ANIMALSCRIPT: An Extensible Scripting Language for Algorithm Animation

Guido Rößling and 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

Published in ACM SIGCSE 2001 Proceedings, pp. 70–74

Abstract

In this paper, we present the ANIMALSCRIPT visualization language. This scripting language uses the flexibility of the ANIMAL system and provides many additional new graphic primitives and animation effects that go beyond the traditional ANIMAL GUI features.

ANIMALSCRIPT can easily be configured by changing the content of a registration file. Users may also have multiple registration files, as ANIMALSCRIPT will always use the first registration file it finds. ANIMALSCRIPT can easily be extended with additional features without needing to read, let alone change, any existing code.

1 Introduction

Most popular animation and visualization systems rely on at least one of the following ways for generating their content: *visually* within a GUI [4], directly from underlying *source code* [1], by API calls [5] or using a *scripting language* [3, 4, 6]. Each approach has its advantages and disadvantages. GUI-based generation usually does not require much knowledge of animations or programming; however, animation generation is very slow as everything has to be done manually. Source code animation does not require any direct user action; however, the visualization is constrained both by the supported language and the list of supported data structures. API calls allow for automating the generation process, but require a firm grasp of programming knowledge in the underlying programming language. Finally, scripting languages also allow for automatic generation, but force the user to learn a specific syntax.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCSE 2001 2/01 Charlotte, NC, USA
© 2001 ACM 1-58113-329-4/01/0002...$5.00

The main drawback of all approaches is that the supported features of the listed visualization systems are fixed. For example, Jeliot [1] is restricted to Java source code and a limited selection of "known" data structures. JAWAA [3] is geared for the computer science context and thus supports some data structures such as arrays and stacks. However, it cannot easily be used for "generic" animations, or algorithms using data structures not included in the implementation.

The same is true for the other systems: if a given feature is not available in the system used, there is often nothing to be done about it. From a pragmatic point of view, this is still true for many open software products. While it is technically possible to adapt and extend them according to one's needs, this usually requires reading and modifying a large part of the source code. Thus, adaptation becomes too time-consuming to tackle for most users, often leading to the development of different - but not necessarily better - tools to address specific needs.

Our Java-based ANIMAL project started from the same motivation - no system offered quite the right set of operations for our purposes. However, we have taken care to make the system as open to extensions as possible, so that others will not be forced to develop new tools just because some feature they look for is not provided in ANIMAL.

In this paper, we focus on the extensibility of ANIMAL's built-in scripting language ANIMALSCRIPT[4]. In section 2, we provide a quick overview of the basic operations of ANIMALSCRIPT. Section 3 introduces some advanced operations of ANIMALSCRIPT. In section 4, we outline how ANIMALSCRIPT dynamically determines its components, followed by a discussion of how new primitives or animation effects can be added in section 5. Section 6 presents our conclusions.

2 ANIMALSCRIPT Basics

ANIMALSCRIPT is a simple but powerful scripting language. ANIMALSCRIPT input is typically a simple ASCII file consisting of a number of lines. Each line may contain either a comment marked by a hash mark #, or exactly *one* ANIMALSCRIPT command. The first word of each com-

mand line is the *keyword*; this must always be unique to allow both easy parsing and dynamic extension.

ANIMALSCRIPT commands typically have a number of parameters, some of which may be optional. The syntax of ANIMALSCRIPT is defined by a set of extended Backus-Naur rules using [ ] for optional parameters and {} for elements that may be repeated an arbitrary number of times.

The basic constructs of ANIMALSCRIPT are the graphic primitives *point, polyline / polygon, text* and *arc (ellipse segment)*. These can be used to build all other object types. For ease of use, ANIMALSCRIPT also provides specific subtypes of the primitives such as *rectangle, square, circle, ellipse*, plus a *list element* primitive with as many pointers as wanted. Pointers can be placed on all four corners of the object. Each object has a *color, location, bounding box* and *depth*. The depth can be used for specifying which object stays on top when two or more objects overlap.

ANIMAL animations built from ANIMALSCRIPT input consist of a number of animation steps each containing an arbitrary number of animation effects. Each command line in an ANIMALSCRIPT file is placed in a new step, unless commands are grouped together with curly braces { }.

All graphic objects may be displayed either at once or after a delay which is appended to the command line. Each graphic object has its unique ID used for referencing it later. The ID is an arbitrary text placed in double quotes "...".

The basic animation commands provided by ANIMALSCRIPT contain the effects *show, hide, move, rotate* and *change color*. All operations can be given both a duration and an offset from the beginning of the step. All timing can be given either in milliseconds or internal units called "ticks". Ticks depend on the graphics power of the currently used machine, making the display smooth on both fast and slow machine. ANIMAL uses the duration information to determine appropriate intermediate animation states.

In addition, object IDs can be swapped to easily support element switching. Users can also specify a delay between steps, and can give a label to each step if desired. The labels are collected in ANIMAL and serve as reference points that can be directly jumped to during the animation.

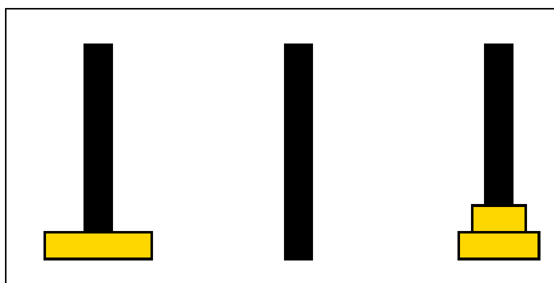

Figure 1: Result of *Towers of Hanoi* animation

Figure 1 shows an example animation screen shot for the *Towers of Hanoi* problem popular for illustrating recursion. All disks are to be moved from the left stick to the middle stick with the right stick as an auxiliary. The main portion of the required ANIMALSCRIPTcode is presented in figure 2. The % character used in the first line is part of the standard ANIMALSCRIPT header format.

```
% Animal 1.4
{
  polygon "ls" (35, 40) (35, 120)\-
   (45, 120) (45, 40) color black\-
   filled fillColor dark Gray
  polygon "cs" (110, 40) (110, 120)\-
     (120, 120) (120, 40) filled
  polygon "rs" (185, 40) (185, 120)\-
     (195, 120) (195, 40) filled
  polygon "d1" (30, 90) (30, 100)\-
     (50, 100) (50, 90) filled \-
     fillColor gold
  polygon "d2" (25, 100) (25, 110)\-
     (55, 110) (55, 100) filled
  polygon "d3" (20, 110) (20, 120)\-
     (60, 120) (60, 110) filled
}
{
  polyline "p"  (35, 90) (35, 20)\-
     (110, 20) (110, 110) hidden
  move "d1" via "p"  within 10 ticks
}
{
  polyline "p0"  (35, 100) (35, 20)\-
     (185, 20) (185, 110)  hidden
  move "d2" via "p0"  within 10 ticks
}
# ( ... skipping 20 lines of code ...)
{
  polyline "p5"  (35, 110) (35, 20)\-
    (110, 20) (110, 90)  hidden
  move "d1" via "p5"  within 10 ticks
}
text "tText" "# Moves: 7" at (113, 140)\-
  centered font SansSerif size 16 bold
```

Figure 2: ANIMALSCRIPT for *Towers of Hanoi* animation

This quick overview of ANIMALSCRIPT's features covers the basic operations available in ANIMAL. The ANIMAL Web page also contains a summary of ANIMALSCRIPT commands. Basic ANIMALSCRIPT is very useful for all standard animation purposes, but lacks many advanced features such as *array* or *source code* support. These are addressed by the extended ANIMALSCRIPT functionality described in the next section.

3 Extended AnimalScript Functionality

AnimalScript provides some extended functionality that offers more comfortable access to "hidden" features of Animal. The simplest extended command is the *group / ungroup* directive which tells AnimalScript to group all object IDs passed in as one object for the next steps, and thus frees the user from much typing.

List linking allows the user to set or clear the pointers of list elements with a single command. In keeping with pointer semantics, the user only has to specify the target object by its ID, and does not have to worry about the exact location of the pointer. This takes a lot of administrative book-keeping from the user, compared to other systems where the precise target location must be given.

AnimalScript provides full array support for both vertical and horizontal alignments. This includes the installation and moving of *array index pointers* with an optional label as well as the put and swap operations. As the last two operations may have an effect on the width of the array cells after the first affected element, the layout of the array is updated, including the possible moving of existing array pointers. Thus, sorting an array using AnimalScript is very easy and consists mostly of a sequence of arraySwap and movePointer commands for changing the elements and moving the (optional) index pointers. *Stacks* are also supported, offering the push, pop, clear and hide operations.

AnimalScript also provides full *source code embedding*. For this purpose, the user declares a codeGroup with some formatting options, such as the font to be used, and gives the upper left corner. Individual code lines or code line elements can then be added to the code group, including an optional indentation depth. The code is laid out dynamically. *Source code highlighting* is easily accomplished by commands that allow the user to highlight a given line or line segment as either the current command or the current context.

All elements may be positioned in one of the following ways:

- using absolute coordinates such as `(10, 20)`,
- relative to an edge of another object's *bounding box*,
- relative to an *arbitrary node* of a given (polyline) object,
- or relative to the *last* location used.

Figure 3 shows both the power of relative placement and AnimalScript's support of list elements.

One of the most helpful features of AnimalScript during animation development, however, is the echo command. This command can be used to print out the *location* of objects (their upper left corner) or their *bounding box*. It can also be used to print the *ids* of grouped objects, or show the ids of all currently visible objects. For assertions, *echo* can print out the current value, and it can also print out any given

```
% Animal 1.4
{
  codegroup "code" (10, 50)
  addCodeLine "Generate first" to "code"
  addCodeLine "Generate second" to "code"
  addCodeLine "Generate third" to "code"
  addCodeLine "Link first, second" to\-
              "code"
  addCodeLine "Link second, third" to\-
              "code"
}
{
  listElement "10" (10, 0) pointers 1
  highlightCode on "code" line 0
}
{
  listElement "11" offset (50, 0)\
    from "10" NE pointers 1
  unhighlightCode on "code" line 0
  highlightCode on "code" line 1
}
{
  listElem "12" move (50, 0) pointers 1
  unhighlightCode on "code" line 1
  highlightCode on "code" line 2
}
{
  setLink "10" to "11"
  unhighlightCode on "code" line 2
  highlightCode on "code" line 3
}
{
  setLink "11" to "12" within 10 ticks
  unhighlightCode on "code" line 3
  highlightCode on "code" line 4
}
unhighlightCode on "code" line 4
```

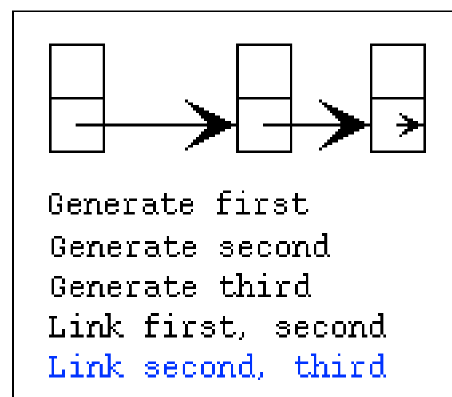

Figure 3: AnimalScript code for object placement

text to the standard output. Thus, *echo* is a very powerful help during animation development in pinpointing the precise reason why something did no go the way it was supposed to. Of course, *echo* can also be turned off. Figure 4 illustrates the scripting power of *echo*, assuming the same animation state as shown in figure 1. The resulting system output is shown in figure 5.

```
# show top right corner of left staff
echo location: offset (0,0) from "ls" NE
# group disk 1, 2 together
group "d1d2" "d1" "d2"
# show the bounding box of the two disks
echo boundingBox: "d1d2"
# show the ids of the composite object
echo ids: "d1d2"
# Finally, print a simple text
echo text: "Echo demo!"
```

Figure 4: The *echo* command (appended to code in figure 2)

```
# location: (45, 40)
# "d1d2" has bounds (105,110), (205,120)
# "d1d2" has ID(s) 13 14
# Echo demo!
```

Figure 5: Output of the *echo* command

In order to address the widest possible target audience with animations, all ANIMALSCRIPT text entries may be given either as a literal string or as a sequence of *localized texts*. In the latter case, the text component is placed in parentheses (), and each component is prefixed with the *language code* followed by a colon: (de: "Ja" en: "Yes" fr: "Oui"). On loading the animation, ANIMALSCRIPT prompts the user for the language to be used for the animation's display, and will layout all components accordingly. Due to the possibility of using relative coordinates, it is usually sufficient to generate *one* animation with multiple languages embedded. Additional extended features of ANIMALSCRIPT include *assertion checking, block charts* and *stop-and-think questions* as used in the *JHAVÉ* [2] system now also using ANIMALSCRIPT.

4 Dynamic Component Determination

While ANIMALSCRIPT already provides many helpful features for animation generation, users are bound to find some features they consider necessary or at least helpful which are not yet covered in the extended ANIMALSCRIPT functionality. For this purpose, we have designed ANIMALSCRIPT in a way that allows for easy extensions by other programmers.

ANIMALSCRIPT relies on a registration file listing all available components. This file may be located in the directory where ANIMALSCRIPT is called, or anywhere in the Java CLASSPATH. ANIMALSCRIPT will first look for the file in the current directory, allowing the user to have a unique registration file in each directory. The registration file simply lists the fully qualified Java class names containing ANIMALSCRIPT primitives, as shown in figure 6. The upcoming release of ANIMAL will also allow the user to change this registration file while the system is running.

```
animalscript.core.BaseAdminParser
animalscript.core.BaseAnimatorParser
animalscript.core.BaseObjectParser
animalscript.extensions.ArraySupport
animalscript.extensions.CodeSupport
animalscript.extensions.StackSupport
```

Figure 6: Example ANIMALSCRIPT registration file

When ANIMALSCRIPT is started, it reads in the registration file and tries to load the classes given in the file. Each class is tested for validity by checking whether it implements the `AnimalScriptInterface` and extends `animalscript.core.BasicParser`. This is necessary for correct handling of the following operations.

If the currently examined class is valid, ANIMALSCRIPT queries a hash table of keywords included in each class, which maps the keyword to the *method* name to invoke. The keywords extracted from the hash table are added to ANIMALSCRIPT's internal master hash table of handled keywords. In this hash table, the keyword is not mapped to a *method*, but to the *class* responsible for handling the keyword.

After the hash table is built, ANIMALSCRIPT starts parsing the input. Whenever a non-comment line is encountered, the first entry is extracted and interpreted as a keyword. ANIMALSCRIPT now looks for this key in its hash table, and if a fitting entry is found, passes the keyword along to the target class and its appropriate handler method for parsing. After the current line is fully parsed, the control returns to the main ANIMALSCRIPT class and the process continues until the whole input file has been parsed.

5 Adding New Primitives or Animation Effects

The main requirements for new classes can be summarized as follows:

1. make sure the new class *extends* the `BasicParser` class for parsing support and *implements* the `AnimalScriptInterface` method;

2. implement a number of methods for parsing the new commands, whether *administrative, graphic primitives* or *animation effects*,
3. initialize the class-specific hash table of handled keywords.

Adding new graphic primitives or animation effects requires a certain knowledge of Java and programming in general. ANIMALSCRIPT comes with an extensive set for parsing support methods, so that actually parsing the input data is comparatively easy to accomplish. This includes support for parsing keywords, optional parameters, comments, color specification, absolute or relative coordinates, and localized texts. Usually, each element in a new command definition requires one method call in the parsing support class.

Furthermore, we are currently working on implementing an auxiliary API that will generate the parser code from a given extended Backus-Naur-Form. Once this work is finished and made available on the Web, future programmers of ANIMALSCRIPT will only have to program the actual graphic operations and the display operations.

Internally, ANIMALSCRIPT uses three hash tables for keeping track of objects. The first hash table is used for mapping the *names* of objects to their internal numeric ID, and is especially helpful for the *swap* operations. The second hash table holds the types for all objects given by ID, and thus allows for easy access checking. For example, basically possible but unsupported operations such as rotating a text can be detected in this way. The third and most important hash table contains the *properties* of the individual objects and object types. This is also the place where *default* values for optional parameters are stored and retrieved. Thus, programmers also have to make sure that they update the three hash tables in their implementations. Note that the advanced features such as *relative positioning* work for **all** object types, so that implementors of new features need not take care of them.

6 Conclusions

We have provided an overview of the basic and advanced features of ANIMALSCRIPT. Apart from the "standard" operations common to today's visualization systems, ANIMALSCRIPT provides several features that support visualization debugging (`echo`), object positioning using relative placement, and localized international versions within a single animation.

The main advantages of ANIMALSCRIPT, however, lies in the easy approach taken for extending the functionality. This requires only a limited amount of programming skill and familiarity with Java. It is *not* necessary to read existing ANIMALSCRIPT code, let alone *modify* it to any extent. The programmer need only change a single line in a registration file to add or remove supported parse components. Once the BNF parser we currently develop is finished, generating new ANIMALSCRIPT support classes will become even easier.

Note: ANIMAL is available for download on the WWW at `http://www.informatik.uni-siegen.de/~roesslin/Animal/index.html`. ANIMALSCRIPT is already embedded in the current distribution of ANIMAL.

References

[1] Haajanen, J., Pesonius, M., Sutinen, E., Tarhio, J., Teräsvirta, T., and Vanninen, P. Animation of User Algorithms on the Web. *IEEE Symposium on Visual Languages* (1997), 360–367.

[2] Naps, T., Eagan, J., and Norton, L. JHAVÉ: An Environment to Actively Engage Students in Web-based Algorithm Visualizations. 31*st* *SIGCSE Technical Symposium on Computer Science Education, Austin, Texas 32*, 1 (Mar. 2000), 109–113.

[3] Rodger, S. JAWAA and Other Resources, 1997. Available at `http://www.cs.duke.edu/~rodger/tools/tools.html`.

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

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

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