Enhanced Expressiveness in Scripting Using AnimalScript 2

Guido Rößling, Felix Gliesche, Thomas Jajeh, Thomas Widjaja
Department of Computer Science / Dept. of Business Administration and Computer Science
Darmstadt University of Technology, Darmstadt, Germany

{guido, gliesche, jajeh, widjaja}@rbg.informatik.tu-darmstadt.de

Abstract

AnimalScript 2 is a new implementation of the visualization language AnimalScript used in the Animal system. The new implementation adds important features, especially conditional and loop statements. It also prepares the ground for further advanced components, such as methods or object templates. Several examples illustrate the expressiveness and ease of use of AnimalScript 2.

1 Introduction

One of the many different approaches for generating algorithm or program visualization content (abbreviated "AV" for the rest of this paper) is *scripting*. Here, the user provides a simple ASCII file containing commands that steer the visualization. The commands are usually held in plain English to make using the underlying scripting language easier. Typical examples for scripting-driven AV systems include JAWAA (Akingbade et al., 2003), JSamba (Stasko, 1998), and the *JHAVÉ* visualization environment with its visualization front-ends GAIGS, JSamba (Naps et al., 2000), and Animal (Rößling and Freisleben, 2002).

Scripting files are normally very easy to create manually. The user requires only a text editor and a certain familiarity with the scripting notation to become productive. Even better, it is relatively easy to modify existing code so that it generates scripting commands for visualization purposes while running the underlying program. Generating *some* working scripting code is normally rather easy. Writing a scripting code that presents a "good" visualization is more difficult. However, the same is true for *any* AV system that allows or forces the user to explicitly layout the visual components.

In this paper, we focus on the added capabilities to the original version of the scripting language AnimalScript provided by the Animal system (Rößling and Freisleben, 2001). To avoid confusion, we will always refer to the new implementation as AnimalScript 2, and use AnimalScript for the original implementation. We first review the main features of interest in the original scripting language and motivate why a new implementation was needed. The added features are then described in detail. The paper concludes with a short overview of the current implementation status and the goals we have set for the final version.

2 A Quick Overview of AnimalScript

Each AnimalScript animation consists of a single file with a set of lines. Each line can contain exactly one command or comment. To make parsing the files easier, each operation starts with a unique keyword. The parser can therefore determine the appropriate action by parsing the first keyword, although later parameters usually determine the actual action taken.

AnimalScript is parsed line-by-line. This means that once a given line is parsed, the appropriate animation commands are added to an Animal animation. Normally, each operation – whether declaration of a new object or animation effect – takes place in a separate animation step. If multiple operations shall take place in the same animation step, the user has to surround them with curly braces { } to indicate a block. Similarly to programming languages, the animation treats this block as a unit placed in the same step.

AnimalScript comes with built-in support for the graphical primitives `point`, `polyline` / `polygon`, `text` and `arc`. There are also specific commands for generating subtypes, such as

`squares, lines,` or `circles`. To enhance the use of ANIMAL for computer science education, the following common complex objects are also supported: `list elements` with an arbitrary number of points, `arrays` in either horizontal or vertical orientation, and `source / pseudo code` including indentation and highlighting.

Most commands have a set of optional parameters for setting specific properties. This includes simple settings, such as color or display depth. Arrows may be added at the beginning or end of a polyline. The user can also switch between polylines and polygons using the boolean *closed* property, using *closed=true* for polygons, and *closed=false* for polyline objects.

ANIMAL offers only a small selection of animation effects at first glance, limiting the operations to `show / hide, move, rotate` and `change color`. Each animation effect can work on an arbitrary set of animation objects at the same time. The expressive power of the scripting language becomes obvious when the set of options for the commands is reviewed. For example, a `move` can be made `to` a certain location, `along` an object defined inside the command, or `via` a previously defined object. The latter supports easy reuse of common move paths inside an animation, for example for sorting problems.

To further enhance the expressiveness of the scripting language, each object type can offer specific subtypes of a given animation effect. These are passed as an optional parameter to the standard animation effect (Rößling, 2001). For example, a polygon may offer the user the following `move` types:

- move the whole object,
- move a single node,
- move an arbitrary set of nodes,
- move the whole object except for a single node,
- move the whole object except for an arbitrary subset of nodes.

In this way, it is very easy to reach rather complex behavior based on a still simple notation. To further support animation authors, the computer science-based primitives also have their own set of commands. This especially concerns the following operations:

- Generating a group of source or pseudo code with user-specified font and color settings. As an exception to the general rule, each code line or line fragment is added as a separate component. This avoids exceedingly cluttered scripting code with several hundred characters in one line, and thus makes the script far easier to read;
- highlighting or unhighlighting a single line of code or a fragment thereof - for example, the boolean condition of a `for` loop;
- generating an array with user-defined font and color settings, either in horizontal or vertical orientation;
- installing an "array index pointer" with an optional label, useful for example to indicate an array position in sorting algorithms;
- putting values into the array and swapping array elements. The latter operation is animated automatically if a positive effect duration is specified;
- creating list elements with an arbitrary number of pointers at either the top, bottom, left or right side;
- resetting or setting a given list pointer. Here, the user can specify either a position or a target object. ANIMAL then figures out the appropriate way to handle the pointer based on the relative positions of the two objects.

Each AnimalScript object has a unique ID. Once the current line is parsed, the animation author can retrieve the current *bounding box* of the defined object, yielding the smallest rectangle that covers the whole object.

All AnimalScript coordinates can be specified in a number of different ways:

absolute coordinates give an explicit pair of (x, y) coordinates on the screen. To yield a visible object, x and y should be positive and within the display window borders;

locations can be defined once and reused as often as necessary;

relative coordinates are the most expressive and powerful option. Here, the location of a given object is determined based on other visible or hidden objects. Typically, the position is determined based on the *bounding box* of a given object by giving one of the eight compass directions or "center" and an (x, y) offset. Polyline or polygon objects also allow placement relative to a given *node*. Components can also be aligned to the *base line* of a text component. Finally, the location can also be defined as an offset from the `previous` coordinate.

A special `echo` command can be used for user feedback. Apart from simply printing a certain text to the command line or main window, the actual bounding box of a given object or set of objects can be retrieved, as well as individual objects and their IDs. In this way, if the layout on the screen does not match the author's expectations, some debug commands using `echo` can be integrated to figure out exactly *what* went wrong. Finally, objects can be grouped or ungrouped to save repeating the objects IDs for objects that are animated in the same way over several operations. It is important to note that a component inside a group can still be animated individually, *without* effect on the other group elements.

The syntax of AnimalScript, JAWAA and JSamba is roughly similar. AnimalScript uses String identifiers for objects instead of integers. Additionally, AnimalScript offers a fine-grained timing, compared to the "instant" or "animated" modes in JAWAA and JSamba. AnimalScript also boasts a far greater flexibility in animating and placing components, as outlined above.

In both JAWAA and JSamba, all parameters have to be given for all commands without an introducing keyword. Thus, commands start with a descriptive keyword, followed by a set of seemingly arbitrary values, typically of type integer. AnimalScript strictly requires keywords between most parameters, such as `color` or `depth`. At the same time, most parameters including their associated keyword are optional. This combination makes the underlying script easier to read, but also somewhat longer, than scripts for JAWAA or JSamba.

As can be seen from this overview, AnimalScript is rather powerful and expressive. However, there is one crucial drawback. As stated before, each line is parsed separately, as the context of the previous lines is retrieved from the Animal-internal animation object. Thus, many of the standard parsing concepts, such as *abstract syntax trees*, are not needed to parse AnimalScript animations. To make the implementation easier and more efficient, we took the ultimately unfortunate implementation decision to stay at a "single line parser". This brings one severe limitation: interesting components such as *loops* or *conditionals* can not be supported by the original AnimalScript parser.

To address this problem, we decided to re-implement the whole parser from scratch. This was also a good opportunity to clean up some the messier parts of the source. Compared to the former implementation with about 7500 lines of code, a team of three students of Business Administration and Computer Science was formed for this task.

3 Added Features in AnimalScript2

AnimalScript 2 is downward compatible to AnimalScript. That is, all commands which worked in AnimalScript will ultimately work in its successor. "Ultimately", because the

extent of the scripting language means that we had to restrict the amount of work we could tackle at one time.

The main goal of developing AnimalScript 2 is changing the line-based parsing approach to one based on abstract syntax trees. Apart from allowing components such as loops and conditionals, this will ultimately allow us to support method invocations. Currently, the most striking additions are the `while` and `for` loops and the `if` conditional with an optional `else` part. The additional `loop` construct iterates the loop body for a number of repetitions specified as an arithmetic expression. Apart from its use as a "shorthand notation", the `loop` construct is also helpful for beginners in programming. The trinary conditional operator `(booleanExpression) ? expression : expression` is not supported. The syntax for the entities follows the syntax used in Java and is shown in Listing 1. `intVarDecl` in the `for` construct stands for the initialization or declaration of a variable.

```
while (booleanExpression) #execute as long as expression is true
{
  command
}

for (intVarDecl; booleanExpression; arithmeticExpression) # as in Java
{
  command
}

loop (arithmeticExpression) # iterate exactly "expression" times
{
  command
}

if (booleanExpression) {
  command
}

if (booleanExpression) {
  command
}
else {
  command
}
```

Listing 1: Loops and conditional constructs

The body of each loop may also contain sub-blocks, just as in a "real" programming language. As shown in Listing 1, the curly braces can appear either on a new line or at the end of the current line. In contrast to C, C++ and Java, the curly braces *must* appear, even if the command body consists of only one command. Users familiar with a C-like syntax should find it easy to learn and effectively use the notation.

To support the loops appropriately, AnimalScript 2 introduces commands for handling arithmetic, boolean and String-based expressions. Arithmetic expressions currently cover the base operators `+, -, *, /` and `%` (modulo). This includes the precedence of multiplication and division over addition and subtraction, as well as parentheses.

AnimalScript 2 also supports integer variables, defined in the same way as in Java: `int nrIterations = 10`. Note that a semicolon can optionally be used to be even closer to the

notation employed in Java, C and C++. Integer variables can be assigned arbitrary (integer) expressions using the assignment operator, e.g. `nrIterations = 5 * i`.

Boolean variables are defined as in Java. They can be assigned either one of the two literals `true / false`, another boolean variable or an arithmetic expression with C semantics (0 is `false`, all other values are `true`). The boolean operators cover conjunction && and disjunction ||, the boolean comparison operators `==` and `!=`, and integer comparisons yielding a boolean result (using $<, <=, ==, >=, >$, and $!=$).

AnimalScript 2 also offers String variables, declared as `string myString = "Hello"` and assigned a new value in the usual way. Strings can be concatenated using the *Perl/PHP*-notation with a point in the middle. Thus, `myString . " world"` yields the String `Hello world`. The concatenation works on String, boolean and integer variables and literal Strings.

Due to the way String variables are expanded, even the names of variables can be generated dynamically. This is mainly useful in loops that generate individual objects with a unique variable name, for example `a1, a2, a3, ...`. A similar effect can be achieved in some scripting languages for programming, notable PHP and Perl, with operators such as `$$` (Lerdorf and Tatroe, 2002).

4 Example Use of AnimalScript 2

The source code shown in Listing 2 swaps the first element of an array with the minimum array value. It therefore constitutes a part of the *Straight Selection* sorting algorithm. We assume the presence of a method `swap` that can swap two elements on a given array.

```
int[] values = new int[]{3, 2, 4, 1, 7};
int pos = 1;
int minIndex = 0;
while (pos < values.length) {
  if (values[pos] < values[minIndex])
    minIndex = pos;
  pos++;
}
swap(values, 0, min);
```

Listing 2: Java code for swapping the minimal array element with the first array element

The original AnimalScript does not provide any loop support. Therefore, the structure has to be "flattened", resulting in something like Listing 3.

```
%Animal 1.4
array "values" (10, 10) length 5 "3" "2" "4" "1" "7"
arrayMarker "pos" on "values" atIndex 1 label "pos"
arrayMarker "minIndex" on "values" atIndex 0 label "minIndex"
moveMarker "minIndex" to position 1 within 5 ticks
moveMarker "pos" to position 2 within 5 ticks
moveMarker "pos" to position 3 within 5 ticks
moveMarker "minIndex" to position 3 within 5 ticks
moveMarker "pos" to position 4 within 5 ticks
moveMarker "pos" to outside within 5 ticks
arraySwap on "values" position 0 with 3 within 10 ticks
```

Listing 3: Example animation code in the original AnimalScript

The script in Listing 3 is easy to read but very hard to understand, as the semantics of the array operations are hidden. In essence, the reader has to build his own hypothesis why the markers change, and what the actual underlying algorithm is. Listing 4 shows the same code, implemented in AnimalScript 2.

```
%Animal 2.0
array "values" (10, 10) length 5 int {3, 2, 4, 1, 7}
int pos = 1
int minIndex = 0
arrayMarker "pos" on "values" atIndex pos label "pos"
arrayMarker "minIndex" on "values" atIndex minIndex label "minIndex"
while (pos < 5) {
  if (values[pos] < values[minIndex]) {
    minIndex = pos;
    moveMarker "minIndex" to position pos within 5 ticks
  }
  pos = pos + 1
  moveMarker "pos" to position pos within 5 ticks
}
arraySwap on "values" position 0 with minIndex within 10 ticks
```

Listing 4: Example animation code in AnimalScript 2

At first glance, the code shown in Listing 4 is longer than the code in Listing 3 (15 lines versus 11 lines of code). It is easy to see that this depends on the actual array: the number of code lines are fixed for both the Java and the AnimalScript 2 listing. For AnimalScript, the number of code lines depends on the array length and the ordering of the elements.

There is a strong similarity between the Java code in Listing 2 and AnimalScript 2. The script contains twelve lines of effective AnimalScript 2 code, if we ignore lines 1, 11 and 14. Six lines of assignments, conditional and loop are identical or almost identical. The mapping from the Java array declaration to AnimalScript 2 is also easy. The main changes concern the commands for installing visible array position markers and moving them in concert with the value assignments. The complete animation code for *Selection Sort* in AnimalScript 2 is shown in Listing 5. The equivalent AnimalScript notation contains 34 lines of code.

```
%Animal 2.0
array "values" (10, 10) length 5 int {3, 2, 4, 1, 7}
int pos = 0
int spos = 0
int minIndex = 0
arrayMarker "pos" on "values" atIndex pos label "pos"
arrayMarker "spos" on "values" atIndex spos label "pos"
arrayMarker "minIndex" on "values" atIndex minIndex label "minIndex"
while (pos < 4) {
  minIndex = pos
  for (spos = pos + 1; spos < 5; spos = spos + 1) {
    if (values[spos] < values[minIndex]) {
      minIndex = spos;
      moveMarker "minIndex" to position spos within 5 ticks
    }
  }
  arraySwap on "values" position pos with minIndex within 10 ticks
  pos = pos + 1
  moveMarker "pos" to position pos within 5 ticks
}
```

Listing 5: Example selection sort animation code in AnimalScript 2

As can be seen, the new features are very helpful for array-based algorithms, such as sorting or searching. They significantly reduce the cognitive effort of coding "programs" into visualizations. Loops and conditional can of course also be used for other programs.

Compared to the original implementation of AnimalScript, the new implementation can offers significant run-time performance advantages. This is especially true for programs that use the new conditional or loop statements. For example, the implementation of *Selection Sort* shown in Listing 5 only has to parse 20 lines from the file. The iterative version implemented for AnimalScript has to parse 34 lines of code. Any reduction of file I/O, especially concerning parsing operations, can greatly improve the run-time, even when buffered streams are employed.

Additionally, the look-up mechanism for object position determination in AnimalScript 2 is faster than in the original AnimalScript. In the original version, the animation had to be fast-forwarded to the "current point in time" to accurately determine the bounding box of a given element. Relative placement commands used in defining new objects or within animation effects were therefore very time-consuming to evaluate. In the new version, the look-up is significantly faster thanks to the (hidden) tree structure used for storing and evaluating the animation.

At the moment, we cannot provide conclusive run-time measurements, as the implementation of the array operation visualizations has not been fully implemented. We plan to do a more extensive evaluation once the implementation of the parsing and execution are completed.

5 Summary and Further Work

AnimalScript 2 is a re-implementation of the scripting language AnimalScript (Rößling and Freisleben, 2001). The previous line-based parser is replaced by a parse tree. The main change that is visible to users are the important base operations for simplifying animation creation: loops, conditionals, variables and expressions.

The additions considerably increase the expressiveness of AnimalScript, pushing it closer to a full-fledged programming language with visualization. This simplifies manual generation, for example of sorting algorithms. We plan to evaluate the effects on (semi-)automatic generation once the implementation is finished. As the user is not required to use the new commands, AnimalScript 2 is at least not "more difficult" to learn and use than the original release.

All new components can be parsed and evaluated. Some of the older (and not very well documented) advanced features of the original scripting language are missing and placed on hold for more important content. This includes importing several scripting files into a single animation and internationalization aspects. Additionally, the support for pre-defined *locations* is still under development.

The team is currently working on getting all object generation commands set up. While this task is per se relatively simple, the size of the Animal system with 216 classes and about 45000 lines of code has to be taken into account. Becoming familiar with all components and their interplay is hardly trivial, as can be seen when studying the reference work (Rößling, 2002). Currently, all additional features can be parsed, evaluated and executed, apart from the occasional bugs to be expected in any significant software project.

Due to the complete redesign and reimplementation of the parsing process, the new version of the scripting language is ready for other advanced extensions. This includes method definitions and blocks that define author-specific objects based on a set of primitives. Due to the size of the implementation team and the other demands on their time, not all goals are realistic - this is only a one-year project without payment!

Note that Animal itself still offers the graphical drag-and-drop user interface. Thus, Animal still supports beginners, but has extended its support for "expert" programmers using AnimalScript 2.

Once the implementation is finished, the new release of AnimalScript 2 will be available online under `http://www.animal.ahrgr.de`. A set of examples will also be available there, as there was just too little space in this paper for more.

References

Ayonike Akingbade, Thomas Finley, Diana Jackson, Pretesh Patel, and Susan H. Rodger. JAWAA: Easy Web-Based Animation from CS 0 to Advanced CS Courses. In *Proceedings of the 34^{th} ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2003), Reno, Nevada*, pages 162–166. ACM Press, New York, 2003.

Rasmus Lerdorf and Kevin Tatroe. *Programming PHP*. O'Reilly & Associates, Sebastopol, CA, 2002. ISBN 1-56592-610-2.

Thomas Naps, James Eagan, and Laura Norton. JHAVÉ: An Environment to Actively Engage Students in Web-based Algorithm Visualizations. *Proceedings of the 31^{st} ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2000), Austin, Texas*, pages 109–113, March 2000.

Guido Rößling. Algorithm Animation Repository. Available online at `http://www.animal.ahrgr.de/` (seen August 14, 2004), 2001.

Guido Rößling. Animal-Farm*: An Extensible Framework for Algorithm Visualization*. PhD thesis, University of Siegen, Germany, 2002. Available online at `http://www.ub.uni-siegen.de/epub/diss/roessling.htm`.

Guido Rößling and Bernd Freisleben. AnimalScript: An Extensible Scripting Language for Algorithm Animation. *Proceedings of the 32^{nd} ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2001), Charlotte, North Carolina*, pages 70–74, February 2001.

Guido Rößling and Bernd Freisleben. Animal: A System for Supporting Multiple Roles in Algorithm Animation. *Journal of Visual Languages and Computing*, 13(2):341–354, 2002.

John Stasko. Smooth Continuous Animation for Portraying Algorithms and Processes. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization*, chapter 8, pages 103–118. MIT Press, 1998.