A Java API for Creating (not only) AnimalScript

Guido Rößling, Stephan Mehlhase, Jens Pfau
CS Department, TU Darmstadt
Hochschulstr. 10
64289 Darmstadt, Germany

`roessling@acm.org`

Abstract

Generating animation content can be tedious and result in "messed-up" code. We present a Java API that can be used for generating AnimalScript-based animations. It was designed to be extendable to other output formats, such as SVG or other scripting languages. Apart from describing the use of the API, we also show a concrete example that was generated using the API to illustrate the API's expressiveness.

1 Introduction

Creating AV content is often a slow and tedious job. Automating this process offers faster - and reusable - generation of new animation content. However, the automation requires either an underlying language notation or a (hopefully well-designed) API, or both. If present, such features can make the content provider's job much easier, and may even allow end-users to create visualization content on-the-fly.

In this paper, we present a Java API for creating animation content which uses AnimalScript as the main output. However, the API can also easily be extended to support other notations or languages, making it an attractive tool for many AV content generators, including those that do not use the Animal AV system.

In the following, we will first briefly describe the underlying AnimalScript language. In Section 3, we then introduce the design for the Java API, followed by an example animation generated using the API. Sections 5 and 6, respectively, present a brief informal evaluation and comments on extending the API to other notations. Section 7 summarizes the work and outlines areas of future research.

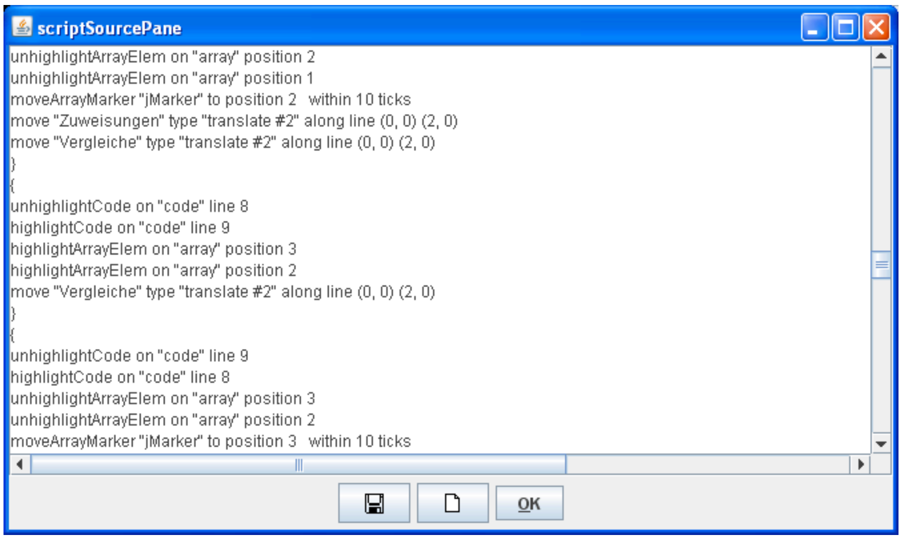
2 A Brief Overview of AnimalScript

AnimalScript (Rößling et al., 2004) is a versatile notation for programming algorithm visualization and animation (AV) content. AnimalScript offers a very flexible placement of elements, using either absolute values or an offset relative to another object's bounding box, text baseline, or an individual node in a polygon or polyline. Almost all animation effects can be assigned a relative starting time as an offset from the beginning of the associated animation step and a duration, which can be specified using either milliseconds or the number of animation frames. Figure 1 shows Animal's built-in editor for AnimalScript code.

Animation effects can also be given a concrete animation "method" as a String parameter, which further describes how the affected object(s) shall be animated. For example, the generic *move* command can be given a method name *translate #2*, which will change the behavior from moving the complete object to moving only the second node of the object(s).

Similarly to all other scripting notations, AnimalScript can easily be entered with a standard text editor. We have recently completed an Eclipse plugin for editing AnimalScript, including code assist and the check for syntax errors (Rößling and Schroeder, 2008).

AnimalScript contains a large selection of optional components in almost all commands, which makes it easier to write for hand-coded scripts. AnimalScript can also be automatically dumped while the underlying algorithm is being executed. However, just as for all other scripting-based approaches, this quickly leads to cluttered code for the actual algorithm. Figure 2 shows an example; here, the actual algorithm code is almost totally obscured by the manually generated visualization code. It may take an "expert" to see that this piece of code actually calculates the step width for Shell Sort.



```
unhighlightArrayElem on "array" position 2
unhighlightArrayElem on "array" position 1
moveArrayMarker "jMarker" to position 2  within 10 ticks
move "Zuweisungen" type "translate #2" along line (0, 0) (2, 0)
move "Vergleiche" type "translate #2" along line (0, 0) (2, 0)
}
{
unhighlightCode on "code" line 8
highlightCode on "code" line 9
highlightArrayElem on "array" position 3
highlightArrayElem on "array" position 2
move "Vergleiche" type "translate #2" along line (0, 0) (2, 0)
}
{
unhighlightCode on "code" line 9
highlightCode on "code" line 8
unhighlightArrayElem on "array" position 3
unhighlightArrayElem on "array" position 2
moveArrayMarker "jMarker" to position 3  within 10 ticks
```

Figure 1: Example AnimalScript code

```
int oldDist = -1;
sb.append("  text \"dist\" \"dist:\" offset (40, 20) from \"array\" E");
for (dist = 1; dist < a.length / 9; dist = 3 * dist + 1) {
  toggleStep();
  if (oldDist != -1) {
    sb.append("  hide \"dist").append(oldDist).append("\"");
  }
  sb.append("  text \"dist").append(dist).append("\" offset (10, 0)");
  sb.append(" from \"dist\" baseline");
  oldDist = dist;
}
```

Figure 2: (Bad) Example of integrated AnimalScript code generation

Before our research presented in this paper, several content authors had already recognized this situation as bad, and developed their own (specialized) "mini-API" to deal with exactly those contents they needed. However, the limited expressiveness of these APIs and (sometimes) their lack of a good software engineering architecture prevented other content authors from adopting any of these APIs. We therefore set out to develop an "official" Java API for generating AnimalScript code that should be easy to use by current and future content authors. In Section 3, we will outline the basic approach for this API.

3 AnimalScript API Design

The main design goals for the AnimalScript Java API were the following:

- Provide a "cleaner" way of generating AnimalScript code than shown in Figure 2,
- Offer a "common" API that all content generators for Animal can use,
- Use a clean inheritance hierarchy using both classes and interfaces as appropriate,
- Support the current language features of AnimalScript in the Java API,
- Provide means to extend the API for future additions to AnimalScript,

- Simulate the "optional elements" inside AnimalScript in a sensible way.

We decided to use one comparatively large *Factory* class (Gamma et al., 1995) called "Language". This abstract class provides a large set of methods for creating new objects of the proper type, e.g. there is a method called *newPoint* for creating a new *Point* object. In fact, there is often more than only one way to create a given object, using a different set of creation parameters. In this case, all such Factory methods will usually be mapped to a single (abstract) method. The concrete implementation of these methods is left to the implementing subclass of class *Language*.

It quickly became obvious that the good software engineering decisions for the third bullet point also opened the door for alternative output languages. Therefore, we decided to provide two "layers" of the API. The outer, abstract layer offers all interesting functions to the programmer, but uses an underlying inner, concrete layer to actually map the functions to output commands. In this way, it is easy to implement support for an alternative output format such as *SVG* (Ferraiolo, 2003). The switching between output instances is decided when the concrete instance of the *Language* Factory is created by providing one of the possible implementing subclasses, as shown in Listing 1.

```
// Generate a new Language instance for content creation
// Parameter: Animation title, author, width, height
Language lang = new ConcreteLanguage("Quicksort Animation",
          "Name of the author", 640, 480);
// Activate step control
lang.setStepMode(true);
```

Listing 1: Example for creating a new *Language* instance

3.1 Step control

The user can determine if all operations should be executed in sequence, or whether some operations may be grouped together and will be performed in parallel. This is achieved by turning the "step mode" on or off, respectively. If the step mode is turned on, a new step has to be introduced by a call to the *nextStep* method in class *Language*, which can also be passed an *int* value for a delay between consecutive steps, measured in milliseconds. An optional String parameter is used for the hypertext-like "table of contents" shown in Figure 3.

3.2 Defining Graphical Objects

The API for creating graphical objects is based on the definitions used by AnimalScript and the "standardized" XML of an ITiCSE 2005 Working Group (Naps et al., 2005), also found in the *XAAL* system (Karavirta, 2005). Essentially, almost all objects require the following creation parameters: location, value, name, display options and visual properties.

The location specifies the placement of the object and can be either absolute or relative to an other object or object node. The value depends on the type of the object, and can for example by an int array for an object of that type (see Listing 2 for an example of creating a visual object). The display options can be used to declare the object as *hidden* (not visible) or specify a delay after which the object should become visible. Finally, the visual properties describe the outward appearance of the object, such as its color. The visual properties will be examined in more detail in Section 3.3.

```
// Create a new int[] object (will normally exist before)
int[] arrayContents = new int[] { 1, 3, 7, 5, 2, 6, 8, 4 };
// Parameters: location, value, name, display opt., visual props.
IntArray array = lang.newIntArray(new Coordinates(10, 30),
      arrayContents, "array", null, arrayProps);
```

Listing 2: Example for creating a new Graphical Object, here a visual int array

The API supports a large set of graphical objects: points, squares, triangles, rectangles, polygons, lines, polylines, circles and ellipses (including segments thereof), and text elements. It also supports many of the most relevant data structures used in Computer Science, such as graphs, arrays and matrices with a base type of *int* or *String*, code blocks including indentation, list elements with an arbitrary number of pointers, and three variants each of stacks and queues (conceptual, list- and array-based). All objects can be created with a single method invocation similar to lines 4-5 in Listing 2.

3.3 Defining Visual Properties

Apart from defining a graphical object, such as the *IntArray* shown in Listing 2, the user should also be able to define the object's visual appearance. The typical approach for this is to use either constructor invocation arguments, for example passing in the color of the object, or explicit API invocations to set the values after the "basic" object was created.

For many of the more complex objects, both of these approaches can be cumbersome due to the following reasons:

- Passing in the concrete values to the constructor can result in a bad design or usage issues. If all values are mandatory, that is, there is only one appropriate constructor, the number of parameters needed may become very large. For example, the IntArray in Listing 2 already has color values for its *outline, elements, cell background, element highlight* and *cell highlight*, the latter two of which are used if the user's attention is to be drawn to certain elements or cells. A user who simply wants to create an IntArray now has to worry about five colors (plus the font settings etc.) - or may leave them *null*, which may or may not lead to other problems.

  If the designer tries to help the user by allowing individual colors to be dropped from the list, the API will quickly have a large set of different IntArray constructors, which may also confuse users.

- The user may assign values to the five colors mentioned in the previous item by invoking one API method per color. This leads to a rather broad API with many simple (and similar) methods, as well as to about six additional method invocations - and thus, six more lines of Java code - to get the colors and fonts "right".

- Visual settings cannot easily be reused by either of the two previous approaches. For the second approach, the user could write a method that calls the appropriate API methods with the same settings. However, that still means additional work for the user.

We have therefore decided to follow an approach closer to *Cascading Style Sheets*. Here, the user can define a given combination of visual attributes once, and then reuse it as often as he likes. For example, to create ten text objects that have exactly the same visual settings, the user would first describe the visual properties once, and then pass these visual properties as a constructor parameter to the ten text objects. Listing 3 gives a brief example of how the visual properties for the IntArray object in Listing 2 can be specified. Note that the properties defined in Listing 3 are already used in Listing 2 in line 5.

```
1 // create array properties with default values
2 ArrayProperties arrayProps = new ArrayProperties();
3 // Redefine properties: border black, filled with gray
4 arrayProps.set(AnimationPropertiesKeys.COLOR_PROPERTY, Color.BLACK);
5 arrayProps.set(AnimationPropertiesKeys.FILLED_PROPERTY, true);
6 arrayProps.set(AnimationPropertiesKeys.FILL_PROPERTY, Color.GRAY);
```

Listing 3: Specifying visual properties (here, for array objects) once

On creation of a new visual properties object, as shown in Listing 3, all possible properties for the object are already set to a default value. Therefore, the user only has to overwrite those settings that he wants to adjust. For example, the array properties actually include 11 different properties.

3.4 Animating Graphical Objects

Once a graphical object was defined, it can be animated. The supported animation methods are invoked directly on the underlying graphical object. The parameters required for the animation effects depend on the type of effect chosen. However, they will usually include the following information:

- a *method name* for specifying subtypes of a given animation effect, as "translate #2" used in Figure 1. For example, a *color change* may concern the array's border or its fill color. The method name specifies which of these is actually meant to change;
- an *offset* relative to the start of the current animation step, which is usually 0 to indicate an immediate start;
- and a *duration.* Both offset and duration can be measured in animation frames or *ms.*

4 Example Animation Generated Using the Java API

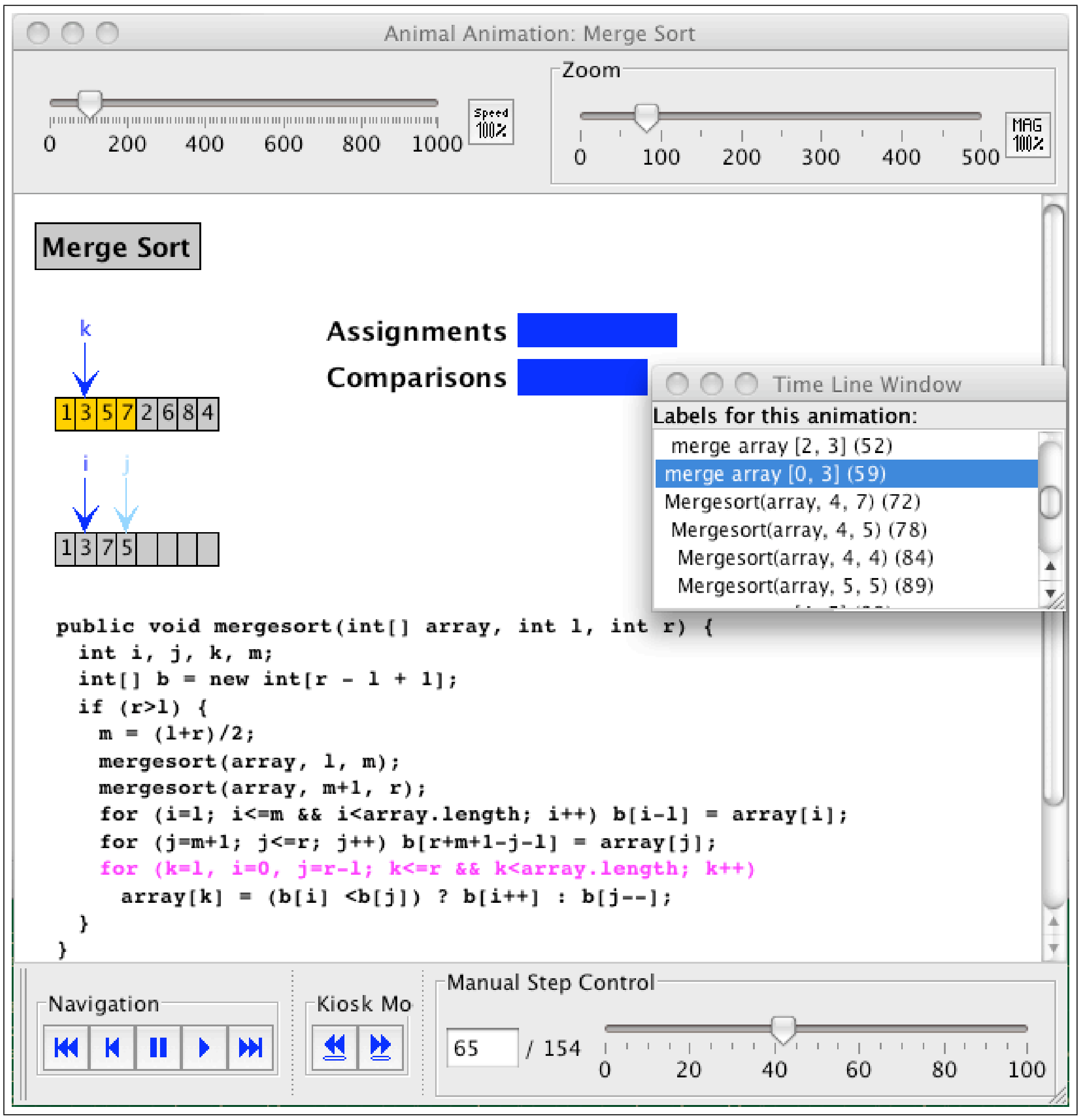
Figure 3 shows an example of a generated animation. The window contains the standard Animal controls for speed and zoom at the top. The animation can be navigated flexibly in both directions and also includes a "kiosk mode", which will display the animation step by step. The user can also jump ahead in the animation by entering the target step or dragging the slider shown on the bottom right. The window overshadowing the animation display contains the labels assigned to the animation, allowing instant access to the associated animation step. In the example, we started from step 59 which starts the merge operation of the first four array elements, and have now reached step 65.

The graphical objects shown in Figure 3 include a boxed text for the title and two int arrays defined similarly to Listing 2. The visual properties used for the arrays are identical to those described in Listing 3. The array markers *i, j,* and *k* are directly installed on the array. They can be moved to another index (using either a "jump" of no duration or a timed "move"), and can also automatically update their position if a change in a cell value occurs, for example by putting a larger number into an array cell.

The source code shown in the example animation is created with a set of API method invocations. The code indentation can be specified separately for each line. Animal will figure out the actual indentation to be used according to the font used for the code lines. The code also highlights the current code line in a configurable color (here, violet).

Finally, the two "counter boxes" for the number of assignments and comparisons also use the AnimalScript API. As there is no "counter box" feature in the API, we instead use a filled rectangle for each box. Whenever an assignment or comparison is made, the associated box is stretched by moving the end point by two pixels per assignment or comparison, as stretching the box by a single pixel would be difficult to see on many larger displays.

Listing 4 shows an excerpt of the concrete code for performing the first two parts of the merge process, when the sorted elements of the left and right subarray are copied into the temporary array. We do not expect the reader to fully understand the code as it is portrayed in this listing, but the main approach should hopefully be understandable. In line 1, we start a new step that is also provided with the "merge array [l, r]" information shown in the list of assigned labels. The parameters *l, r* indicate the current subarray bounds, while *depth* represents the recursion depth and determines the number of spaces to indent the label.



Labels for this animation:

```
merge array [2, 3] (52)
merge array [0, 3] (59)
Mergesort(array, 4, 7) (72)
Mergesort(array, 4, 5) (78)
Mergesort(array, 4, 4) (84)
Mergesort(array, 5, 5) (89)
```

65 / 154

```
public void mergesort(int[] array, int l, int r) {
  int i, j, k, m;
  int[] b = new int[r - l + 1];
  if (r>l) {
    m = (l+r)/2;
    mergesort(array, l, m);
    mergesort(array, m+1, r);
    for (i=l; i<=m && i<array.length; i++) b[i-l] = array[i];
    for (j=m+1; j<=r; j++) b[r+m+1-j-l] = array[j];
    for (k=l, i=0, j=r-l; k<=r && k<array.length; k++)
      array[k] = (b[i] <b[j]) ? b[i++] : b[j--];
  }
}
```

Figure 3: Screenshot from a generated animation

Lines 3-4 (as well several other lines in the listing) adjust the counters for the number of assignments and comparisons. In lines 3-4, these are for the initialization and condition of the *for* loop in line 6-12. Inside the loop, the values from the “main” array *array* are copied into the “helper” array *bArray*. The *null* parameters indicate that the effect happens instantaneously without duration or delay. Lines 13 and 25 toggle the display of the current code line from the first to the second and from the second to the final loop shown in Figure 3.

```
 1  lang.nextStep(createMergeLabel(l, r, depth)); // provide a label!
 2
 3  incrementNrAssignments(); // i = l in init of foor loop
 4  incrementNrComparisons(2); // i <=m, i < array.getLength() in for
 5                                   // copy first subarray
 6  for (i = l; i <= m && i < array.getLength(); i++) {
 7    bArray.put(i - l, array.getData(i), null, null); // copy value
 8    incrementNrAssignments();        // counts as one operation
 9    bArray.unhighlightElem(i - l, null, null); // unhighlight
10    incrementNrAssignments();        // i++ in for loop
11    incrementNrComparisons(2);       // comparisons in for loop
12  }
13  code.toggleHighlight("copyLeftside", "copyRightside");
14  lang.nextStep();                   // change step
15
16  incrementNrAssignments();          // j = m + 1 in for
17  incrementNrComparisons();          // j <= r in for loop
18  for (j = m + 1; j <= r; j++) {     // copy second subarray
19    bArray.put(r + m + 1 - j - l, array.getData(j), null, null);
20    incrementNrAssignments();
21    bArray.unhighlightElem(r + m + 1 - j - l, null, null);
22    incrementNrAssignments();
23    incrementNrComparisons();
24  }
25  code.toggleHighlight("copyRightside", "loop");
26  lang.nextStep();                   // next step: merge in loop
```

Listing 4: A subset of the code used for MergeSort

5 API Evaluation

The API presented in this paper is currently used by roughly 120 different algorithm animation content generators. To be fair, many of these generators only differ by nuances, such as the language used for textual output or for program code (e.g., Java versus pseudo code). About 45 of these generators previously used a generation approach similar to the “interwoven” code in Figure 2. Changing these to the API represented a fair amount of work, but was definitely useful, as the modified code is now far more readable and also far shorter.

Currently, a set of students is working with the Java API in a lab about algorithm visualization. We expect to be able to provide more information about their experiences for the final submission to the Workshop, and certainly for the presentation in Madrid, as our summer term has just started two weeks before the submission deadline. The first feedback is positive, claiming that is easy to start working with the API based on the initial English slides we provide at the ANIMAL home page (Rößling, 2008) under “Downloads”.

6 Extending the API to other output languages

Extending the API to other output languages is very easy. The programmer first creates a new Java package for the new output language. A new base factory implementation of the abstract *Language* Factory then has to be created and implemented. Tools like Eclipse can help by automatically filling in the methods that have to be implemented; this is currently a set of 26 Factory methods for the objects listed in Section 3.2.

The implementation of the Factory methods then usually requires the creation of additional classes representing the created object, for example a *SVGIntArray* representing an *int[]* in SVG. This class only has to map the existing *IntArray* object into SVG; it does not have to provide any internal representation of the array or other "business logic".

The implementation for the AnimalScript language currently consists of 30 Java classes with a total of 6,336 lines (including all empty lines, comments, package and import statements). By far the largest class is the base Factory class with about 900 lines, many of which are either blank (87) or contain *import* (90) statements or *JavaDoc* comments (344). When we also disregard the declaration of methods, less than 300 lines of actual code are left.

7 Summary and Future Work

We have presented the basic design and use of the AnimalScript Java API. The API is a large help in separating the algorithm from the visualization code. It is easy to use, once the programmer has understood the concept of the *visual properties*, and highly expressive.

In the future, we want to extend the API to other output languages, especially the XML code defined by the ITiCSE 2005 Working Group (Naps et al., 2005), also used in the *XAAL* system (Karavirta, 2005). Other target formats include *SVG* and potentially *ActionScript* (used for Adobe Flash). We would also like to offer the API to any interested party; it can be freely downloaded (Rößling, 2008). This especially concerns AV content creators and the authors of other systems, who may be interested in adopting the API.

References

Jon Ferraiolo. Scalable Vector Graphics (SVG) 1.1 specification. `http://www.w3.org/TR/SVG`, September 2003.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

Ville Karavirta. XAAL - Extensible Algorithm Animation Language. Master's thesis, Department of Computer Science and Engineering, Helsinki University of Technology, 2005.

Thomas Naps, Guido Rößling, Peter Brusilovsky, John English, Duane Jarc, Ville Karavirta, Charles Leska, Myles McNally, Andrés Moreno, Rockford J. Ross, and Jaime Urquiza-Fuentes. Development of XML-based Tools to Support User Interaction with Algorithm Visualization. *SIGCSE Bulletin inroads*, 36(4):123–138, December 2005.

Guido Rößling. Animal2 Home Page. WWW: `http://www.algoanim.info/Animal2`, 2008.

Guido Rößling and Peter Schroeder. Animalipse - An Eclipse Plugin for AnimalScript. In *Proceedings of the Fifth Program Visualization Workshop, Universidad Rey Juan Carlos, Madrid, Spain*, page (in print), July 2008.

Guido Rößling, Felix Gliesche, Thomas Jajeh, and Thomas Widjaja. Enhanced Expressiveness in Scripting Using AnimalScript V2. In *Proceedings of the Third Program Visualization Workshop, University of Warwick, UK*, pages 15–19, July 2004.