Journal of Visual Languages and Computing (2002) **13,** 000–000
doi:10.1006/yjvlc.239 available online at http://www.idealibrary.com on IDEAL®

Animal: A System for Supporting Multiple Roles in Algorithm Animation

Guido Rössling[*] and Bernd Freisleben[†]

[*]*Department of Computer Science, Darmstadt University of Technology, Alexanderstraße 6, D-64283 Darmstadt, Germany, E-mail: roessling@acm.org and* [†] *Department of Electrical Engineering and Computer Science, University of Siegen, Hölderlinstr. 3, D-57068 Siegen, Germany, E-mail: freisleb@informatik.uni-siegen.de*

Many algorithm animation tools have been developed over the last years. The users of such tools can be separated into four roles: the original algorithm programmer, developers of the animation tool, visualizers that generate the animation and end users viewing the animation. Most tools focus on providing features for only one or at most two of these roles. The Animal system is designed to present valuable benefits for the last three roles. The principal research contributions of this work lie in dynamic extensibility, internationalization of GUI components and animation content, reversible animation display and flexible import and export facilities. We also present several core features of Animal including dynamic reconfiguration, internationalization in both GUI and animations, display scaling, export facilities and full video player controls. © 2002 Published by Elsevier Science Ltd.

1. Introduction

Algorithm animation (AA) is a subtopic of software visualization focusing on the dynamic visualization of the higher level abstractions which describe software [1]. Thus, it covers the dynamic display of actual implementation code, pseudo-code or other abstract views. The interest in algorithm animation has grown over the last years, as indicated by the growing number of publications, for example [2–5].

One of the first examples of algorithm animation is the film *Sorting out Sorting* [6]. Shown at many universities all over the world, the movie introduces nine different internal sorting methods, including an efficiency analysis. The movie is described in more detail in [7]. Since then, many algorithm animations tools have been developed. Most of these tools come from universities and are freely available. However, the interpretation of what algorithm animation is differs between the tools, and thus each provides a slightly different approach.

The standard book for the more general topic of software visualization [2] provides two different taxonomies for software visualization. The first taxonomy is based on a framework of six categories: *scope, content, form, method, interaction and effectiveness*. These categories are then further refined into subcategories with up to three levels [1]. The second taxonomy for classifying algorithm animation displays uses a three-dimensional grid with the axes *persistence, content and transformation type* [8].

1045-926X/02/$-see front matter © 2002 Published by Elsevier Science Ltd.

Price *et al*. [1] define four different roles in algorithm animation: *user, visualizer, software visualization software* developer (or simply *developer*) and *programmer*. The user views and interacts with an animation specified by the *visualize*r. The underlying animation system is designed and implemented by *developer*s. Finally, the *programmer* is the implementer of the visualized algorithm, for example, Quicksort. Note that in this context, the programmer may be unaware of animation plans by the visualizer. Therefore, most animation systems cannot provide services for the programmer role.

Each role has different expectations of algorithm animation systems. Users want to have a tool that runs smoothly, is easy to use and offers features such as video-player-like controls. Visualizers require comfortable and flexible animation generation, different ways of generating animations may be required to address personal preferences or experience levels. Developers may be interested in how easily the system can be updated or extended according to visualizer or user demands.

Since 1998, we have been working on an algorithm animation system called ANIMAL [9]. The system is geared to address these expectations by providing valuable services to the three roles concerned with algorithm animation. *Users* can freely adjust the magnification and display speed, export animations to other formats, and choose the language used in the graphical user interface and localized animations. The structuring of animations and the flexible display capabilities reduce the chance of getting lost. *Visualizers* can choose between three different approaches for generating animations: graphically in a GUI, by scripting, or using an API. They also profit from the flexible object placement features, internationalization support, precise timing of all actions and the generic graphical primitives and animation effects. *Developers* can implement additions that may be added or removed while the system is running. These additions are configurable on a directory basis, allowing for different views of the same tool in a shared environment. In addition, a new language for the GUI can be added without touching any system code.

The paper is organized as follows. Section 2 gives a short survey of representative algorithm animation systems. Section 3 describes the features of ANIMAL for the different roles. Section 4 presents our conclusions and areas of future research.

2. Related Work

Algorithm animation is used in a variety of contexts including education, source code debugging and presentation demands. Our review of the related systems focuses on their educational use. This also constitutes the background of our own work, which began as an implementation for an introductory computer science course. Over 150 software visualization prototypes and systems have been built over the last 20 years [1]. Instead of trying to cover all of them, we will illustrate how the different roles described in Section 1 are addressed by some representative tools usable in an education context.

Most animation systems offer a set of control features for the *user* role. Some tools such as Jsamba [10] allow the user to adjust the speed of the display. Most systems can run in a single-step mode. A slide show mode for a set of steps or the full animation is also common. Support for going backwards for a fixed number of steps as in *DDD* [11] is relatively uncommon. Most systems do not support this operation at all. There are some notable exceptions such as *ZStep* 95 [12] and *Leonardo* [13] which offer fully reversible execution.

Several tools focus on supporting the *visualizer* role. The approach taken for generating a visualization varies between tools. Dershem and Brummund [3] describe an approach for visualizing web-based sorting algorithms by decorating the algorithm with special method invocations. Several other tools including the well-known *POLKA, POLKA-RC* and *XTANGO* [14] provide special API methods for visualization. *Algorithma 99* [15] lets the user enter algorithms in a predefined pseudo-code language which can then be visualized.

Several systems including *JAWAA* [16] and *JSamba* [10] visualize input provided in a special command language. *JHAVÉ* [5] extends the language used in JSamba with interactive quiz questions and links to external documentation. Generic presentation tools such as *Microsoft Powerpoint*™ and *Sun StarImpress*™ are also often employed by instructors. Most often, algorithm animations are added to lecture slides. The GUI-based generation offered by presentation tools places the least demand on user skills, but at the same time also prevents automation. Furthermore, the lack of support for specific data structures such as lists makes animation generation both awkward and time-consuming.

Most published tools do not offer special support for the *developer* role. In many cases, the source code for the system is not available, preventing developers from adding new features. Even if the source code of the system is available, the user may be forced to figure out where additions have to be placed. A component-based system architecture as used, for example, in *Algorithma 99* [15] may make it much easier to add new elements. Pattern-based approaches as described by Nguyen and Wong [17] for sorting algorithms also support the developer. For example, the visualizer used in [17] can easily be replaced by another implementation.

By definition, the *programmer* role may be unaware of future plans for animation when implementing the algorithm. Rasala [18] describes how array algorithms can be automatically animated. This requires adapting the array to a template and using a special implementation that visualizes the operations. Thus, a small change to the code is required.

Several systems supporting the programmer role rely on debugger data, while others are actually debuggers with a graphical front-end that can be used for AA. Probably the most wide-spread representative of the latter type is the *GNU Data Display Debugger* (*DDD*) [11]. DDD is reported to have more than 250 000 users world-wide and acts as a front-end for command line debuggers such as GDB. DDD is typically used to locate bugs in programs, but also offers a selection of visualization features. Objects are displayed in boxes with references shown as pointers. When the underlying program stops, DDD retrieves the current values from the debugger and updates the display. This can be regarded as an algorithm animation. The standard display modes include a list display or a selection of numerical value plots. The plots can also be 'animated' by updating the display at breakpoints combined with a *continue* directive.

There are several similar algorithm animation tools based on interpreting source code, for example *ZStep 95* [12]. Used for functional programming, this tool also supports going back-wards in the display. The *KAMI* system [19] relies on debugger data and uses a 'paper-slide' display mode consisting of a set of overlapping slides illustrating nested method invocations. *Jeliot* [13] interprets Java source code and automatically generates a visualization. *Jeliot* 2000 [21] is a special re-implementation of *Jeliot* for high-school students. Algorithms can be edited and compiled within the tool and are animated including the evaluation of conditions. *Leonardo* [13] interprets *C* source code. The execution model also allows fine-grained control over the animation including reverse execution. Alas, Leonardo is only available for Macintosh computers.

Several systems are geared for specific contexts. Example contexts include formal languages [22], recursive methods [23], scheduling and page replacement algorithms [24], image compression [25], computer architecture models [26] or grading purposes [4, 27]. Some of these systems [24, 25] show the previous state next to the current state.

3. Animal Features

Most related systems cater for one of the four roles, with only few systems supporting two roles. We wanted our system to address most of the roles. We start with an example animation and then outline the core features ANIMAL offers for the different roles. The *programmer* is the only role we have to exclude. We are currently looking into a way to extract relevant data from a program without having to adapt the original code manually. However, this development is still in the early stages.

3.1. Example ANIMAL Animation

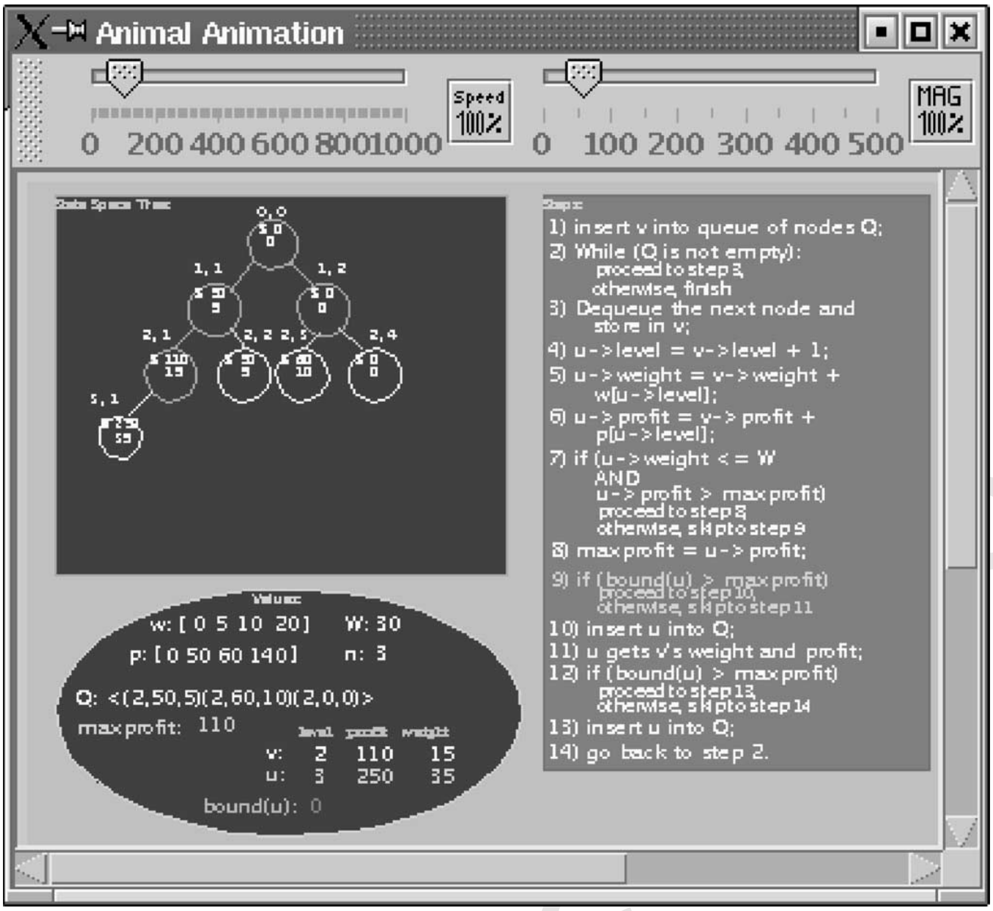
ANIMAL animations consist of a linked sequence of animation steps. An animation step can contain an arbitrary number of animation effects, each of which can work on several objects at the same time. The animation can be structured by labeling individual steps, with the label acting as a hyperlink in the display front-end [28]. The next animation step is shown when the user activates special GUI elements, or automatically after an optional delay specified by the visualizer.

Figure 1 shows a screen shot of an ANIMAL animation. The window is split into three segments, with two floatable control segments embracing the central content area. A control tool bar at the top of the window allows adjusting the display speed and magnification. A second tool bar at the bottom contains control elements for triggering the animation display. For layout reasons, we have dragged the control bar from this figure. It is discussed in detail in Section 3.2. The center of the display is occupied by the canvas presenting the animation itself.

The tool bar at the top of the window contains two groups of control elements. The slide ruler to the left adjusts the animation speed between 0 and 1000% of the original speed. The button next to the slide ruler resets the speed to 100%. The second slide ruler is used for setting the display magnification. It offers values between 0 and 500%. The magnification can also be reset to 100% using the button at the top right. Scaling the animation window for the available page layout introduces some display artifacts, especially regarding text elements and the labeling of the slide ruler ticks.

The animation shown illustrates the branch and bound solution to the dynamic knapsack problem. The state space tree is shown at the left of the display, containing the individual solution nodes with costs. The ellipse at the bottom shows the currently calculated values and the state of the queue. Finally, the right-hand side of the display contains the pseudo-code of the algorithm. The current step is highlighted by a different color shade, in the screen shot, this is step 9.

The example animation was created by four students at the University of Wisconsin Park-side using ANIMAL's graphical front-end. It contains 173 animation steps with a total of 299 graphical primitives. Stored in ANIMAL's compressed ASCII-based format, the animation occupies 9958 bytes.



```
1) insert v into queue of nodes Q;
2) While (Q is not empty):
      proceed to step 3,
      otherwise, finish
3) Dequeue the next node and
      store in v;
4) u->level = v->level + 1;
5) u->weight = v->weight +
      w[u->level];
6) u->profit = v->profit +
      p[u->level];
7) if (u->weight <= W
      AND
      u->profit > maxprofit)
      proceed to step 8
      otherwise, skip to step 9
8) maxprofit = u->profit;
9) if (bound(u) > maxprofit)
      proceed to step 10,
      otherwise, skip to step 11
10) insert u into Q;
11) u gets v's weight and profit;
12) if (bound(u) > maxprofit)
      proceed to step 13,
      otherwise, skip to step 14
13) insert u into Q;
14) go back to step 2.
```

```
w: [ 0 5 10 20]      W: 30
p: [0 50 60 140]     n: 3
Q: <(2,50,5)(2,60,10)(2,0,0)>
maxprofit: 110      level  profit  weight
               v:     2     110      15
               u:     3     250      35
          bound(u): 0
```

Figure 1. Screen shot of ANIMAL's animation window

3.2. Features for Users

The user viewing an animation is one of the most important clients of any AA system. If the users are dissatisfied with the tool, they will not want to use it. This also renders the work of the other three roles useless. Many different considerations have to be taken into account when implementing graphical user interfaces. Faulkner [29] offers a compact overview of general user interface considerations. This section lists some special user features in ANIMAL which are independent of the generic considerations.

One key aspect in tool usability is the language in which the tool is held. Most users are likely to prefer seeing the tool's elements in their native language, assuming the translation is well done. Therefore, we have embedded flexible internationalization support for the graphical user interface components, as well as all messages displayed during program execution. To change the language of the interface, the user simply selects one of the provided language keys from a *Language* menu. All elements are updated at once to the changed language settings, including menu items, button labels and activation hotkeys.

Translating the graphical user interface to another language such as Spanish may be helpful for users fluent in Spanish. However, this is not very effective if the animation itself is in German. The built-in scripting language ANIMALSCRIPT [30] supports the generation of an arbitrary number of language versions within a single animation. The user can choose the animation language on loading the animation from the set of available translations.

A second central area of user support lies in the display controls. Some systems can only display the animation in the forward direction. Only few systems such as *ZStep 95* [12] or *Leonardo* [13] offer full reverse playing. The reversibility is achieved by a special interpretation engine. The underlying semantics of the interpreted programming language limits the applicability of the system.

Several systems have limited support for going backwards in the animation, bought at the cost of memory for storing the history, for example [11]. Additionally, some systems including *JAWAA* [31] and *JSamba* [14] let the animation run through without user interaction, unless explicitly paused. A short distraction may be sufficient to miss a relevant event without being able to go back.

One challenge in designing the ANIMAL system therefore was providing efficient random access to the animation steps. Part of the problem lies with irreversible effects, such as scaling objects with a factor of 0. Additionally, simply displaying the previous step with reverse animation may be insufficient for user understanding. Anderson and Naps [32] state that efficient rewinding is one of the most important 'open questions' in AA.

With ANIMAL, we propose a solution to this problem. Clones of the primitives are used to replace the actual primitives in transformations. Irreversible effects can therefore easily be 'undone' by starting from the beginning of the animation in the worst case. Smooth reverse playing is achieved by treating effects as a change of properties on a percentage scale, with the start of the effect at 0% and the end at 100%. Our implementation efficiently supports smooth reverse playing by simply letting animation effects start at 100% and change their state to 0% instead of in the opposite direction.

ANIMAL provides several different control mechanisms for selecting the current animation step: a control tool bar, a list of labeled animation steps, a slide ruler and a text field for direct input. The control tool bar, shown in Figure 2, provides the same functionality as a video player. The buttons, from left to right, offer the following functions:

- rewind to the first animation step,
- return to the previous step, without displaying the transformations in the step,
- run the animation in reverse slide show mode,
- display the current step backwards, like performing an animated 'undo' operation,
- pause at the end of the current step,

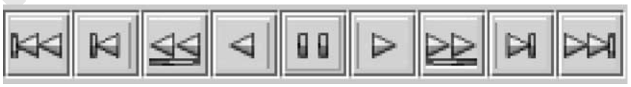


Figure 2. ANIMAL's animation display control tool bar

- play the current step,
- run the animation in slide show mode,
- go to the next step,
- and go to the last step in the animation.

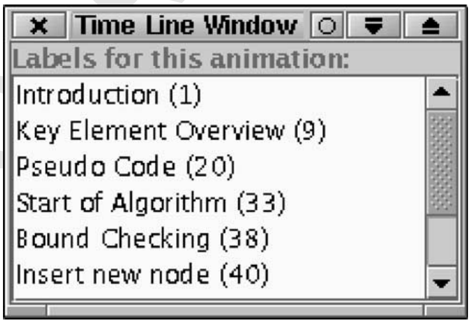
The slide show modes temporarily link all steps, causing the full animation to be displayed without additional interaction. This does not affect manually specified delay times. Steps which normally wait for a key press are linked with a delay time that can be set in the configuration window. The slide show mode is therefore similar to the standard display in *JSamba* [10]. Gloor's second 'commandment of algorithm animation' [28] states that users should interact with the system at least every 45 s. Therefore, we regard the slide show mode as an added bonus, not as the central control.

The slide ruler for adjusting the animation progress shows the execution state on a percentage scale. Dragging the ruler acts like a fast forward or backward operation. The slide ruler's tool tip text states the current percentage of the animation. The text field expects the input of a valid step number and changes to the step. If the requested step does not exist, the input has no effect and the step number is reset to the current step.

The time line window shown in Figure 3 gathers all labeled animation steps. Each line in the window contains a step label, followed by the step number in parentheses. Clicking on one of the labels immediately sets the animation to the associated step. This follows the suggestion of Gloor's third 'commandment of algorithm animation' [28] by enabling the accentuation of logical algorithmic units.

Another issue in animation display is the variety of screen resolutions. Especially when using laptop computers with a display of only 800 × 600, the animation window may be smaller than the animation content. Another possible cause for size mismatches is desktop layout preference. ANIMAL automatically adds scrollbars to the display as needed. However, scrolling through the animation display for each step is not attractive.

We resolve this problem by allowing the user to scale the animation display. A slide ruler is used for adjusting the magnification to any value between 0 and 500%. ANIMAL resizes the display fluidly. The representation of pixel coordinates as a pair of integer values implies that some magnifications work better than others. This is especially the case for text elements, as Java cannot allocate fonts of arbitrary real-valued pixel sizes. Therefore, text that was placed inside a box may become larger than the box in some magnification scales. Changing the scaling factor by a few percentage points is usually



Introduction (1)
Key Element Overview (9)
Pseudo Code (20)
Start of Algorithm (33)
Bound Checking (38)
Insert new node (40)

Figure 3. ANIMAL's time line window with labels acting as hyperlinks

sufficient to fix these problems. The current magnification scale is displayed on the slide ruler's tool tip. For convenience, a button for resetting the magnification to 100% is added.

Finally, ANIMAL supports diverse import and export filters. Most animation systems do not seem to offer any import or export facilities. For example, the scripting languages employed in *JAWAA* [31] and *JSamba* [10] are syntactically very similar; however, neither tool can parse the other tool's notation. ANIMAL's open data exchange architecture allows the user to plug in new filters once they are made available on the WWW.

The standard ANIMAL import filters allow reading and writing the formats of ANIMAL, as well as a beta version of a *JSamba* import filter. We also plan to add a filter for *JAWAA*. Depending on the features of the output format, the user can choose between exporting a single snapshot for each of the selected export steps, or exporting the step with full dynamics. Animations can currently be exported to validated XML and several image formats including BMP, JPG, PNG and Photoshop PSD, as well as *Quicktime* videos. The generation of videos is currently restricted to Windows and MacOS, as the underlying *Quicktime for Java* API [33] incorporates native code.

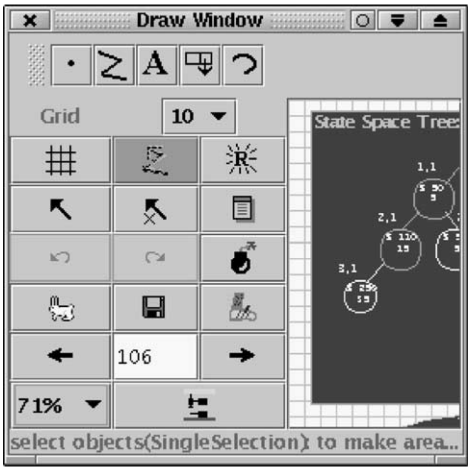
3.3. Features for Visualizers

The visualizer has full timing control over each animation effect by specifying both a duration and an offset from the start of the animation step. The intermediate states of the animated objects are interpolated based on the current state of execution and the duration of the effect. Thus, the second frame of a *move* effect that spans five consecutive display frames will show the target objects at 20% of the way to their target positions.

We want to address a wide range of visualizers with our system. Therefore, three different ways of generating animations are offered: graphically, by scripting and by API calls. In addition, animations can be imported into the system. ANIMAL treats all loaded animations in the same way, regardless of their origin. Thus, all animations can also be further refined in the graphical editors.

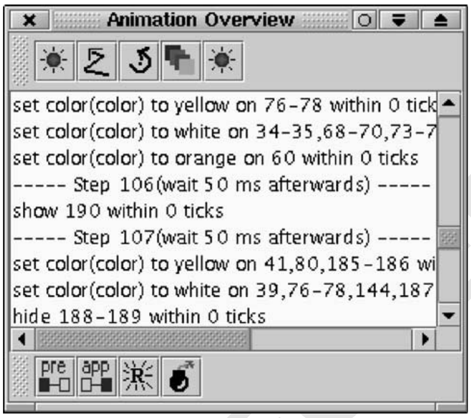
ANIMAL's graphical front-end for animation generation offers access to the full functionality. We have kept the interface as simple as we could to avoid confusing novice users with large menus. The graphical interface contains a drawing window and an animation overview window for assembling the animation effects. The complete graphical interface can be translated into another language with a single menu item selection. We currently provide English and German interface versions. Additional languages can be incorporated by translating a message file and adapting the configuration without touching the system's code.

The drawing window, shown in Figure 4, represents a static view of a given animation step with all animation objects used in the step. New objects are added by clicking on the generation buttons at the top, and then following the instructions in the status line at the bottom of the window. A specific editor window opens when objects are generated or edited. The editors can be used to adapt object properties such as the color or depth information. The drawing interface supports drag and drop on objects or individual nodes. A pop-up menu containing the most common operations is also included. Other features include cloning the current object selection, scaling the display and adjusting settings such as the grid width.



select objects(SingleSelection) to make area...

Figure 4. ANIMAL's drawing window



```
set color(color) to yellow on 76-78 within 0 tick
set color(color) to white on 34-35,68-70,73-7
set color(color) to orange on 60 within 0 ticks
----- Step 106(wait 50 ms afterwards) -----
show 190 within 0 ticks
----- Step 107(wait 50 ms afterwards) -----
set color(color) to yellow on 41,80,185-186 wi
set color(color) to white on 39,76-78,144,187
hide 188-189 within 0 ticks
```

Figure 5. ANIMAL's animation overview window

The animation overview window, shown in Figure 5, is used for assembling an animation. Animation steps or effects can be added or removed. Animation effects are generated by clicking on one of the buttons at the top. An effect-specific pop-up editor window can then be used to assemble the properties of the effect, such as the timing information and the affected objects.

Scripting-based generation uses the built-in ANIMALSCRIPT [30] language, which supports all operations available in the graphical front-end. It also offers command short-cuts that combine multiple objects. For example, ANIMALSCRIPTsupports arrays, lists and code elements. Array support includes index pointers, putting and swapping elements. Lists allow the setting and clearing of their pointers.The number of pointers can be any natural number, and the position can be either to the left, right, above or below the list element value. Code support incorporates highlighting code lines or individual line elements, as well as indentation based on the font size.

ANIMALSCRIPT animations can easily be generated within program code. ANIMALSCRIPT offers several additions compared to other scripting languages such as those employed in *JAWAA* [31] and *JSamba* [10]. Animation internationalization is supported by either embedding the translations into the animation, or linking each translatable text to an external resource file. The translated texts must be provided by the visualizer. Note that the translation of any text is highly likely to impact the text width.

ANIMALSCRIPT offers highly flexible object placement mechanisms to avoid overlapping elements or empty spaces in different language versions. Individual locations can be defined using absolute (x, y) coordinates, by an offset from another object, relative to a stored location, or relative to the last location used. The offset operations use the eight compass directions and the *center* designation, similar to *XTANGO* [14]. Offsets from an object refer to the object's bounding box or an individual node of a polyline object. Relative placement is resolved on loading the animation. Using a given object's bounding box also solves the problem of text width differences in internationalized animations. Locations can be stored and used in any of the specification methods. Relative movement is similar to the forward operation employed in *Logo* and simplifies the generation of many object types such as function graphs.

ANIMALSCRIPT supports debugging using the echo command. Most commands have several optional parameters, so that the user is not forced to specify unwanted entries. We also provide a separate Java API for generating animations. To reduce redundant implementations, the API acts as a wrapper for generating customized ANIMALSCRIPT commands. We expect most visualizers to use either the scripting language or the API after they have become familiar with the system.

3.4. Features for Developers

Developers expect a system to be well-documented, easy to learn and extend, and adaptable to specific needs. Ideally, both recompilation and source code modifications should be unnecessary for adding new features. To achieve this goal, ANIMAL is dynamically assembled from independent components at start-up according to a configuration file. Java's dynamic loading facilities also make adding or removing components at runtime possible.

The configuration is stored in a properties file in the current directory. Thus, developers can add experimental features to one configuration without impacting other configurations. This is especially relevant in the context of a shared installation using a common base configuration. If a given component is unwanted or does not work properly, it can also be removed at runtime. The sole exception to this rule are the core components of the system. We plan to gather all available extensions on the ANIMAL home page [34] including a description of the added features.

ANIMAL developers can implement their own extensions following our (forthcoming) implementation guidelines. Example extension areas are adding graphical primitives, animation effects, import or export filters, and new scripting commands. In all cases, the developer has to extend a base class that encapsulates much of the administrative tasks. The new class must be placed in the appropriate Java package and follow naming conventions.

Implementing new features is comparatively easy, as the developer usually does not have to modify any of the underlying code. Much of the implementation is rather straightforward, although a certain amount of Java knowledge is required. New graphical primitives can be created easily by combining existing primitives, or alternatively by implementing them from scratch.

Graphical primitives are decoupled from animation effects using a special intermediate handler class. Each handler specifies the names of all available animation effects for its underlying primitive. The handler is also responsible for mapping given effects to a set of method invocations on the primitive. Graphical primitives and animation effects can therefore be treated as fully independent components.

Providing a new animation effect requires modifications to at least one primitive handler class. The developer has to add a name for the animation effect and map the execution to appropriate primitive method invocations. Both operations are simple text modifications that can often also be accomplished without Java programming experience. Without the modifications, no primitive type is registered for the new effect, thus rendering it unreachable.

Developers can easily provide a new language to be supported in the GUI. This requires the translation of a simple text file containing all localized messages. Additionally, four lines have to be added to a configuration file. They specify the label of the language menu item, the tool tip text, the language and country code and the name of an image. The image should symbolize the country or language, for example, by using the country's flag.

4. Conclusions

In this paper, we have presented the features the algorithm animation system ANIMAL offers for the three user roles *user, visualizer* and *develope*r. *Users* can adjust the display of the animation in various ways. The video player control bar includes forward and fully dynamic backward slide show mode. Animation steps may possess a label that gives structure to the animation and also acts as a hyperlink to the step. The magnification can be set to a value appropriate for the current environment. Even large animations can be displayed on small display sizes, with a very modest quality degradation. The language used in the GUI can be changed with one menu selection. Animations built with ANIMALSCRIPT can also provide multiple language versions without overlapping elements or white spots. The language support depends on the languages available in the current setting.

Visualizers profit from the highly flexible graphical primitives and animation effects provided by the base system, as well as from additions loaded from the WWW. They can choose between a graphical interface, scripting or API for generating animations according to their preferences and skill levels. Each animation effect can be precisely scheduled using an offset from the start of the animation step and a duration. Scripting and API generation add powerful commands which are mapped to a set of primitives or effects. Examples include extensive support for internationalization, flexible object placement and special primitive support as for arrays, lists or code elements.

Developers should find it easy to extend or adapt ANIMAL to their preferences, including adapting the language used in the GUI front-end. Most possible ANIMAL extensions do not require any modification of the system code. If code has to be touched, the modification is typically localized to a single class.

Further plans for ANIMAL include providing new graphical primitives, such as images and item lists. We plan to publish a documentation of the steps for extending the system on the tool page. Since 2001, ANIMALSCRIPT is also supported by the JHAVÉ environment [5]. The benefits of this cooperation regarding multiple-choice quizzes and links to external, dynamic HTML documentation seem promising, though it is too early for a detailed evaluation.

We are also working on providing an applet version of the ANIMAL player front-end. The applet might not fully resemble the application due to the tight integration with Swing and the lack of Swing support in current browsers. An older applet version is available at the ANIMAL home page [34]. Note that this applet may not work with some versions of Netscape due to bugs in Netscape's security manager implementation.

ANIMAL is available on the Internet at http://www.animal.ahrgr.de/. All currently registered ANIMAL animations as well as future extensions of ANIMAL will also be collected at this location.

Acknowledgements

The authors would like to thank the students P. Ahlbrecht, J. Brodowski, A. Flöper, M. Schöuler, and M. Smith for their work in implementing parts of ANIMAL.

References

1. B. Price, R. Baecker & I. Small (1998) An introduction to software visualization. In: *Software Visualization* (J. Stasko, J. Domingue, M. H. Brown & B. A. Price, eds). MIT Press, Cambridge, MA, Chapter 1, pp. 3–27.
2. J. Stasko, J. Domingue, M. H. Brown & B. A. Price (eds) (1998) *Software Visualization: Programming as a Multimedia Experience*. MIT Press, Cambridge, MA.
3. H. L. Dershem & P. Brummund (1998) Tools for Web-Based Sorting Animations. *29th ACM SIGCSE Technical Symposium on Computer Science Education* (*SIGCSE' 98*), Atlanta, GA, pp. 222–226.
4. D. Jarc, M. B. Feldman & R. S. Heller (2000) Assessing the benefits of interactive prediction using web-based algorithm animation courseware. *31st ACM SIGCSE Technical Symposium on Computer Science Education* (*SIGCSE 2000*), Austin, TX, pp. 377–381.
5. T. Naps, J. Eagan & L. Norton (2000) JHAVÉ: an environment to actively engage students in web-based algorithm visualizations. *31st ACM SIGCSE Technical Symposium on Computer Science Education* (*SIGCSE 2000*), Austin, TX, pp. 109–113.
6. R. Baecker & D. Sherman (1981) Sorting out Sorting. 30 minute color film, Dynamic Graphics Project, University of Toronto (excerpted and 'reprinted' in SIGGRAPH Video Review 7, 1983). Distributed by Morgan Kaufman Publishers, Los Allos, CA.
7. R. Baecker (1998) *Sorting Out Sorting*: a case study of software visualization for teaching computer science. In: *Software Visualization* (J. Stasko, J. Domingue, M. H. Brown & B. A. Price, eds). MIT Press, Cambridge, MA, pp. 369–381.
8. M. H. Brown (1998) A taxonomy of algorithm animation displays. In: *Software Visualization* (J. Stasko, J. Domingue, M. H. Brown & B. A. Price, eds). MIT Press, Cambridge, MA, Chapter 3, pp. 35–42.
9. G. Rößling, M. Schüler & B. Freisleben (2000) The ANIMAL algorithm animation tool. *5th Annual ACM SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education* (*ITiCSE 2000*), Helsinki, Finland, pp. 37–40.

10. J. Stasko (1998) *Samba Algorithm Animation System*. Available at http://www.cc.gatech.edu/gvu/softviz/algoanim/samba.html.
11. A. Zeller (2001) Animating data structures in DDD. *First International Program Visualization Workshop*, Porvoo, Finland. University of Joensuu Press, pp. 69–78.
12. H. Lieberman & C. Fry (1998) ZStep 95: A reversible, animated source code stepper. In: *Software Visualization* (J. Stasko, J. Domingue, M. H. Brown & B. A. Price, eds). MIT Press, Cambridge, MA, Chapter 19, pp. 277–292.
13. P. Crescenzi, C. Demetrescu, I. Finocchi & R. Petreschi (2000) Reversible execution and visualization of programs with LEONARDO. *Journal of Visual Languages and Computing* **11,** 125–150.
14. J. Stasko (1998) Smooth continuous animation for portraying algorithms and processes. In: *Software Visualization* (J. Stasko, J. Domingue, M. H. Brown & B. A. Price, eds). MIT Press, Cambridge, MA, Chapter 8, pp. 103–118.
15. A. I. Concepcion, N. Leach & A. Knight (2000) Algorithma 99: an experiment in reusability & component based software engineering. *31st ACM SIGCSE Technical Symposium on Computer Science Education* (*SIGCSE 2000*), Austin, TX, pp. 162–166.
16. W. Pierson & S. H. Rodger (1998) Web-based animation of data structures using *JAWAA*. *29th ACM SIGCSE Technical Symposium on Computer Science Education* (*SIGCSE'98*), Atlanta, Georgia, pp. 267–271.
17. D. Nguyen & S. B. Wong (2001) Design patterns for sorting. *32nd ACM SIGCSE Technical Symposium on Computer Science Education* (*SIGCSE 2001*), Charlotte, NC, pp. 263–267.
18. R. Rasala (1999) Automatic array algorithm animation in C++. *30th ACM SIGCSE Technical Symposium on Computer Science Education* (*SIGCSE'99*), New Orleans, LA, pp. 257–260.
19. M. Terada (2001) Animating C programs in paper-slide-show. *First International Program Visualization Workshop*, Porvoo, Finland. University of Joensuu Press, pp. 79–88.
20. J. Haajanen, M. Pesonius, E. Sutinen, J. Tarhio, T. Teräsvirta & P. Vanninen (1997) Animation of user algorithms on the web. *IEEE Symposium on Visual Languages*, pp. 360–367.
21. R. B.-B. Levy, M. Ben-Ari & P. A. Uronen (2001) An extended experiment with Jeliot 2000. *First International Program Visualization Workshop*, Porvoo, Finland. University of Joensuu Press, pp. 131–140.
22. T. Hung & S. H. Rodger (2000) Increasing visualization and interaction in the automata theory course. *31st ACM SIGCSE Technical Symposium on Computer Science Education* (*SIGCSE 2000*), Austin, TX, pp. 6–10.
23. C. E. George (2000) EROSI—visualizing recursion and discovering new errors. *31st ACM SIGCSE Technical Symposium on Computer Science Education* (*SIGCSE 2000*), Austin, TX, pp. 305–309.
24. S. Khuri & H.-C. Hsu (1999) Visualizing the CPU scheduler and page replacement algorithms. *31st ACM SIGCSE Technical Symposium on Computer Science Education* (*SIGCSE'99*), New Orleans, LA, pp. 227–231.
25. S. Khuri & H.-C. Hsu (2000) Interactive packages for learning image compression algorithms. *5th Annual ACM SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education* (*ITiCSE 2000*), Helsinki, Finland, pp. 73–76.
26. W. Yurcik & L. Brumbaugh (2001) A web-based little man computer simulator. *32nd ACM SIGCSE Technical Symposium on Computer Science Education* (*SIGCSE 2001*), Charlotte, NC, pp. 204–208.
27. S. Bridgeman, M. T. Goodrich, S. G. Kobourov & R. Tamassia (2000) PILOT: an interactive tool for learning and grading. *31st ACM SIGCSE Technical Symposium on Computer Science Education* (*SIGCSE 2000*), Austin, TX, pp. 139–143.
28. P. A. Gloor (1998) User interface issues for algorithm animation. In: *Software Visualization* (J. Stasko, J. Domingue, M. H. Brown & B. A. Price, eds). MIT Press, Cambridge, MA, Chapter 11, pp. 145–152.
29. C. Faulkner (1998) *The Essence of Human–Computer Interaction*. Prentice-Hall, Englewood Cliff, NJ.
30. G. Rößling & B. Freisleben (2001) ANIMALSCRIPT: an extensible scripting language for algorithm animation. *32nd ACM SIGCSE Technical Symposium on Computer Science Education* (*SIGCSE 2001*), Charlotte, NC, pp. 70–74.

31. S. Rodger (1997). *JAWAA and Other Resources*. Available at http://www.cs.duke.edu/~rodger/tools/tools.html.
32. J. M. Anderson & T. L. Naps (2001) A context for the assessment of algorithm visualization system as pedagogical tools. *First International Program Visualization Workshop*, Porvoo, Finland. University of Joensuu Press, pp. 121–130.
33. Apple Computers, Inc. (2001) Quicktime for Java API. WWW:http://developer.apple.com/quicktime/qtjava/index.html.
34. G. Rößling (2001). ANIMAL *Home Page*. WWW: http://www.animal.ahrgr.de.