Enhanced Expressiveness in Scripting Using AnimalScript V2

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

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

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. Typically, the commands are 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), *JHAVÉ* (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. While 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 built into the Animal system (Rößling and Freisleben, 2001). To avoid confusion, we will always refer to the new implementation as AnimalScript V2, 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 currently 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 only has to envelop them in 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 display properties, such as the outline and fill color (for closed objects) or the display depth.

The scripting language 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, *moves* 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. The precise notation is defined at (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 make things even easier for 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 has to be added as a separate component. This avoids exceedingly cluttered lines of several hundred characters and makes the scripting 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 usually unique ID. If IDs are reused, accessing the previous holder of the ID is no longer possible. However, the object can still be removed from the display using the *hideAll* command. 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.

To improve the animation layout, 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* location used.

There is also a special "echo" command which 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.

As can be seen from this overview, AnimalScript is rather powerful and expressive. However, there is one crucial drawback. As stated before, each line can be 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 more efficient and easier, 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 single person implementation, which already contained about 7500 lines of code, a team of three students of business administration was formed for this task.

3 Added Features in AnimalScript V2

AnimalScript V2 will be 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 some compromises in the amount of work we could heap on the students is limited. The current state of implementation is described in the next section. Here, we will focus on the added components of AnimalScript V2.

The main goal of developing AnimalScript V2 was to change the line-based parsing approach to one based on abstract syntax trees. Apart from allowing components such as loops and conditionals, this ultimately will also allow us to support method invocations. Currently, the two most striking additions are the *while, for* and *loop* loops and the *if* conditional with an optional *else* part. The syntax for the three entities follows the syntax used in Java, except that the trinary operator *(boolean expression) ? expression : expression* is not supported.

The structure for the loops is shown below, where *command* is a place-holder for a set of commands. Note that the set of commands may also contain subblocks, just as in a "real" programming language. Thus, users familiar with C-like syntax should find it easy to learn and effectively use the notation. The hash mark # introduces a comment spanning the rest of the line.

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

```
{
  command
}

for (init; booleanExpression; arithmeticExpression) # complex loop
{
  command
}

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

Similarly, the notation of the *if / if ... else* conditionals is identical to that found in C, C++, and Java.

To support the loops appropriately, we also had to introduce commands for handling arithmetic and boolean expressions. *Arithmetic* expressions currently cover the four base operators *+, -, *, /*. This also includes the precedence of multiplication and division over addition and subtraction, as well as parentheses.

To allow for useful loops, we introduced integer variables, which are defined in the same way as in Java except for the missing semicolon, e.g. *int nrIterations = 10*. 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 ||, as well as the boolean comparison operator == and integer comparisons yielding a boolean result (using $<, <=, ==, >=, >$ and $!=$).

Finally, AnimalScript V2 also offers *String* variables. They are declared as *string myString = "Hello"* and assigned a new value in the usual way. Additionally, they can be concatenated using the Java-notation with a point in the middle. Thus, *myString . " world"* yields the String *Hello world*. The concatenation works on String, boolean and integer variables as well as literal Strings.

4 Conclusions and Further Work

AnimalScript V2 extends the functionality of the scripting language AnimalScript (Rößling and Freisleben, 2001) by offering important base operations for simplifying animation creation: *loops, conditionals, variables* and *expressions*.

The additions considerably increase the expressive power of AnimalScript, pushing it closer to a full-fledged programming language with visualization. This makes manual generation, for example of sorting algorithms, far easier. We will evaluate the effects on (semi-)automatic generation once the implementation is finished.

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.

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 - getting 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. We hope that we will have a stable version of the new scripting language in time for PVW 2004.

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!

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.

Thomas Naps, James Eagan, and Laura Norton. JHAVÉ: An Environment to Actively Engage Students in Web-based Algorithm Visualizations. 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 July 14, 2003), 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. 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.