An Algorithmic Animation Platform for the Web

M. Esponda
University of Applied Sciences Gießen-Friedberg, Friedberg, Germany

Abstract—**This paper describes a system for generating animations of algorithms for use in the classroom. The animations emulate the individual steps of an algorithm in graphical form and can be posted to the Web. The system is based in Flash. In order to generate an animation the code of an algorithm is extended with annotations which generate the script code. We call our scripting language "Flashdance".**

Index Terms—**Animation, Algorithms. Educational Technology, Programming.**

I. INTRODUCTION

This papers deals with techniques for the design and production of appealing algorithmic animations and its use in computer science education. A good visual animation is both a technical artifact and a work of art which can greatly enhance the understanding of an algorithm's workings. The paper shows how to generate animations which represent data structures in a visually intuitive manner. The animations described in this paper have been implemented and used in the classroom for courses at the university level.

During the founding years of computer science, the development of visualization tools was guided by the interests and possibilities of the academic community. In the case of computer graphics, the driving force has switched from the universities to companies such as Pixar (general animation), Macromedia, or many of the computer games foundries. Such companies set now de facto standards which are very difficult to ignore [1].

If we look back and review the history of algorithmic animation, it is striking to see that almost all systems built in the 1980s and 1990s had to provide their own animation engine. This was the case for the BALSA and for the Tango family, as well as for most other systems. When Java arrived, there was at least the possibility of doing animation with a graphical standard engine. Java, however, was not conceived for animation.

Macromedia, as a company, has much more freedom to define and modify its Flash animation engine. Introduced in 1995-96 (first with the name FutureSplash by a company later bought by Macromedia), Flash has gone through several generations and has transformed into a de facto Internet standard. Today, high-quality animations for the Web are most likely produced in Flash. Since the player is free, the market penetration of Flash is well above 90% for the previous versions of Flash and is growing steadily for the latest version.

In this paper the emphasis is put on the production of high-quality animations, which will be animated with Flash. We want to export those animations to the Web. The complete system is called Flashdance, in the tradition of naming algorithmic animation systems by a kind of dance. No other algorithmic animation system until now has used a standard animation engine with the popularity and user base of Flash. Attempts in this direction are represented by those vintage animation systems based on Hypercard which later disappeared from the scene [2,3]

II. FLASH ANIMATIONS - BASICS

We decided to adopt the Flash animation engine for the production of high quality animations mainly because of the following reasons:

- *Flash is a de facto standard for the Internet*
- *Flash animations can be posted in Web pages*
- *Flash offers esthetically pleasing graphical objects*
- *Flash animations can be stored in small files*
- *Flash animations are streamed*
- *The playing format is in the public domain*
- *The Flash ActionScript language*

Macromedia developed ActionScript, a scripting language for Flash animations. ActionScript is a prototype object-based language, with a very similar syntax to JavaScript. Like many script languages, ActionScript is a loosely typed language.

The actual authoring environment of Flash integrates drawing, animation and programming tools in the same work environment. The ActionScript code reacts to events on the timeline. Algorithms must be modified to be able to produce animations and make then fit the ActionScript style.

As a result, learning the Flash authoring environment is very time intensive. Only the drawing tools are easy to learn. The ActionScript language can be very frustrating for most programmers used to work with programming environments for languages like Java or C++. ActionScript is not a suitable language for algorithm design or for beginner students of computer science.

Flash has two important advantages though: the graphical quality of the animations and the ease with which object libraries can be built. It is then possible to use an intermediate language to create Flash animations, which can be generated from any programming language. Less effort is necessary and results are obtained faster.

Fig. 1 shows the general structure of a Flash animation. A film consists of scenes, which are played one after the other (unless control code and user interaction determine a "non-linear" flow). Each scene consists of one or more frames. A frame contains one or more layers. Layers are placed one on top of each other. Layers are containers for graphical objects, interaction objects, or animation objects.

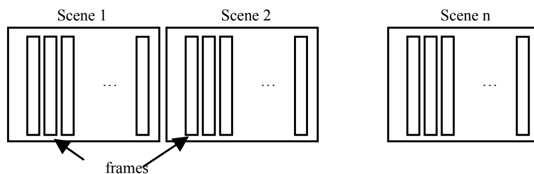

Figure 1. A Flash film may contain many scenes, and each scene may have many frames.

Fig. 2 shows four layers on top of each other. The background of layers can be transparent. Graphical objects can also have some degree of transparency, so that other objects in layers further down in hierarchy can also be seen through them.

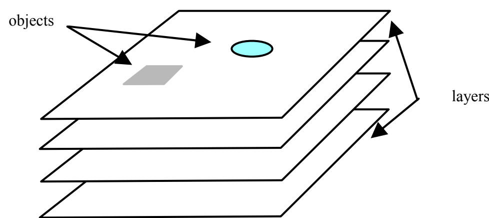

Figure 2. Scenes consist of layers, stacked on top of each other. Layers contain drawings and other objects.

Flash animations are frame-based. When a user develops an animation, she has to define all the frames that will be played, for all the layers it contains. Complex animations, with many frames, can be produced more easily by defining "key" frames in the animation. Flash can then interpolate additional frames between the key frames, a process called "tweening".

The sequence of frames in an animation defines the "timeline". The different layers share a timeline. Objects in each overlay are in principle independent of the other objects and can move or change aspect in any frame. Tweened objects must be in their own layer. Objects in different layers can also coordinate their movement of change of appearance. Fig. 3 shows an example of a scene with two layers. The upper layer contains a sphere; the lower layer a shaded square. The scene consists of 19 frames, in which the sphere moves in front of the square covering it partially.

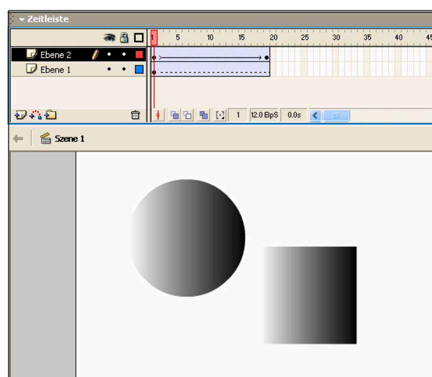

Figure 3. The timeline with two layers.

The key frames in this animation are the first and the last. Intermediate frames are generated by Flash, using the "tweening" option (for in-betweening). It is also possible to assign a curved trajectory to the sphere, from the first to the last frame. In that case an additional layer is used to define the path.

Tweening can also be used to interpolate frames when the form or the color of an object is changed. In this way, smooth transitions between transformations of an object can be produced.

Therefore, creating a Flash animation by hand usually involves the following steps:

- The film is first divided in scenes, and scenes in overlays.
- At least one key frame for each scene is inserted.
- Graphical objects that will be reused are defined as symbols and saved in the project library. Symbols can be reused as "instances". Parameters of symbol instances (color, form, etc.,) can be changed.
- The objects are arranged in the key frames for each overlay. This is similar to the way a slide presentation is created using PowerPoint.
- Where tweening-frames are to be interpolated between key frames the tweening-options must be set.

Even a short Flash animation can involve many overlays and many frames. The programmer can determine the frame rate at which the film will be played. A complete animation can then be exported as a SWF file, and the Flash development environment is not needed to view it. Any Flash player will do.

A handcrafted Flash animation is produced by drawing and redrawing frames, and by interpolating between the important frames. But there is one more powerful feature of Flash that makes Flash animations so compelling. Symbols in Flash can be themselves self-animated objects. In this case, they are called movieclips. A movieclip pasted on an overlay has it own timeline and plays its own animation when it is used. The timelines of the main scene and the timeline of the movieclip run at the same frame rate. One could, for example, animate a person walking. The eyes could be a movieclip. The movement of the eyes could be defined inside the eyes movieclip. When the person walks in the animation, the eyes will be moving. In this way it is possible to create complex and powerful hierarchical movieclips.

The full power of Flash animations is unleashed, when ActionScript is used. ActionScript, the Flash scripting language, gives the programmer full access to all these features and more. An animation can then consist of a single frame which contains the script code. When the script code runs, it generates all frames of the film. Objects used by ActionScript can contain ActionScript code themselves, so that a Flash animation running is a collection of objects executing their code concurrently.

With ActionScript it is possible to produce an animation directly from an instrumented algorithm. Flashdance, my own algorithmic animation language, is converted into ActionScript by an interpreter, which then takes advantage of the powerful Flash animation engine.

III. OVERVIEW OF THE FLASHDANCE-SYSTEM

Fig. 4 shows an overview of the architecture of the Flashdance system. An algorithm provides events which

can activate instrumented classes, or it directly provides the instructions which are accumulated in a script file ("name.ans"). The Flashdance interpreter executes the animation using the standard library of animation objects as well as user defined libraries. The result is visible on the screen. The user has some buttons to control the speed and appearance of the animation, as well as its rendering in overlays.

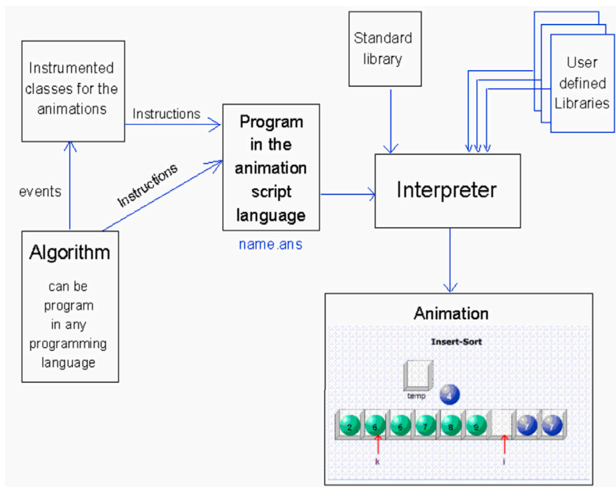

Figure 4. Architecture of the Flashdance system

A Flashdance animation is played on a single scene. We can create the animation directly on the scene background or we can have many views.

An important difference between Flashdance and other algorithmic animation systems is that, in Flashdance a single class of animation objects is used (called AObjects). They provide a high abstraction level for the programmer. All objects in an animation are instances of the AObjects class. An AObject has properties such as *x*- and *y*-coordinate, width, height, color, and also name, type (the shape), types of possible highlighting, etc. An AObject behaves like a software agent: it can change any of its properties autonomously, including its form (shape or type). An AObject can execute any instruction of the Flashdance script language, with exception of the setTime and setStop instructions, which control global temporal properties of the animation.

A Flashdance animation consists of a sequence of messages to AObjects, commanding them to change their properties. A View is also an AObject with the special property that it can contain other AObjects. If an AObject of subtype "view" is modified, all those objects contained in the view are modified. This opens the possibility of implementing more sophisticated animations which can zoom into the data structures in which relevant changes are taking place. In our interpreter, AObjects are an extension of Movieclips with the mentioned additional properties such as name, label, type, etc., and also additional highlighting methods.

AObjects are subdivided into two categories: those which are built completely when they are needed and those which are instances of predefined objects from the library of the Flash animator. The following objects are built completely during execution: Points, Lines, Rectangles, Ovals, and Views. The following AObjects are instances of objects in the library: Bubble, Ball, Rabbit. The second category allows the user to define many types of objects which can be used in animations. This approach was followed in my implementation of Flashdance and also in the Java classes which were instrumented for animations.

An example is the following: An AArray is a class capable of animating any relevant change in an array. All the visible components of the AArray are AObjects. The elements of the array, the boxes in the array, and the indices used to point to operations in the array are AObjects. The AArray provides all methods and operations necessary for modifying the array, and also the corresponding animated version. There are several methods for AArray. Some of them animate state transitions, others do not. For example "swap" is the method to exchange two array elements without animating, whereas "aSwap" swaps two elements and produces the animation.

An AArray frees the user from the computational details needed to animate an element of the array, when it moves from one position to another, or when the indices change. It substantially simplifies the programming of the animations based on this kind of data structures. Other instrumented classes, AQueue, AStack, AGraph, ATree, were programmed in the same way.

Overlays can be defined by the programmer of the algorithmic animation with the definition of different Views. Views can be placed over each other and can have a transparent background.

Flashdance has a pre-defined library of animation-objects which can be very easily extended by the user (see section 6.5).

IV. THE FLASHDANCE SCRIPT LANGUAGE

The Flashdance animation language was designed for simplicity [4]. An animation should be easy to produce, without having to master a very extensive set of commands. The quality of the vector graphics should be preserved. This is guaranteed by allowing the user to select objects from a library of standard object types. The user should be able to import or produce graphical libraries in advance.

In what follows we give an overview of the instructions in the Flashdance scripting language.

A. *Instructions*

The name of each command is given in bold face. Mandatory parameters follow. Optional parameters are enclosed in square brackets. Parameters are separated by blank space. Commands are written in one line, here they sometimes wrap around producing two lines of text.

new object-type object-name x y width height [label] [colour]

Primitive object-types for animations are:

View view-name x y width height [label] [background-colour]

String object-name x y width height string-text [colour] [style]

Line object-name x1 y1 x2 y2 [line-width] [colour]

Rectangle object-name x y width height [line-width] [colour]

Oval object-name x y width height [line-width] [colour]

With the instruction **new** a new animation object from the library is selected and an instance of it is inserted at the position (*x,y*) of the current view. Object-type is the name of the animation object (movieclip-symbol) in the library. Object-name is the name of the new instance. All objects in an animation must have different names. Only objects in different views may have the same name. *x* and *y* is the position on the view-window of the animation. If no views were defined and set at the time the instruction new is executed, the object will be positioned directly in the background of the scene. The new instruction must have at least the four arguments listed. If width or height is not specified, the object is drawn with a standard size.

Most objects of the library have a label option which can be written when the object is created.

An animation-object (movieclip) can be of type "View", that is, a window at a specific position on the animation screen and of size width × height in pixels. The background colour can be given as an option. The default colour is transparent.

Views are Flash-movieclips and can be created one on top of another. This is a very significant difference to other algorithmic animation systems, which do not offer the option of overlaying views.

A new object can be of type String, Line, Point, Rectangle, or Oval. These simple standard animation Objects do not really exist in the library. They are created at run time for more flexibility and because of their simplicity. For each one of these objects it is possible to give some special options like colour, line-width, style (for strings), etc.

remove object-name1 . . . object-namen

This instruction removes one or more animation objects from the current view.

removeAll

This instruction removes all objects from the current view.

change object-name property1 value1 property2 value2 . . . propertyn valuen

This command is used to change one or more properties of an object to one or more values (for example colour, width, height, etc.)

exchange object-name object-type [x y width height] [label]

Redefines the object with the name object-name to have the type object-type. The original object is removed and a new animation-object is created with the same object-name but the new object-type. The new type uses the parameters of the old object, except if optional parameters are explicitly given.

This instruction is very useful when properties like color or shape of complicated objects have to be changed.

moveTo object-name1 x1 y1 . . . object-namen xn yn

With this command an object, or several objects, are beamed to a new position with coordinates (*x,y*).

animTo object-name1 x1 y1 . . . object-namen xn yn [path]

animTo produces a smooth movement of an object from its current to a new position. Several objects can be animated simultaneously. A different path can be used as an option. If no path is specified the movement is a straight line.

highlight highlight-type object-name1 . . . object-namen

A visual cue is produced to highlight one or more objects simultaneously. Types of possible highlight are blinking (twinkle) and a momentary change of size (swelling).

swap object-name1 object-name2 [path]

This instruction is used to swap two objects with a smooth movement. It is not necessary to pass the position of the two objects as parameter. An optional path gives the trajectory for the movement of both objects.

setView view-name

All instructions following a setView instruction refer to view-name, until a new setView instruction is executed.

setTime duration

Set the execution time for each instruction following, until a new setTime instruction is used.

stop [label]

Set a stop mark with an optional name label in the animation script. The animation can be restarted pressing the "run" button in the viewer.

B. The object library

One of the major advantages of adopting a standard animation engine such as Flash for algorithmic animation is that all the editing and animation tools developed by Macromedia can be inherited for our task. There is no need to generate bad looking objects from scratch, when we can define a library of graphically appealing objects using the tools of the system.

Flashdance animations use symbols stored as movieclips in a library of objects. The new command accesses these objects and makes them available for the animation. For our first animations we defined a simple library of objects using the Flash editing tools. Fig. 5 shows a selection of some of the animation objects. The name of the object is shown above each one (in black). There are several types of spheres, for example, with names _B, _G, _R, Rb, etc. The library includes spheres, discs, containers for arrays, bars, rulers to measure objects, arrows, circles, rectangles, and even rabbits (which can perform fully animated jumps).

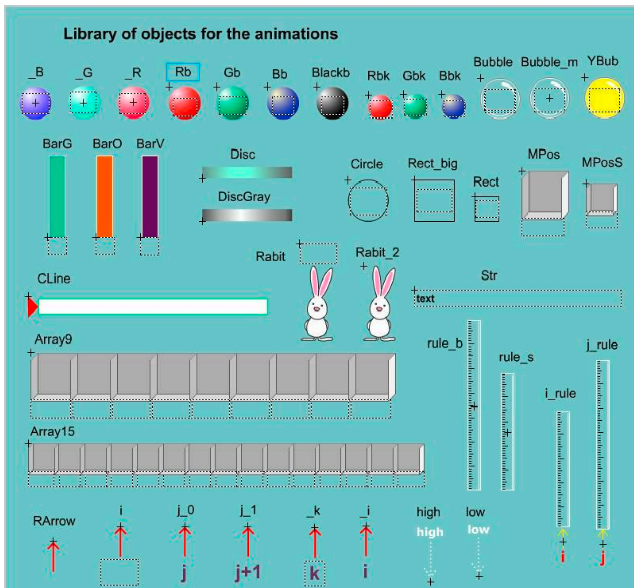

Figure 5. Library of some of the animation objects available in Flashdance.

In Fig. 5 some of the objects have a dashed rectangle in front of them. This rectangle can be used to write a label for the animation. The spheres, for example, can be labeled with the number they represent. Array locations can be labeled with their index value, and so on.

The movieclips library is the heart of the pictorial representation. It is very easy to draw good-looking objects using the refined Flash editing and drawing tools. A movieclip-symbol can be modified at any time, or can be completely substituted by another representation.

Objects in the library can be animated objects themselves. The spheres, for example, could have a texture and could be rotating spheres during the animation or could be growing during the animation. The algorithm animator does not have to take care of such details. This is done in advance by the person designing the object library. This library is defined once and much design work is saved by reusing symbols later.

C. *Instrumenting a program*

It is easy to instrument programs which produce the animation script. Let us review a simple example: the Quicksort algorithm, as it can be written and animated in Python.

The main program for the animation consists of the following few lines:

```
f = open('quicksort.ans','w')
f.write("&prog_text=\n")
S = [12,5,13,8,9,1,3,10,14,4,7,6,15,2,11]
for i in range(0,len(S),1):
    f.write("new Oval o%s %s 300 %s %s \n" %
                    (S[i],20+i*30,15+S[i],15+S[i]*5))
qsort(S)
f.close()
```

In this program, a file “quicksort.ans” is opened as writable file. The list to be sorted is S. The call to Quicksort is “qsort(S)” and the animation script file is closed. A circle is defined and painted in the animation window using the “new Oval” command. Each circle is an object, the object number is given by S[i]. The position of the circles has been defined to be at row 300 on the screen. The column position increases by 30 pixels each time (with an offset 20). The height of the circles (ellipses) is 15+S[i]*5, the width is 15+S[i].

The Quicksort algorithm is as in the first edition of [5]. Two indices are used. In the minimal version defined below, the indices are not being shown on the screen, only the movement of the array elements. All movement is concentrated in the function “swap”. This interchanges two elements in the array, and at the same time, writes the script animation command in a file.

The script command is “swap ox oy”, where *x* and *y* are the objects (numbers) being swapped. There is no need to write their coordinates, they are implicit when we refer to the objects by name. The trajectory followed during the swap is a rectangle, that is, both objects move vertically upwards, then horizontally to their new horizontal coordinated, and then vertically downward to fill in-place.

The instrumented code of Quicksort is the following:

```
import random

def qsort( A ):
      quicksort(A,0,len(A)-1)
def quicksort(A,low,high):
   if low < high:
      m = partition(A,low,high)
      quicksort(A,low,m-1)
      quicksort(A,m+1,high)
def partition(A,low,high):
   pivot = A[high]
   i = low-1
   for j in range(low,high):
      if A[j]<=pivot:
         i = i+1
         swap(A,i,j)
   swap(A,i+1,high)
   return i+1
def swap(A,i,j):
   temp = A[i]
   A[i] = A[j]
   A[j] = temp
   f.write("swap o%s o%s rect 0\n" % (A[i],A[j]))
def generate(A,low,high):
   i=low
   while i<high:
     A[i]=random.randrange(1,400)
     i=i+1
```

The instrumented code remains very readable, as can be seen. To animate pointers, a “new” instruction has to be defined for each pointer (to generate an arrow) and every time a pointer is updated, the arrow has to move. This is best done by defining an “update_pointer” function which sets a pointer to its new value and writes the animation command to a file. In this way the scripted program

remains short and readable. Since this example consists of a single view and has no overlays, no view commands are needed.

V. THE INTERPRETER

The general structure of the Flashdance interpreter is shown in Fig. 6. The Flashdance script produced by a program is parsed, in order to identify the individual instructions. Each instruction is then given to the instructions interpreter, which starts the ActionScript sequence corresponding to the instruction found in the Flashdance stream. The instructions interpreter accesses the predefined object library in order to create the objects for the animation.

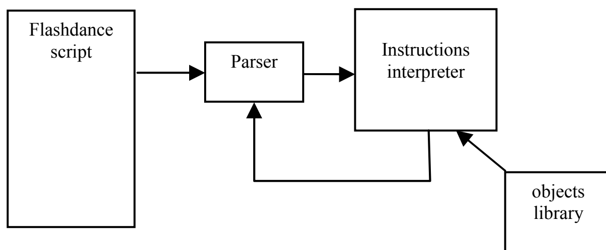

Figure 6. Block diagram of the Flashdance interpreter.

The structure of the interpreter is simple, but powerful. Drawing commands are almost not needed; the library of objects contains already the most common graphical primitives, which are then accessed by name.

When the Flashdance interpreter is run, a screen appears with a drawing window for the views of the algorithm and with some buttons to control the animation.

In this section, we will review some animations created with Flashdance. The production time for most of them was very low. Most of the effort went into defining the object library, but this is a one-time effort whose result can be reused many times.

A. *The Game of Life*

The Game of Life was invented by John Conway around 1970 [6]. It is a kind of mathematical recreation, which nevertheless has led to many implementations and even serious research about the computational capabilities of cellular automata. The Game of Life is universal, that is, any computable function can be implemented with the 0-1 code and with the matrix used by the game.

Life is played on a matrix of cells. Each of them can be dead (0) or alive (1). The game proceeds by generations. Out of an initial state, cells can become alive or dead in the next generation. A cell which is dead becomes alive in the next generation if it has exactly three live neighbors in the current generation. Each cell can have up to eight possible neighbors in the 3 by 3 matrix with this cell at the center. A cell which is alive stays alive in the next generation only if it has two or three live neighbors. In all other cases the cell dies.

Fig. 7 shows the start of a game. The red cells have been predefined as alive, the blue cells are dead. The pattern in the middle is called a “glider” since it reproduces after four generations, but displaced diagonally. The pattern to the left is a stationary one, which “twinkles”, that is, alternates between a vertical and horizontal bar in each generation.

Fig. 8 shows two pictures of the evolution of the game after several generations. The glider is going across the matrix, whereas the stationary pattern keeps alternating between its two states.

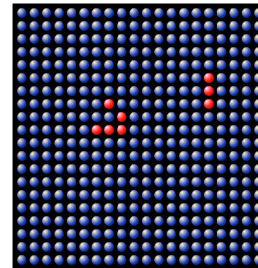

Figure 7. Initial configuration for a game of Life. Red cells are alive, blue cells are dead.

The algorithm for the game was written in Python. The Python program produced a script by writing to the script file. For example, the matrix of cells is initialized with two loops and the write command:

```
f.write("new Bb c%sc%s %s %s 0 0\n" % (i,j,70+i*20,100+j*20))
```

This command tells Flashdance to define a new cell, a blue small ball (Bb), with name “c<i>c<j>”, where <i> and <j> are the numerical decimal values of the indices of the entry (i,j) in the game matrix. Each cells is positioned at the pixel coordinates (70+i*20,100+j*20).

When the simulation runs, only a “change” command is needed, every time a cell changes state, to order a ball to change its color from blue to red, or vice versa.

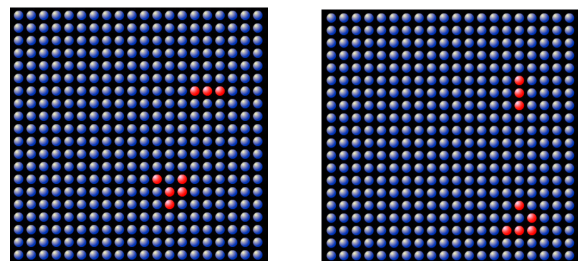

Figure 8. A glider moving to the right, a blinker blinking on place.

The complete animated Python code for the Game of Life is just a few lines long.

B. *Quicksort*

A further improvement to this kind of animations is to include the Java code of the algorithm in an additional view, which is played alongside the algorithm. This has been done with another version of the Quicksort algorithm. The Java code is written in an auxiliary text file, which is read by the interpreter. In the Java code, where Quicksort is implemented, it is necessary to produce an instruction for the Flashdance interpreter to let it know which line of the code to highlight. This is the “setCodeLine” command, followed by the line number.

A portion of the animation script, for example, is the following:

highlight swelling o7 o14 animTo i1 297 173 setCodeLine 13

This tells the animation engine to highlight objects o7 and o14 by swelling them. Then object i1 is smoothly moved to its new position (297,173) and line 13 in the Java code is highlighted. This gives the impression that the algorithm is running concurrently with the data view.

Fig. 9 shows the start of the animation. The Java code has been loaded and the array has been initialized, as well as the pointers i and j.

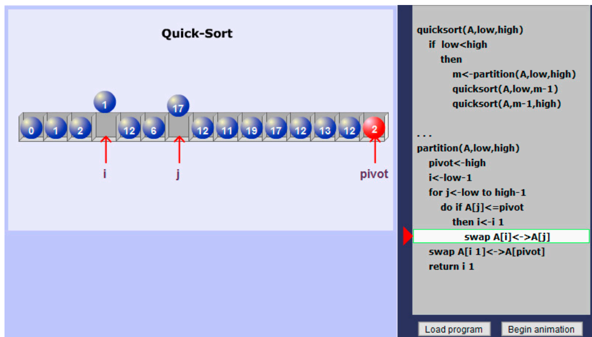

Figure 9. Start of the Quicksort animation.

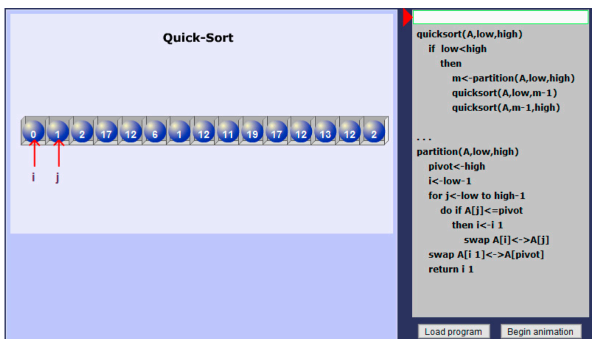

Figure 10. Exchange of two numbers at the index positions i and j.

In the next screenshot, Fig. 10, Quicksort is exchanging two numbers (1 and 17), and the pseudocode is highlighted at the “swap” operation.

In the next screenshot, Fig. 11, Quicksort has further progressed. The first recursive evaluations have returned and the numbers at the beginning of the array are sorted. They are thus colored green.

The speed of the animation can be controlled by the user or by the teacher explaining the algorithm to a class. The animation can be exported to a Web site also.

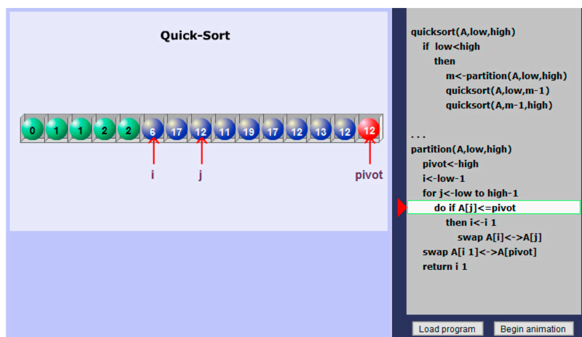

Figure 11. Sorted subarrays are colored green.

C. *Radix Sort*

Radix sort is a sorting algorithm with linear complexity. Fig. 12 shows the algorithm running, using a set of dates as the numbers to be sorted. The first set of bars shows the days, the second row the months, and the third row the years. A date is composed on one bar in each row (one day, one month, and one year). Radix sort starts sorting first the days, as shown in Fig. 12, where the days (in green) are being copied to a second array. After the dates have been sorted according to the day, the next sorting sweep sorts the months (Fig. 13, in blue). Now the dates are copied from the lower part of the screen to the upper part. Repeating this process for the years, the dates are finally sorted. Each sort was implemented using the counting sort algorithm.

Using dates to illustrate Radix sort makes the algorithm easier to understand. It also becomes obvious that different numerical bases can be intermixed, for different portions of the input. The animation is reversible (see Section 6.9) and the code is shown on the panel on the right.

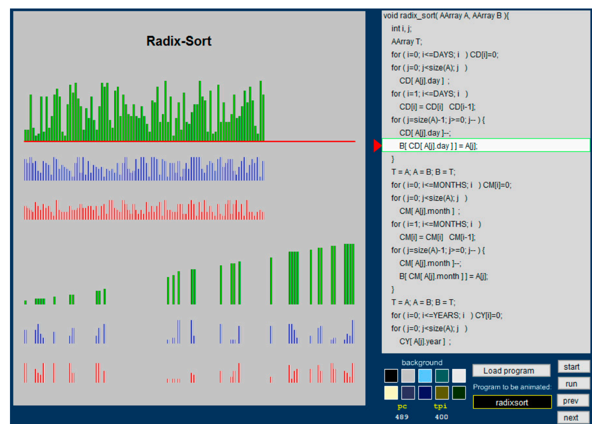

Figure 12. Radix sort running. Dates are being sorted. Days are represented by green bars, months by blue bars, and years by red bars. The dates on the upper portion of the window are being copied to the lower portion, ordering by day.

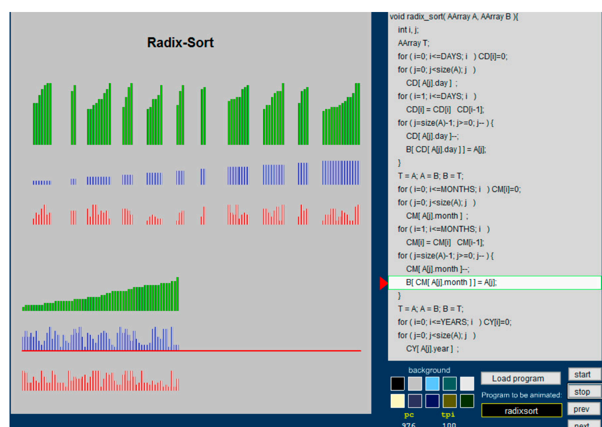

Figure 13. In the next sort sweep the dates are ordered by month and are copied from the lower to the upper portion of the window.

D. *Depth First Search*

The next example is an animation of an algorithm that works on graphs. Starting from a given node, depth first search looks for a way to access all nodes in a graph,

generating a spanning tree of all reachable nodes. The algorithm preserves a set of nodes to be visited, and is depth first, because explores as far as possible along each branch before backtracking.

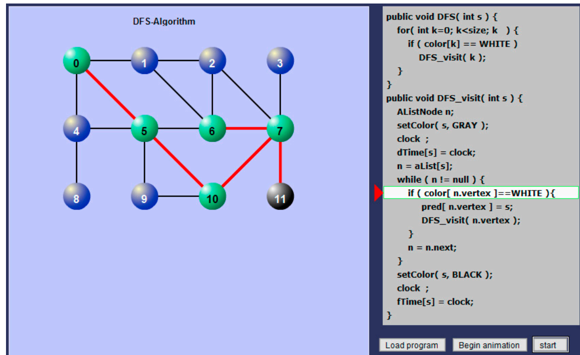

Figure 14. Depth First Search in a graph.

The screenshot in Fig. 14 shows the algorithm running. The spanning tree being formed is colored red. Nodes, whose children have been completely considered, are colored grey. Nodes which have been discovered are colored green.

The screenshot in Fig. 15 shows the final state of the simulation: all nodes have been reached, the spanning tree is complete, and there are no more nodes to continue processing.

The last screenshot of the DFS Algorithm in Fig. 16 shows more information about the internal representation of the graph. You can see the time stamps of the nodes when they are discovered or finished and the adjacency list of the node neighbors.

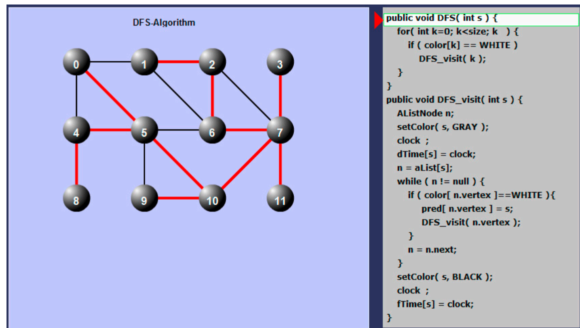

Figure 15. End of the DFS algorithm. The spanning tree is shown in red.

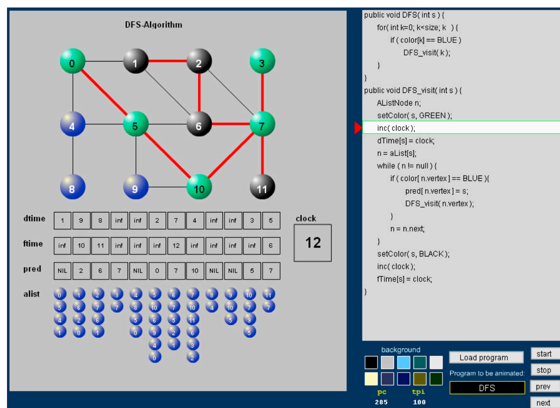

Figure 16. DFS algorithm with a second view of the graph representation.

E. *The towers of Hanoi*

The towers of Hanoi are one of the classical examples used for explaining recursion and one favorite theme of algorithmic animation systems. The screenshot in Fig. 17 shows a Flashdance animation of the Java code shown in the right window. Instrumenting the animation was very simple, scripting commands had to be included in just one function call.

The animation shows the position of the plates during an animation run. The plates are inserted into three poles. Plate number 4 is moving from the central to the left pole. The lines below the poles show the successive movement of the plates: a red line represents a movement from the right to the left, a blue line a movement of a plate from the left to the right. The pseudocode is shown on the right, highlighted at the current instruction.

Now that we have seen Flashdance being used in several algorithmic animations, let us look at a few important features in more detail.

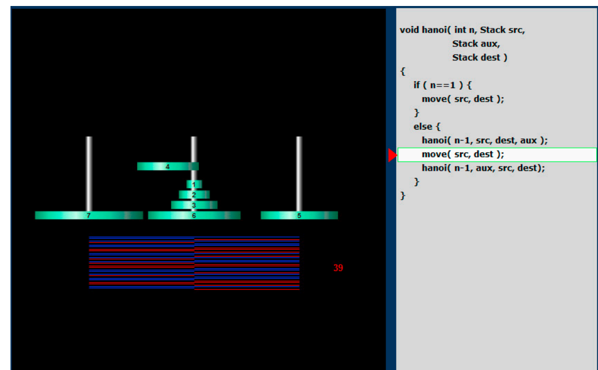

Figure 17. The towers of Hanoi animation.

F. *Reversibility and Overlays*

Reversibility is an important feature for algorithmic animations. This feature gives the viewer the possibility of mentally "zooming" into an operation in order to examine the conditions and context in which a data structure is modified. Reversibility is difficult to implement in systems which do their own rendering (for example Zeus) because it is easier to draw an object on top of an existing picture than it is to remove the object from the composite image. If a single rendering layer is used, the set of pixels covered by an object has to be determined and managed. This imposes a high overhead on the simulation system.

The user can backtrack or go forward in an animation by undoing or redoing a previous operation. This is the approach we followed in Flashdance: when an animation runs, it is made reversible by generating the inverse instructions and saving them on a stack. The user can press the forward button activating so the next operation in the Flashdance code, or she can decide to go backwards pressing the "step back" button which executes the next operation from the undo stack. Reversibility is achieved through the interplay of the programmed sequence of commands and the undo stack. Reversibility is easier to implement in Flash than in other systems, because Flash animations are drawn in layers – the runtime system maintains each layer and redraws it automatically, respecting the object occlusion constraints.

Flashdance commands were defined from the beginning with reversibility in mind: for every operation there is a

corresponding inverse operation. The inverse operation is generated at runtime; each inverse operation is pushed into the undo stack before her corresponding Flashdance instruction is executed.

The inverse operations, for the most important instructions, are shown in Table 1. All inverse instructions are generated when the program is executed.

TABLE I.
SOME FLASHDANCE COMMANDS AND INVERSE OPERATIONS

Command	Inverse command	Comments
remove *name*	**new** *type name x y width height label color*	The arguments *type, name, x, y, width, height, label* and *color* must to be read from object *name* before remove is executed.
new *type name x y width height* [*label*]	**remove** *name*	Deletes object with ID *name*
change *name* $property_1$ $value_1$ [$property_2$ $value_2$]...	**change** *name* $property_1$ $value_{o1}$ [$property_2$ $value_{o2}$]...	The parameters $value_o$ (old value) are read from the object before they are changed.
exchange *name type* [*x y width height label*]	**exchange** *name* $type_o$ x_o y_o $width_o$ $height_o$ $label_o$ $color_o$	The arguments $type_o$, *name*, x_o, y_o, $width_o$, $height_o$, $label_o$ and $color_o$ must be read from the object before, the exchange is executed.
moveTo $name_1$ x_1 y_1 [$name_2$ x_2 y_2] . . .	**moveTo** $name_1$ x_{o1} y_{o1} [$name_2$ x_{o2} y_{o2}] . . .	The old position of the objects must be read to construct the reverse instruction
animTo $name_1$ x_1 y_1 [$name_2$ x_2 y_2] . . . [*path*]	**animTo** $name_1$ x_{o1} y_{o1} [$name_2$ x_{o2} y_{o2}] . . . [*path*]	The position of the objects must be read, before the instruction is executed. The animation path is the same.
Highlight *type* $name_1$ [$name_2$] ...	**Highlight** *type* $name_1$ [$name_2$]...	Both instructions are equal
swap $name_1$ $name_2$ [*path*]	**swap** $name_1$ $name_2$ [*path*]	Both instructions are equal
setView *view-name*	**setView** *original-view*	The original view where the animation was running, must be read before changing it
removeAll	list of **new** instructions	A **new** instruction for each objects is pushed into the stack
setCodeLine *line*	**setCodeLine** $line_o$	The original $line_o$ for the program pointer is used before updating the pointer.

Some Flashdance commands and inverse operations. The first column lists the instructions with its parameters. The second column provides the corresponding inverse instructions. The third column gives some information about the parameters used. The subindex "o" (for original) refers to the object parameters before they are modified by a Flashdance instruction.

Fig. 18 shows the panel for controlling an animation. The name of the Flashdance program is entered in the text window. The button "load program" loads the code and generates the reversible code (undo stack). Pressing "start" lets the simulation run. The button changes its label to "stop"; if pressed again this button stops the simulation. A stopped simulation can be operated in single steps forward or backward, using the buttons "next" and "prev", respectively. The color of the background can be changed by selecting one of the colors on the left. The program counter is shown under the label "pc", and the time interval for a single step under the label "tpi" (time per instruction). This number can be changed, making the animation run faster or more slowly.

A nice example of a reversible animation and its usefulness is this implementation of Dehornoy's algorithm for bringing braids into a canonical form [7]. A braid is a set of *n* lines starting from ordered positions 1 to *n*, which overlap in the way shown in Fig. 19. A braid can be simplified, in order to make it comparable to other braids. Figure 19 shows the start of Dehornoy's algorithm and the construction of a braid from its description as a list of positive and negative numbers, which represent crossings. The screenshot on the right of Fig. 19 has been executed reversibly and brings the construction process some steps back.

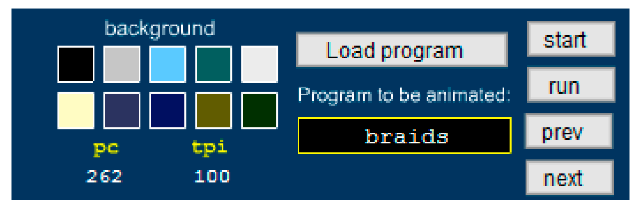

Figure 18. Control panel for a Flashdance animation showing the button for reversible execution.

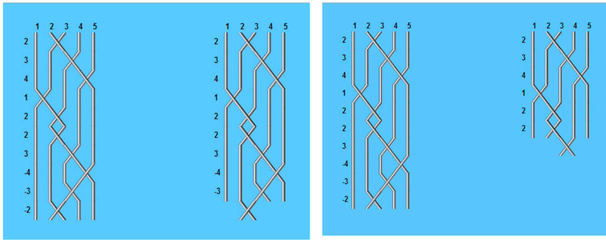

Figure 19. Dehornoy's algorithm running forward (left), and running backwards (right).

Fig. 20 shows the final step of the braid simplification process. As can be seen, only the crossings that remain in the braid to the right are essential. Other crossings are not essential and can be discarded.

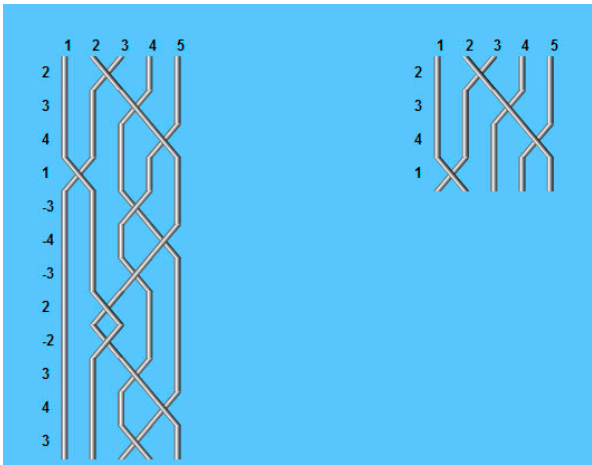

Figure 20. Final simplification: the braid to the left is equivalent to the braid to the right.

G. Overlays

The general idea of overlays is to generate an animation using different superimposed views, which can be switched on and off when the animation runs. This allows the viewer to control the amount of informational detail that she or he wants to receive. Overlays are another way of focusing the attention on the important details of an animation.

Flash animations are based on the concept of animation layers. Layers are superimposed on each other and can be switched off manually using the Flash user interface. In Flashdance we make this functionality available in the simulation window itself. For every view the interpreter creates a button which can be toggled by the user, and which switches on or off the display of an animation view. The views are still present and are updated continuously, but they are made visible or invisible according to the corresponding overlay button setting.

An example for the use of overlays is the animation of an algorithm for finding the convex hull of a set of points. Fig. 21 shows the algorithm running in the Flashdance environment. The points are visible to the left. The algorithm code is on the right. When the algorithm runs, the segments tested as possible components of the convex hull are marked in black. The buttons below the blue window, are the overlay buttons. There are four overlays in this simulation: two for the left side of the convex hull, and two for the right side. With one of the buttons for the left side, the tested segments can be shown or not as black segments. With the other button, it is possible to turn on or off the left side of the convex hull.

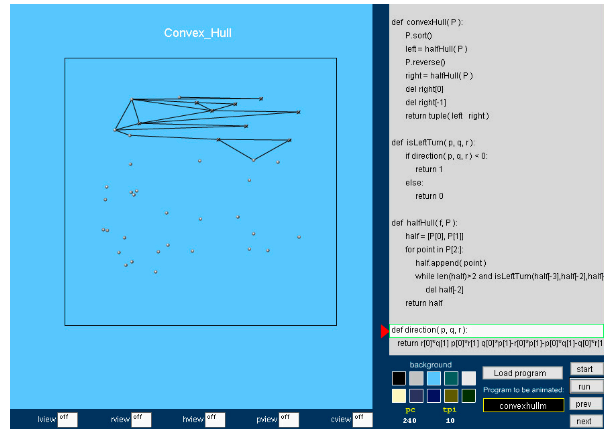

Figure 21. The convex hull algorithm running.

Fig. 22 shows the five overlays of the algorithm superimposed on each other (left side). The right side shows two of the overlays switched off, namely those containing the information about the segments which were tested. The left side marks tested segments in black. For the right side of the convex hull construction, the segments were marked in red. As can be seen with this experiment, the amount of information displayed by the animation can be controlled directly by the user, while the algorithm is running, providing a better way of spatially zooming in and out of an algorithm. Temporal zooming is available in Flashdance through the control of the animation step. Overlays play the same role, from the perspective of the objects shown by the animation.

The use of overlays could allow to produce automatic animations. Using dataflow analysis, the data flow of a program could be automatically distributed on several layers. The user can then just switch off those layers which are not relevant for the operations she wants to focus on.

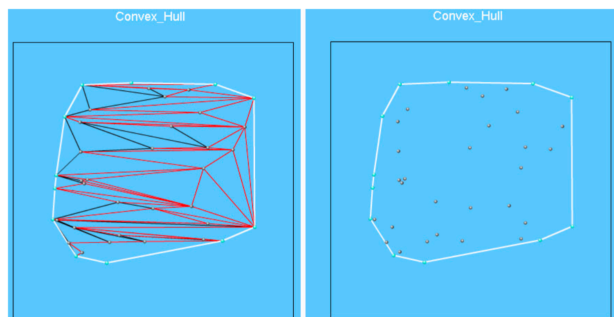

Figure 22. The convex hull algorithm running. On the left side we see all overlays. On the right side, the tested segments overlays for the laft and right convex hull have been switched off. This can be done while the algorithm is running.

H. Instrumented Java classes

An alternative to the inclusion of inline code in an algorithmic animation is to provide instrumented classes, which supersede the standard array or linked data classes, providing the same functionality plus an animation. This approach has been used by authors of algorithmic animation systems [8,9]. Instrumented classes have been

used to visualize lists in Java [10]. A JVALL class overloads the Java LinkedList class and the user can switch between a linear or a circular visualization of a linked list. The same technique has been used to animate arrays in C++ [11].

Flashdance code can be produced by instrumented Java or C++ classes and can be played with the Flashdance interpreter. It would be easy to take a large library of algorithms and data structures, such as LEDA, and add the necessary code in order to have instrumented classes. This work could be done by a group of students now that the necessary infrastructure is in place.

As an illustration of how instrumented Java classes can be used for animating algorithms, we instrumented a small library of Java methods for handling trees. Fig. 23 shows a screenshot of a tree being built by repetitively inserting nodes into a tree (using the corresponding Java method). When the Java program runs, it produces the Flashdance commands which when played animate the sequential construction of the tree.

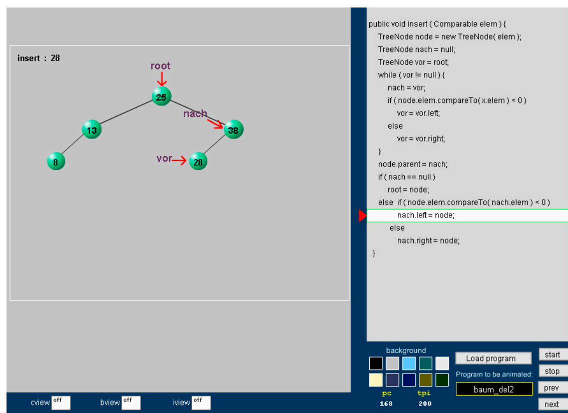

Figure 23. Construction of a tree by an instrumented Java class.

In Fig. 24 we see how a node is being deleted by another Java method. Pointers help to find the node, which is then erased from the tree. The tree pointers are updated as needed. The pointers are drawn on an overlay which can be switched on or off.

This example suffices to show that all the machinery needed to instrument classes in any programming language (Java, C++, Python) is available in the Flashdance system. Significant libraries of instrumented classes in these languages can be created as part of students projects.

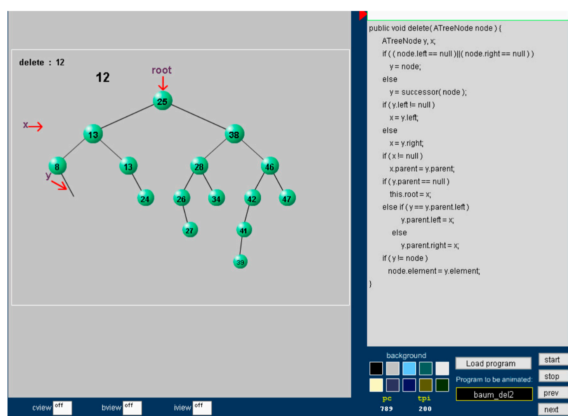

Figure 24. Deleting a node from a tree by an instrumented Java class.

VI. SUMMARY AND DISCUSSION

In this paper, we have shown how to harness the Flash engine for our own purposes, namely algorithmic animation. Flash is a powerful animation system, which has gained widespread acceptance in the Internet. We have shown why: the vector oriented representation used by Flash leads to compact yet good-looking animations, which can be streamed. The difference to Java animations is startling. Flash animations are much smaller, yet more appealing.

This paper discussed ActionScript and the general structure of Flash animations, but the user of Flashdance does not have to be aware of these technicalities. Flashdance is a scripting language which effectively insulates the user from all Flash issues. ActionScript itself is changing; it has evolved from year to year. It would be annoying to have to modify the algorithms already implemented, in case a new version of ActionScript is released. This is not needed, since the Flashdance interpreter takes care of providing the correct interpretation of the scripted code.

The Flashdance script language has been designed with simplicity in mind. It should be easy for students and researchers to animate code in a few minutes. Flashdance offers an option not present in other algorithmic animation systems: overlays. Taking advantage of the fact that Flash animations are organized in layers, we can also organize algorithmic animations in different views. One view can show the algorithm itself, another layer the number of operations or data exchanges. Overlays can be switched on and off by the viewer of the animation and provide a way of transporting more information to the final viewer.

As shown in this paper, instrumenting a program to produce an animation is very simple. It would be easy to instrument classes in object oriented programming languages to extend them with animation capabilities.

Many other algorithmic animation systems have gone into oblivion because the implementation platform has disappeared. Flashdance is a simple scripting language for which players can be written fairly easily. We expect Flash to be around for at least ten more years and ActionScript animations to be upward compatible at least for a decade. In ten more years, it could be that animation features are already part of the operating system and then other animation scripting languages could become more popular.

The popularity of Flash animations is also transforming the Linux world. Players for Linux are already available, and also Web servers for Flash content. The intention of Macromedia is to position Flash as the interface for Rich Internet Applications, that is, applications delivered through the Web, and compatibility across platforms becomes important. Sun Microsystems, for example, started delivering Flash as a component of its Java Desktop in 2003 and the Flash player is an integral part of the open source Mozilla browser. Therefore, using Flash as the graphical front-end for my own system seems to be a good bet for the future.

Flashdance is the first algorithmic animation system which takes advantage of the inherent animation capabilities of an animation engine for the Web. We assume that other systems will emerge in the future and will follow this lead. In this paper we have provided examples of many algorithmic animations with high-

quality graphics, reversibility, overlays, and coupling between a code and an animation window. The interested user can follow an animation, stop it, and zoom on a step by going forwards and backwards.

VII. FUTURE WORK

Right at the beginning of *The Visual Display of Quantitative Information* Edward R. Tufte summarizes his rules for graphical excellence. This paper is concerned with graphical quality, but first and foremost, with algorithmic animation excellence. The difference is important: while Tufte only needs to consider rules for the esthetic and efficient display of static data, we are confronted with the more challenging problem of representing movement, changes, and individual steps of algorithms, that is, dynamics [12]. However, we can paraphrase Tufte's original rules, adapting them to the problem of algorithmic animation [13]. What we obtain is a useful set of heuristic rules which can applied to algorithmic animation. Tufte's modified rules (my modifications are highlighted using italics) are now:

Excellence in algorithmic animation consists of complex ideas communicated with clarity, precision, and efficiency. Algorithmic animation should

- show the data transformations
- induce the viewer to think about the substance rather than about methodology, graphic design, the technology of rendering, or something else
- avoid distorting what the algorithm has to say
- present many steps in small space
- make large data sets coherent
- encourage the eye to compare different algorithm steps
- reveal the algorithm at several levels of detail, from a broad overview to the fine structure
- serve a reasonable clear purpose: description, exploration, learning or decoration
- be closely integrated with the verbal descriptions of the algorithm.

The animations discussed in this paper try to put the spotlight on all data transformations, using explicit movement of data objects or highlighting them. The idea is always to convey the essence of an algorithm to the viewer, making her or him concentrate in the most important operations. The scripting language can deal with small and with large data sets. Small data sets were used extensively in the Flash animations. Large data sets were handled as examples with E-Chalk Animator. The animations try to guide the eye of the observer, connecting a view of the pseudocode with views and overlays of the data. Algorithm steps can be reviewed, either by rerunning the algorithm or by letting it execute backwards. The algorithmic animation tools described in this paper serve the main purpose of teaching students about such algorithms, and both E-Chalk Animator and Flashdance can be enhanced with sound and verbal descriptions.

The scripting language is a general purpose animation tool, and, of course, it can be misused. Bad animations can still be produced with the best animation engine available, in the same way that a blackboard can be used to give bad or good lectures. The animation engine is a clean slate in which the algorithm animator can imprint his or her understanding of an algorithm. The best animations are those in which the mental data structures proposed by the algorithm correspond best to the algorithms data structures. Or to put it in the words of Bertin: "The entire problem is one of augmenting this natural intelligence (of the user, ME) in the best possible way, of finding the artificial memory that best supports our natural means of perception" [14]. This is the intended evolution path for further versions of Flashdance, originally described in [15], and for which new versions are currently in development.

REFERENCES

[1] Rhyne T-M., "Computer Games' Influence on Scientific and Information Visualization," *IEEE Computer*, December 2000, pp. 154–156.

[2] Gloor P., "AACE Algorithm Animation for Computer Science Education," *Proceedings of the 1992 IEEE Workshop on Visual Languages*, Seattle, WA, September 1992, pp. 25–31.

[3] Gloor P., Dynes S. and Lee I., *Animated Algorithms*. MIT Press, Cambridge, MA, 1993. (CD ROM)

[4] "Algorithmic Animation in Computer Science Education with the Flashdance System", 2. Workshop Grundlagen Multimedialen Lehrens und Lernens, Berlin, March 15-18 2004.

[5] Cormen T., Leiserson C. and Rivest R., *Introduction to Algorithms*, MIT Press, Cambridge MA, 1990.

[6] Gardner M., "The Fantastic Combinations of John Conway's Solitaire Game 'Life'," *Scientific American*, Vol. 223, No. 10, 1970, pp. 120–123.

[7] Dehornoy, P., "A fast method of comparing braids," *Advances in Mathematics*, Vol. 125, 1997, pp. 200–235.

[8] Hausner A., "Web Based Animation of Geometric Algorithms", unpublished PhD Thesis, Princeton University, November 2001.

[9] Ben-Ari M., Myller N., Sutinen E. and Tarhio J., "Perspectives on program animation with Jeliot," *Software Visualization: International Seminar*. Dagstuhl Castle, Germany, Lecture Notes in Computer Science Vol. 2269, 2002, pp. 31–45.

[10] Dershem H., McFall R. and Uti N., "Animation of Java Linked Lists," *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, Cincinnati, KY, 2002, pp. 53–57.

[11] Rasala R., "Automatic Array Algorithm Animation in C++," *Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education*, New Orleans, LO, 1999, pp. 257–260.

[12] Foley J. and McMath C., „Dvnamic Process Visualization," *IEEE Computer Graphics and Applications*, Vol. 6, No. 2, March 1986, pp. 16–25.

[13] Tufte E., *The Visual Display of Quantitative Infürmation*, Graphics Press, Cheshire, CT, 1983.

[14] Bertin J., *Semiology of Graphics*, University of Wisconsin Press, Madison, WI, 1983.

[15] Esponda, M, „A New Algorithmic Animation Framework for the Classroom and for the Internet", PhD Thesis, Department of Computer Science, Freie Universität Berlin, 2004.

AUTHOR

M. Esponda is with the University of Applied Sciences Gießen-Friedberg, Friedberg, Hessen, Germany (e-mail: margarita.esponda@mnd.fh-friedberg.de).

Manuscript received 12 January 2008. Published as submitted by the author.