Software Visualization Generation Using ANIMALSCRIPT

Guido Rößling and Bernd Freisleben

Many different software visualization tools are available to the public. Most, however, only address specific application areas, forcing users to shift between tools. The visualization tool ANIMAL, with its built-in visualization language ANIMALSCRIPT, is an algorithm animation tool including a graphical editor that can be used for arbitrary software visualizations. This article introduces the basics of ANIMAL and ANIMALSCRIPT as well as some special features including internationalization support. ANIMALSCRIPT is also easy to adapt and extend with more advanced features.

Keywords: Software Visualization, Algorithm Animation, Internationalization Issues, Dynamic Placement, Dynamic Extension.

1 Introduction

The increasing popularity of algorithm or program visualization has led to the development of several tools, most of which are designed to address a certain field of visualization. The advantage of implementing such specific tools is that they may be highly specialized within their field and thus may offer great field-specific features. However, they are usually not usable outside their specific field. For example, a tool geared towards visualizing run-length encoding algorithms such as *RLE* [Khuri/Hsu 00] is highly useful in its field, but not usable for other topics. One of the areas that shows growing interest in visualizations is academic education. Here, a fair number of teachers are interested in using generated algorithm animations in their lectures instead of having to perform the operations themselves on the blackboard. However, if the tool used for the visualization is highly specialised, they will have to shift between using the tool and the normal mode of presentation. This becomes even more relevant in conferences, where presenters who want to show the usage of such a tool have to change between the visualization tool and the tool used for the presentation, such as *PowerPoint*™ .

Having a single tool that is generic enough to handle both the presentation and the content display saves the time needed to move between tools. More importantly, it also prevents the presenter from having to learn how to use a number of different tools. However, the tool must be sufficiently generic that it can handle the typical requirements of presentations as well as to support context-specific operations. It is unlikely that a single tool could offer built-in support for all the types of software visualizations that users might want. Therefore, it may be beneficial to have a tool that can be dynamically extended by users to address their specific needs without forcing them to read and modify large portions of the underlying code. This article presents the scripting language ANIMALSCRIPT that extends the capabilities of the ANIMAL visualization system [Rößling et al. 00]. ANIMALSCRIPT offers the basic operations required for software visualization, but can easily be adapted to address other context-specific requirements. For example, ANIMALSCRIPT supports relative object placement and addresses internationalization issues.

The article is organized as follows. Section 2 presents some tools specific to a certain usage area. Section 3 gives a short overview of the underlying visualization concept of ANIMAL and how visualizations can be generated. Section 4 presents the capabilities of ANIMALSCRIPT including several screen shots of the system. Some possibilities for adapting and extending the capabilities of both ANIMAL and ANIMALSCRIPT are outlined in section 5. Section 6 concludes the paper and outlines areas for further research.

2 Classification of Available Tools

Due to the increasing popularity of software visualization, many systems have been introduced during the last years. Since we cannot cover all available tools, we will focus on some tools that are representative of their categories. The tools can be divided into different categories based on the type of

Guido Rößling received the diploma degree in Computer Science in 1996 from the Darmstadt University of Technology. He has been a full-time research assistant at the University of Siegen since 1996 and from 1997, he has given lectures on Computer Science and Programming at the Berufsakademie Mannheim. He is currently working on his Ph.D. thesis which describes a dynamically extensible and configurable software visualization tool. His research interests include the effective use of multimedia in teaching, algorithm animation and automatic generation of data representations. He is a member of ACM, the IEEE Computer Society, AACE and GI. <roessling@acm.org>

Bernd Freisleben is associate professor of computer science in the department of Electrical Engineering and Computer Science at the University of Siegen. He received mis Master's degree in computer science from the Pennsylvania State University, USA, in 1981, and the Ph.D. degree in Computer Science from Darmstadt University of Technology, in 1985. His research interests include distributed systems, network computing, and multimedia in education.

input required to generate a visualization: mouse actions within a GUI, source code, method calls in a specialized method library, or scripting language commands.

Classic presentation tools such as *PowerPoint*™ from Microsoft or Sun's *StarImpress*™ typically expect the user to provide all contents of the visualization manually within a graphical user interface (GUI). The advantage of this approach is that the user has complete control over all elements, at least within the scope of what the tool can do. However, being forced to do everything manually also means that the generation cannot be automated. If a second visualization of the same topic but with changed values is required, the user basically has to start again from scratch. Furthermore, presentation tools are often more geared towards generating stunning visual displays for short presentations than addressing the needs of presenting the semantics of dynamic processes such as algorithms or data structures. Thus, they often offer a fair selection of ways in which new elements can be added, but may be limited in how elements that are already visible may be modified. Furthermore, the underlying concept of the presentation is a sequence of slides. This means that the next slide will initially be empty and thus not reflect the contents of the last slide – which is unfortunate for visualizing algorithms that perform several operations spread over a number of "slides". While presentation tools are very useful for presentations, they are thus less useful for presenting the behaviour of systems or algorithms.

Source-code based visualization systems such as *DDD* [Zeller 00], *Jeliot* [Haajanen et al. 97] / *Jeliot 2000* [Ben-Ari 00] and *WinHIPE* [Naharro-Berrocal et al. 00] automatically present a visualization of the underlying source-code. This also means that it is very easy to automatically generate visualizations: the user only needs to provide the source code, or modify parameter values according to the specific requirements. However, a number of restrictions apply to these systems. Firstly, the programming language that can be used is fixed by the system. For example, *DDD* addresses machine-code programs whose input typically will come from *C* or *C++*, while *Jeliot* and *Jeliot 2000* work on Java source code and *WinHIPE* is restricted to functional programming. If the programming language used in the presentation context is not supported by the tool, the tool cannot be used. Due to the automatic generation of the visualization, the user will also have only a limited ability to change the display. For example, it is unlikely that the system will be able to deduce the semantically most appropriate representation of a given data structure from the underlying code, e.g. a stack implemented using a linked list might not be represented as a stack, but as a standard list. Finally, presenters might wish to add explanatory text, or take a more abstract view of the underlying algorithm (such as a pseudo-code implementation) to avoid confusing the viewers with implementation details. These types of tools do typically not support both operations. Finally, if the topic to be presented is not available as an algorithm, the tools cannot be used at all.

Tools such as Silicon Graphic's *JAL* [Silicon Graphics 98] rely on method calls within a specific programming library (API) for generating the visualization. The ease of use of the library, as well as the extent to which operations are supported, depends on the state of the library. Two effective limitations are clear: (1) the user has to be sufficiently proficient in programming to write a program that correctly invokes the library methods and (2) the language in which the library is implemented must match the language used for the underlying code. If both are true then visualization APIs can be very useful; otherwise, the library is inappropriate and thus useless for the target application.

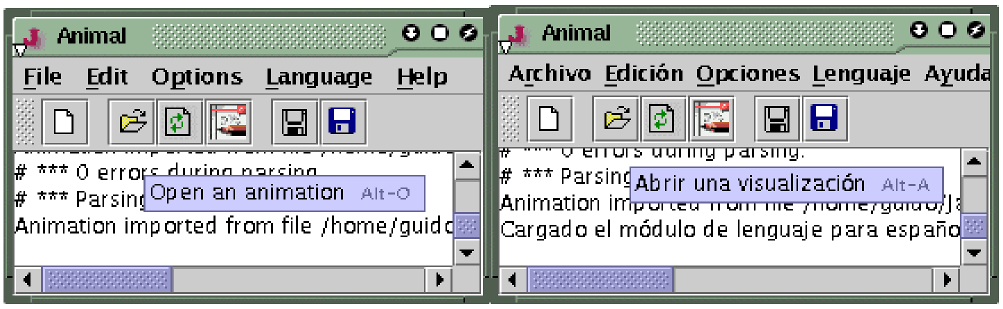
Systems that expect input in a specific format such as a scripting language offer a partial solution to the problem of inappropriateness. The tool does not care how the scripting input is generated – be it manually by the user, or automatically by the underlying code. Thus, the user does not have to be proficient in programming. However, algorithms implemented in any language may be modified so that they generate the appropriate scripting commands. Due to this fact, tools that work on scripting input have become popular over the last few years: *JSamba* [Stasko 98], *JAWAA* [Pierson/Rodger 98] and highly specific tools such as *JellRep* and *JFlap* [Hung/Rodger 97] for compiler construction topics. The scripting language underlying *JSamba* is kept generic, while most other tools offer a scripting language geared towards a specific context. For example, *JAWAA* is geared for algorithm and data structures, thus offering elements such as trees and graphs. However, it is less useful for presentations, as it does not support elements such as item lists. Additionally, going backwards one or more steps in the visualization is often not supported by the tools.

Users should generally choose the type of tool that is most appropriate to their intentions. They should reflect on the characteristics and limitations of each approach, especially if they wish to reuse the tool in other, less specific applications. Scripting-based tools offer a reasonable compromise between ease of generation and automation support. If the scripting language is not highly specialized, they also offer easy reuse in other contexts.

In the following, we will examine the scripting language ANIMALSCRIPT. As ANIMALSCRIPT is embedded in a visualization system, we will first take a look at the central features of this system.

3 The ANIMAL Visualization System

The ANIMAL visualization system [Rößling et al. 00] treats a visualization as a sequence of discrete steps. Each step may contain an arbitrary number of transformations. Each transformation in turn works on an arbitrary number of objects at the same time. ANIMAL offers the basic graphic primitives point, polyline / polygon, arc and text. As a concession to its initial use for introductory computer science visualizations, it also offers list element primitives. Each of these primitives contains a number of user-adjustable properties. Some properties are common to all primitives, for example, colour and depth information to resolve display conflicts caused by overlapping elements. Other properties depend on the given primitive. For example, a polygon object may have a fill colour, while this has no semantic meaning for a polyline. On the other hand, polylines may have a pointer at the beginning or the end,


Animal
File Edit Options Language Help
Open an animation Alt-O
Animal
Archivo Edición Opciones Lenguaje Ayuda
Abrir una visualización Alt-A

```
# *** 0 errors during parsing
# *** Parsing
Animation imported from file /home/guido
```

```
# *** 0 errors during parsing.
# *** Parsing
Animation imported from file /home/guido/J
Cargado el módulo de lenguaje para españo
```

Fig. 1: How translating ANIMAL's GUI affects titles, messages and keyboard short-cuts

which is not appropriate for polygons. Note that ANIMAL does not restrict the number of nodes a polyline can have.

ANIMAL offers transformations for changing the visibility state of arbitrary objects, rotating or moving objects and changing their colour properties. The extent to which these operations are supported depends on the type of objects they are executed on. For example, when moving a list element, users can choose between moving the element as it is, adjusting only one or more list pointers, or moving the element with fixed pointers. An arbitrary polyline or arc can be used as the object along which objects are to be moved, enabling complex move operations. Colour changes are applicable to all colour properties of a given object, so it is possible to have a polygon with a fill colour different from the outline colour. All operations can be scheduled by specifying a start time and a duration.

The individual visualization steps are usually chosen by using the video player-like buttons provided. These also include functionality for going backwards in the visualization. The author of a visualization may also specify that a given step will automatically advance to the next step after a certain amount of time has passed. The user may use a slide ruler displaying the percentage of the visualization already shown to adjust the current state. Finally, the author may provide labels to steps that are collected in a separate window and which can be used to jump to the associated step from any place in the visualization with a single mouse click.

In order to make ANIMAL usable for the widest possible target audience, we have followed the principle "keep it simple". ANIMAL offers a small graphical interface allowing users to edit any presentation using the mouse with standard features such as a grid and drag and drop support. The number of both graphic primitives and transformations has been kept intentionally small. Users who want a more powerful interface are expected to move to the built-in scripting language ANIMALSCRIPT once they have become familiar with using ANIMAL. The graphical interface and all output messages can be translated into another language by a single mouse click. At the moment, we can only provide German and English language support. Support for other languages such as Spanish, French and Italian requires the translation of a text file, without recompilation. Figure 1 shows ANIMAL's main window after the language has been changed. Note that the translation affects menus and their items, explanatory texts and keyboard short-cuts.

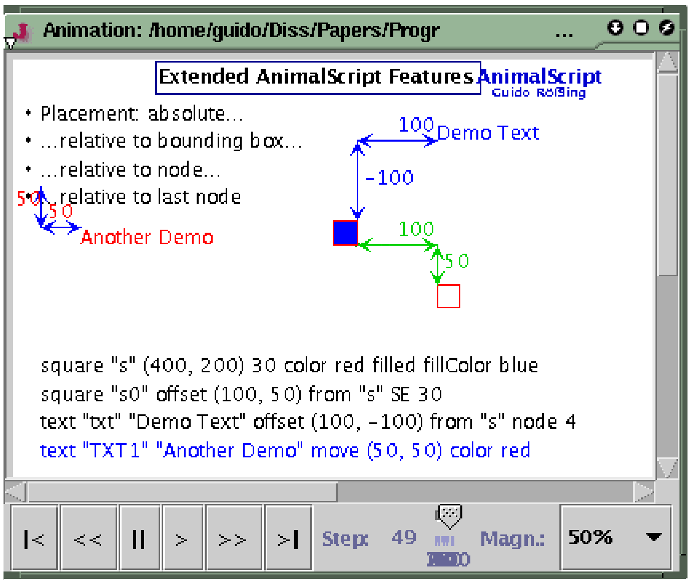
ANIMAL also offers an interface for import filters and provides a number of export filters. Currently, visualizations can be imported only from *JSamba* but be exported as a QuickTime movie, a variety of image formats and in XML format. Other formats are relatively straightforward to implement if the precise definition of the target format is available. All filters are registered in a configuration file that can be updated whenever a new filter becomes available. Adding new import or export filters does not require access to the base ANIMAL code, as no recompilation is necessary.

4 Selected Features of ANIMALSCRIPT

ANIMALSCRIPT supports a large number of features, so that we cannot cover all of them here. Therefore, we will start with a short introduction to ANIMALSCRIPT and then focus on three special features. These are relative object placement, internationalization issues and diagnostic output support. ANIMALSCRIPT contains commands for all the features of ANIMAL listed in the previous section. It also includes specific commands for common subtypes, such as squares instead of only generic polygons. Like several other scripting languages for visualization [Pierson/Rodger 98], [Stasko 98], ANIMALSCRIPT is line-based – each line contains exactly one command or comment. All graphic elements used during a visualization have to be given a name to allow their use in later transformations. These names can be exchanged using a swap command, making many transformations much easier for the user.

Some of the more advanced features of ANIMALSCRIPT go beyond the operations available within the graphical user interface of ANIMAL. For example, a single scripting command is sufficient to link or unlink list elements. Other special commands built into ANIMALSCRIPT support array data structures including pointers to array cells, overwriting of cell values and swapping the contents of two cells in the same array. To provide greater flexibility for source or pseudo code usage, a special code environment is also provided that handles indentation and offers highlighting for the current line of code or the context, such as the enclosing loop statement.

The main difference between ANIMALSCRIPT and the scripting languages used in other systems such as *JAWAA* and *JSamba*, however, lies in the areas of relative element placement, in-



• Placement: absolute...
• ...relative to bounding box...
• ...relative to node...
• ...relative to last node

```
square "s" (400, 200) 30 color red filled fillColor blue
square "s0" offset (100, 50) from "s" SE 30
text "txt" "Demo Text" offset (100, -100) from "s" node 4
text "TXT1" "Another Demo" move (50, 50) color red
```

Fig. 2: Object placement using AnimalScript: absolute coordinates, relative to bounding box or node, and as offset from the last node used

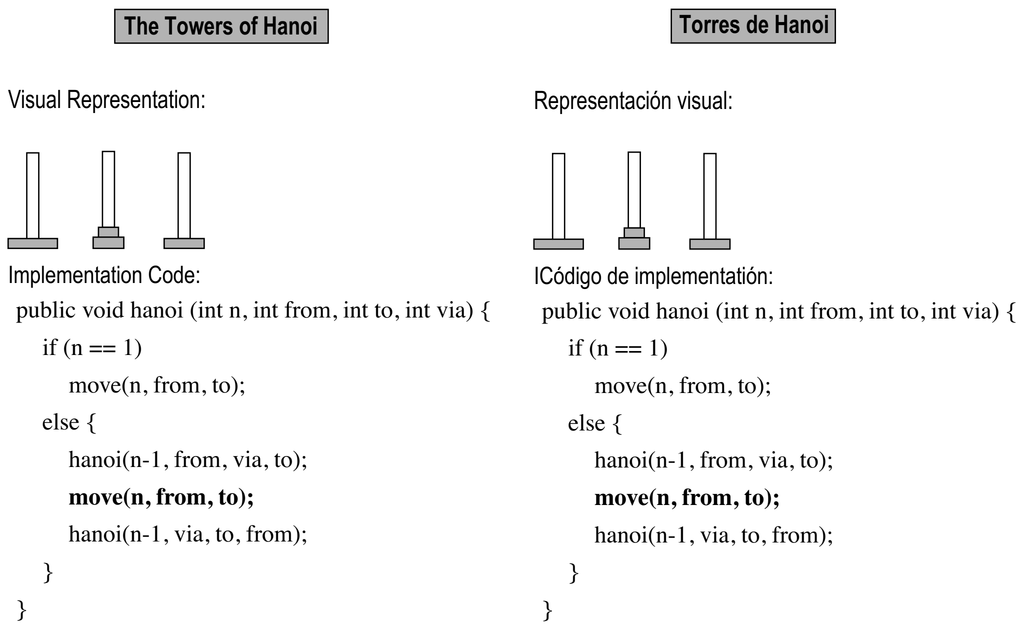
ternationalization and diagnostic output. The relative element placement of AnimalScript frees authors from keeping track of where each object is at any given point in time by allowing direct access to its bounding box, the smallest rectangle containing the whole element. Users have the following choices for specifying the coordinates of any element:

- absolute coordinates, such as (400, 200),
- placement relative to a corner of a previously defined object. Any of the eight compass needle points plus the centre of the object's bounding box can be used as the reference point,
- placement relative to an arbitrary node of a previously defined polyline or polygon,
- placement relative to the last defined coordinate, which is helpful for operations such as "go forward 20 pixels",
- or relative to a previously defined location that may have been defined using any of the techniques listed above.

Figure 2 shows these applications and the AnimalScript code required to produce them. Note that the filled square has been given absolute coordinates. The empty box's location is specified from an offset of the south east corner of the filled square and the text at the top right is placed relative to a node of the square. The last node used in the visualization was the top left corner of the text *"relative to last node"*, causing the text at the left centre of the screen to be placed relative to the text's top left corner.

As mentioned above, Animal offers a method to translate all entries of the graphical user interface. AnimalScript goes a step further by allowing the user to specify a translation for each text element. The languages supported in the visualization must be listed by language code at the start of the visualization file. Each text entry can then be given a translation prefixed by the language code. On loading the visualization, the user is prompted for the language to be used.

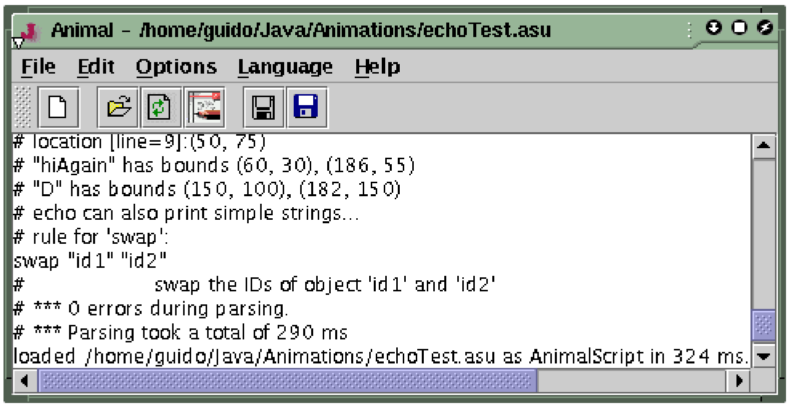
Note that the translation is highly unlikely to have the same length in pixels as the original element. For example, German texts tend to be longer than their English equivalent. Simply providing a translation is therefore not sufficient – elements that are supposed to be placed to the right of a translatable mes-



```
public void hanoi (int n, int from, int to, int via) {
    if (n == 1)
        move(n, from, to);
    else {
        hanoi(n-1, from, via, to);
        move(n, from, to);
        hanoi(n-1, via, to, from);
    }
}
```

```
public void hanoi (int n, int from, int to, int via) {
    if (n == 1)
        move(n, from, to);
    else {
        hanoi(n-1, from, via, to);
        move(n, from, to);
        hanoi(n-1, via, to, from);
    }
}
```

Fig. 3: Example visualization using the internationalization support of AnimalScript


Animal – /home/guido/Java/Animations/echoTest.asu
File Edit Options Language Help

```
# location [line=9]:(50, 75)
# "hiAgain" has bounds (60, 30), (186, 55)
# "D" has bounds (150, 100), (182, 150)
# echo can also print simple strings...
# rule for 'swap':
swap "id1" "id2"
#              swap the IDs of object 'id1' and 'id2'
# *** 0 errors during parsing.
# *** Parsing took a total of 290 ms
loaded /home/guido/Java/Animations/echoTest.asu as AnimalScript in 324 ms.
```

Fig. 4: Example applications of the echo command

sage would suddenly have an offset, or might overlap the message itself. By using relative placement aligned on the bounding box of the translated element, this problem can be solved. After the text has been read in and translated to the chosen language, its location and extent is known, so that elements and effects depending on its placement can be correctly resolved. One implication of this is that users who work in an international context are not forced to generate separate visualizations according to the target audience. Instead, they can embed the translations into the same visualization and choose the language according to the context they are in at any given time. Figure 3 shows an example visualization with an English and a Spanish version. Note the changed width of the header rectangle that uses relative coordinates to fit under the heading.

The diagnostic output support of ANIMALSCRIPT is very helpful, especially for novice users. The functionality of ANIMALSCRIPT's echo command includes the printing of the location (upper left corner) or the bounding box of objects, the numbers of the components of a grouped object or currently visible objects, arbitrary text comments, or the rule syntax for ANIMALSCRIPT commands. Some of these applications are shown in Figure 4.

5 Extending ANIMALSCRIPT

We believe that ANIMALSCRIPT already contains several features that prove beneficial for users and allow the tool to be used in a generic context. However, we are of course aware that there are many application areas for which specific support would be helpful. For example, there is currently no support for graph structures or theoretical applications such as Turing machine or finite state automaton simulation. As we cannot possibly hope to satisfy all user wishes, we have adapted the structure of how ANIMALSCRIPT is parsed for easy extensibility.

Users who want to provide a specific extension to ANIMALSCRIPT should first check whether there is already a third-party implementation available. Otherwise, we can provide them with a set of comparatively simple steps to follow for providing extensions to ANIMALSCRIPT. To make this task as easy as possible, we provide a parser that translates a special subtype of Backus-Naur rules into Java code for parsing the commands and extracting properties. The user will then only have to provide some parts of the coding, particularly the mapping of properties into a set of ANIMAL transformations and primitive objects.

ANIMALSCRIPT parsing components are loaded dynamically from a configuration file. Therefore, users who have either implemented or downloaded an extension need only update the configuration file. No change to the ANIMAL system or ANIMALSCRIPT itself is necessary and no recompilation is required. This is also true for the process of extending the functionality of ANIMALSCRIPT, which does not require modifications or recompilation of existing code. The sole exception to this rule concerns changes to already implemented features, such as bug fixes.

6 Conclusions

In this article, we have outlined selected features of ANIMAL and ANIMALSCRIPT. ANIMAL is useful for generic software visualizations and does not require special user knowledge or skills due to the graphical editor. ANIMALSCRIPT adds powerful functionality for users willing to tackle animation generation by scripting, as is common in some other systems [Pierson/Rodger 98], [Stasko 98]. ANIMALSCRIPT currently supports only a limited number of operations. However, these operations already include advanced features that are not commonly found in many comparable applications, such as relative object placement, internationalization support and diagnostic output. In order to address as large a client audience as possible, both ANIMAL's GUI and ANIMALSCRIPT visualizations can easily be translated into other languages, especially when used in conjunction with relative object placement. Authors simply provide the translations without having to adapt object locations.

We hope that we will find users willing to help us add new features to ANIMALSCRIPT once we have documented how this can be achieved. Most changes do not require modifying ANIMALSCRIPT code. Further work in extending ANIMALSCRIPT includes support for images, specific computer science data structures or displays and the typical features found in presentation tools. In our experience, generating a visualization by scripting is much faster than doing the same visually, once the user has become familiar with the underlying concepts and commands. This is even true for conference presentations, despite the fact that most presentation tools rely heavily on graphical editing. Despite the lack of explicit ANIMALSCRIPT commands for item lists, we have used ANIMAL in conjunction with ANIMALSCRIPT as the main tool for conference presentations since February 2000 without problems or complaints.

References

[Ben-Ari 00]
M. Ben-Ari: An Extended Experiment with Jeliot 2000. First International Program Visualization Workshop, Porvoo, Finland, 2000.

[Haajanen et al. 97]
J. Haajanen, M. Pesonius, E. Sutinen, J. Tarhio, T. Teräsvirta, P. Vanninnen: Animation of User Algorithms on the Web. IEEE Symposium on Visual Languages Proceedings (VL'97), Capri, Italy 1997, pp. 360–367.

[Hung/Rodger 97]
T. Hung, S. H. Rodger: Increasing Visualization and Interaction in the Automata Theory Course. 31st SIGCSE Technical Symposium on Computer Science Education, Austin, Texas 1997, pp. 6–10.

[Khuri/Hsu 00]
S. Khuri, H.-C. Hsu: Interactive Packages for Learning Image Compression Algorithms. 5th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2000), Helsinki, Finland 2000, pp. 73–76.

[Naharro-Berrocal et al. 00]
F. Naharro-Berrocal, C. Pareja-Flores, and J. Á. Velázquez-Iturbide, Automatic generation of algorithm animations in a programming environment, 30th ASEE/IEEE Frontiers in Education Conference, Volume II, Stipes Publishing L. L. C., 2000, pp. 6–12 (session S2C).

[Pierson/Rodger 98]
W. C. Pierson, S. Rodger: Web-based Animation of Data Structures Using JAWAA. 29th SIGCSE Technical Symposium on Computer Science Education, Atlanta, Georgia 1998, pp. 267–271.

[Rößling et al. 00]
G. Rößling, M. Schüler, B. Freisleben: The ANIMAL Algorithm Animation System. 5th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2000), Helsinki, Finland 2000, pp. 37–40.

[Silicon Graphics 98]
Silicon Graphics: JAL Algorithm Animation, 1998. Available online at http://reality.sgi.com/austern/java/demo/demo.html

[Stasko 98]
J. Stasko: Samba Algorithm Animation System, 1998. Available online at
http://www.cc.gatech.edu/gvu/softviz/algoanim/samba.html

[Zeller 00]
A. Zeller: Dynamic Display of Data Structures Using DDD. First International Program Visualization Workshop, Porvoo, Finland 2000.