1999) is a tool for interacting with automata theory concepts. With JFLAP one can create a finite automaton, pushdown automaton, and 1 or 2-tape Turing machine, drawn as a transition diagram. The user can enter input strings and watch the execution, even for nondeterministic machines. In addition, one can explore the conversion of several different forms including NFA to DFA, DFA to minimum state DFA, NFA to regular grammar, regular grammar to NFA, regular expression to NFA, NFA to regular expression, NPDA to CFG, and CFG to NPDA. JFLAP has been designed with general definitions and can be used with most automata theory textbooks such as (Linz, 2001).

Without JFLAP, students create their automata on paper and turn them in, usually with errors in them. It is too tedious to trace through to check for correctness. With JFLAP, their automata come to life.

6 Using JFLAP in Courses

JFLAP is used at Duke in CPS 140, the automata theory and formal languages course. It could also be used in the early phases of a compiler course. We describe how JFLAP can be used in several ways during lecture, and later by students outside of lecture.

6.1 Instructors Demo JFLAP

Instructors can demo JFLAP to students in lecture to show them how to use JFLAP. Demoing can show students the ease of use and its functionality, including moves they may not figure out on their own. As an example, in earlier years when JFLAP was not demoed in class, students learned it on their own for homework. Many students would turn in awful looking drawings, because they did not realize that states could be moved after they are created.

6.2 Instructors Solve Problems With the Class

Instructors can solve problems using JFLAP during lecture with input from the students. A typical example is giving the students an automaton and to ask them if it is correct or not, and if not to fix it. Figure 5 shows an incorrect Turing machine for recognizing the language $\{a^n b^n c^n \mid n > 0\}$. Students are asked during lecture to take a few minutes to trace the machine to see if it is correct. Tracing is easier and shorter than asking them to write the complete machine during class. Almost all students will say the machine is correct. The instructor can then run the machine on an input string suggested by a student such as **aaabbbccc**, which accepts. At this point the instructor can ask the class if there are strings that the Turing machine does not work correctly for, and then run the TM on the suggested strings. The instructor can also suggest the string *aaabbbbccc*, a string not in the language, run the TM and show the TM accepts it! Figure 6 shows a snapshot of this run. Here students can see that the TM is not checking to see if there are extra b's or c's. Students can be given a few minutes to determine how to fix the machine. A volunteer can suggest fixes which the instructor can implement and try on the previous input string.

6.3 Instructors Experiment with Difficult Concepts

Instructors can use JFLAP to experiment with difficult concepts. Students have difficulty creating a nondeterministic pushdown automaton (NPDA) since they are not used to thinking in a nondeterministic way. An example is to ask them to write an NPDA for $\{ww^R|w \in \Sigma^+\}, \Sigma = \{a, b\}$ during class. The majority of them will be stumped. They want to find the middle first, which is impossible since this language is not a deterministic context-free language. With some help in guiding them, the class can create the NPDA solution, shown in Figure 7.

Most students will not believe this NPDA is correct without some experimentation. There are three ways to run an input string. The first way to run it is to select the *Fast run*. JFLAP runs the

NPDA on the input string and prints a message when it completes (or a message that it is taking a long time and prompts whether or not to continue). In running the above NPDA on the string a^{16} a message stated that the string was accepted, 89 configurations were generated, and the path of acceptance was of length 18. This gives the instructor a chance to explain how the NPDA runs and that there are several configurations being tried at the same time. The second way to run the NPDA is the *Step run* which shows all the current configurations at any time (up to a limit of 15). By stepping through and showing how to control the run (the user can freeze configurations for later expansion), students can experience the nondeterminism. Figure 8 shows a snapshot of the run on a^{16} in which 8 configurations are currently active and one configuration cannot proceed any further. Each configuration shows the current state, input symbols remaining, and current stack.

6.4 Instructors Work Examples of Transformations

Instructors can give an example of a transformation either before or after studying a formal proof. For example, one proof students study is that every NPDA can be converted to a context-free grammar (CFG) that represents the same language. The instructor can start JFLAP and choose an NPDA, such as the one in Figure 7, and then step through the steps of converting that NPDA into the equivalent grammar.

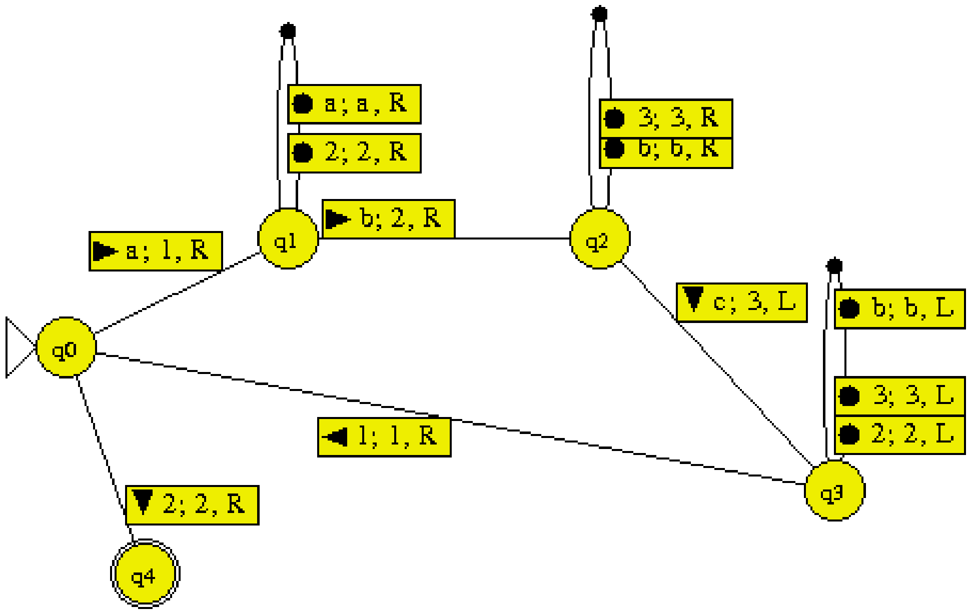


Figure 5: Incorrect JFLAP Turing machine for $a^n b^n c^n$

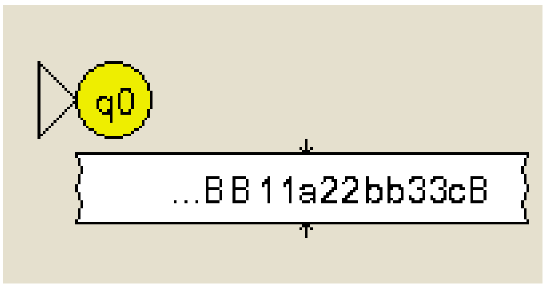


Figure 6: JFLAP Turing machine run on *aaabbbbccc*

6.5 Instructors Relate Concepts From Other Courses

Instructors can use JFLAP to show how automata relate to other computer science courses, by studying some of the same concepts. For example, with JFLAP one can revisit the problem of developing test data, in this case to test out an automaton. A third run feature is *Multiple run* mode, a window in which one can enter up to 10 input strings and JFLAP will run on all of them providing accepts or rejects. Another example is to study the running time of one and two-tape Turing machines. With JFLAP one can run them and see how long (how many steps) they take to figure out the running time.

6.6 Students using JFLAP

Students can use JFLAP outside of class for homework and for additional study problems they create. There are the obvious problems such as building an automaton and creating a set of test data, and stepping through one of the many transformations included in JFLAP. Students can also use JFLAP to study properties. For example, given two DFAs, students should create the DFA that represents the intersection of the two, and create test data to determine if they have been successful. One more use of JFLAP is that students can load files from lecture to recreate the instructor's examples presented in class.

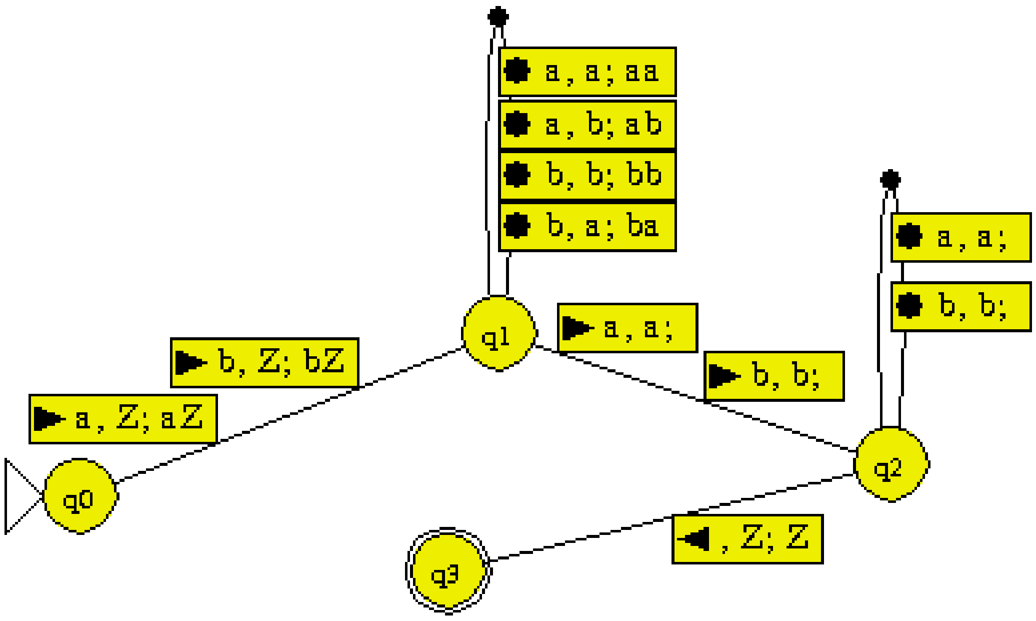
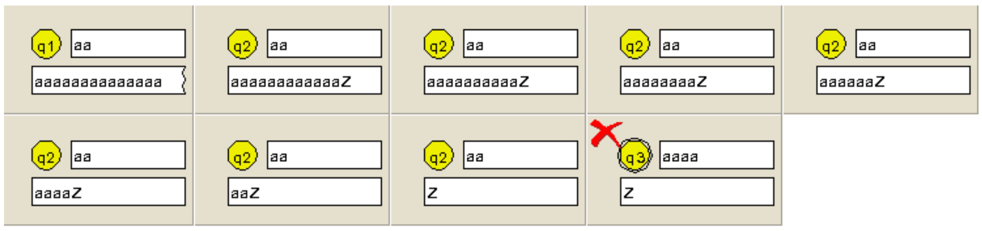


Figure 7: JFLAP NPDA for ww^R



q1 aa
aaaaaaaaaaaaaa

q2 aa
aaaaaaaaaaaaZ

q2 aa
aaaaaaaaaaZ

q2 aa
aaaaaaaaZ

q2 aa
aaaaaaZ

q2 aa
aaaaZ

q2 aa
aaZ

q2 aa
Z

q3 aaaa
Z

Figure 8: JFLAP run of NPDA for ww^R

7 Other Tools for Hands-On Visualization

There are several other tools for creating animations with scripts. Samba (Stasko, 1997) has been used to create animations for many computer science topics including CS 1, CS 2, algorithms, and operating systems. AnimalScript (Rossling and Freisleben, 2001) is a scripting language that includes a list element primitive and many other features.

We mention several other tools that can be used for teaching automata theory with hands-on visualization. Pâté (Hung and Rodger, 2000) is a tool for parsing restricted and unrestricted grammars and a grammar transformer from a context-free grammar to Chomsky Normal Form. Given a grammar and an input string, an exhaustive search parser shows either the textual derivation or the parse tree. This tool can be used in conjunction with JFLAP in the example above in which JFLAP converts an NPDA to a CFG. The resulting CFG can then be tested with input strings to convince the students it recognizes the same input strings as the NPDA.

JeLLRap (Rodger, 2002a) is an instructional tool for building LL(1), LL(2) and LR(1) parse tables. JeLLRap guides the user through several steps in constructing a parse table. When the table is complete, the user can enter an input string and step through the building of the parse tree. JFLAP is a Java version of LLparse and LRparse (S. Blythe and Rodger, 1994).

Other tools include Turing's World(Barwise and Etchemendy, 1993), a tool for experimenting with Turing machines as building blocks, and Snapshots of the Theory of Computing (C. Boroni and Ross, 2001), a hypertextbook for the web.

8 Future Work

We are currently working on a new version of JAWAA that will be easier to create animations by novices and by instructors of a class. To create an animation one will be able to create two JAWAA files, one using an editor to setup the animation, and then a second file created from a program to finish the animation We are also updating JFLAP to include features from JeLLRap and Pâté.

Acknowledgements Thanks to the many students who worked on these tools, including Anna Bilska, Thomas Finley, Eric Gramond, Ted Hung, Pretesh Patel, Willard Pierson, Magda Procopiuc, Tavi Procopiuc, and Jason Salemme.

References

J. Barwise and J. Etchemendy. *Turing's World.* CSLI, 1993.

M. Grinder C. Boroni, F. Goosey and R. Ross. Engaging students with active learning resources: Hypertextbooks for the web. *Thirty-second SIGCSE Technical Symposium on Computer Science Education*, pages 65–69, 2001.

E. Gramond and S. H. Rodger. Using jflap to interact with theorems in automata theory. *Thirtieth SIGCSE Technical Symposium on Computer Science Education*, pages 336–340, 1999.

T. Hung and S. H. Rodger. Increasing visualization and interaction in the automata theory course. *Thirty-first SIGCSE Technical Symposium on Computer Science Education*, pages 6–10, 2000.

P. Linz. *An Introduction to Formal Languages and Automata, Third Edition.* Jones and Bartlett, 2001.

W. Pierson and S. H. Rodger. Web-based animation of data structures using jawaa. *Twenty-ninth SIGCSE Technical Symposium on Computer Science Education*, pages 267–271, 1998.

S. Rodger. Visual and interactive tools, 2002a. www.cs.duke.edu/∼rodger/tools/.

S. H. Rodger. Introducing computer science through animation and virtual worlds. *Thirty-first SIGCSE Technical Symposium on Computer Science Education*, pages 186–190, 2002b.

G. Rossling and B. Freisleben. Animalscript: An extensible scripting language for algorithm animation. *Thirty-second SIGCSE Technical Symposium on Computer Science Education*, pages 70–74, 2001.

M. James S. Blythe and S. H. Rodger. Llparse and lrparse: Visual and interactive tools for parsing. *Twenty-fifth SIGCSE Technical Symposium on Computer Science Education*, pages 208–212, 1994.

R. Sedgewick. *Algorithms*. Addison Wesley, 1988.

J. Stasko. Using student-built algorithm animations as learning aids. *Twenty-eighth SIGCSE Technical Symposium on Computer Science Education*, pages 25–29, 1997.

Visualization of computer architecture

C. Yehezkel
Department of Science Teaching, Weizmann Institute of Science, Rehovot 76100 Israel

ntcecile@wisemail.weizmann.ac.il

1 Introduction

The explanation of the internal operation of the computer is not an easy task, because the processes involved are complex and occur at a very high speed. Students of computer science and engineering acquire a cognitive model of the internal computer operation in courses and labs; this model is often incomplete or erroneous. During a course in computer architecture and assembly language programming, students learn about the internal structure of the computer and how instructions are executed (Loui, 1988). There are several professional tools available for the development of assembly language programs, such as editors, assemblers and debuggers, however, they do not meet the needs of students at the high school and undergraduate level (Foley, 1992; Simeonov and Scheinder, 1995).

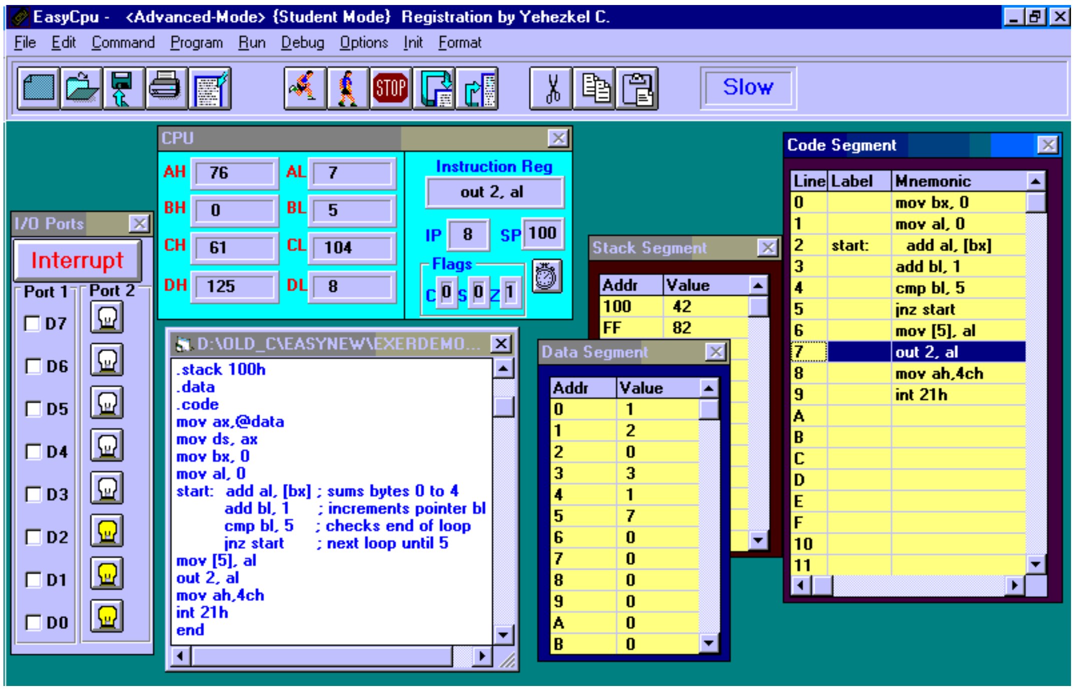
The first step of this research was to design a computer-aided learning environment called EasyCPU (Yehezkel et al., 2001) that was especially developed for teaching novices. A set of learning activities based on EasyCPU was also developed. EasyCPU runs on PC/Windows and simulates the operation of a subset of the instructions of a microprocessor from the Intel X86 family. EasyCPU presents the student with a development environment for editing, assembling and debugging programs. The second step of this research is to investigate the effectiveness of the environment in order to improve the teaching process and to formulate guidelines for the use of visualization in teaching (Petre et al., 1998).

2 Description of the EasyCPU

EasyCPU has two modes of operation: basic mode and advanced mode. Basic-mode enables the novice student to learn the syntax and semantics of individual instructions. The display shows the main units of the computer: CPU, memory segments and elementary I/O (arrays of buttons and LEDs). The units are connected by control, address and data busses, which are animated. The student can simulate an instruction and follow the data transfer between the CPU's registers or follow the data flow between the units. The advanced-mode (see Figure 1) is used by students after they have acquired a basic knowledge of assembly language. It provides these students with a development environment for creating their own programs. The students can edit a new program or existing examples, assemble, link and execute the program continuously or step-by-step. EasyCPU can control external hardware in addition to on-screen I/O simulation and therefore the student can read and write binary data to external hardware in order to develop his own hardware project.

2.1 Insertion of EasyCPU in the Curriculum

By itself, a tool is of little use; it must be integrated with learning materials such as textbooks and lab assignments that require the student to use the environment. Anderson and Naps claim (Anderson and Naps, 2001) "the greater the degree of interaction allowed by the instructional design of the AV system, the greater degree of understanding on the part of the student." To create a computer-aided learning environment we have developed simulation-based activities. They were developed for an introductory course in computer architecture and assembly language. The course is taught to tenth or eleventh-grade high school students toward matriculation examination. Since 1995, over thousands of students in 40 high schools have used the program and the activities.


EasyCpu - <Advanced-Mode> {Student Mode} Registration by Yehezkel C.
File Edit Command Program Run Debug Options Init Format
Slow
CPU
I/O Ports
Interrupt
Port 1
Port 2
D7
D6
D5
D4
D3
D2
D1
D0
D:\OLD_C\EASYNEW\EXERDEMO...
Stack Segment
Data Segment
Code Segment

AH	AL	BH	BL	CH	CL	DH	DL
76	7	0	5	61	104	125	8

Instruction Reg: out 2, al
IP 8 SP 100
Flags: C 0 S 0 Z 1

```
.stack 100h
.data
.code
mov ax,@data
mov ds, ax
mov bx, 0
mov al, 0
start:  add al, [bx] ; sums bytes 0 to 4
        add bl, 1    ; increments pointer bl
        cmp bl, 5    ; checks end of loop
        jnz start    ; next loop until 5
mov [5], al
out 2, al
mov ah,4ch
int 21h
end
```

Stack Segment

Addr	Value
100	42
FF	82

Data Segment

Addr	Value
0	1
1	2
2	0
3	3
4	1
5	7
6	0
7	0
8	0
9	0
A	0
B	0

Code Segment

Line	Label	Mnemonic
0		mov bx, 0
1		mov al, 0
2	start:	add al, [bx]
3		add bl, 1
4		cmp bl, 5
5		jnz start
6		mov [5], al
7		out 2, al
8		mov ah,4ch
9		int 21h
A		
B		
C		
D		
E		
F		
10		
11		

Figure 1: Advanced mode of EasyCPU

3 Problems with simulations and visualizations

Many simulators are currently being used to support the teaching of introductory courses on assembly language programming (Yehezkel et al., 2001), and some simulators include visualization of material likely to be difficult to learn. There is a large variety of simulators, each targeted to a small student population: each simulator was intuitively designed to overcome specific teaching difficulties. Unfortunately, we were not able to find empirical evaluation of hardware simulators that could contribute to effective design. Recently the results of a meta-study of algorithm visualization effectiveness were published, concluding that the form of learning activity is more important that the form of the visualization itself (Hundhausen et al., 2002). Our research is centered on the assessment of the EasyCPU environment impact on the learning process. This impact will depend on many variables, among them: the amount of time invested in using the tool, the characteristics of the student population and the working environment. We are investigating the impact of the EasyCPU during the learning process, to compare among different student populations, and to compare the EasyCPU environment with the Turbo-Assembler environment.

4 Research method

The research was based on the use of EasyCPU environment in a year-long course on a wide students population (150 students). We developed a pretest and posttest to determinate whether the visualization actually helps the student to understand the salient features of the computer architecture and operation. We are also investigating if the visualization contains elements whose interpretation, while obvious to an expert, is ambiguous for the novice. From the preliminary analysis, we were often surprised by students' interpretation of some visualization elements and modified our design accordingly. For example, students thought that I/O instructions are unique to the EasyCPU environment because they used them to control on-screen I/O simulation (Figure 1). It has motivated us to enable the student to control real hardware like the I/O of a PC mother-board or external hardware connected to the computer.

Qualitative research was used to determine how students actually used the EasyCPU environment. We developed simulation-based activities that focus on the crucial program development phases: writing and debugging. We have used video recording to observe pairs of students working together in order to analyze their interaction during the activities. We found a tendency among students to use the EasyCPU to build their program directly into the environment and to check parts of the programs during the writing process. Stasko and Lawrence (1998) state that empirical studies help to better understand the pedagogical value of algorithm animation.

We performed qualitative research to determine the specific contribution of the EasyCPU environment to the development of students' debugging skills. The research was carried out on tenth-grade high school students (two classes: 43 students) studying the course computer organization unit towards the matriculation examination. One class used the EasyCPU environment and the other used the Turbo Assembler environment. The classes were composed by the school on the basis of equivalent average ability, but the TA class happened to be better as measured by a pretest in an introductory course on algorithms and programming. Because the course prepares students for the matriculation examination, it was inflexible and the same for the two classes.

In an analysis task, performed as an in-class assignment, the students were asked to find two logic errors in a program that checks if an array of unsigned natural numbers is an arithmetic progression or not. The two errors were intentionally located in different parts of the program. The experiment was first run on two pairs of students (volunteers), one who used the EasyCPU environment and a control pair who used the Turbo Assembler environment. The task execution was videotaped. Following this experiment we had the remaining students of the two classes perform the same task.

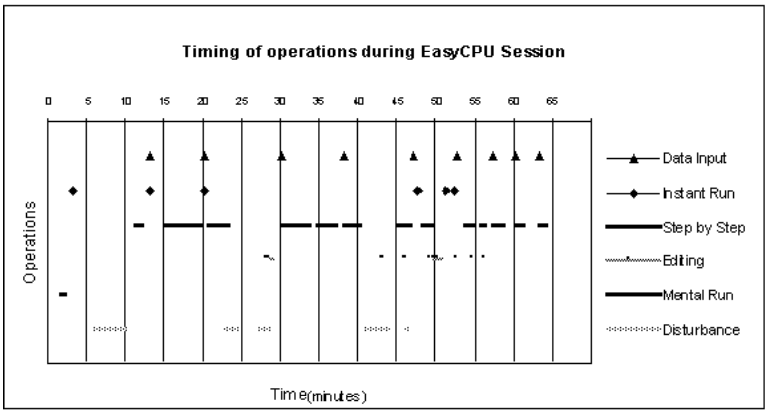
5 Results and discussion

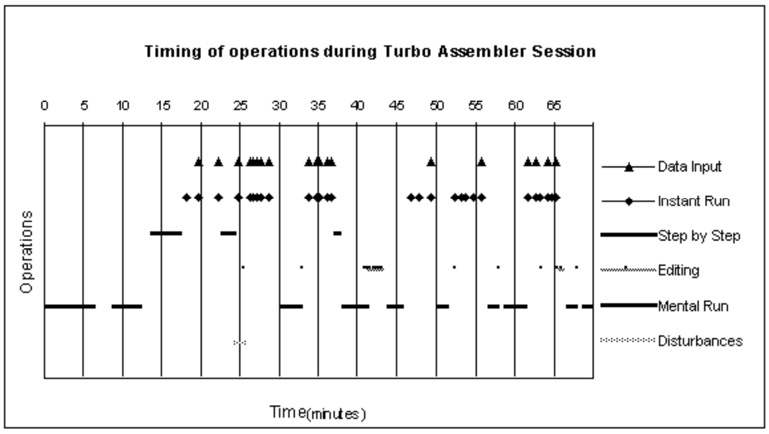
We characterized the student activities during the execution by defining categories of operations: data input, mental run, instant run and step-by-step run. This way we could graphically trace the operations during the performance of the task. The figures below present the operations performed by each pair of students plotted in function of time. The first one is the plot of EasyCPU students' operations and the second one is the plot of the Turbo Assembler students' operations during the execution of the task. The operations traced are not performed simultaneously, except for *data input* and *continuous run* which are nearly concurrent operations.

This allowed us to compare the ways of performing the task and to conduct a fine-grained analysis of characteristics of the activities, such as timing, frequency and duration. Very different patterns of operation emerged from this analysis. The TASM students overused continuous run. They related to the program as a black-box, changing data inputs and checking data outputs. Then, they diagnosed the mismatch between the actual and the predicted output by characterizing the problematic data input. They made very little use of the debugger tools. They tried to check the program by mental runs. By comparison, the EasyCPU students fully used the EasyCPU debugging tools and they analyzed the execution step-by-step to locate the errors. Further analysis of the recorded protocols confirmed these findings.

In the second stage, we were able to find meaningful differences in the performance of the students of the two groups as they searched for the errors. The analysis allowed us to conclude that the findings of the first stage were applicable to the entire population.

Quantitative research methods were then used to compare the performance of students in the two environments in a large scale experiment. We compared the performance of two groups on the matriculation examination, which is a laboratory examination and thus quite appropriate for assessing the impact of the environment on the students' learning. The examination grade was not simply a single number, but included detailed assessments of the students' knowledge of the working environment, understanding of the computer operation, facility with assembly language and programming skills. Students who used the EasyCPU environment achieved significantly higher grades in the matriculation examination than did students in the control group.





6 Conclusion

Our results so far have shown that the characteristics of a visual environment can significantly impact the learning style and the learning outcome. EasyCPU encourages a higher cognitive level of debugging skill. We believe that this research will provide evidence for the usefulness of visualization tools for teaching, in particular in the field of teaching computer architecture, which has been poorly empirically investigated.

Acknowledgements

I would like to thank my advisors Mordechai Ben-Ari and Tommy Dreyfus for their valuable help and support.

References

J. M. Anderson and T. L. Naps. A context for the assessment of algorithm visualization system as pedagogical tools. In *Program Visualization Workshop*, pages 121–130, Joensuu, Finland, 2001.

D. Foley. Microcode simulation in the computer architecture course. *SIGCSE Bulletin*, 24(3):57–59, 1992.

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.

M. L. Loui. The case of assembly language. *IEEE Transaction on Education*, 31(3):160–164, 1988.

M. Petre, B. Blackwell, and T. Green. Cognitive questions in software visualization. In Stasko et al. (1998), pages 453–480.

S. Simeonov and M. Scheinder. MISM: An improved microcode simulator. *SIGCSE Bulletin*, 27(2): 13–17, 1995.

J. Stasko, J. Domingue, M. H. Brown, and B.A. Price, editors. *Software Visualization: Programming as a Multimedia Experience*. MIT Press, Cambridge, MA, 1998.

J. Stasko and A. Lawrence. Empirically assessment algorithm animation as learning aids. In Stasko et al. (1998), pages 419–438.

C. Yehezkel, W. Yurcik, M. Pearson, and D. Armstrong. Three simulator tools for teaching computer architecture: EasyCPU, Little Man Computer, and RTLSim. *Journal of Visual Languages and Computing*, 1(4), 2001. http://portal.acm.org/.

VisuSniff: A Tool For The Visualization Of Network Traffic

Rainer Oechsle, Oliver Gronz and Markus Schüler
University of Applied Sciences, Trier, Postbox 1826, D-54208 Trier

oechsle@informatik.fh-trier.de

1 Introduction

Computer networks are a well established topic in computer science education. The curriculum comprises the protocol layering and data encapsulation principles in general, and the Internet protocols like TCP, UDP, IP, DNS, HTTP as specific examples. For better understanding the protocol implementations of the lower layers within the operating system kernel and distributed application software, a visualization of all the exchanged network data is of major importance. At the application level one may consider standard client-server applications (DNS Name Service or the World Wide Web, e.g.), or the students' self-written client-server applications using sockets, RMI (Remote Method Invocation) or CORBA.

In this paper we present VisuSniff, a tool which captures all the data which is sent or received by a network device (usually an Ethernet adapter), and which visualizes the captured data in a well structured and understandable way. The outline of this paper is as follows: In section 2 we discuss related work. In section 3, the main part of the paper, we present VisuSniff from a user's point of view. Finally, in section 4, we summarize our work and report on ongoing work.

2 Related work

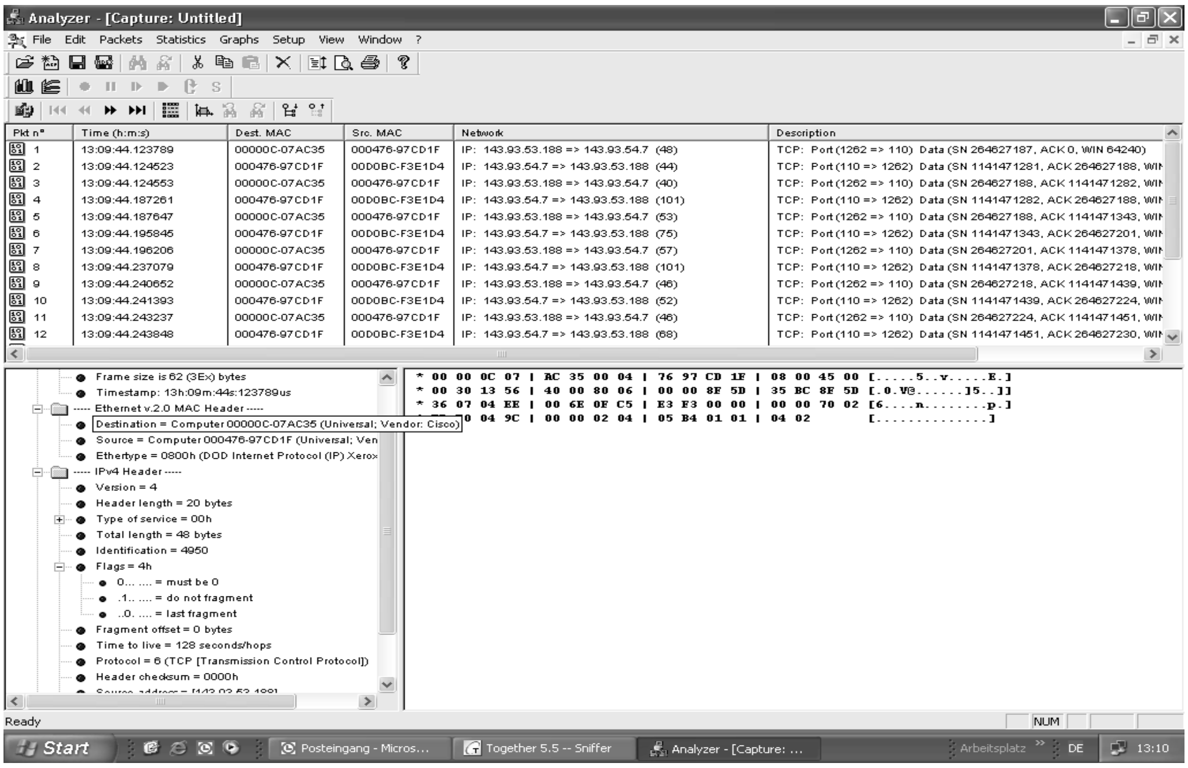
There are several tools available for capturing and showing network traffic. These tools can be classified as command-line tools and tools with a graphical user interface (gui).

2.1 Command-Line traffic capturing tools

TCPDUMP and WINDUMP are the most famous command-line tools for capturing network traffic (Netgroup-WinDump). The captured data is written to standard output and thus purely text-based. This way of displaying information is tedious and annoying for non-expert users; they usually are lost in captured data space. Because there are several applications concurrently running on a host, there is typically a huge amount of data captured even within a short period of time. To reduce the amount of data, a filter expression can be given. However, indicating appropriate filter expressions is only possible for network experts, but not for novices like students. There is a risk that the filter is too fine such that important and interesting data is filtered out und thus not displayed. If the filter is too coarse, the poor users are still flooded by the displayed data.

2.2 Traffic capturing tools with graphical user interfaces

There are different types of network capturing tools with graphical user interfaces (we consider Analyzer (Netgroup-Analyzer) as one of the best of these tools, see Fig. 1). Some of them present details of the captured network packet headers, some of them display only the application level data like transmitted web pages or emails, others focus on statistical data. Although all these programs have a graphical user interface, the information usually is not shown graphically, but textually. Especially, there is little support for navigating through the huge amount of captured data. Usually, all the captured packets are shown in chronological order. It is not possible to group the packets according to different communication relationships (e.g. all packets belonging to a single TCP connection). In addition, it is usually not possible to display several packets at the same time (for example, a request and the corresponding response). These tools are not suited for network novices.



```
Pkt n°  Time (h:m:s)      Dest. MAC       Src. MAC        Network                                    Description
1       13:09:44.123789   00000C-07AC35   000476-97CD1F   IP:  143.93.53.188 => 143.93.54.7  (48)    TCP:  Port (1262 => 110)  Data (SN 264627187, ACK 0, WIN 64240)
2       13:09:44.124523   000476-97CD1F   00D0BC-F3E1D4   IP:  143.93.54.7 => 143.93.53.188  (44)    TCP:  Port (110 => 1262)  Data (SN 1141471281, ACK 264627188, WIN
3       13:09:44.124553   00000C-07AC35   000476-97CD1F   IP:  143.93.53.188 => 143.93.54.7  (40)    TCP:  Port (1262 => 110)  Data (SN 264627188, ACK 1141471282, WIN
4       13:09:44.187261   000476-97CD1F   00D0BC-F3E1D4   IP:  143.93.54.7 => 143.93.53.188  (101)   TCP:  Port (110 => 1262)  Data (SN 1141471282, ACK 264627188, WIN
5       13:09:44.187647   00000C-07AC35   000476-97CD1F   IP:  143.93.53.188 => 143.93.54.7  (53)    TCP:  Port (1262 => 110)  Data (SN 264627188, ACK 1141471343, WIN
6       13:09:44.195845   000476-97CD1F   00D0BC-F3E1D4   IP:  143.93.54.7 => 143.93.53.188  (75)    TCP:  Port (110 => 1262)  Data (SN 1141471343, ACK 264627201, WIN
7       13:09:44.196206   00000C-07AC35   000476-97CD1F   IP:  143.93.53.188 => 143.93.54.7  (57)    TCP:  Port (1262 => 110)  Data (SN 264627201, ACK 1141471378, WIN
8       13:09:44.237079   000476-97CD1F   00D0BC-F3E1D4   IP:  143.93.54.7 => 143.93.53.188  (101)   TCP:  Port (110 => 1262)  Data (SN 1141471378, ACK 264627218, WIN
9       13:09:44.240652   00000C-07AC35   000476-97CD1F   IP:  143.93.53.188 => 143.93.54.7  (46)    TCP:  Port (1262 => 110)  Data (SN 264627218, ACK 1141471439, WIN
10      13:09:44.241393   000476-97CD1F   00D0BC-F3E1D4   IP:  143.93.54.7 => 143.93.53.188  (52)    TCP:  Port (110 => 1262)  Data (SN 1141471439, ACK 264627224, WIN
11      13:09:44.243237   00000C-07AC35   000476-97CD1F   IP:  143.93.53.188 => 143.93.54.7  (46)    TCP:  Port (1262 => 110)  Data (SN 264627224, ACK 1141471451, WIN
12      13:09:44.243848   000476-97CD1F   00D0BC-F3E1D4   IP:  143.93.54.7 => 143.93.53.188  (68)    TCP:  Port (110 => 1262)  Data (SN 1141471451, ACK 264627230, WIN
```

```
    ● Frame size is 62 (3Ex) bytes
    ● Timestamp: 13h:09m:44s:123789us
⊟ ----- Ethernet v.2.0 MAC Header -----
    ● Destination = Computer 00000C-07AC35 (Universal; Vendor: Cisco)
    ● Source = Computer 000476-97CD1F (Universal; Ven
    ● Ethertype = 0800h (DOD Internet Protocol (IP) Xerox
⊟ ----- IPv4 Header -----
    ● Version = 4
    ● Header length = 20 bytes
  ⊞ ● Type of service = 00h
    ● Total length = 48 bytes
    ● Identification = 4950
  ⊟ ● Flags = 4h
        ● 0... .... = must be 0
        ● .1.. .... = do not fragment
        ● ..0. .... = last fragment
    ● Fragment offset = 0 bytes
    ● Time to live = 128 seconds/hops
    ● Protocol = 6 (TCP [Transmission Control Protocol])
    ● Header checksum = 0000h
```

```
* 00 00 0C 07 | AC 35 00 04 | 76 97 CD 1F | 08 00 45 00 [.....5..v.....E.]
* 00 30 13 56 | 40 00 80 06 | 00 00 8F 5D | 35 BC 8F 5D [.0.V@......]5..]]
* 36 07 04 EE | 00 6E 0F C5 | E3 F3 00 00 | 00 00 70 02 [6....n........p.]
  0 04 9C | 00 00 02 04 | 05 B4 01 01 | 04 02       [..............]
```

Figure 1: Screen shot of Analyser

3 VisuSniff seen from a user's perspective

Because of the shortcomings of existing network sniffing tools, we decided to develop our own network sniffer now called VisuSniff. The main goal of VisuSniff is the presentation of captured network traffic that is especially suited for computer science education. We strongly believe that the best way to achieve our goal is the presentation of the information in a graphical form. After data has been captured, an overview diagram shows all captured communication relations. By clicking on specific items in the overview diagram (hosts, ports, communication relations), the interesting information can be selected easily. A sequence diagram (like in UML) then visualizes all exchanged data packets chronologically for the selected items. After clicking on an interaction arrow in the sequence diagram, a visual representation of the exchanged data is shown. The data is presented in a way which is typically used in lectures and text books. In the following we show a typical usage of our tool in more detail. We consider the sending and receiving of email.

3.1 Selecting The Data Sources

First of all, the source of the data to be visualized has to be selected by the user. There are two possibilities: A new capturing process can be started (on-line mode) or a file can be opened which contains data that was captured previously (off-line mode). The off-line mode enables instructors to present captured data in a lecture room without being on-line or without having to rely on a specific interesting network situation. If on-line mode has been selected, all network adapters of the computer are listed. The user has to select the adapter from which data is going to be captured. In addition, the file has to be selected into which all captured data is stored. Although not recommended for novice users, a TCPDUMP / WINDUMP filter expression can be given (host 143.93.53.7 and (udp or tcp), e.g.). After having started the capturing process, the programs whose transmitted and received data is going to be captured are used (sending and receiving email with the user's favorite mail program, e.g.). During the capturing process, a dialog window shows the elapsed time and the number of captured packets. The capturing process is then stopped by the user. For all captured packets ViusSniff resolves all source and destination IP addresses to DNS names and stores these names together with the captured data in a file. Thus, at a later point in time, when VisuSniff is used in off-line mode, it is possible to display host names even if the computer is not connected to a network and therefore not able to contact a DNS server for the resolution of IP addresses into host names. If VisuSniff is used in off-line mode, a file containing captured network traffic has to be selected as data source. The file may have been written by VisuSniff, or by TCPDUMP or WINDUMP, because VisuSniff uses the same de-facto standard format called PCAP for writing the captured data like TCPDUMP and WINDUMP do.

3.2 Selecting the interesting data

After the data source has been selected, an overview diagram displays all hosts that appear in the captured data as sender or receiver, and their communication relations (see Fig. 2). This diagram gives

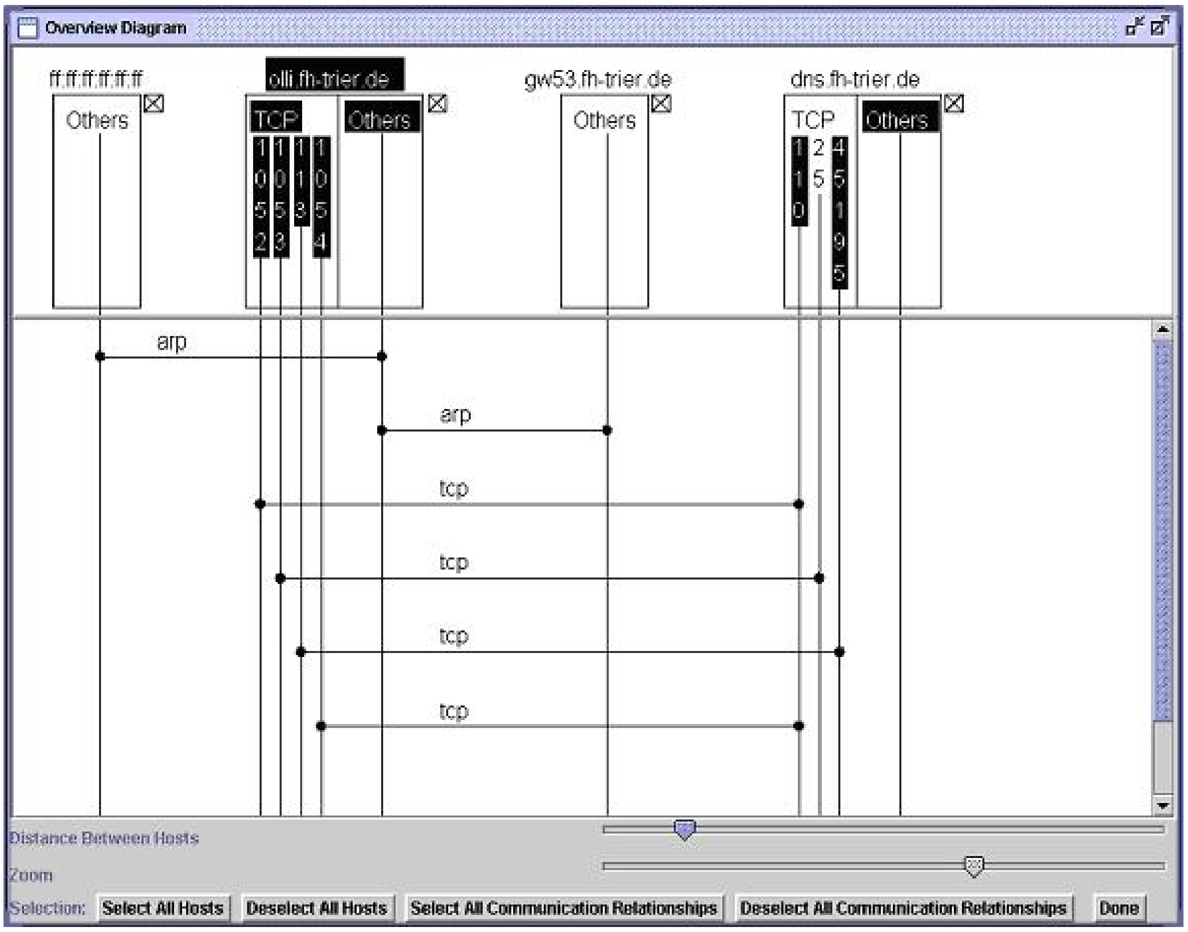


Figure 2: Overview diagram: hosts, ports, and their communication relationships (selected items inverse).

a clear overview of what has been captured. The hosts are located in the upper part of the diagram. The hosts are labelled by their DNS name. If the name is not known, the IP address is shown. If even the IP address is not known, the MAC address is used instead. For each host all used TCP and UDP ports are displayed (for layout reasons, the digits of the port numbers are drawn one below the other). Other protocols (ARP, ICMP, IGMP) for which there is no corresponding port number, are summarized as Others beside the TCP and UDP port numbers.

Broadcast and multicast addresses which appear only as destinations, never as sources, are represented as hosts as well. An alternative would be to draw a line from the sender to all receivers.

Unfortunately, it is not possible to know the set of receivers. Therefore, we decided to represent broadcast and multicast addresses as virtual hosts.

The lower part of the overview diagram summarizes all packets of the same protocol type (ARP, IGMP, ICMP) with the same source and destination address as a single link. Communication relationships for UDP and TCP are further differentiated by port numbers. In Figure 2 it can be seen that host olli.fh-trier.de sent one ore more ARP messages to all hosts connected to his local network (the MAC address ff:ff:ff:ff:ff:ff is the MAC broadcast address). The host olli.fh-trier.de is the one which captured the data. In addition, four TCP connections between olli.fh-trier.de and dns.fh-trier.de have been captured (dns.fh-trier.de is olli's DNS and email server), for instance from port 1052 on olli to port 110 on dns for receiving email via the POP3 protocol, and from port 1053 on olli to port 25 on dns for sending email via the SMTP protocol.

When the overview diagram is scrolled vertically, the host representations in the upper part remain at their fixed positions. The two sliders enable the user to change the size of the overview diagram. A small-sized diagram allows a better overview, a large-sized diagram may be needed for a beamer presentation within a lecture.

It is possible to select or deselect all hostsor some of the hosts completely, all UDP or all TCP ports of some hosts, some UDP or TCP ports, all other protocols of some hosts, or some communication relationships. Selected items are displayed in inverse. If a host is selected completely, all packets that this host has sent or received are displayed in the following diagram (see section 3.3). If a port is selected, then all packets with this port as source or destination port are displayed, even if the corresponding peer port has not been selected. In Figure 2 host olli.fh-trier.de has been selected completely, but only ports 110 and 45195 and Others from host dns.fh-trier.de have been selected.

Often the overview diagram contains hosts in which the user has no interest at all. The user can eliminate these hosts from the overview diagram by clicking into the small box with an X to the right of each host representation. After removing a host, all other ports or hosts which have no communication relationships any more, are removed, too. This enables users to simplify still confusing overview diagrams step by step.

3.3 Displaying the selected network traffic

After having finished the selection, a window containing a sequence diagram (message flow diagram) is opened which contains the selected hosts and ports and their corresponding peers (see Fig. 3). The representation of the hosts and ports is the same as in the overview diagram. But instead of showing summarized communication relationships, there is an arrow for each captured data packet. In the beginning, only a single arrow representing the first packet is displayed. Each time the button labelled + is pressed, the next packet is added to the diagram. By pressing the button labelled -, the last added packet can be removed again. The possibility to add packets step by step can be used in lectures. Instead of seeing all packets at once, the instructor can ask the students what will be the next packet. The information which is shown as label of the arrows can be configured by the users. For each protocol type the shown information can be separately selected or deselected. For TCP, e.g., the TCP flags, the sequence and acknowledgement numbers can be used as arrow labels.

In Figure 3 we see all the items selected in Figure 2 as well as the partners with which there is a communication relationship. Packet no. 1 ist an ARP request from host olli.fh-trier.de sent to the broadcast address in order to resolve the next router's IP address into a MAC address. Packet no. 2 is the ARP response from the router gw53.fh-trier.de. Packets no. 3, 4, and 5 represent a TCP connection setup to port 110 of host dns.fh-trier.de (3-way handshake). In packet no.3 the SYN flag is set. The server replies a packet with the SYN and ACK flags set. Packet no. 5 is an acknowledgement to packet no. 4. It follows (not visible in Figure 3) the authentication by the transmission of userid and password from the client to the server. The server tells the client the number and size of all the emails. The emails are then sent from the server to the client and the POP3 connection is closed. Sending emails from the client to the server via SMTP over a newly established TCP connection is very similar.

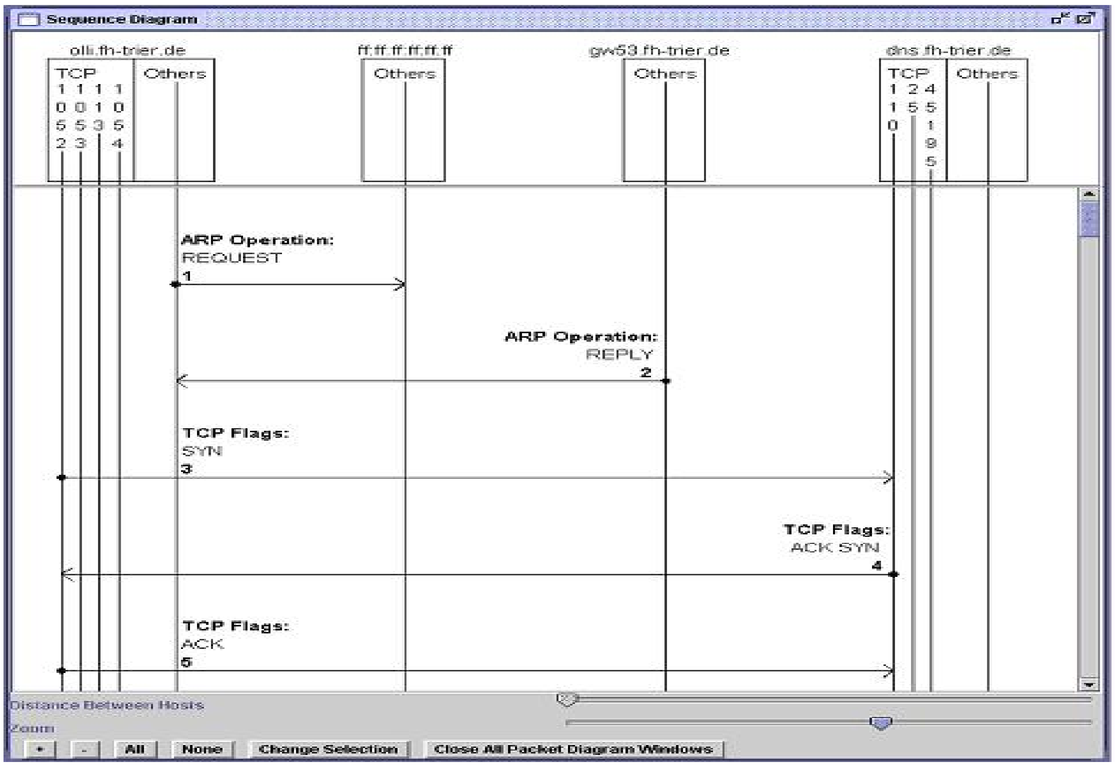


Figure 3: Sequence diagram

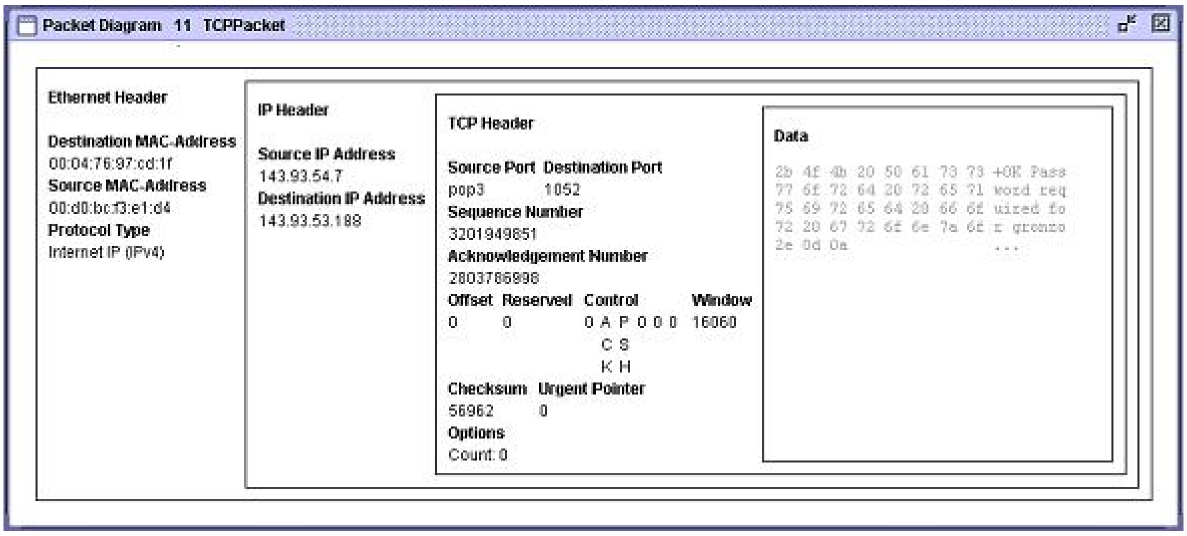
3.4 Displaying packets

Clicking on an arrow in the sequence diagram results in the opening of a new window which shows detailed information for the selected packet (see Fig. 4). The user can open many such windows and is thus able to compare easliy the details of different packets. The packet representation again has a layout which is usually used in lectures and computer networks text books. The representation emphasizes the encapsulation of data by the different protocol layers. In Figure 4 the application data is encapsulated within a TCP segment which is encapsulated within an IP packet which is encapsulated within an Ethernet frame. After the opening of a packet window only the most important information is shown. Clicking repeatedly into a header results in displaying additional information in a step-by-step manner until finally the header is totally expanded and all header information is visible.

Figure 4 shows packet no. 11 from the sequence diagram of Figure 3. This packet was sent from the mail server to the mail client. The application data is shown in hex code and - if possible - in ASCII characters. The application data of packet no. 11 can easily be interpreted because the used protocol POP3 is a pure ASCII protocol. The mail server is telling the client that it accepted the client's userid and is requesting now the client's password (which is transmitted in the next packet no. 12 as plain ASCII text). The TCP header is fully expanded which means that all TCP header information is shown. Well-known ports are displayed as service names (pop3) instead of numbers (110). Similarly the protocol type in the ethernet frame header is shown as text (Internet IP) instead of hex code (0x0080).

3.5 Implementation Of VisuSniff

VisuSniff is based on our version of JPCAP (S. Bitteker), an open source Java class library for capturing network traffic. We had to modify JPCAP for several reasons which we will not discuss here, because we would have to describe many details. JPCAP is written in Java and C. Our part of VisuSniff is pure Java. The main architecture of VisuSniff follows the MVC design pattern (model - view - controller).



```
Packet Diagram  11  TCPPacket

Ethernet Header
Destination MAC-Address
00:04:76:97:cd:1f
Source MAC-Address
00:d0:bc:f3:e1:d4
Protocol Type
Internet IP (IPv4)

IP Header
Source IP Address
143.93.54.7
Destination IP Address
143.93.53.188

TCP Header
Source Port  Destination Port
pop3         1052
Sequence Number
3201949851
Acknowledgement Number
2803786998
Offset  Reserved  Control        Window
0       0         0 A P 0 0 0    16060
                    C S
                    K H
Checksum  Urgent Pointer
56962     0
Options
Count: 0

Data
2b 4f 4b 20 50 61 73 73 +OK Pass
77 6f 72 64 20 72 65 71 word req
75 69 72 65 64 20 66 6f uired fo
72 20 67 72 6f 6e 7a 6f r gronzo
2e 0d 0a                ...
```

Figure 4: Packet diagram showing a TCP segment

4 Summary and further work

VisuSniff (Visual Sniffer) is a network traffic capturing and visualization tool. An overview diagram provides a good feeling of what has been captured by showing all captured communication relationships. It is possible to select interesting items (hosts, ports, communication relationships) by mouse clicks. A sequence diagram shows the next level of detail by drawing all exchanged packets for the selected items in chronological ordering. Each packet can be examined in all details. We are convinced that VisuSniff is a tool that displays captured network traffic in a much clearer and better understandable way - especially for network novices - than other protocol analyzers that we are aware of.

Up to now the development of VisuSniff was focussed on the lower layers (layers 2, 3, and 4). Application data is presented in hex code and ASCII code. Our main future work will be oriented towards a better representation of the application protocols and application data. Currently, our special focus is on Java RMI (Remote Method Invocation) which is an easy-to-use mechanism for the communication of distributed Java applications. Jini, e.g., is based on RMI. We are planning to display the transmitted arguments and return values in a graphical form as object diagrams like in the JAVAVIS system (R. Oechsle, 2002). JAVAVIS is a visualization tool which shows the dynamic behavior of a running Java program by displaying UML object diagrams and UML sequence diagrams. The JAVAVIS system uses the Java Debug Interface (JDI), so there are no modifications needed in the Java source code for the extraction of information. To achieve our goal for the visualization of RMI arguments and return values, VisuSniff needs to analyze the JRMP protocol (Java Remote Method Protocol) (Sun-Microsystems-Inc). JRMP uses the RMI Wire Protocol and the Java serialization mechanism.

We believe that the following types of users will benefit from VisuSniff:

1. Students: Students will better understand the lower layer protocol software and distributed application software when VisuSniff is used in lectures, labs, or for self-studies.

2. Instructors: Often, the descriptions of protocols in text books is not precise enough, valid only for certain cases, and sometimes even wrong. Instructors may use VisuSniff for verifying their knowledge or for exploring scenarios that they do not know yet.

3. Network Administrators: Network administrators may use VisuSniff for a better understanding what is happening in their local area networks. This may lead to measures that improve the performance, the responsiveness, or the security of the network.

4. Software Developers: Software developers may use VisuSniff in the future for the understanding and debugging of distributed Java RMI applications.

5 Acknowledgments

We like to thank Nikolaus-Koch-Stiftung, Trier, for funding the VisuSniff project.

References

Netgroup-Analyzer. A public domain protocol analyzer. URL `http://netgroup-serv.polito.it/analyzer`.

Netgroup-WinDump. Tcpdump for windows. URL `http://netgroup-serv.polito.it/windump`.

T. Schmitt R. Oechsle. Javavis: Automatic program visualization with object and sequence diagrams using the java debug interface (jdi). *Stephan Diehl (ed.): Software Visualization*, pages 176–190, 2002. URL `http://www.springer.de`.

J. Haddad S. Bitteker, P. Charles. Network packet capture facility for java. URL `http://sourceforge.net/projects/jpcap`.

Sun-Microsystems-Inc. Java remote method invocation specification. URL `http://java.sun.com/j2se/1.3/docs/guide/rmi`.

Towards Improved Individual Support in Algorithm Visualization

Guido Rößling
Department of Computer Science, Darmstadt University of Technology, D-64283 Darmstadt, Germany

Thomas L. Naps
University of Wisconsin, Oshkosh, Oshkosh, WI, USA 54901

roessling@acm.org
naps@uwosh.edu

1 Introduction

Algorithm visualization (AV) has seen an increasing amount of interest in the past years, as evidenced by the growing number of publications, as well as the formation of the biannual International Program Visualization Workshop. However, in the context of education, AV has still failed to make a breakthrough regarding the effective support for learner understanding. Systematic assessments of the effect of AV have often not shown a significant improvement in learner understanding (Stasko and Lawrence, 1998).

Why is this so and what can be done about it? We believe that the reasons are essentially twofold. First, most of today's AV systems do not support several pedagogical requirements. However, the required features are essential for deeper understanding of the learners, or simply for making the system attractive to use for learner and educator. Second, most AV systems are too impersonal in their interaction with the learner, and thus do not provide as much learner motivation as might be possible.

We want to address these issues with a prototypical AV system that addresses pedagogical requirements and also pushes AV into the direction of personalized instruction and intelligent tutoring. The proposed system combines two existing systems that have already been adapted to support the pedagogical requirements.

The paper is structured as follows. In Section 2, we present the set of pedagogical requirements that shall be offered by a personalized and effective AV system. Section 3 adapts the requirements to the usage context of personalized instruction. Section 4 illustrates the solution approach we plan to take during the summer of 2002. Section 5 concludes the paper and outlines areas of future research.

2 Pedagogical AV Requirements

Rößling and Naps (2002) summarize key findings of current research projects in AV use. They outline nine different pedagogical requirements for algorithm visualization systems:

1. use *applications* instead of applets to overcome the restrictions in expressiveness due to the dependency on the browser's Java virtual machine support,

2. prefer *general-purpose systems* over topic-specific systems due to the chance for reuse and better integration into a given course (Bazik et al., 1998),

3. allow users to *provide input* to the algorithm, using appropriate input helpers,

4. support for visualization *rewinding* so that users can return to the place where they lost track of the content (Anderson and Naps, 2001; Stasko and Lawrence, 1998),

5. offer a view of the visualization's *structure* that can also be used to jump to associated visualization steps (Gloor, 1998),

6. *interactive prediction* of the algorithm's behavior (Naps, 2001),

7. integration with a *database* for course management facilities (Jarc et al., 2000),

8. inclusion of hypertext *documentation* that explains the visual display (Stasko and Lawrence, 1998),

9. offering the user the choice between *smooth* visualization transitions and *discrete* steps. A break between consecutive steps, or at least a pause button, shall also be provided (Anderson and Naps, 2001).

This list incorporates several research papers on AV systems. The goal of the list is to provide a testbed for the effectiveness of AV used in education. However, for making the transition to improvied individual support systems, the requirements given are not sufficient. How can we make this transition possible? We will examine this question in the next section.

3 Towards Improved Individual Support

To bring AV into the realm of improved individual support, additional steps have to be taken. The AV system has to offer the features requested for pedagogical reasons, as outlined in the previous section. In addition, it also has to offer advanced functionality that goes beyond the standard expectations as follows.

Input data specification. The degree to which the learner controls the input of data to the algorithm being visualized will vary greatly from algorithm to algorithm. In some cases all data input may occur before actual execution (and visualization) of the algorithm begins. This is the case, for example, for a student viewing an animation of a graph algorithm such as a shortest path or minimal spanning tree. The graph itself and specifications for vertices critical to the particular run are all entered prior to execution of the algorithm. The geometric parameters critical to the visualization may then be determined and the algorithm executed with further input from the user limited to interactive prediction (see below). On the other hand, for an algorithm that is tracking the growth of a data structure as data arrives for insertion (for example binary search trees and their derivatives), having the viewer specify all data to be inserted before the algorithm executes is less than ideal. Here we want the viewer to be able to see a partially grown structure and react to that by entering a value that may cause a particularly interesting event, for example a rotation in the case of a balanced tree.

Visualization display, navigation and control. To enhance the learner's control over the visualization display and navigation, the AV system has to be able to perform a smooth reverse visualization of effects. This enhances the ability of the learner to see the difference that a single transformation makes by being able to switch between the current and previous state in arbitrary order. Seeing the transition performed smoothly in forward direction but then only having it jump back to the previous state in reverse mode is insufficient. It places the burden of detecting the difference between steps on the learner, not on the system.

Additionally, learners shall be able to adapt the display to their current environment. This includes the choice of display background color to account for diverse lighting situations, transition speed and display magnification. The latter two settings may be performed on a percentage scale that allows values both smaller and larger than 100%, as well as offering a facility for resetting the value to 100%. The speed setting depends on the perceptiveness of the current learner, as well as on the processing power of the underlying hardware. Similarly, the display magnification depends on the target platform. For example, if the content is presented by an educator over a beamer, a large scale has to be chosen to enable reading the display for learners sitting at a distance. On the other hand, notebook computers usually have display restrictions that may prevent the learner from seeing the whole content at one time unless the display is scaled down, for example to 71%.

Interactive prediction. Most interactive prediction approaches such as (Jarc et al., 2000; Naps et al., 2000; Rößling and Naps, 2002) focus on querying the learner's prediction and marking it as either correct or wrong. However, from a pedagogical point of view, receiving merely a reply of the type "Sorry, you are wrong" is not motivating. Additionally, learners may easily become bored if they are requested to perform essentially the same prediction multiple times. This is especially the case when they have already shown that they have understood the specific topic. Therefore, more advanced concepts have to be explored for intelligent tutoring.

First, a didactical feedback has to be incorporated in the question. Typical feedback shall include information that helps the learner understand the topic better. In the case of correct answers, this can be links to additional material, related topic areas or applications. For incorrect answers, the system can strive to provide hints on what the learner may have understood incorrectly and also hint toward the correct answer. Note that a comment that simply gives the correct answer is not helpful for most learners, as it provides neither a justification of why the shown answer is correct nor any insight into what the learner may have misunderstood.

Second, the AV author has to be able to group questions based on the question topic area. Once the current learner has sufficiently shown his or her understanding, the questions for this topic can be dropped to avoid boring students. The number of related questions to answer correctly for a specific topic has to be specified by the AV author.

Finally, the AV author may assign a certain number of points to each question and inform the learners on their progress. Of course, if questions are grouped and dropped after a sufficient number of correct answers, the learner must receive the same number of points for this group as if all questions had been answered.

External documentation. Anderson and Naps (2001) propose three different types of external documentation. Static documentation is unaware of the current visualization state and therefore does not change while the visualization is being displayed. Algorithm-sensitive documentation is aware of the current execution stage, but not of the specific values used in the visualization. Therefore, it cannot explicitly mention specific properties of the current execution. Accordingly, dynamic documentation is algorithm-sensitive and also aware of the values used. We also consider here a fourth type of documentation that could be termed *pedagogically dynamic*. This type of documentation is not only sensitive to the algorithm state and values being used but also to the degree of expertise being demonstrated by the student. More detailed documentation is presented for the viewer having problems understanding the algorithm - similar to the didactic feedback mechanism described under interactive prediction. From the viewpoint of improved individual support, pedagogically dynamic documentation holds the greatest promise for helping learners.

Reusable visualization modules. The generation of visualizations for a given topic becomes considerably easier if it can be structured in reusable modules. For example, most sorting algorithms can be structured into the following six parts: introduction to the algorithm, data acquisition, initialization, actual sorting, efficiency, and a statement of time complexity. The introduction to the algorithm – if included – presents a textual overview of how the algorithm works and can be defined once for each algorithm. It may also be placed in the external documentation, just as the time complexity statement and efficiency measurement. However, including it into the visualization increases the likelihood that learners will read and reflect on this information. The data acquisition may be reused by all sorting algorithms, as it covers the definition of the contents of the array to sort, common to nearly all sorting algorithms. These components may be presented as modules, provided that the number of steps actually performed for the current input values can be passed between visualization modules. The information may be presented textually or using other approaches, such as an efficiency gauge that acts as a meter counting the number of steps.

Support for reusable visualization modules makes the visualization generation easier for AV authors, as they can use established content and simply plug in their current visualization data. This is perhaps most valuable in the case of algorithms that manipulate non-linear data structures such as trees and graphs. Determining esthetically pleasing geometric layouts for such structures can be extremely time-consuming and laborious for animation designers. Modules that automate this process, shielding the animation designer from the geometric details, greatly enhance the energy that can be devoted to more pedagogic goals such as thought-provoking interactive prediction. However, module support also requires an AV system that is able to load multiple visualization files and merge them into a single visualization. It must also be able to substitute variables with their concrete values, for example for the number of steps performed. Note that counting the number of steps performed is not the same as determining the number of calculation steps, as some steps may be expanded over multiple steps for explanation purposes, or compressed to avoid tiresome repetitions. Additionally, a general-purpose AV system does not have enough context knowledge to be able to detect and count the instructions underlying the visualization.

Other possible avenues for making AV more accessible and helpful for both learners and educators are discussed in a working group held at the ITiCSE 2002 conference. We will strive to incorporate the results of this working group into our work.

4 Our Solution Approach

We propose to realize the ambitious goals presented in the previous section by combining the expressiveness of two existing AV systems, JHAVÉ (Naps et al., 2000) and ANIMAL (Rößling and Freisleben, 2002). The main reasons for this decision are as follows:

- The combination of JHAVE and ANIMAL has already been shown to be highly expressive and helpful for addressing the pedagogical requirements in Section 2 (Rößling and Naps, 2002).
- ANIMAL acting as the visualization front-end for JHAVÉ supports extensibility in various areas. This was documented by adapting Animal to JHAVÉ's stop-and-think quizzes (Rößling and Naps, 2002).
- ANIMAL incorporates the extensible scripting language ANIMALSCRIPT (Rößling and Freisleben, 2001) which can be used as the base point for implementing the features stated in Section 3. Scripting languages are relatively easy to use for generating visualization content, and may support the integration of separate modules well.

Currently, the combination of the two systems does not yet support the list of features we require. Due to the distance between the project partners, both geographically and regarding the time difference, we are planning to implement the functionality during the summer break of our respective universities. We plan to achieve the goals we have set us as follows.

Input data specification. It is convenient to think of the algorithm and the renderer of the algorithm's state as being in a client-server relationship. The algorithm itself is the server. As the algorithm executes, it provides information about its state to the renderer/client, which in turn knows how to parse that information and display it in appropriate graphic fashion. In the current design of JHAVÉ and ANIMAL, the JHAVÉ server receives information from the client about input for the algorithm and then executes the algorithm to completion, after which it relays the script to be rendered to the remote client. If input were to be provided by the client after the algorithm begins to execute but before the algorithm terminates, the server would have to suspend the execution of the algorithm at key steps, waiting for input from the client. This would necessitate the server's being able to remember states between its interactions with the client. Two ways of preserving this state information are to use Java servlets or the Remote Method Invocation (RMI) API.

Visualization display, navigation and control. Our current system already incorporates both smooth transitions and discrete steps at the learner's discretion. Additionally, it is also one of the few systems that provide both rewinding and smooth reverse execution. This is highly unusual for general-purpose AV systems, as evidenced by the statement that "efficient rewind is one of the most important 'open questions' in AV" (Anderson and Naps, 2001). All other requested features, including adjustment of the background color, display size and speed, are also addressed.

Interactive prediction. Our prototype currently cannot provide any of the advanced features required in Section 3. However, we plan to provide an extended ANIMALSCRIPT syntax that allows the incorporation of (optional) didactical comments to each prediction query. Other requirements such as grouping and scoring questions can also be embedded in the underlying scripting language.

External documentation. Pedagogically dynamic documentation requires resolving a set of obstacles similar to that described under our discussion of input data specification. As the server generates the documentation, it must remember the state of the viewer's understanding of the algorithm as the algorithm progresses, for example using servlets or RMI.

Reusable visualization modules. The newest release of ANIMALSCRIPT includes support for module loading and variable passing. Visualization modules that contain an explanation of the complexity of the sorting algorithm can also give the number of comparisons and assignments for the chosen data. Advanced display options, such as an efficiency gauge, are still left for development.

5 Conclusions

In this paper, we have discussed several pedagogical requirements for AV systems and how they have to be modified for bringing AV into the realm of improved individual support systems. The increased level of personal interaction with the system shall raise the learner's motivation. Additionally, the features incorporated into the system such as the structural view and adaptive documentation shall improve learner retention.

The current prototype we have implemented can already address all pedagogical requirements. Further work is needed for also addressing the requirements for intelligent tutoring. As outlined in Section 4, we have already determined our development strategy, and plan to set this into action during summer 2003. Further research is needed for analyzing the benefits gained from the prototype once the implementation is finished.

References

Jay Martin Anderson and Thomas L. Naps. A Context for the Assessment of Algorithm Visualization System as Pedagogical Tools. *First International Program Visualization Workshop, Porvoo, Finland. University of Joensuu Press*, pages 121–130, July 2001.

John Bazik, Roberto Tamassia, Steven P. Reiss, and Andries van Dam. Software Visualization in Teaching at Brown University. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization*, chapter 25, pages 382–398. MIT Press, 1998.

Peter A. Gloor. User Interface Issues for Algorithm Animation. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization*, chapter 11, pages 145–152. MIT Press, 1998.

Duane Jarc, Michael B. Feldman, and Rachelle S. Heller. Assessing the Benefits of Interactive Prediction Using Web-based Algorithm Animation Courseware. 31st *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2000), Austin, Texas*, pages 377–381, March 2000.

Thomas Naps, James Eagan, and Laura Norton. JHAVÉ: An Environment to Actively Engage Students in Web-based Algorithm Visualizations. 31st *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2000), Austin, Texas*, pages 109–113, March 2000.

Thomas L. Naps. Incorporating Algorithm Visualization into Educational Theory: A Challenge for the Future. *Informatik / Informatique, Special Issue on Visualization of Software*, pages 17–21, April 2001.

Guido Rößling and Bernd Freisleben. ANIMALSCRIPT: An Extensible Scripting Language for Algorithm Animation. 32nd *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2001), Charlotte, North Carolina*, pages 70–74, February 2001.

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

Guido Rößling and Thomas L. Naps. A Testbed for Pedagogical Requirements in Algorithm Visualizations. 7th *Annual ACM SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education (ITiCSE 2002), Århus, Denmark*, pages 96–100, June 2002.

John Stasko and Andrea Lawrence. Empirically Assessing Algorithm Animations as Learning Aids. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization*, chapter 28, pages 419–438. MIT Press, 1998.

The Algorithms Studio Project

C. D. Hundhausen
Laboratory for Interactive Learning Technologies, University of Hawai'i, Honolulu, HI 96822 USA

hundhaus@hawaii.edu

Abstract

I introduce the idea of the "algorithms studio," a novel approach to integrating algorithm visualization technology into undergraduate courses on algorithms. I then present a five-year research project, funded by the National Science Foundation, for developing and empirically evaluating the idea. Finally, I report on the current status of the research.

Interactive, Animated Hypertextbooks for the Web: Ongoing Work

M.T. Grinder
Computer Science Department, Montana Tech of the University of Montana, Butte, Montana, USA

`mgrinder@mtech.edu`

R.J. Ross
Computer Science Department, Montana State University, Bozeman, Montana, USA

`ross@cs.montana.edu`

1 Introduction

A hypertextbook project has been underway in the Webworks Laboratory at Montana State University for a number of years now. This project arose as a natural extension of work already being done by the project director in animated visualization systems for computer science education. The idea of integrating educational visualization systems into a hypertextbook that would augment, or indeed supplant, traditional textbooks presented itself as an appealing possible solution to one of the most perplexing problems facing educational visualization researchers: the lack of actual use of visualization systems in the classroom. A convincing argument was made that if a comprehensive, seamless teaching and learning resource could be constructed (i.e., a hypertextbook) that became the primary teaching and learning resource for a particular class, students and instructors alike would quite naturally use all aspects of the resource, including any embedded visualizations.

This paper outlines some of the key points learned during our work on two hypertextbooks for the web, one on the theory of computation and one on biofilm engineering. Although online courses and electronic books that incorporate some visualizations have been around for a while, no other efforts that we know of are as comprehensive as the work we are doing, particularly with respect to active learning visualization systems. In the remainder of this paper, we outline some of the major issues facing hypertextbook authors.

2 Hypertextbook Design Issues

In this section we briefly highlight the major issues we have encountered in our work on hypertextbooks that feature active learning visualizations. The intent is not to discuss these issues in detail here, but to provide a framework for understanding the unique problems faced by the hypertextbook author. The following subsections identify and briefly outline the issues.

2.1 Platform

On which computer platform should a hypertextbook run? We have chosen to design our hypertextbooks for the web. This has the appealing advantage of platform independence. Of course, platform independence is a somewhat elusive objective, especially when one is working with the latest versions of the Java Virtual Machine (JVM), which not all browsers for all platforms implement at the same time. One is also restricted to features (e.g., HTML) that the standard browsers can process. Finally, the wars between Microsoft and the rest of the world conspire to impede progress on some important platform independence issues.

2.2 Java Applets for Active Learning Visualization Modules

Once the web is chosen as the platform for delivery of a hypertextbook, one can incorporate active learning visualization models into the hypertextbook in the form of Java applets. For our hypertextbook projects we currently have the following active learning visualization models for use as standalone applications or for seamless integration as applets in computer science hypertextbooks:

- A program animator (which runs Pascal programs in visual fashion)
- A finite state automaton animator
- A regular expression animator
- A context free grammar animator
- An LL(1) table construction animator

For use in hypertextbooks relating to biofilm engineering, we have:

- A fugacity model animation
- A biofilm accumulation model animation

These applets all have a number of features that encourage active learning on the part of users. For example, the finite state automaton applet mentioned above allows students to construct their own automata from scratch (or to modify existing ones), run those automata on arbitrary inputs, and then compare their results with hidden, correct automata. Feedback is provided to aid the learner in correcting detected mistakes. Active learning features like these contribute to the success of the applets as learning tools.

2.3 Seamless Integration of Visualization Components

One of the primary issues we have been forced to confront in the design of visualizations for use in a hypertextbook is the necessity of integrating visualization applets seamlessly into the fabric of the hypertextbook. We have discovered that in order for visualizations to be useful and used in a hypertextbook, they must appear as a naturally integrated part of the whole. It is distracting, confusing, and counterproductive to the goal of encouraging students and instructors to use visualizations if these visualizations must be opened in windows outside the normal flow of the text. It isn't clear in such situations, for example, when to close a newly opened window or how to return to the main textbook presentation. Furthermore, if the presentation of visualizations becomes too disjointed through the opening and closing of auxiliary windows, the visualizations might simply be ignored.

We have tackled this issue by constructing the visualization applets so that they can be embedded directly within the text of the hypertextbook at arbitrary locations as preconfigured examples for students to follow or as preconfigured exercises that students must complete. These same applets can also be included as standalone models in an appendix for arbitrary student and instructor use.

2.4 Cooperation Between Visualizations

Not only must visualizations be interwoven seamlessly into the fabric of a hypertextbook, but they often must also be constructed to interact with each other. For example, in the hypertextbook on the theory of computing that we are building, the finite state automaton applet, the regular expression applet, and the context free grammar applet must all be made to work with each other, so that, for example, animations can be included that demonstrate conversion algorithms from one form to another (e.g., from finite state automata to regular expressions).

2.5 Applet Authoring Tools

It is important to provide a prospective hypertextbook author with tools for configuring and incorporating applets into a hypertextbook as desired. For example, if an exercise that requires a learner to construct a finite state automaton is to be included in a hypertextbook at a certain point, the author must be able to configure the finite state automaton applet to be invoked at that point with

- a description of the exercise

- a blank work area
- a correct finite state automaton that is hidden from the student (for comparison with the version constructed by the student)

An author must not be required to be cognizant of the implementation details of any particular applet in order to embed that applet in appropriate ways and places in a hypertextbook.

2.6 Representation and Presentation Independence of Visualizations

One key issue that took us a while to uncover is that to be most effective and adaptable, visualizations designed for the web must have two components that are independent of each other. The first component is the *internal representation* of the object being visualized, and the second is the *visualization* of that object. That is, the internal representation of the object should not dictate how the object is to be visualized. For example, a description of a finite state automaton should be given in a form that provides all of the components necessary to define the automaton in generic fashion. The visualization applet, then, must use this generic definition to present the automaton in an appropriate visual form (e.g., as a directed labeled graph, or as a table). This independence of representation from presentation provides great flexibility in the forms a graphical display of an object can take. It also makes cooperation between different visualization applets much easier to accomplish (e.g., in demonstrating the conversion of a regular expression to a finite state automaton) for obvious reasons.

We are taking this idea of independence of representation from presentation one step further by requiring that all of the internal representations of objects for visualization have a standard definition in the eXtensible Markup Language (XML). This provides the added bonus of having an internal description for visualization objects that is in a standard form for the web.

2.7 Other Considerations

There are a number of issues other than the incorporation of visualization applets into a hypertextbook of which a prospective author needs to be aware. A few are listed below:

- The hypertextbook should have an attractive, inviting portal (cover).
- The hypertextbook should appear to the learner as a single, integrated component, not a collection of disorganized pieces.
- The hypertextbook should have a polished, professional look and feel that gives the learner confidence in its use. A consistent design theme and a consistent appearance to all of the pages and the linking structure are important.
- The use of distracting “bells and whistles” should be avoided.
- An organization of the material by way of hyperlinks that leads learners through the hypertextbook in different ways based on different learning levels or styles should be provided.
- Audio and video should be incorporated where it would be effective.

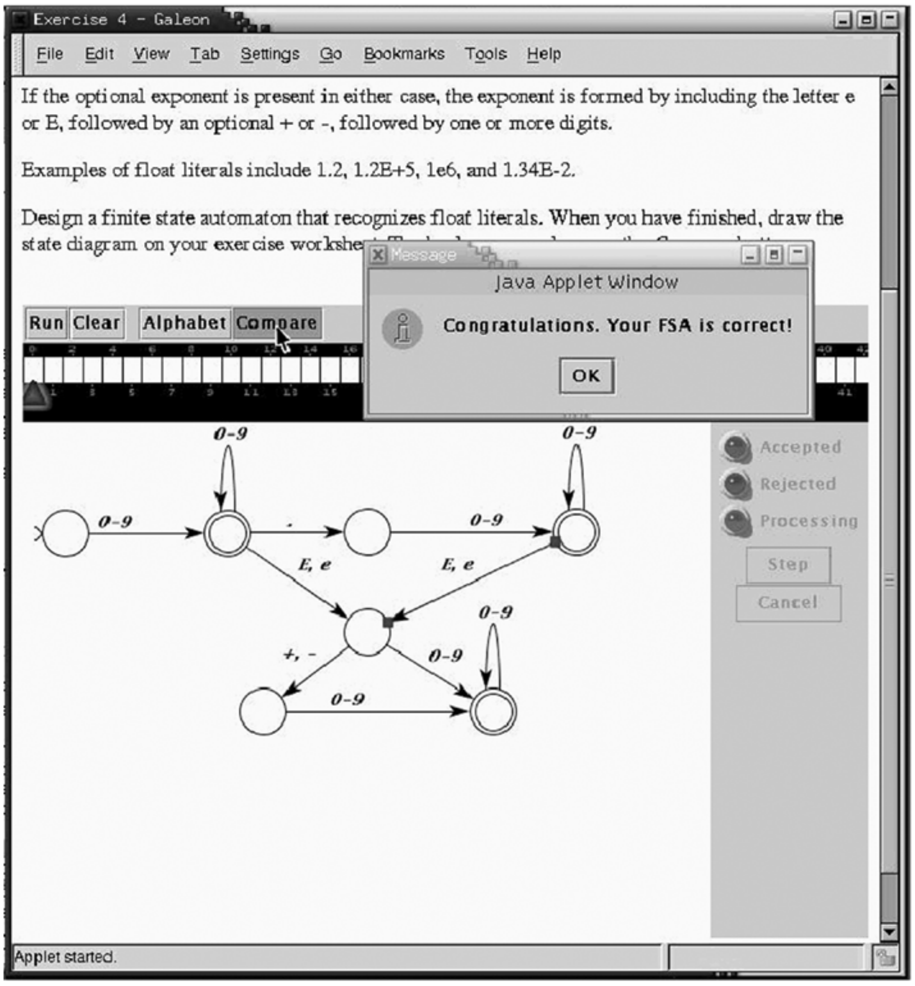
In other words, a hypertextbook should have a professional look and should make use of the unique capabilities offered through hyperlinking to reach out to students with varying learning needs.

3 Problems

No endeavor this large is without problems. One large hurdle is the amount of work needed to write a hypertextbook. The effort entailed in writing a traditional textbook is plenty already. The construction of a hypertextbook requires the additional work of providing an organization that caters to different learning needs, of (possibly) recording and integrating audio and video clips appropriately, and of designing and including effective active learning applets. This additional work is substantial. Consider just the development of visualization applets. Even passive learning applets, in which the learner participates only as a viewer, are time-consuming to design. Now consider further the effort involved when such applets must be constructed to support active learning, to be adaptable to various situations, to be integrated into a hypertextbook, to interact with different but related applets, and to come supplied with authoring tools. It should be clear that such applets require substantial effort to design and implement well.

4 An Example

Let's consider an example. The figure below is taken from a mockup of a portion of a hypertextbook on the theory of computing.



If the optional exponent is present in either case, the exponent is formed by including the letter e or E, followed by an optional + or -, followed by one or more digits.

Examples of float literals include 1.2, 1.2E+5, 1e6, and 1.34E-2.

Design a finite state automaton that recognizes float literals. When you have finished, draw the state diagram on your exercise workshe

Congratulations. Your FSA is correct!

Even though the figure is small, there are a number of points to notice.

- The window that frames the entire figure is a standard browser window—the learner in this example has accessed the hypertextbook simply by linking to the hypertextbook through a browser in platform independent fashion.
- There is text at the top of the window and a finite state automaton applet at the bottom. The text includes a description of an exercise that is to be carried out with the finite state automaton applet. The finite state automaton applet is embedded directly in the text where it should naturally appear and is not required to be opened in a separate window.
- The finite state automaton applet is a fully self-contained visualization system for finite state automata.

  - There is a menu bar at the beginning of the applet with menu items for running a finite state automaton, clearing the finite state automaton pane (to allow construction of a finite state automaton from scratch), selecting an alphabet over which a constructed finite state automaton is to operate, and comparing a constructed finite state automaton against a correct solution.
  - There is a tape for input (the linear sequence of cells below the menu and above the finite state automaton pane). The learner can type an arbitrary string onto this tape and then run the automaton in the pane below on that tape.
  - There is a large pane (the one that displays the automaton of this example) in which a finite state automaton (either deterministic or nondeterministic) can be constructed and modified quite easily through a set of mouse clicks and drags.
  - There is a set of "processing indicators" to the right of the automaton pane that display the status of the finite state automaton as it is being run on an input string (including acceptance or rejection of the input string).

- The learner in this example has apparently constructed a finite state automaton in the automaton pane in completing the exercise described in the text in which the applet is embedded. To check whether the constructed solution is correct, the learner has also apparently clicked the Compare button on the menu. The applet has returned a message saying that the automaton is correct. Had the automaton not been correct the message returned would have stated this fact and provided a sample string that the learner's automaton accepts that is not in the language of the exercise, or a string that the learner's automaton does not accept that is in the language of the exercise, thus providing guidance to help lead the learner to a correct solution.

The above observations from just this one example should help one gain an appreciation for just how much time must be invested in designing good active learning applets for inclusion in a hypertextbook. Way back in 1974 in *The Mythical Man Month* Fred Brooks observed that if one were to design a software system for personal use that took time N to complete, then if that same system were to be designed for use by others, or if it were to be designed to be integrated into some other existing system it would take time $3N$ to do. If instead the original system were to be designed to be used by others *and* to be included in some other existing system it would take time $9N$ to complete. We have certainly informally validated this observation during our attempts to design software like the finite state automaton applet illustrated above, which is expressly intended to be used by novices and which also must be integrated into a hypertextbook and be able to interact seamlessly with other applets, such as a regular expression applet and a regular grammar applet.

The time factor for constructing useful applets is further increased when one wants to incorporate active learning. The integration of guidance and feedback mechanisms is a challenging task. In the illustration above, for example, the determination of whether a learner has constructed a finite state automaton that meets the specifications of an exercise requires the following:

- The person designing the exercise must provide a correct finite state automaton for the exercise that is hidden from the learner and which the applet must be able to maintain somewhere.

- The learner's attempt at creating a correct finite state automaton must be compared with the correct, hidden finite state automaton. The comparison cannot be done in the applet by comparing the two automata, because there are in general an infinite number of finite state automata recognizing any given language, so somehow the applet must determine whether the *languages* recognized by the two automata are the same.

- Using known results from the theory of computing, a finite state automaton M_1 is be constructed from the learner's automaton such that M_1 recognizes the language that is the intersection of the language recognized by the learner's automaton, M_L, and the *complement* of the language recognized by the correct automaton, M_C. If this new automaton, M_1, accepts any strings at all, this means that the learner's automaton, M_L, accepts some strings that the correct automaton M_C does not accept. Examples of such strings can be found by doing a depth-first search on the graph for M_1, which can be presented to the learner for feedback. Similarly, a finite state automaton M_2 can be constructed that recognizes the language that is the intersection of the *complement* of the language recognized by the learner's automaton, M_L, and the language recognized by the correct automaton, M_C. If M_2 accepts any strings at all, then there are strings that the correct automaton accepts that the learner's automaton does not accept. Again, a depth first search on the automaton M_2 will reveal examples of such strings for feedback to the user. If both M_1 and M_2 recognize the empty language, then the learner has constructed a proper automaton, that is, one that recognizes the same language as the hidden correct automaton.

Since active learning and feedback are important objectives of applets used for visualizing educational concepts, it should be clear from this description that the applets are going to necessarily be more complex and time-consuming to construct than similar passive learning applets.

5 Evaluation

Do hypertextbooks work? Phrased this way, the question is perhaps too broad. More specifically, we want to know the answers to some of the following questions:

- Do students learn better in traditional learning environments with hypertextbooks that incorporate active learning modules?
- Do students learn faster in traditional learning environments with such hypertextbooks?
- Can students learn better and/or faster in non-traditional learning environments (e.g., in distance learning situations) when hypertextbooks that incorporate active learning modules are available for the subject?
- Do students perceive that their opportunities to learn are enhanced when such hypertextbooks are used?
- Do students enjoy learning more when such hypertextbooks are used as opposed to traditional textbooks?

We would be happy to determine, for example, that students learned at least as well with hypertextbooks as with traditional textbooks if at the same time they were more satisfied with the learning process. This might encourage better learning and even entice more underrepresented classes of students (e.g., women) to complete a degree in computer science.

Our hypertextbook projects have only recently begun to undergo preliminary evaluation. The thesis cited in the References section below describes some of these evaluations.

6 Summary

In this paper, we have presented an outline of our investigations into the feasibility and effectiveness of hypertextbooks, particularly those that incorporate active learning visualization applets. Our ongoing work can be followed at

`http://www.cs.montana.edu/webworks`

7 References

Rather than providing a small list of references that would fit in this paper, we encourage readers to obtain a copy of the thesis *Active Learning Animations for the Theory of Computation* by the first author. The thesis contains a much more detailed look at the construction of active learning applets and includes an extensive list of references. The thesis can be obtained online at

```
http://www.cs.montana.edu/webworks
```

and at

```
http://www.cs.mtech/~grinder
```

The authors can also be contacted directly for information about the hypertextbook projects.

Author index