Using ANIMALSCRIPT to Generate Animations*

Dipl.-Inform. Guido Rößling
FB 12, Parallel Systems
University of Siegen
Hölderlinstr. 3
D-57080 Siegen

EMail: roessling@acm.org

April 6, 2000

Contents

*Covers ANIMAL Release 1.4

1 Introduction

ANIMAL is a compact, efficient and easy to use animation tool developed at the University of Siegen starting in 1998. The motivation for developing ANIMAL came from a lack of satisfying tools for animations that are both easy to generate and edit and helpful in lectures.

ANIMAL is an acronym for A NEW INTERACTIVE MODELER FOR ANIMATIONS IN LECTURES, which already shows the main area of usage for ANIMAL.

ANIMAL's strengths lie mainly in the following areas:

- very easy to use graphic interface incorporating *drag and drop*,
- animations can be automatically generated using either the scripting language ANIMALSCRIPT or the ANIMALGENERATOR API,
- platform neutral implementation in Java,
- high flexibility.

On first using ANIMAL , the program might appear somewhat limited in the animation effects as well as the primitive graphic types offered. Other programs seem to offer a far more operations at first glance.

However, a closer look will quick reveal that this impression is not necessarily accurate. Some other programs tend to offer the same basic effect not as a configurable effect, but rather as a collection of effects. ANIMAL offers highly flexible, adaptable effects that can usually perform the same other animation programs also offer - although the list of "supported effects" at first seems much shorter. Check out section 4 on page 6 for an in-depth description of the basic effects available and what operations can be done by using them wisely.

Finally, the high flexibility mentioned above comes from the simple fact that practically all operations and objects ANIMAL uses are easy to edit and adapt to specific needs. Furthermore, the available object or animation effect types are not coded directly into the implementation. Thus, developers should find it very easy to add extensions to ANIMAL without having to touch much existing code.

Using the new additions to ANIMAL, it is very easy to generate interesting animations to enhance viewers' motivation and understanding of the topics presented.

ANIMAL is freely available on the *World Wide Web*, complete with a extensive collection of animations including screen shots and description. Check out the official ANIMAL Home page at **Home Page**

`http://www.informatik.uni-siegen.de/~inf/Animal/index.html`

for more information.

2 Requirements

Java

ANIMAL is implemented completely in Java. This means several things, one of which being that the performance is - alas - at times somewhat slower than one might wish.

It also means, however, that the *hardware / software* requirements for using ANIMAL are very easy to specify: your computer must be able to run Java. This is something that practically *all* current computer models can do, although with a wide difference in performance. We recommend a moderately modern computer with a "enough" RAM, since *Java* itself is somewhat memory-intensive on most platforms. For example, ANIMAL runs smoothly on my laptop (Pentium II Mobile, 300 MHz, with 64 MB RAM).

JDK: 1.1.5+
Swing 1.1.1

As stated above, ANIMAL is implemented completely in *100% Pure Java*, using SUN's *JDK 1.1.5* or above. The only further requirement is Swing , the graphic package collection that provides a far more attractive alternative to Java's original *Abstract Windowing Toolkit (AWT)*. ANIMAL is currently only available for the current Swing 1.1.1 release also included in JDK 1.2+.

Tested platforms

ANIMAL has been developed both on Digital AlphaStation running Digital OSF v4.0 – also known as Digital Unix – and Intel-based computers running Linux, FreeBSD, Windows 95 / 98 and Windows NT. It has also been tested on an Apple Powerbook and thus underscores Java's claim of "write once, run everywhere".

JDK support

Note that changes in the class libraries, especially the renaming of the Swing packages, prevents us from providing support for both *JDK 1.1.5+* and *JDK 1.2 / 2.0* with the same **source** code. The binary Java code for JDK 1.1.5+ will run under JDK 1.2, so this is the main release we support

3 Starting and Playing ANIMAL Animations

ANIMAL's animation player is included in ANIMAL and is also useful for checking the animation currently under development.

Starting

To start the player, simply type the following command:

```
java animal.main.Animal
```

If this leads to the following message

```
Can't find class animal.main.Animal
```

you must update your `CLASSPATH` environment variable first to include the ANIMAL archive file `Animal.jar`. This is usually done by locating the place where `CLASSPATH` is set and appending an entry like

CLASSPATH

```
C:\Java\Animal.jar
```

for Windows users, or

```
$HOME/Java/Animal.jar
```

for users of UNIX-based operating systems. If you are unfamiliar with how to do this, take a look at the Java documentation which should including information about setting the `CLASSPATH`.

Start Player

To start the ANIMAL animation player, choose the entry `Show Animation` in the `Edit` menu of ANIMAL's main window shown in figures 3 on page 7. See figure 4 on page 8 for a screen shot of this operation.

The player itself consists of a single window with video player-like functionality, magnification setting and step choice. Figure 1 on the next page shows an example window.

Most of the player is taken up by the *animation canvas* in which the actual animation is shown.

Components

Below this are the controls for the current animation. These consist of a series of buttons which are used to

- jump to the animation's first step $< |$; **Jump to first** $< |$
- go to the previous step *without* playing out the animators contained in the step $<<$; **Previous Step** $<<$
- *pause* before changing the to the next step $||$; **Pause** $||$

  Note that this is only useful if there is a *timed link* between the current step and the next step.
- *play* the current step, ie. activate all animators $>$; **Play step** $>$
- go to the next step *without* playing out the animators contained in the step ensuremath¿¿; **Next Step** $>>$
- jump to the animation's last step $> |$. **Last Step** $> |$

Step Slider

The next component is a *slider* showing how far the animation has progressed. By clicking on the slider icon and dragging the mouse, a "skip forward" effect can be obtained, resulting in a execution of the steps dragged over. The slider can thus also be used to jump to a different step (whether backward or forward) in the animation.

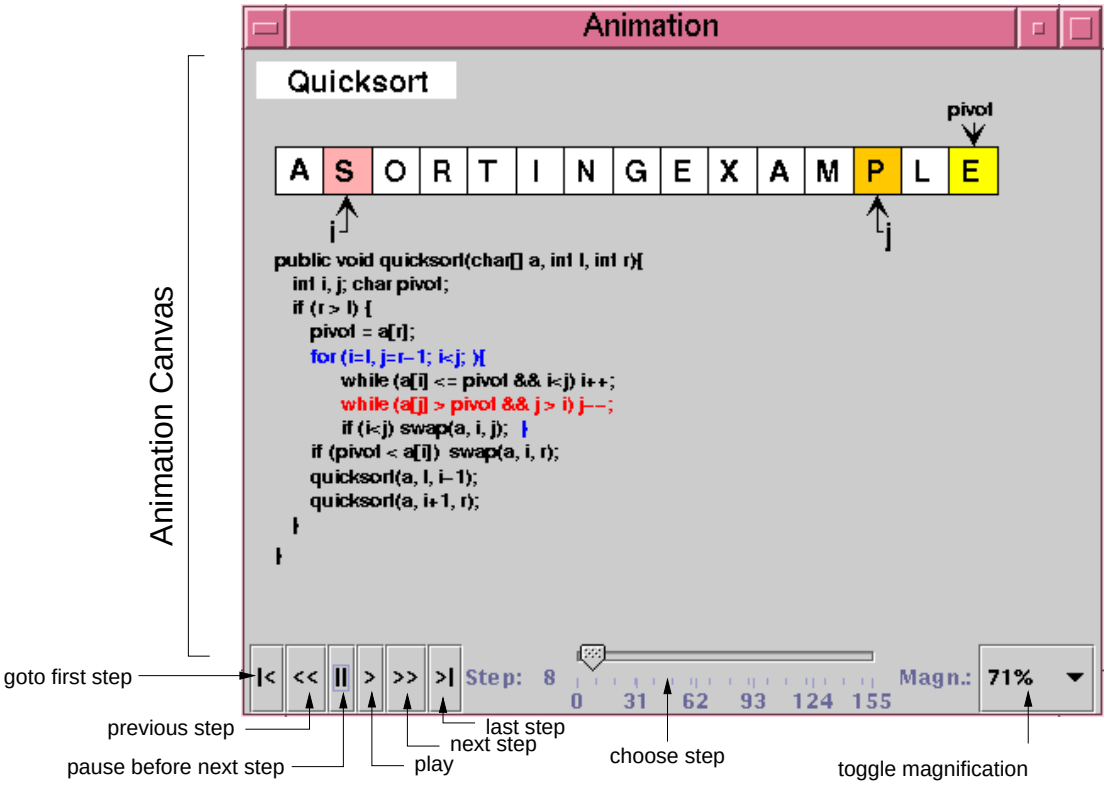


Figure 1: ANIMAL Player front end

Note that this may cause problems while generating animations, as the numbers of the animation steps need not be *always* sequential. If you encounter such problems, simply save your animation and reload it, and the problem should be solved.

Magnification

The next component allows the user to select a *magnification* for the display. This is especially helpful for very broad or high animations, grabbing screen shots or scaling the components to allow a switch from *computer presentation* to *beamer presentation* in lectures.

Due to scaling anomalities, only the following “sane” scaling factors are supported:

- 50%,
- 71%,
- 100% (default),
- 141%,
- 200%

4 Generating A New Animation Using ANIMALSCRIPT

In this example, you will use a few simple steps to generate a short but interesting animation about the behavior of the data structure *singly-linked list*. This animation will illustrate how to use ANIMAL to easily visually build animations.

The final result of this process will look roughly as follows: **Result**

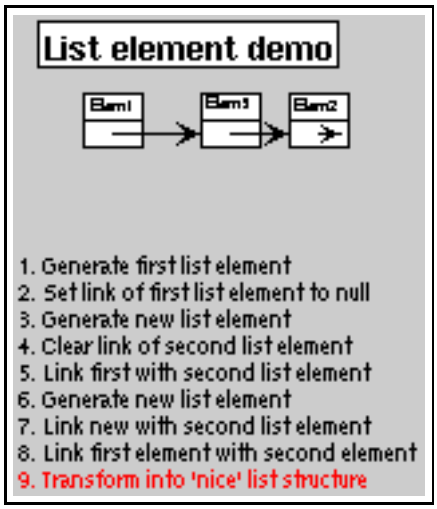


Figure 2: Final result of the tutorial animation

Don't worry, reaching this result is really not difficult. But now, let's get going!

4.1 Preparation for the new Animation

Generating animations using ANIMALSCRIPTbasically only requires an arbitrary *text editor*. So, start your favourite text editor on a new file called – for example – `demo.asu`. Note that the extension *.asu* is reserved for *uncompressed* ANIMALSCRIPT. **asu**

ANIMALSCRIPTfiles generally start with the following header line:

```
%Animal 1.4
```

Make this the first line of the file and end it with *Return*. Note that the version number *1.4* given in the file reflects the version of ANIMALSCRIPTused and thus may change in the future.

Next, you might want to give some information about the animation itself. Append these following lines, exchanging the *title* and *author* according to your wishes:

```
title "AnimalScript Tutorial Animation"
author "Guido Roessling <roessling@acm.org>"
```

Now, you can already start ANIMAL and load in your animation. Start the ANIMAL player as described on page 4. After a while spent on initialization and loading the **Main Window**

initial animation, (at least) ANIMAL's main window is shown. The main window is shown in figure 3.

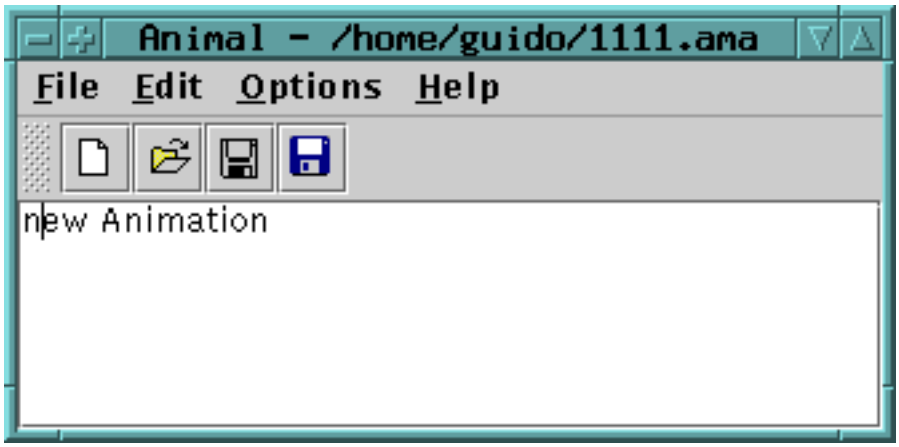


Figure 3: ANIMAL's Main Window

Menus

This window contains menus for *file operations* (`File`), opening and closing the windows (`Edit`) used for editing and viewing the animation, setting the *Options* (menu `Options`), and `Help`. Furthermore, it has a list of buttons which serve as a shortcut for – from left to right – *New Animation, Load Animation, Input* ANIMALSCRIPT*, Save Animation, Save Animation As*. . .

Select *Load* in one of the following ways:

- Click on the *second* button in ANIMAL's main window showing a *opened folder*,
- or click on the menu `File` and select its second entry, `Load`.

Shortcuts

You can also use shortcuts by pressing the shortcut key and the letter highlighted in the menu - in this case, `L`, so press both `ALT` and `L`, and the menu will be displayed. If not, you probably have to replace `ALT` by `CTRL`. If this does not work either, ask your system administrator for the local configuration details.

After the menu is shown, pressing `L` – the letter shown after the entry `Load` – is the same as clicking on `New`.

You can see the title and author information in the *Animation Information*. Select the menu `Help` and then `About this Animation` to show the information dialog.

4.2 Generating the Title

The first thing we are going to generate is the appropriate *title*. As this is a simple *text*, we use the **text** command as follows:

```
text "header" "List element demo" at (20, 30)
```

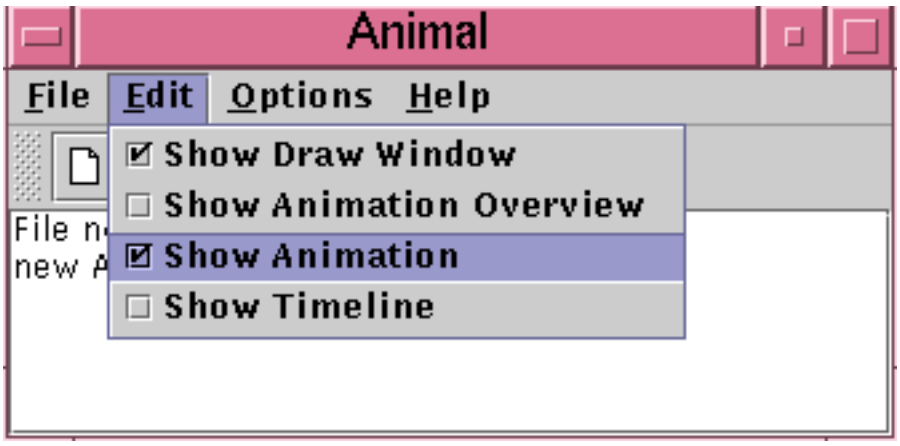


Figure 4: Selecting the displayed windows. Here, both *Animation* and *Draw Window* are opened.

Reload your animation, and you should see the header, but possibly not quite with the right "look". To change this, we assign the header a new font: SansSerif, size 24, **bold**. To do so, simply change the line to the following:

```
text "header" "List element demo"  at (20, 30)\
  font SansSerif size 24 bold
```

However, **do not break the line in your editor!** All ANIMALSCRIPTcommands must be given in a single line, so you should probably turn off *word wrapping*, too. Due to space limitations, the command does not fit in a single line in this display. Your command lines can have an arbitrary length in the file, though.

Reloading the animation will show you a somewhat more appropriate setting for a title. Also notice that ANIMAL has set the text color to the default color *black* without your saying so.

Now, we want to underscore the importance of the title by placing a *filled rectangle* below the title. This requires the use of a second command, **rectangle**.

There is a small problem in doing so, however. Do you know just how high and wide the text is? Experimenting with different setting would be rather time-consuming and tiresome, after all. . ..

Luckily, ANIMALSCRIPTaddresses this problem by allowing *relative placement*. Instead of giving the *absolute coordinates* for the rectangle points, you can substitute one (or all) of them using *offsets* from another object. In this case, we know that the rectangle should cover the whole text, so it should start a bit to the upper left of the text and extend to the text's lower right.

Try the following line and see what happens when you reload the animation:

```
  font SansSerif size 24 bold
rectangle "hRect" offset (-5, -5) from "header" NW\
```

You will now see *two* different animation steps, as each ANIMALSCRIPTcommand normally starts a new animation step. To make sure both commands are executed in the *same* step, put an opening curly brace { in a *single* line before the **text** command, and a matching closing brace } in a separate line after the **rectangle** command. Now reload and see what happens.

Hmmm... the rectangle completely hides the text! This is not really surprising, as ANIMAL draws the objects in the order they were specified. Changing the order of the commands will not work, as the rectangle will then not have access to the text coordinates. However, ANIMALSCRIPTalso offers the optional **depth** attribute for all graphic objects. **depth** influences the order in which objects are drawn: objects with *less* depth are *closer to the foreground*, and will be drawn *after* object that lie "deeper". Therefore, change your lines to the following:

```
{
text "header" "List element demo"  at (2y0, 30)\
  depth 1 font SansSerif size 24 bold
rectangle "hRect" offset (-5, -5) from "header" NW\
  offset (5, 5) from "header" SE depth 2 \
  filled fillColor white
}
```

Reload and see that now the header behaves properly.

You might also try adding the following line to your animation:

```
echo bounds: "header"
```

This will output the exact bounding box of the title when the animation is loaded in.

4.3 Embedding Code Lines

Now, we are going to enter the code describing the commands of this animation. This consists of the following text entries:

- `1. Generate first list element`
- `2. Set link of first list element to null`
- `3. Generate new list element`
- `4. Clear link of second list element`
- `5. Link first with second list element`
- `6. Generate new list element`
- `7. Link new with second list element`
- `8. Link first with new element`
- `9. Transform into 'nice' list structure`

To enter this code, you could use the **text** command as shown previously. However, ANIMALSCRIPTalso offers the concept of a *code group* of related lines of code (in this case, pseudo code!) which allows operations such as *indentation* and especially *highlighting*.

To generate a new code group, type in the following command:

```
codeGroup "listSource" at (10, 200) color black highlightColor red
```

Note that this command will *not* enter a new graphic object at all, but will only prepare ANIMAL for handling code lines. The coordinate `(10, 200)` is the *upper left corner* for the code group. Instead of the absolute coordinates, you could also have used *relative placement*, for example

```
codeGroup "listSource" offset (0, 80) from "hRect" SW color black
```

To enter the code elements, you should place them *including* the **codeGroup** command inside a *single* animation step, so that they will all appear at the same time. Therefore, put curly braces in the line before and after the **codeGroup** command.

Entering a new code fragment is very easy - you only have to say

```
addCodeLine "text" to "codeGroupID"
```

In our case, simply enter the following commands:

```
{
codeGroup "listSource" at (10, 200) color black \
    highlightColor red
addCodeLine "1. Generate first list element" \
  to "listSource"
addCodeLine "2. Set link of first list element to null"\
  to "listSource"
addCodeLine "3. Generate new list element"\
 to "listSource"
addCodeLine "4. Clear link of second list element"\
 to "listSource"
addCodeLine "5. Link first with second list element"\
 to "listSource"
addCodeLine "6. Generate new list element"\
 to "listSource"
addCodeLine "7. Link new with second element"\
 to "listSource"
addCodeLine "8. Link first with new element"\
 to "listSource"
addCodeLine "9. Transform into 'nice' list structure"\
 to "listSource"
}
```

Again, keep in mind that these commands must all appear on a *single* line each, so continue lines whenever a backslash \ is shown in the listing.

4.4 Code Highlighting

Now that the preparations are done, we can start with the contents of this animation. The first command to be executed is the first code line. To show this execution, you can highlight the code line as follows:

```
highlightCode on "listSource" line 0
```

Note that as is usual in **C** and **Java**, ANIMALSCRIPTstarts counting at 0, so the first code line has number **0**.

The effect of this statement as shown on reloading the animation is setting the line in its *highlightColor* **red**. Check the **codeGroup** command you entered – the color **red** was explicitly given as the highlight color. If you prefer other colors, feel free to change this entry.

4.5 Generating A List Element

Now that the command has been highlighted, you can generate the first list element. This is done using the **listElement** command as follows:

```
listelement " firstListElem " (100, 80) pointers 1
```

However, this leaves the element without a text entry. Therefore, insert a **text "Elem 1"** between the coordinates and the **pointers** command. Furthermore, the element should not be shown at once, as we first want to draw attention to the command itself. Therefore, use the command as follows:

```
listelement "elemA" (100, 80) text "Elem1" pointers 1\
  after 20 ticks
```

The **after** command determines the amount of time to wait until the action is performed - in this case, 20 ticks will take place before the element is shown. **Ticks** are an ANIMAL -internal time unit that allows for smoother effects than precise timing on a millisecond base does[1].

On reloading the animation, you will find that the operations have ended on separate steps, so change the code to include curly braces around the two commands.

4.6 Highlighting and Unhighlighting Code

Next, you should start working on the second command. This requires the following operations to take place in *one* step:

- *unhighlight* the first line of code (**line 0**),
- *highlight* the second line of code (**line 1**),
- clear the link of the new list element.

[1] If you are wondering about this, note that statements like **after 200 ms** require the computer to act at a *precise* point of time, or perform all operations within a *presice* amount of time. If the computer is busy or the actions are time-consuming to perform, it may fall behind the schedule and try to catch up by skipping intermediate animation frames. This will often lead to a jerky display.

If you paid close attention, the second operation should not pose any problem by now. Rather unsurprisingly, the command for *unhighlighting* code is called **unhighlightCode**.

Finally, the command for *clearing* a link is called **clearLink** and requires you to give the name of the object to be treated this way.

Thus, the next commands should look as follows:

```
{
  unhighlightCode on "listSource" line 0
  highlightCode on "listSource" line 1
  clearLink "elemA"
}
```

Now, we will repeat these actions to generate the next list element, clear its link, and update the source highlighting accordingly. The second element should be placed to the right of the first list element, say **(130, 0)** from the first list element's top right corner (called **NW**). The commands thus look as follows:

```
{
  unhighlightCode on "listSource" line 1
  highlightCode on "listSource" line 2
  listelement "elemB" offset (130, 0) from "elemA" NE\
   text "Elem2" pointers 1
}
{
  unhighlightCode on "listSource" line 2
  highlightCode on "listSource" line 3
  clearLink "elemB"
}
```

4.7 Linking list elements

The next command requires you to link the two list elements. This is easily accomplished using the **setLink** command, which expects first the name of the *base list element*, then the keyword **to** and the *target list element's name*. In this case, the operation should also take a certain amount of time, for example the **20 ticks** used before.

The required commands are as follows:

```
{
  unhighlightCode on "listSource" line 3
  highlightCode on "listSource" line 4
  setLink "elemA" to "elemB" within 20 ticks
}
```

Note that the **within** keyword defines the *duration* of an operation, while **after** defines the *offset*. If **both** are used, you have to specify the *offset* first.

I will leave the generation of the next few steps as a small exercise. Using what you have done before, it should not be difficult to update the *code highlighting*, *gen-*

erate a new list element called **elemC**, and setting the links between the list elements. However, the complete code is also give a bit later in this document.

Please place the new list element roughly in the middle of the two elements, but **not** precisely. For example, place it at offset **(80, 50)** from the **NW** of the first list element.

4.8 Moving objects

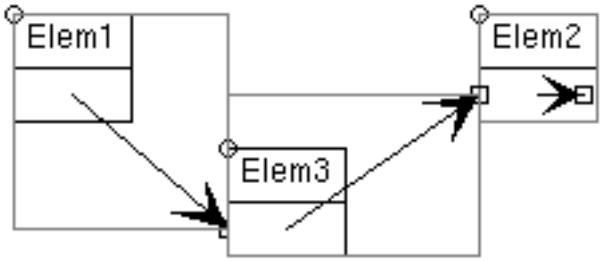
The last step calls for transforming the list into a *nice* structure. To do so, you have to perform the following actions:

- Updating the code highlight for the last line of code (not described here),
- move the **new** element between the two list elements *without* changing its pointer,
- move the **first** list element's **pointer** without moving the rest of the object.

The last two operations call for a **move** operation which is quite difficult, but also shows how powerful ANIMALSCRIPTis.

Note that the bounding box of any object is a rectangle encapsulating *all* components of the element. Thus, the bounding box of the *middle* list element (the last one you inserted) also includes the *pointer* to the last list element. If you used the positioning given above, you will notice that the pointer extends the bounds to the top and the right.

Now, we want to move both the new list element – *without* changing its pointer! – **and** set the first element's pointer *without* changing the element's position. To do so, we first have to decide on how to move the elements. The following screenshot should be helpful, showing the bounding box of the elements.



If you take a close look, you will find that you can use the following edges of the bounding boxes for precise placement:

- the middle element's **SE** corner, as this includes the base line of the object *and* the pointer location;
- the **SW** corner of the last list element, as this gives the target baseline *and* the target coordinates for the pointer.

First, we will generate a *line* object with these coordinates that should be marked as **hidden** to prevent it from disturbing the display:

```
  line "moveLine1" offset (0,0) from "elemC" SE\
    offset (0, 0) from "elemB" SW hidden
```

Now, we can use this line for moving both objects. However, we have to use two *special* subtypes of the **move** command for this purpose:

- for setting only the pointer *without* changing the element, we need a move of **type "setTip #1"**
- and for only setting the pointer *without* moving the element, we need **type "translateWithFixedTip"**

The following code does the trick:

```
  move "elemC" type "translateWithFixedTip"\
    via "moveLine1" after 20 ticks within 20 ticks
  move "elemA" type "setTip #1" via "moveLine1"\
    after 20 ms within 20 ticks
```

And now... the animation is finished! Simply reload and enjoy.
The complete animation code is as follows:

```
%Animal 1.4
title "AnimalScript Tutorial Animation"
author "Guido Roessling <roessling@acm.org>"
text "header" "List element demo"  at (20, 30)\
  font SansSerif size 24 bold
rectangle "hRect" offset (-5, -5) from "header" NW\
  offset (5, 5) from "header" SE filled fillColor white
{
text "header" "List element demo"  at (2y0, 30)\
  depth 1 font SansSerif size 24 bold
rectangle "hRect" offset (-5, -5) from "header" NW\
  offset (5, 5) from "header" SE depth 2 \
  filled fillColor white
}
{
codeGroup "listSource" at (10, 200) color black \
    highlightColor red
addCodeLine "1. Generate first list element" \
  to "listSource"
addCodeLine "2. Set link of first list element to null"\
  to "listSource"
addCodeLine "3. Generate new list element"\
 to "listSource"
addCodeLine "4. Clear link of second list element"\
 to "listSource"
```

```
addCodeLine "5. Link first with second list element"\
 to "listSource"
addCodeLine "6. Generate new list element"\
 to "listSource"
addCodeLine "7. Link new with second element"\
 to "listSource"
addCodeLine "8. Link first with new element"\
 to "listSource"
addCodeLine "9. Transform into 'nice' list structure"\
 to "listSource"
}
{
highlightCode on "listSource" line 0
listelement "elemA" (100, 80) text "Elem1" pointers 1\
  after 20 ticks
}
{
  unhighlightCode on "listSource" line 0
  highlightCode on "listSource" line 1
  clearLink "elemA"
}
{
  unhighlightCode on "listSource" line 1
  highlightCode on "listSource" line 2
  listelement "elemB" offset (130, 0) from "elemA" NE\
   text "Elem2" pointers 1
}
{
  unhighlightCode on "listSource" line 2
  highlightCode on "listSource" line 3
  clearLink "elemB"
}
{
  unhighlightCode on "listSource" line 3
  highlightCode on "listSource" line 4
  setLink "elemA" to "elemB" within 20 ticks
}
{
  unhighlightCode on "listSource" line 4
  highlightCode on "listSource" line 5
  listelement "elemC" offset (80, 50) from "elemA"\
   NW text "Elem3" pointers 1
}
```

```
{
  unhighlightCode on "listSource" line 5
  highlightCode on "listSource" line 6
  setLink "elemA" link 1 to "elemC"
}
{
  unhighlightCode on "listSource" line 6
  highlightCode on "listSource" line 7
  setLink "elemC" link 1 to "elemB"
}
{
  unhighlightCode on "listSource" line 7
  highlightCode on "listSource" line 8
  line "moveLine1" offset (0,0) from "elemC" SE\
    offset (0, 0) from "elemB" SW hidden
  move "elemC" type "translateWithFixedTip"\
    via "moveLine1" after 20 ticks within 20 ticks
  move "elemA" type "setTip #1" via "moveLine1"\
    after 20 ms within 20 ticks
}
```

List of Figures