Program Visualization Using ANIMALSCRIPT

Guido Rößling
Bernd Freisleben
Department of Electrical Engineering and Computer Science
University of Siegen
Hölderlinstr. 3, D-57068 Siegen, Germany
roessling@acm.org, freisleb@informatik.uni-siegen.de

Abstract

ANIMALSCRIPT is a powerful scripting language for generating program visualizations using the ANIMAL animation tool. ANIMAL itself offers a small but powerful set of graphical operators. ANIMALSCRIPT extends these operators by providing support for subtypes, code inclusion and highlighting, composite objects, and data-type specific operations. All operations can be given a precise timing on a millisecond basis or using a built-in timing. Animations generated by ANIMALSCRIPT can also be edited visually. The paper presents ANIMALSCRIPT and evaluates its functionality in comparison to other animation tools.

1 Introduction

ANIMAL (Rößling & Freisleben, 2000a, 2000b, 2000c; Rößling, Schüler, & Freisleben, 2000b, 2000a) is a flexible and powerful animation tool we started developing in 1998. ANIMAL supports three separate approaches for generating animations: *visually, by scripting,* and *by API calls.*

In order to keep ANIMAL usable by the widest possible range of users, the *visual interface* is kept as simple as possible, offering only five buttons each for generating animation objects and animation effects called *animators*. Both objects and animators are easily customizable to more specific subtypes and incorporate special features, such as color, depth, or timing information.

By now, the scripting language ANIMALSCRIPT is the preferred avenue for animation generation due to its powerful commands. The ANIMALGENERATOR API was implemented to actually generate ANIMALSCRIPT commands for each

function call, stressing the role of ANIMALSCRIPT. Both ANIMALSCRIPT commands or ANIMALGENERATOR API method calls are easily embedded into programs, allowing dynamic generation of program visualizations.

This paper focuses on ANIMALSCRIPT and describes some aspects of the scripting language that go beyond what several other comparable tools such as JAWAA (Rodger, 1997) and JSamba (Stasko, 1998) offer.

The paper is organized as follows. In the following section, some basic features of ANIMALSCRIPT are presented. This is followed by a short overview of the supported object types and operations. We then evaluate ANIMALSCRIPT in comparison to both JAWAA and JSamba, and conclude with our plans for further work.

2 Features of ANIMALSCRIPT

ANIMALSCRIPT is a simple, line-based language driven by text commands. This means that

- each command must be given on a *single line*,

  This also means that the lines may become somewhat long, so that users should refrain from using auto wrapping in their editors.

- all entries are given in ASCII text,

- and each line starts with a unique command, making it easy for ANIMALSCRIPT to parse the line.

Each ANIMALSCRIPT file should start with the standard ANIMALSCRIPT header describing the version of ANIMALSCRIPT used in the first line, optionally followed by a *title* and *author* line. Both title and author must be placed in double quotes "...".

A standard ANIMALSCRIPT file thus starts similar to the following entry:

```
%Animal 1.4
title "Demo animation"
author "Guido Roessling"
```

All ANIMALSCRIPT operations give full timing control to the user, supporting the specification of both *offset* and *duration*. Object definitions can also be given an *offset*. Timing is measured either in *ms* (milliseconds) or *ticks*, an internal unit that allows for smoother operations.

In the following examples, a backslash \ is used to indicate that entries must actually be placed in *the same line.* Note that the examples are used to illustrate the commands discussed in each section; they are *not* meant to generate a meaningful animation. An example ANIMALSCRIPT animation is given at the end of the paper.

In the next sections, we describe some of the currently supported objects, followed by the available animators. Finally, we compare ANIMAL's features with those of other tools.

3 Supported Objects

The basic command structure for object generation in ANIMALSCRIPT is as follows:

```
type "ID" placement [options] [delay]
```

Supported *types* currently include *arc, array, code, list element, point, polyline / polygon* and *text.* There are several subtypes offered for most of these entries. For example, apart from *arcs*, users can also define *ellipses, circles* or *circle segments*, and can choose between open and closed segments. Polylines can also be given a forward or backward *arrow* pointing away from the last or first node, respectively. This is especially useful for using pointers to other elements.

Placement can be given in absolute coordinates in the form `(x, y)`. However, ANIMALSCRIPT also supports placing objects *relative* to another object using the **offset** keyword. As the anchor for this operation, each edge and middle point of the base object's bounding box can be used as the reference point. These reference points are labelled as compass needle points, e.g. **NE, S** and **C**. The use of absolute and relative coordinates can also be mixed.

Thus, the following code generates two points placed 10 pixels apart from each other on the same line:

```
point "p1" (100, 50) 30
point "p2" offset (10, 0) from "p1" NE
```

Optional parameters define the specific appearance of a given object and can be omitted. In this case, ANIMALSCRIPT will use the *previous* value used for a component of the same type If no component of this type was previously encountered, a default value is used, for example the color *black.*

Optional parameters are used widely in ANIMALSCRIPT and support settings such as *color, depth*, specific attributes such as whether objects are *closed* or *filled*, and the optional *delay* before the component is shown.

To resolve the display of overlapping components, ANIMALSCRIPT supports a *depth* tag for all objects. Comparing two elements which overlap, the following

distinction determines the object place atop of the other one:

- if the depth values differ, the object with *smaller* depth value is placed on top,
- else the *more recently defined* object is placed on top.

ANIMALSCRIPT supports *arrays* placed either in horizontal or vertical direction. The cells may be shown all at the same time, or in cascaded display using a special timing information. A specific operation for installing a *pointer* to a given array position is also included, requiring only the *target array ID* and the target index.

Furthermore, *code support* allows for defining a *code group* in which an arbitrary number of code lines or fragments thereof can be placed. Each code element can be given an *indentation level* and is easily referenced later on for effects such as *code highlighting.* These operations are outlined in the next section.

List elements with as many pointers as the user wants can be easily defined. Operations for manipulating the list pointers are described in the next section.

Note that animations are organized in distinct *animation steps*, each of which may contain an arbitrary number of effects and object definitions. However, no step may contain *two* operations on the same object. Normally, each line starts a *new step.* If more than a single operation is supposed to take place in the same step, they have to be grouped using braces { }.

Figure 1 shows a few of the more advanced ANIMALSCRIPT object types. It illustrates optional parameters such as depth and delay, relative placement, grouping object definitions in a single step, and some supported types.

In the code, we first generate a partial arc, followed by an array with a pointer to the second array element. Then, we install some pseudo code with one indentation level and finally add two list elements with two pointers each.

Note that ANIMALSCRIPT allows the inclusion of an arbitrary amount of whitespace. Thus, the lines in the example above could also have been indented. However, due to layout restrictions, no indentation was used in this example.

4 Supported Operations

ANIMALSCRIPT offers five generic operations, several operations only available for specific object types, and general commands, which we will address in this order. All generic and object-specific operations allow for precise timing using both *delay* and *duration.*

```
arc "A" (50, 50) radius (30, 20) angle 180 starts 60
{
  array "B" offset (20, 30) from "A" NE vertical \
    length 4 "H" "A" "V" "E"
  arrayIndex "C" on "B" atIndex 1 label "i"
}
{
  codeGroup "code" at (10, 200) color black\
    highlightColor red contextColor blue \
    font Monospaced size 16 depth 10
  addCodeLine "Demonstrate List Usage" to "code" \
    indentation 1
  addCodeLine "Generate list elements" to "code" \
    indentation 1
  addCodeLine "Link elements" to "code"
}
listElement "D" (150, 100) text "1" pointers 2 depth 10
listElement "E" (200, 100) text "2" pointers 2
```

Figure 1: ANIMALSCRIPT example of supported object types

4.1 ANIMALSCRIPT Generic Operations

The generic operations are

- **show / hide** for toggling the visibility of a given object,
- **move**,
- **rotate**,
- and **color change**, supporting the setting of *any* appropriate color for the given object. Apart from the basic color, most objects also have a *fill color*, and some also posses a *highlight color*.

The **move** command allows for very flexible moving of arbitrary components in either of the following modes:

- **via** a previously defined *moveline*

- **along** a moveline defined *within* the command,
- **to** a given location, specified either in absolute or relative coordinates.

The term *moveline* here refers to either a *polyline, polygon, arc* or *circle* object. Thus, moves along both straight lines and arcs are possible for *all* objects.

Furthermore, most objects offer more than a single method for moving them. Thus, you can, for example, move a single point of a component or the whole component.

The following example highlights the use of some commands, referring to the objects defined in previous examples. It will first hide the arc component installed in the last section. After moving *both* list elements simultaneously along the arc component, just the first list element is moved. To illustrate the power of the **move** command, all three options listed above are used.

```
hide "A" after 10 ticks
move "D" "E" via "A" within 10 ticks
move "D" along line (10, 20) (20, 30)
move "D" to offset (50, 30) from "p1" C
```

The use of a simple line as a moveline again comes from layout restrictions. ANIMALSCRIPT supports movelines with an arbitrary number of nodes.

4.2 Object-Specific Operations

The object-specific operations deal with *array manipulation, code maintenance* and *list operations.*

For *array operations,* users can choose between operations for entering values into a given array (**arrayPut**), swapping array elements (**arraySwap**), and moving any installed array pointer to another element.

Code maintenance can be used for *adding* code, toggling *code highlighting* for a given line or element, or *hiding* the whole set of code. Highlighting further distinguishes between normal highlighting and *context mode*. The latter is useful if you wish to show users the current context, e.g. the enclosing method, loop, or condition statement.

List operations focus on setting the link(s) of a given list element by either *clearing* them (resetting to `null`), or setting to another list element. Note that users do *not* have to keep track of where list elements are placed - they need only specify the target ID of the element to be linked to. ANIMAL will then figure out where the objects are located relative to each other, and choose an arbitrary moveline for resetting the pointer location.

It highlights the first line of code, then changes it to *context* mode and highlight the second line of code. One array entry is then substituted by a new value, and two entries are swapped. After updating the pointer to the array, the highlighted line of code is changed, followed by linking the two list elements. Finally, after a delay of 20 ticks, the whole code is hidden at the same time.

Due to the structure of the examples in this paper, the second code line with content "`Generate list elements`" is highlighted only after both elements are already inserted. Of course, this is not a restriction ANIMALSCRIPT places on the user. Furthermore, a hash mark # can be used to start a comment line.

```
highlightCode on "code" line 0
highlightCode on "code" line 0 context
highlightCode on "code" line 1
arrayPut "N" on "B" position 0 \
    after 20 ticks
arraySwap on "B" position 1 with 3
moveArrayMarker "C" to position 2
{
  unhighlightCode on "code" line 1
  highlightCode on "code" line 2
  setLink "D" link 1 to "E"
}
# now hide the complete code!
hideCode "code" after 20 ticks
```

4.3 Maintenance Commands

In addition to the commands outlined in the last two subsections, ANIMALSCRIPT also contains some easy maintenance commands.

The **delay** command can be used to install a timed delay between animation steps. The default behavior of ANIMAL is to wait until the "play" button > shown in figure 2 on page 9 is pressed.

The **group / ungroup** allow merging separate objects under a new identity. This makes it easier to specify actions to be performed on *all* objects of the group without having to repeat all object IDs again and again.

Finally, the **echo** command can be used to send any given text to `System.out`. It also allows the user to print out absolute coordinates or the resulting point of a relative placement, making it easier to adapt a given setting.

The following example illustrates these commands:

```
# wait 1 second, then show next step
delay 1000 ms
# the usual way of moving objects:
{
polyline "pl" (10, 10) (30, 100)
move "D" "E" via "line" within 10 ticks
}
# using grouping in (new) group "elems"
{
group "elems" "D" "E"
move "elems" via "line" within 10 ticks
}
# Now show where the objects are!
echo text: "Location of first list elem:"
echo boundingBox: "D"
echo location: offset (20,20) from "D" C
```

The **echo** commands provide the following output:

```
# Location of first list elem:
# "D" has bounds (168, 290), (231, 375)
# location: (219, 352)
```

5 Comparison to Other Tools

In the following, we present the main differences between ANIMALSCRIPT and the scripting languages employed in JAWAA and JSamba.

JAWAA is tightly focused on the computer science context. It also supports some structures ANIMALSCRIPT does not *currently* support including *trees, stacks* and *queues.* We are working on embedding these types into ANIMALSCRIPT as soon as possible. JAWAA's tight focus also limits its usability outside the Computer Science context. The basic display of JAWAA is a slide-show with user-adjustable speed and single-step capability. It is not possible to go back one step.

JAWAA does not offer any timing, nor does it have any type of *arc* components except for circles and tree nodes. Support for *code* is very scarce - it can only be added as text and manually formatted, while ANIMALSCRIPT includes code indentation and highlighting. Relative placement of objects is not possible, unless the user keeps track of the coordinates of all objects. Finally, the **move** functionality is very restricted compared to the features ANIMALSCRIPT offers.

JSamba, on the other hand, is less CS-specific than JAWAA and provides support for *multiple views* and *zooming*. JSamba displays the animation in a slide-show mode similar to JAWAA, and also does not support going back steps comfortably.

JSamba also does not offer timing control, and has no support for *arcs*, *code*, or *arbitrary polyline / polygon* objects, limiting them to at most eight vertices. As in JAWAA, source code embedding and highlighting is only possible by using simple text components, placing the burden of indentation and highlighting on the user.

Finally, JSamba's move functionality is very restricted when compared to ANIMALSCRIPT, supporting only a smooth move (without adjustable timing) and direct jump. The more advanced move subtypes are not supported.

Figure 2 shows a screenshot of an ANIMAL animation. You can see that ANIMAL offers a video player-like functionality including going to the previous step(s). The *slider* can be used to enter a *slide-show like* presentation by dragging it over the individual steps.

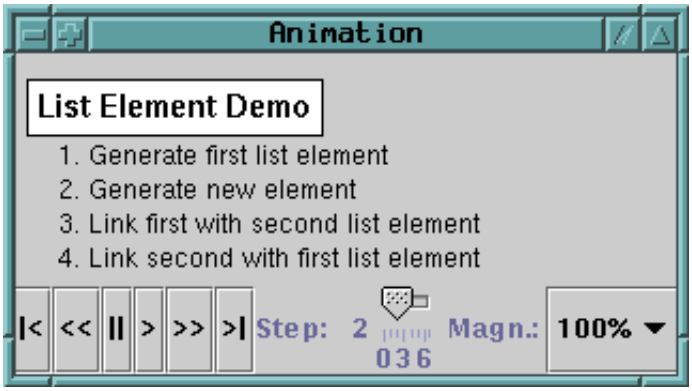

Figure 2: ANIMAL Player at start of animation

6 Example Program Visualization

The following example gives the source code for an animation dealing with doubly-linked lists. It contains several of the advanced features described above.

```
%Animal 2
{
 rectangle "hdrBox" (25, 25) (250, 60) \
  filled fillColor white depth 5
 text "demo" "List element demo" \
  offset (5, -5) from "headerRect" SW \
  font SansSerif size 24 bold depth 0
}
{
 codeGroup "listSource" at (10, 200) \
  color black highlightColor red
 addCodeLine "1. Generate first list\
  element" to "listSource"
 addCodeLine "2. Generate new element" \
  to "listSource"
 addCodeLine "3. Link first with second\
  list element" to "listSource"
 addCodeLine "5. Link second with first\
  list element" to "listSource"
}
{
 highlightCode on "listSource" line 0
 listelement "elem1" (100, 80) \
  text "Elem1" pointers 2 after 20 ticks
}
{
 unhighlightCode on "listSource" line 0
 highlightCode on "listSource" line 1
 listelement "elem2" offset \
  (40, 0) from "elem1" NE \
  text "Elem2" pointers 2
}
{
 unhighlightCode on "listSource" line 1
 highlightCode on "listSource" line 2
 setLink "elem1" link 1 to "elem2"
}
```

```
{
 unhighlightCode on "listSource" line 2
 highlightCode on "listSource" line 3
 setLink "elem2" link 1 to "elem1"
}
unhighlightCode on "listSource" line 3
```

This code will lead to the program visualization shown in figure 2 and 3.

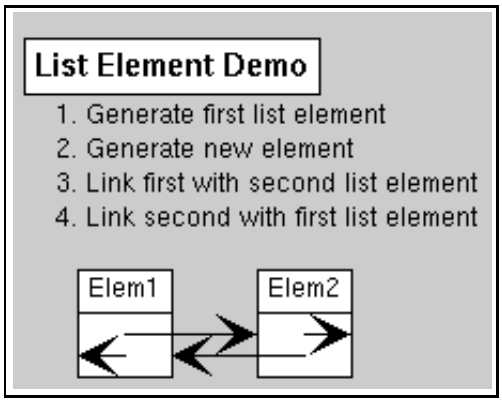

Figure 3: Display at end of example animation

7 Further Work

We plan to extend ANIMALSCRIPT with further features including more advanced data structure support as in JAWAA. This will also be reflected by changes in the ANIMALGENERATOR API functions for generating ANIMALSCRIPT source.

Finally, we are working on changing ANIMALSCRIPT into a component-based collection of small parser libraries, allowing users to extend ANIMALSCRIPT with their *own* commands. In order to keep the language easy to parse by making sure all commands are unique, we will maintain a central library of all extensions implemented by us or other users.

With this change, it should be comparatively easy to generate extensions of ANIMALSCRIPT tailored to specific needs *without* having to touch much – if any – of the original ANIMAL code. The extensions will be maintained dynamically, allowing for easy addition of other features without having to modify the system.

References

Rodger, S. (1997). *JAWAA and Other Resources.* (Available at `http://www.cs.duke.edu/~rodger/tools/tools.html`)

Rößling, G., & Freisleben, B. (2000a). Approaches for Generating Animations for Lectures. *Proceedings of the 11th Society for Information Technology and Teacher Education Conference*, 809–814.

Rößling, G., & Freisleben, B. (2000b). Experiences In Using Animations in Introductory Computer Science Lectures. *SIGCSE 2000*, 134–138.

Rößling, G., & Freisleben, B. (2000c). Flexible Generation of Animations Using ANIMAL. *International Conference on Mathematics / Science Education & Technology*, 432.

Rößling, G., Schüler, M., & Freisleben, B. (2000a). The ANIMAL Algorithm Animation Tool. *5th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2000)*, 37–40.

Rößling, G., Schüler, M., & Freisleben, B. (2000b). ANIMAL: A New Interactive Modeler for Animations in Lectures. *Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education, 32*(1), 437.

Stasko, J. (1998). *Samba Algorithm Animation System.* (Available at `http://www.cc.gatech.edu/gvu/softviz/algoanim/samba.html`)