6 Flashdance – An Algorithmic Animation Platform for the Web

6.1 The Flash animation engine

During the founding years of computer science, the development of visualization tools was guided by the interests and possibilities of the academic community. Even in the case of computer languages, most of them were proposed and developed in academia, with the notable exception of FORTRAN (an IBM product). Now, the situation has changed. A multibillion computer and software industry is continually pushing the boundaries of the state of the art. It is very difficult for academic projects to compete against the likes of Microsoft, Sun, Intel or IBM. 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 [Rhyne 00]. Even Java, which has penetrated so much of the educational curriculum, started as a proposal by Sun Microsystems before it was put into the public domain.

If we look back to Chapter 2 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. Java animations still look like graphical products of the 1980s. The reason is that for Java to run on many machines, it had to settle for the minimum common denominator of all these machines, mostly regarding the user interface, i.e., windows, buttons, and graphics. New versions of Java, the Swing library and the 3D extensions have alleviated this problem, but the need for compatibility at the browser level has led to a slow evolution of the language.

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 (Table 6.1). Macromedia Inc. claims an installed base of 436 million users and a projected market penetration of 90% for Flash 6 in December 2003.

Table 6.1: Flash player market penetration (percent of computers with the player installed)

Worldwide Ubiquity of Macromedia Flash by Version - June 2003

	Macromedia Flash 2	Macromedia Flash 3	Macromedia Flash 4	Macromedia Flash 5	Macromedia Flash 6
US	97.4%	97.3%	97.0%	94.8%	86.3%
Canada	97.5%	97.3%	96.8%	95.6%	86.9%
Europe	97.7%	97.7%	97.5%	97.1%	87.2%
Asia	96.1%	95.3%	94.2%	92.6%	82.7%

Sources: Macromedia Inc, NPD Online Worldwide Survey — conducted June 2003

Figure 6.1, taken from the Macromedia Inc. Web site, compares the market penetration of several browser plug-ins for multimedia content for a US sample. The survey was conducted by NPD Online. The Flash player and Java are the more popular platforms, although Flash is more geared towards animation and Java towards general programming.

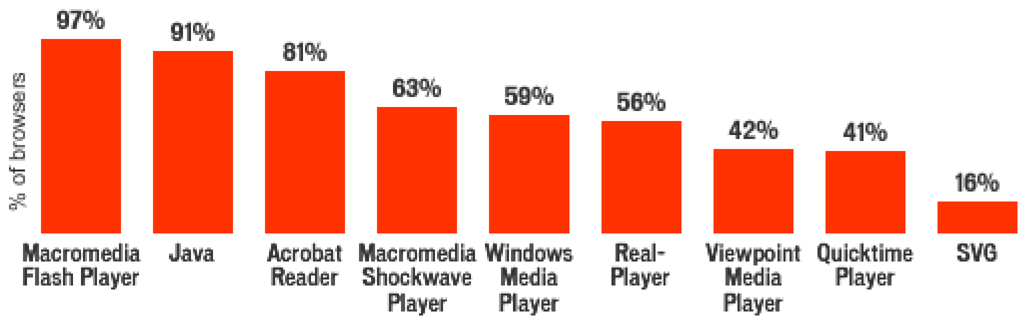


Figure 6.1 Market penetration of browser plug-ins

Therefore, after having dealt in the previous two chapters with algorithmic animations produced for an electronic blackboard, we switch gears in this chapter. Now the emphasis will be put on the production of high-quality animations for the Web, which will be animated with Flash. We want to export those animations to the Web, but we want the animation script to be compatible with E-Chalk animations. With this in mind, I defined a single script language for algorithmic animation but with two distinct players, one for E-Chalk and one for Flash. 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 [Gloor 92, 93]. Java was supposed to provide this functionality, but it has still many deficiencies as an animation language. Thus *Flashdance*.

6.2 Flash animations - basics

I decided to adopt the Flash animation engine for the production of high quality animations mainly as a graphical front-end because of the following reasons:

- *Flash is a de facto standard for the Internet*

As explained above, Flash players have become widely available and Flash is a real alternative to Java, when considering high quality animations. Third party libraries are growing and many are being put in the public domain.

- *Flash animations can be posted in Web pages*

Flash animations can contain interactive buttons and objects, which allow the user to control the parameters of an animation or navigate a Web site.

- *Flash offers esthetically pleasing graphical objects*

Flash is based on vector graphics. This makes the rendering of graphical objects independent of the screen resolution. Flash graphics are of the highest quality possible today.

- *Flash animations can be stored in small files*

Since Flash animations are vector oriented, the files are smaller. Objects reused in an animation are downloaded only once but can be reused many times. Vector objects can be downloaded faster and animations start much faster than Java Applets. This is one of the features that probably explains the popularity of Flash animations.

- *Flash animations are streamed*

Flash animations can start rapidly because they are streamed from a server to the client. In the case of large animations, the first scenes can start playing before the whole file has been received, reducing the waiting time for the viewer.

- *The playing format is in the public domain*

Flash animations are stored as SWF files. The SWF format is open source and is handled by the SWF organization. The SWF format was designed to optimize performance and delivery through a computer network. It is extensible and simple.

- *The Flash ActionScript language*

Macromedia has developed over the last years 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 2004 integrates drawing, animation and programming tools in the same work environment. The ActionScript code reacts to events on the timeline. Algorithms must extensively been modify to be able to produce animations and to make then fit the ActionScript style.

As a result, learning to use the authoring environment is very time intensive. Only the drawing tools are easy to learn. Efficient programming is still difficult with the current programming tools.

Although the syntax of ActionScript is similar to JavaScript, to find bugs in a Flash animation is more difficult because the code is distributed among many components.

The ActionScript language can be very frustrating for the 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.

I decided to use Flash because of two main advantages: 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 produced from any programming language. Less effort is necessary and results are obtained faster.

It is easy to import and export libraries of graphical elements for animations. The algorithms themselves can be written in any language and the animations commands are produced by inline print instructions.

General structure of an animation produced direct with the Flash-GUI.

Fig. 6.2 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.

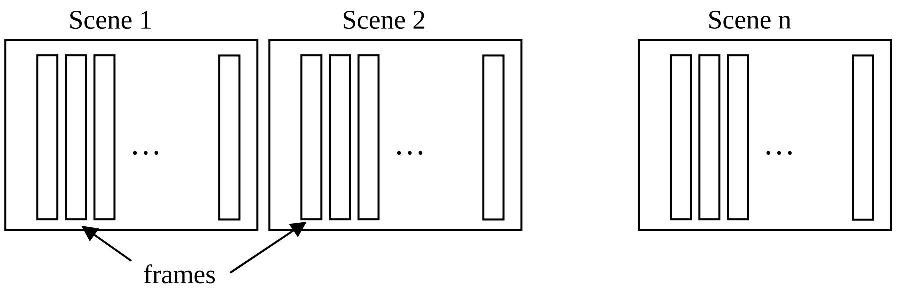


Figure 6.2 A Flash film may contain many scenes, and each scene may have many frames.

Figure 6.3 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.

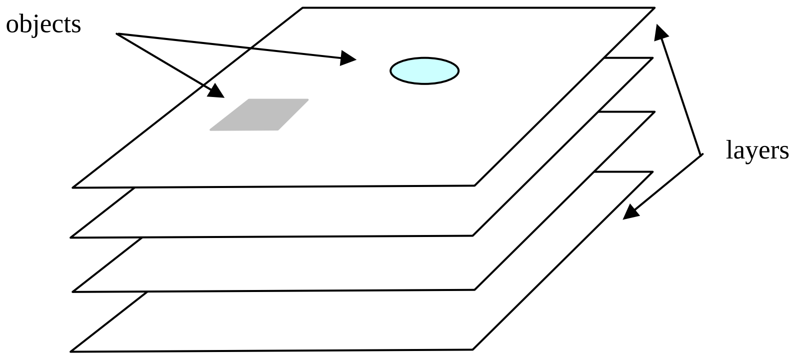


Figure 6.3 Scenes consist of layers, stacked on top of each other. Layers contain drawings and other objects.

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. 6.4 shows an example of a scene with two layers. “Ebene 2” is the upper layer, “Ebene 1”, the lower layer. 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.

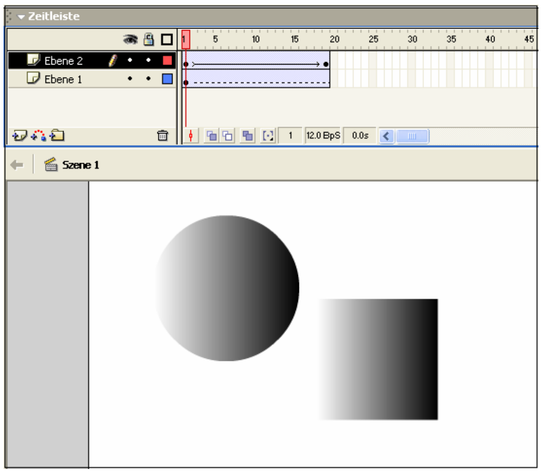


Figure 6.4 The timeline with two layers..

The key frames in this animation are the first and the last in “Ebene 2”. 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.

In the next section we look closer at Flash's scripting language. 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.

6.3 ActionScript

I said before, that Flash animations consist of sequences of frames. Flash animations can also contain interactive objects or components (buttons or check boxes, for example) which can be activated by the user. The animation flow changes according to user actions and therefore some way of specifying such changes is needed. ActionScript is used to produce non-linear animations, i.e. animations without a fixed frame sequence.

ActionScript is based on Netscape's JavaScript. The European Computer Manufacturers Association adopted JavaScript in 1998 as the model for a standard scripting language for Web applications. The ECMA-262 standard defines a language very similar to JavaScript. Compliant implementations must cover the language specification but can provide additional data types and features not specified in the standard. ActionScript is mostly ECMA-262 compliant, although there are a few incompatibilities, mainly regarding data types, exception handling, and case sensitivity. Following the ECMA standard closely is a way for software companies to guarantee that their products can be adapted to current and future Web services with low effort.

The main strength of ActionScript is its consistency across different operating systems and browsers. A Flash animation rarely fails. The players have been written by Macromedia and they provide the same animation on every computer. The vector graphics format provides a way of scaling the output, which is difficult to do in Java and one of the most frequent problems encountered when designing Java Applets for the Internet.

As we saw above, Flash animation can be produced by drawing frames, one after the other. However, frames can also be declared "action frames", and in this case code is included in the action window of the frame to tell the animation engine which actions to perform.

First, a simple example: ActionScript code can be included as "actions" to be followed by symbols when certain events take place. A circle, which has been transformed into a movieclip symbol, can contain this code in its action field:

```
onClipEvent (load) {
// sets the initial x position of the circle
        this._x = 70;
}


onClipEvent (enterFrame) {
//moves the circle 7 pixels to the right when entering this frame
        this._x = this._x+7;
}
```

This code means that the coordinates of the circle are set to 70 when the movieclip is first loaded. When each subsequent frame starts, the x-coordinate of the circle is shifted 7 pixels to the right. It is therefore fairly easy to associate an action with symbols and events in a Flash animation. The actions can be triggered by the animation itself, or by user interaction.

ActionScript code can "catch" events when they happen. Events can be user or system events. If the user clicks with the mouse, this is a mouse down event and it is passed to all objects in all overlays in the scene currently running. The Action-

Script code can decide to react to this event or can ignore it. System events are produced when a new frame is loaded, when a frame is started, when an exception occurs, etc. System events coordinate all objects playing in an animation. Figure 6.5 shows an animation running and receiving events. All objects in all layers receive the events.

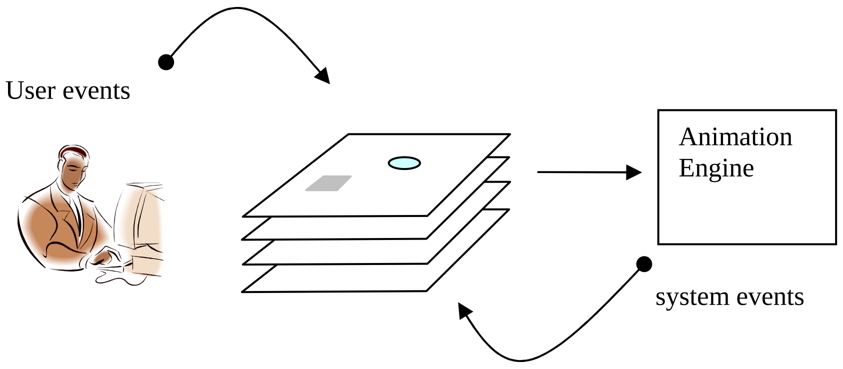


Figure 6.5 Events are produced by the user or by the system.

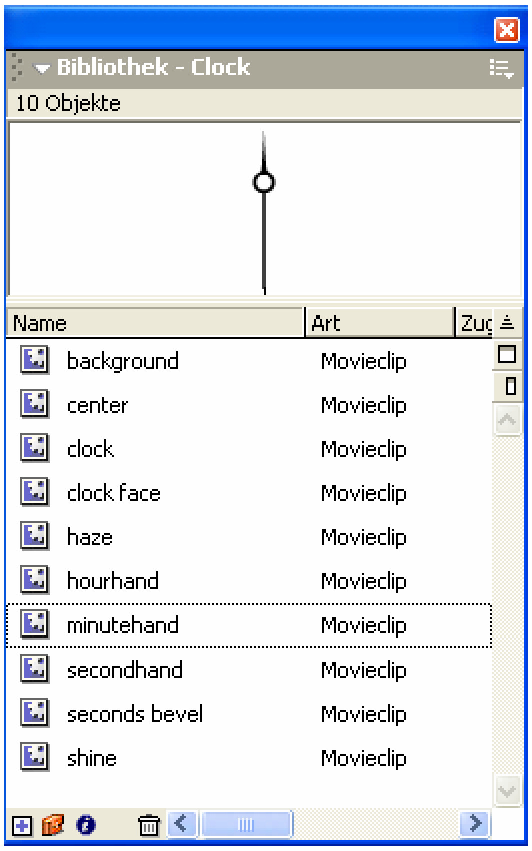
A more complex example, from the Flash library of animations can help to summarize what I have said above.

Figure 6.6 shows an animation of a clock. The hours, minutes and seconds hands move accordingly to the computer's internal time. The animation consists of one single frame, played repetitively, and 11 overlays.

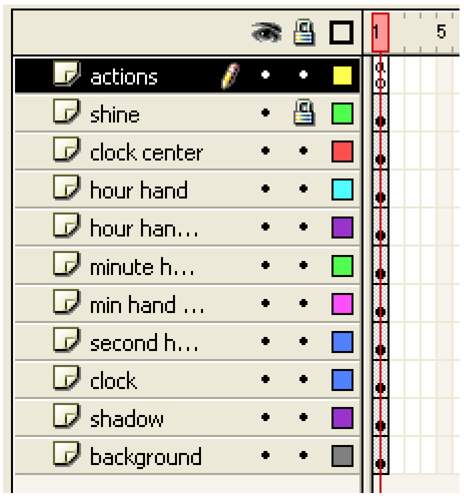
Figure 6.6 An animated clock.

Figure 6.7 shows the timeline and the library of objects for this animation. The timeline contains one frame and layers for different parts of the clock. The shine of the clock cover is above all other graphical overlays. The hour hand is above the minute hand, and so on. An "empty" overlay (actions) contains the code for the animation. There are no graphical objects in this overlay, only code.

The graphical objects to be animated are defined as movieclips. The minute-hand has been selected in the menu, and its graphical definition is visible in the small window.



Name	Art
background	Movieclip
center	Movieclip
clock	Movieclip
clock face	Movieclip
haze	Movieclip
hourhand	Movieclip
minutehand	Movieclip
secondhand	Movieclip
seconds bevel	Movieclip
shine	Movieclip



actions
shine
clock center
hour hand
hour han...
minute h...
min hand ...
second h...
clock
shadow
background

Figure 6.7 Layers and timeline of the clock animation (left picture). The library of movieclips for the animation (right picture).

The main code in the actions overlay is the following:

```
myDate = new Date();

// rotate clock hands and shadows
hourHand._rotation = myDate.getHours()*30+(myDate.getMinutes()/2);
minuteHand._rotation = myDate.getMinutes()*6+(myDate.getSeconds()/10);
secondHand._rotation = myDate.getSeconds()*6;
```

This code first generates a new object of type Date. The object has three methods: getHours(), getMinutes() and getSeconds(). Using the methods, it is possible to compute the position (angle of rotation) of the hours, minutes and seconds hands. The parameter "_rotation" of each hand is set, and the frame is finished. This parameter "_rotation" is a property of all movieclips which can be modified to produce an animation.

Since the animation runs in a loop, the position of the three hands is being updated continually.

These examples should suffice as an overview of the capabilities of the Action-Script language. Additional details can be found in [Müller 03].

6.4 A first overview of the Flashdance-System

Figure 6.8 shows an overview of the architecture of the Flashdance system. An algorithm provides events which can activate instrumented classes (see section 6.11), 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.

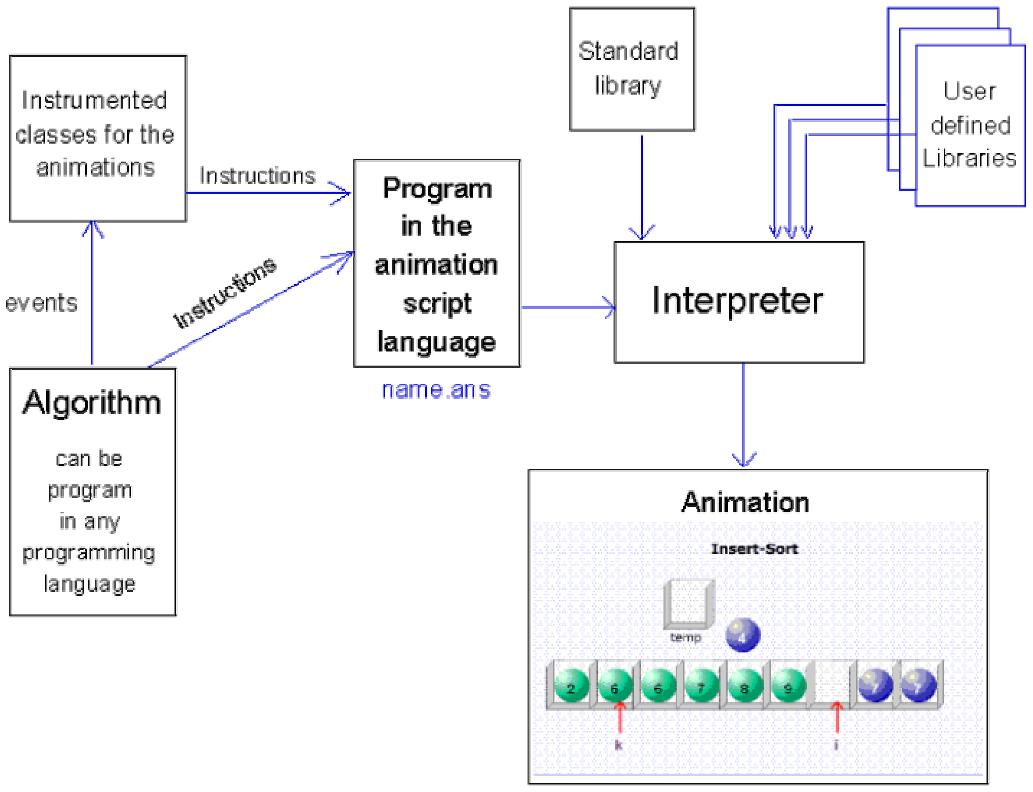


Figure 6.8. Architecture of the Flashdance system

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 my 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 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).

6.5 The Flashdance script language

The Flashdance animation language was designed for simplicity [Esponda 04a]. 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.

I had already listed the different instructions of the Flashdance script language in Chapter 4, without going into their description. Here I offer a more detailed overview of them.

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.

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

special primitive *object-types*:

View *view-name x y width height* [*label*] [*background-colour*]
String *object-name x y width height string-text* [*colour*] [*style*]
Line *object-name* x_1 y_1 x_2 y_2 [*line-width*] [*colour*]
Rectangle *object-name x y width height* [*line-width*] [*colour*]
Oval *object-name x y width height* [*line-width*] [*colour*]

With this instruction 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 position 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 will be 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-name$_1$ . . . object-name$_n$*

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 property$_1$ value$_1$ property$_2$ value$_2$ . . . property$_n$ value$_n$*

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-name*$_1$ *x*$_1$ *y*$_1$ . . . *object-name*$_n$ *x*$_n$ *y*$_n$

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

animTo *object-name*$_1$ *x*$_1$ *y*$_1$ . . . *object-name*$_n$ *x*$_n$ *y*$_n$ [*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 use as an option. If no path is specified the movement is a straight line.

highlight *highlight-type* *object-name*$_1$ . . . *object-name*$_n$

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-name*$_1$ *object-name*$_2$ [*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.

6.6 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 with our own system, when we can define a library of graphically appealing objects using the tools of the system.

Flashdance animations use symbols stored as movieclip in a library of objects. The *new* command accesses these objects and makes them available for the animation. For my first animations I defined a simple library of objects using the Flash editing tools. Figure 6.9 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).

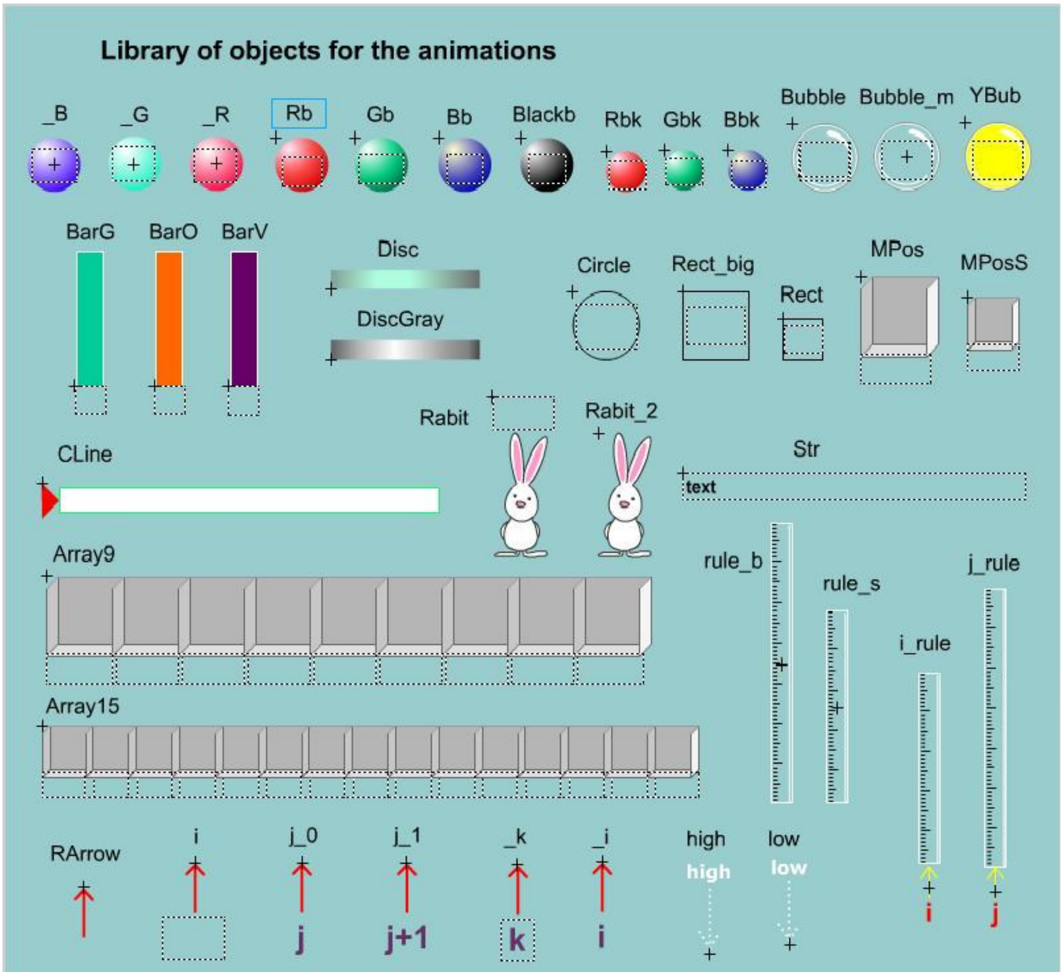


Figure 6.9 Library of some of the animation objects available in Flashdance.

In Figure 6.9 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 labelled with the number they represent. Array locations can be labelled 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.

6.7 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 [Cormen 90]. 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 o*x* o*y*”, 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 indexes (pointers), a “new” instruction has to be defined for each index (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.

6.8 The interpreter

The general structure of the Flashdance interpreter is shown in Figure 6.10. 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.

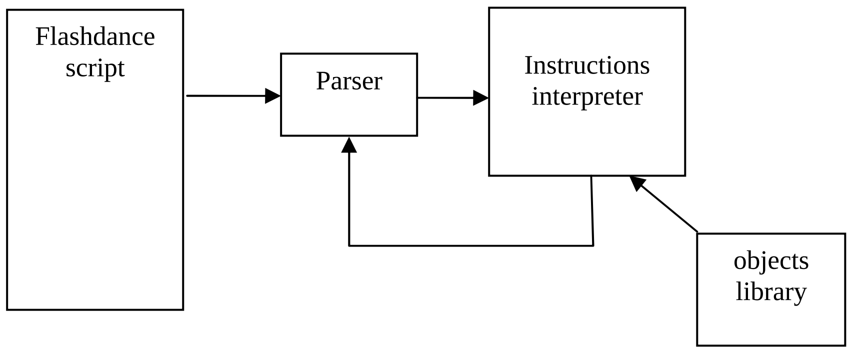


Figure 6.10 Block diagram of the Flashdance interpreter.

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…

6.9 Examples of animations

In this section, I 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.

6.9.1 The Game of Life

The Game of Life was invented by John Conway around 1970 [Gardner 70]. 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.

Figure 6.11 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.

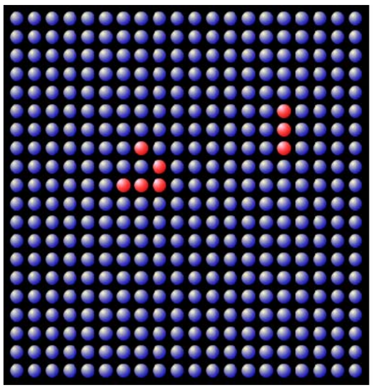


Figure 6.11 Initial configuration for a game of Life. Red cells are alive, blue cells are dead.

Figure 6.12 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.

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.

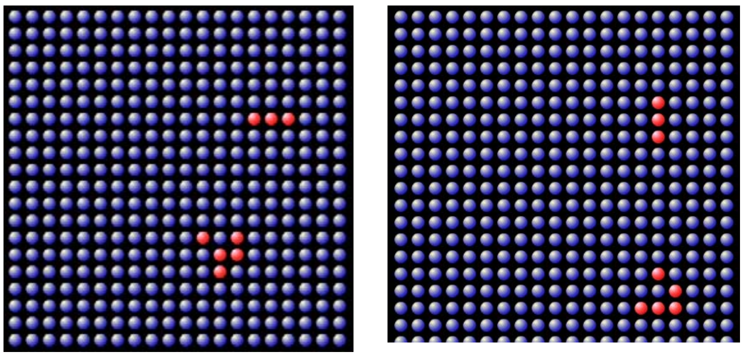


Figure 6.12 A glider moving to the right, a blinker blinking on place.

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

6.9.2 Sorting Algorithms

Insertion Sort

Our next example is the insertion or insert sort algorithm. Figure 6.13 shows the image of the animation running. The sorting algorithm was written in Java and it produced the animation script for this example.

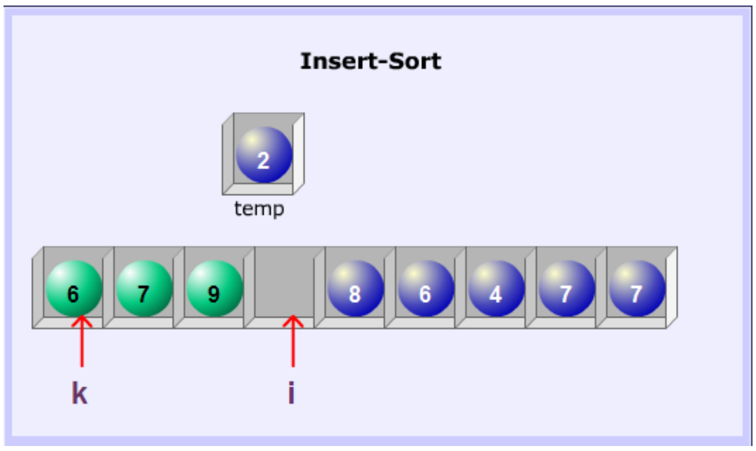


Figure 6.13 Insertion sort, illustrated with an array of boxes.

The algorithm is sorting nine numbers, represented by the spheres. Numbers already inserted into the array are colored green. In the picture, the number 2 has been copied to a temporary variable, from which it will be inserted at the appropriate position, shifting first some green numbers to the right.

Figure 3.14 and Figure 6.15 show the progression of the animation. The numbers have been shifted and the 2 is inserted in front of the array. The index j points at the last position which has been sorted, the index k+1 at the position where a number is inserted into the array. The animation runs showing smooth movements of the objects (picture in the lower left). The last picture shows the sorted array.

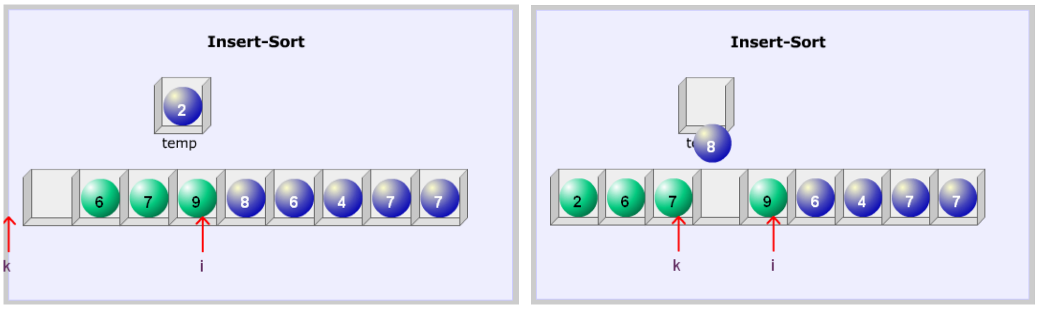


Figure 6.14 The number 2 input at the beginning of the array, after having shifted the green colored spheres.

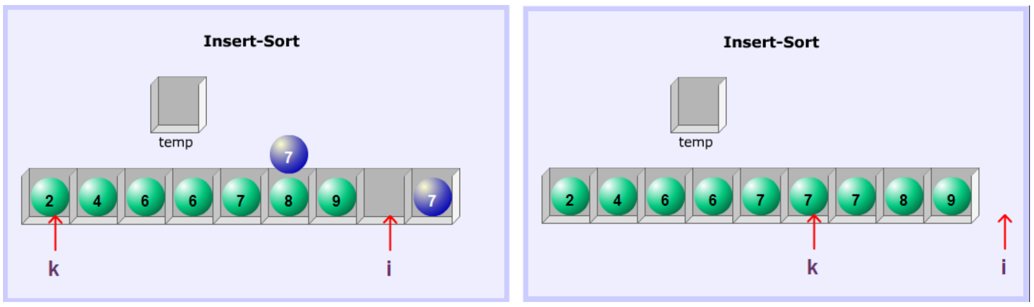


Figure 6.15 Further progression and end of the algorithm.

In this animation only three kind of graphical objects were used: a box container for variables, spheres to represent the numbers and the index-objects (i, k). The labels of the spheres correspond to the stored numbers.

Quicksort

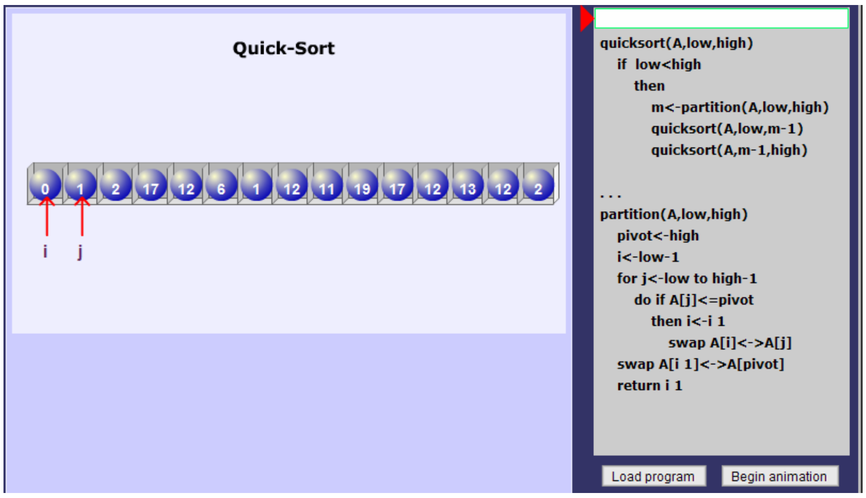
A further improvement to this kind of animations is to include the Java code of the algorithm in an additional view, which is played alongside the algorithm. This has done with another version of the Quicksort algorithm. The Java code is written in an auxiliary text file, which is read by the interpreter. 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.

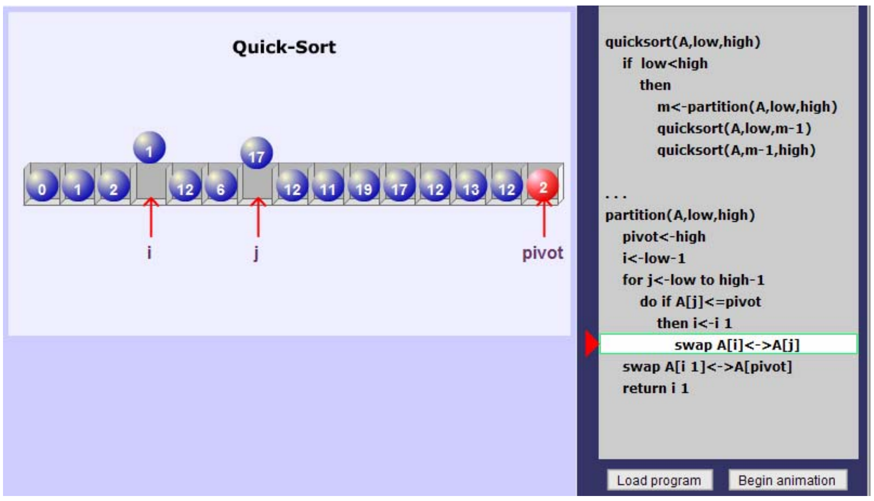
Figure 6.16 shows the start of the animation. The Java code has been loaded and the array has been initialized, as well as the pointers *i* and *j*.



```
quicksort(A,low,high)
  if low<high
    then
      m<-partition(A,low,high)
      quicksort(A,low,m-1)
      quicksort(A,m-1,high)

...
partition(A,low,high)
  pivot<-high
  i<-low-1
  for j<-low to high-1
    do if A[j]<=pivot
      then i<-i 1
        swap A[i]<->A[j]
  swap A[i 1]<->A[pivot]
  return i 1
```

Figure 6.16 Start of the Quicksort animation.



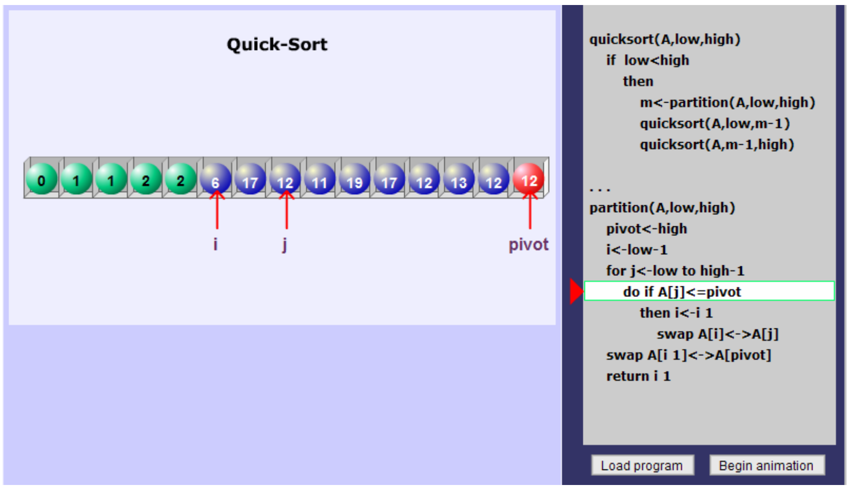
```
quicksort(A,low,high)
  if low<high
    then
      m<-partition(A,low,high)
      quicksort(A,low,m-1)
      quicksort(A,m-1,high)

...
partition(A,low,high)
  pivot<-high
  i<-low-1
  for j<-low to high-1
    do if A[j]<=pivot
      then i<-i 1
        swap A[i]<->A[j]
  swap A[i 1]<->A[pivot]
  return i 1
```

Figure 6.17 Exchange of two numbers at the index positions *i* and *j*.

In the next screenshot, Figure 6.17, Quicksort is exchanging two numbers (1 and 17), and the pseudocode is highlighted at the “swap” operation.

In the next screenshot, Figure 6.18, 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.

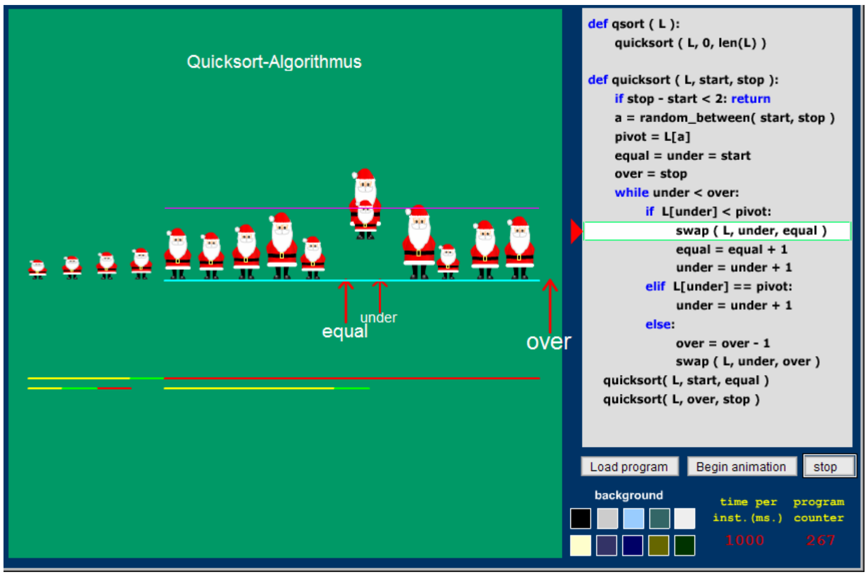


```
quicksort(A,low,high)
  if  low<high
    then
      m<-partition(A,low,high)
      quicksort(A,low,m-1)
      quicksort(A,m-1,high)

. . .
partition(A,low,high)
  pivot<-high
  i<-low-1
  for j<-low to high-1
    do if A[j]<=pivot
      then i<-i 1
        swap A[i]<->A[j]
  swap A[i 1]<->A[pivot]
  return i 1
```

Figure 6.18 Sorted subarrays are 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.



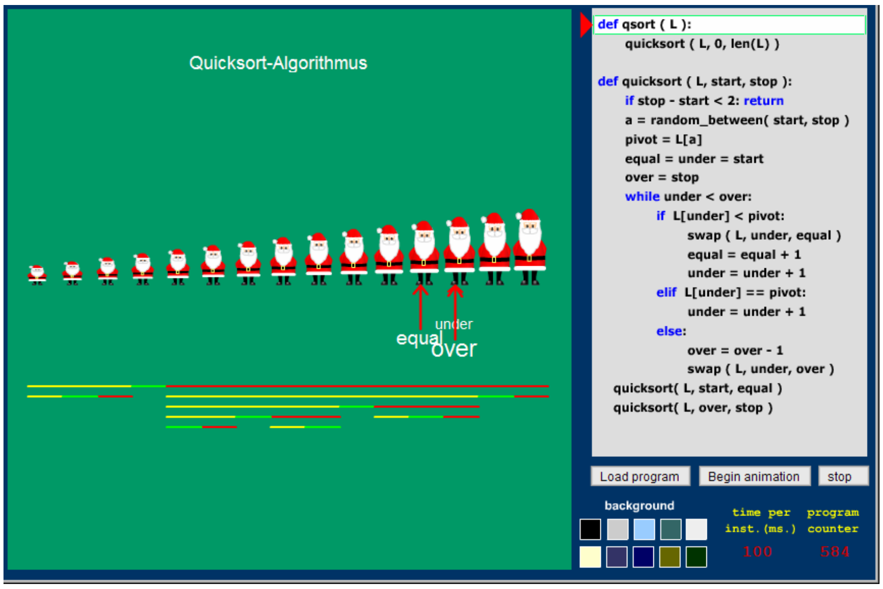
```
def qsort ( L ):
    quicksort ( L, 0, len(L) )

def quicksort ( L, start, stop ):
    if stop - start < 2: return
    a = random_between( start, stop )
    pivot = L[a]
    equal = under = start
    over = stop
    while under < over:
        if  L[under] < pivot:
            swap ( L, under, equal )
            equal = equal + 1
            under = under + 1
        elif  L[under] == pivot:
            under = under + 1
        else:
            over = over - 1
            swap ( L, under, over )
  quicksort( L, start, equal )
  quicksort( L, over, stop )
```

Figure 6.19 Quicksort in Python (code in the right window). The magnitude of the numbers is represented by the size of the objects. The horizontal colored lines show the displacement of the indices on each recursive pass.

The screenshot in Figure 6.19 shows a second implementation of Quicksort, now in Python (as in section 6.6). A simple change of the objects selected from the animation library yields the row of Santa Claus with different sizes. The algorithm works with three pointers: all elements below the “equal” pointer are lower than the pivot. All elements between the “equal” and “under” pointers are equal to the pivot. All elements above the “over” pointer are larger than the pivot. In the screenshot two elements are being exchanged.

The animation highlights the Python code (on the right) and uses another view below the array being sorted. The green segment shows the position of the pivot in one pass. The yellow line shows the elements smaller than the pivot. The red line shows the position of the elements larger than the pivot. A recursive call yields a new line level. Figure 6.20 shows the end of the animation. All array elements are sorted and the colored lines below illustrate the sequence of recursive calls and the array used for them.



```
def qsort ( L ):
    quicksort ( L, 0, len(L) )

def quicksort ( L, start, stop ):
    if stop - start < 2: return
    a = random_between( start, stop )
    pivot = L[a]
    equal = under = start
    over = stop
    while under < over:
        if  L[under] < pivot:
            swap ( L, under, equal )
            equal = equal + 1
            under = under + 1
        elif  L[under] == pivot:
            under = under + 1
        else:
            over = over - 1
            swap ( L, under, over )
  quicksort( L, start, equal )
  quicksort( L, over, stop )
```

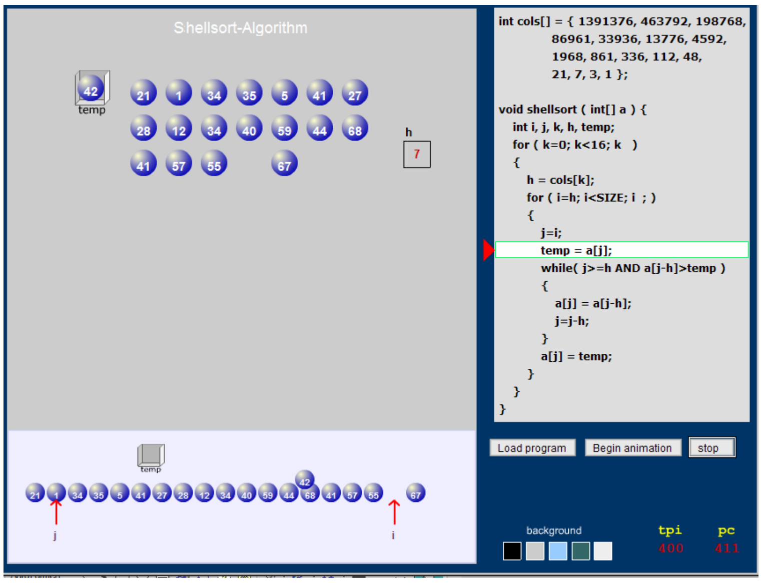
Figure 6.20 Final state of the Quicksort animation.

This animation was featured in the Web site of the Institute of Computer Science, FU Berlin, during the Christmas season of 2003.

Shellsort

The next example, the Shellsort algorithm, is usually difficult to explain. In Shellsort, sorting is done by dividing an array into *h* virtual arrays, using the position of each element modulo *h*. The equivalence classes modulo *h* produce *h* pieces of the array. Each piece is then sorted using insertion sort. The constant *h* is progressively decreased until it is equal to 1.

The idea of the algorithm is to move small elements fast to the beginning of the array, and large elements fast to the back. The same positions of the different pieces of the array are sorted in parallel with insertion sort. Exchanges, when *h* is still large, lead to long position jumps. When *h* comes down to 1, Shellsort dilutes into insertion sort, but with a very favorable array, in which not many element shifts will be needed.



```
int cols[] = { 1391376, 463792, 198768,
           86961, 33936, 13776, 4592,
           1968, 861, 336, 112, 48,
           21, 7, 3, 1 };

void shellsort ( int[] a ) {
   int i, j, k, h, temp;
   for ( k=0; k<16; k   )
   {
      h = cols[k];
      for ( i=h; i<SIZE; i  ; )
      {
         j=i;
         temp = a[j];
         while( j>=h AND a[j-h]>temp )
         {
            a[j] = a[j-h];
            j=j-h;
         }
         a[j] = temp;
      }
   }
}
```

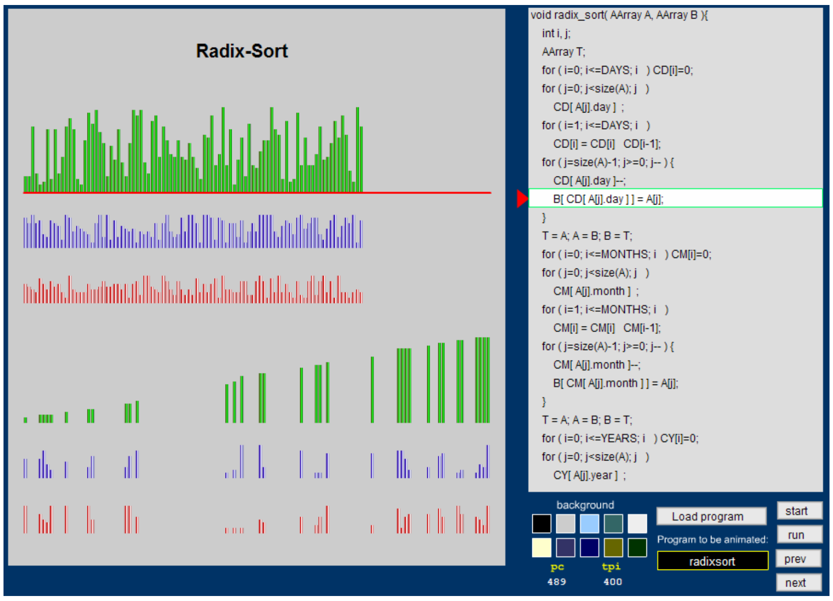
Figure 6.21 Shellsort running. The step size is *h* = 7. The virtual arrays are on top, the array being sorted below, in the view with a lighter background.

The screenshot shows the original array (at the botton) and seven equivalence classes being sorted independently (for *h* equal to 7, as shown in the small square). Code is also highlighted to the right. When a pass of the algorithm finishes, the elements rearrange into a new set of equivalence classes, corresponding to the change of *m*.

Radix Sort

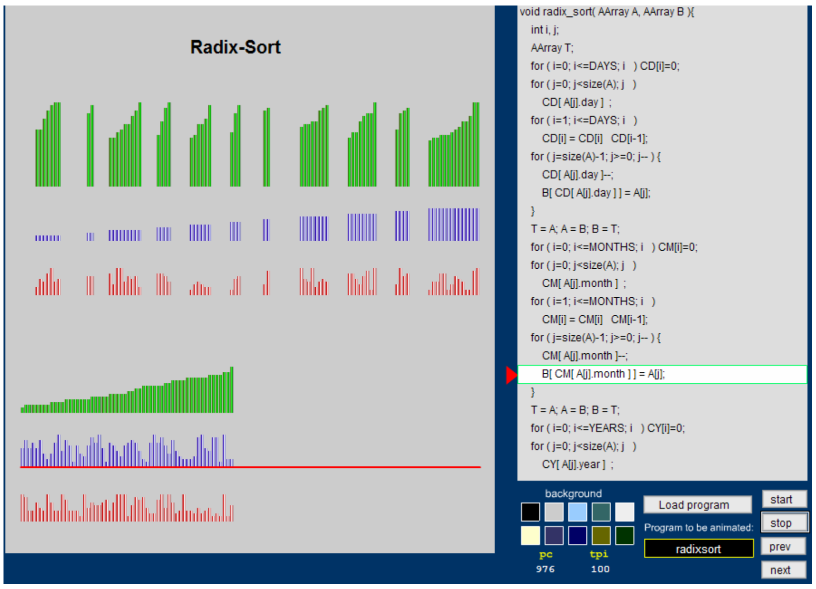
Radix sort is a sorting algorithm with linear complexity. Figure 6.22 shows the algorithm running, using a set of dates as the numbers to be sorted. The first set of bars shows the days, the second row the months, and the third row the years. A date is composed on one bar in each row (one day, one month, and one year). Radix sort starts sorting first the days, as shown in Figure 6.22, 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 (Figure 6.23, 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.



```
void radix_sort( AArray A, AArray B ){
   int i, j;
   AArray T;
   for ( i=0; i<=DAYS; i   ) CD[i]=0;
   for ( j=0; j<size(A); j   )
      CD[ A[j].day ]  ;
   for ( i=1; i<=DAYS; i   )
      CD[i] = CD[i]   CD[i-1];
   for ( j=size(A)-1; j>=0; j-- ) {
      CD[ A[j].day ]--;
      B[ CD[ A[j].day ] ] = A[j];
   }
   T = A; A = B; B = T;
   for ( i=0; i<=MONTHS; i   ) CM[i]=0;
   for ( j=0; j<size(A); j   )
      CM[ A[j].month ]  ;
   for ( i=1; i<=MONTHS; i   )
      CM[i] = CM[i]   CM[i-1];
   for ( j=size(A)-1; j>=0; j-- ) {
      CM[ A[j].month ]--;
      B[ CM[ A[j].month ] ] = A[j];
   }
   T = A; A = B; B = T;
   for ( i=0; i<=YEARS; i   ) CY[i]=0;
   for ( j=0; j<size(A); j   )
      CY[ A[j].year ]  ;
```

Figure 6.22 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.

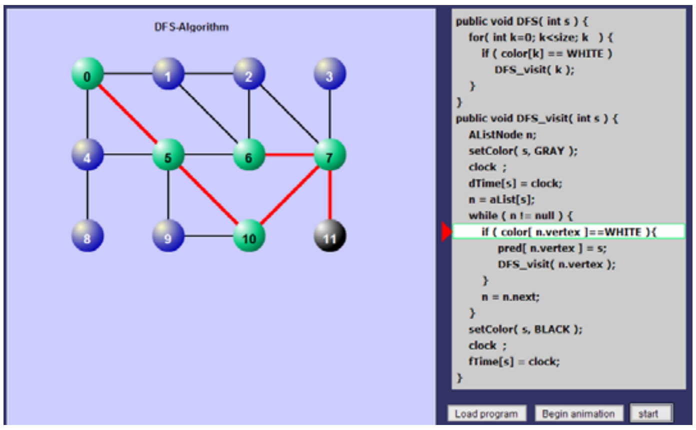


```
void radix_sort( AArray A, AArray B ){
   int i, j;
   AArray T;
   for ( i=0; i<=DAYS; i   ) CD[i]=0;
   for ( j=0; j<size(A); j   )
      CD[ A[j].day ]  ;
   for ( i=1; i<=DAYS; i   )
      CD[i] = CD[i]   CD[i-1];
   for ( j=size(A)-1; j>=0; j-- ) {
      CD[ A[j].day ]--;
      B[ CD[ A[j].day ] ] = A[j];
   }
   T = A; A = B; B = T;
   for ( i=0; i<=MONTHS; i   ) CM[i]=0;
   for ( j=0; j<size(A); j   )
      CM[ A[j].month ]  ;
   for ( i=1; i<=MONTHS; i   )
      CM[i] = CM[i]   CM[i-1];
   for ( j=size(A)-1; j>=0; j-- ) {
      CM[ A[j].month ]--;
      B[ CM[ A[j].month ] ] = A[j];
   }
   T = A; A = B; B = T;
   for ( i=0; i<=YEARS; i   ) CY[i]=0;
   for ( j=0; j<size(A); j   )
      CY[ A[j].year ]  ;
```

Figure 6.23 In the next sort sweep the dates are ordered by month and are copied from the lower to the upper portion of the window.

6.9.3 Depth and Breadth First Search

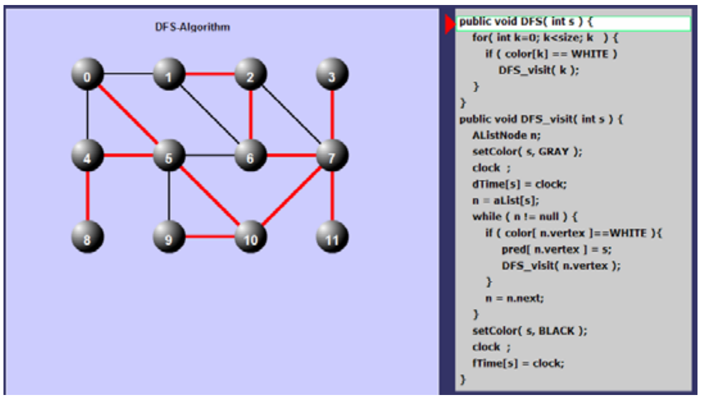
The next example is an animation of an algorithm that works on graphs. Starting from a given node, depth first search looks for a way to access all nodes in a graph, generating a spanning tree of all reachable nodes. The algorithm preserves a set of nodes to be visited, and is depth first, because explores as far as possible along each branch before backtracking.



```
public void DFS( int s ) {
   for( int k=0; k<size; k   ) {
      if ( color[k] == WHITE )
         DFS_visit( k );
   }
}
public void DFS_visit( int s ) {
   AListNode n;
   setColor( s, GRAY );
   clock  ;
   dTime[s] = clock;
   n = aList[s];
   while ( n != null ) {
      if ( color[ n.vertex ]==WHITE ){
         pred[ n.vertex ] = s;
         DFS_visit( n.vertex );
      }
      n = n.next;
   }
   setColor( s, BLACK );
   clock  ;
   fTime[s] = clock;
}
```

Figure 6.24 Depth First Search in a graph.

The screenshot in Figure 6.24 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.

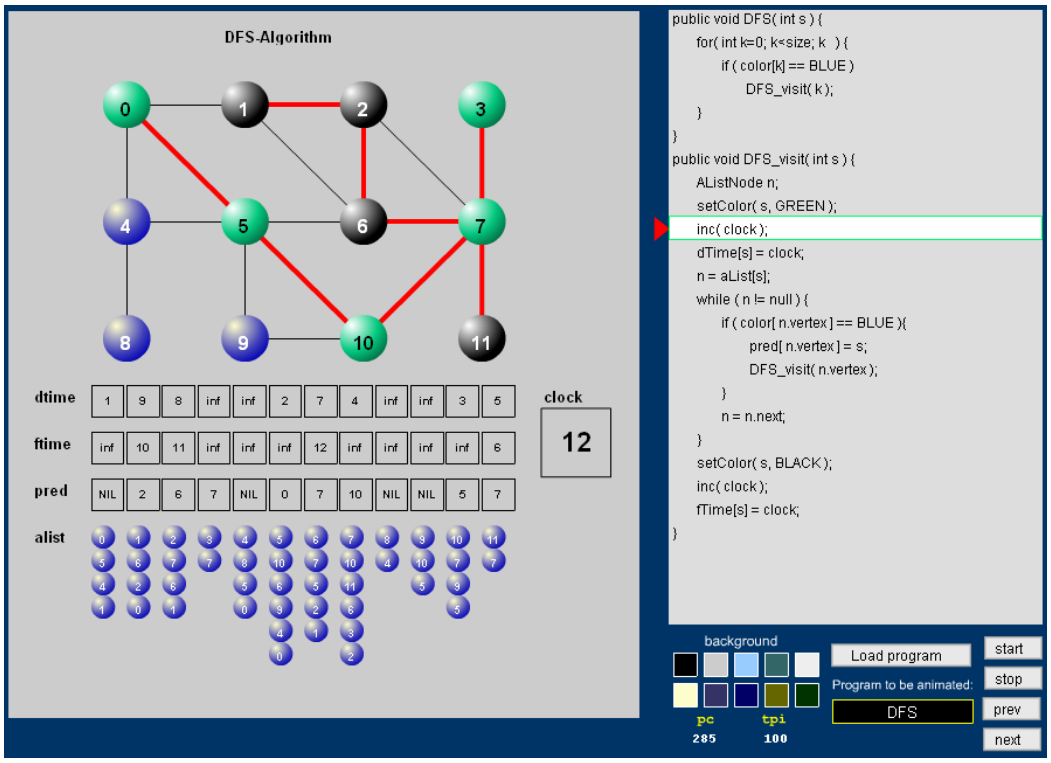


```
public void DFS( int s ) {
   for( int k=0; k<size; k   ) {
      if ( color[k] == WHITE )
         DFS_visit( k );
   }
}
public void DFS_visit( int s ) {
   AListNode n;
   setColor( s, GRAY );
   clock  ;
   dTime[s] = clock;
   n = aList[s];
   while ( n != null ) {
      if ( color[ n.vertex ]==WHITE ){
         pred[ n.vertex ] = s;
         DFS_visit( n.vertex );
      }
      n = n.next;
   }
   setColor( s, BLACK );
   clock  ;
   fTime[s] = clock;
}
```

Figure 6.25 End of the DFS algorithm. The spanning tree is shown in red.

The screenshot in Figure 6.25 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 Figure 6.26 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 neighbours.



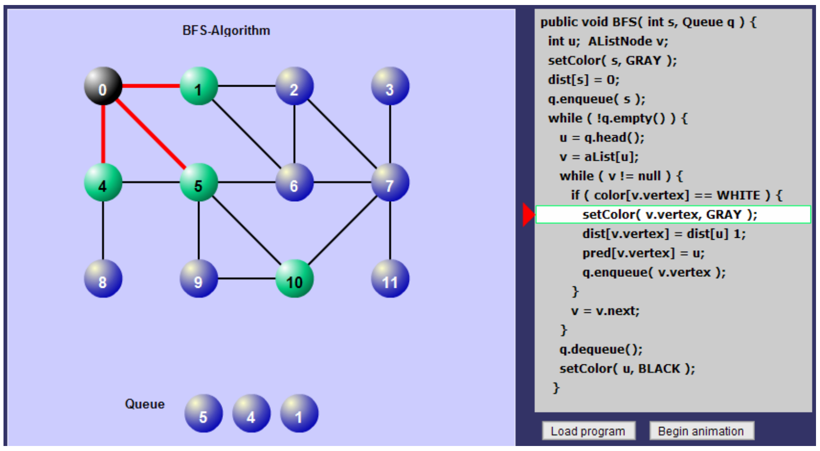
dtime	1	9	8	inf	inf	2	7	4	inf	inf	3	5
ftime	inf	10	11	inf	inf	inf	12	inf	inf	inf	inf	6
pred	NIL	2	6	7	NIL	0	7	10	NIL	NIL	5	7

clock 12

```
public void DFS( int s ) {
    for( int k=0; k<size; k  ) {
        if ( color[k] == BLUE )
            DFS_visit( k );
    }
}
public void DFS_visit( int s ) {
    AListNode n;
    setColor( s, GREEN );
    inc( clock );
    dTime[s] = clock;
    n = aList[s];
    while ( n != null ) {
        if ( color[ n.vertex ] == BLUE ){
            pred[ n.vertex ] = s;
            DFS_visit( n.vertex );
        }
        n = n.next;
    }
    setColor( s, BLACK );
    inc( clock );
    fTime[s] = clock;
}
```

Figure 6.26 DFS algorithm with a second view of the graph representation.

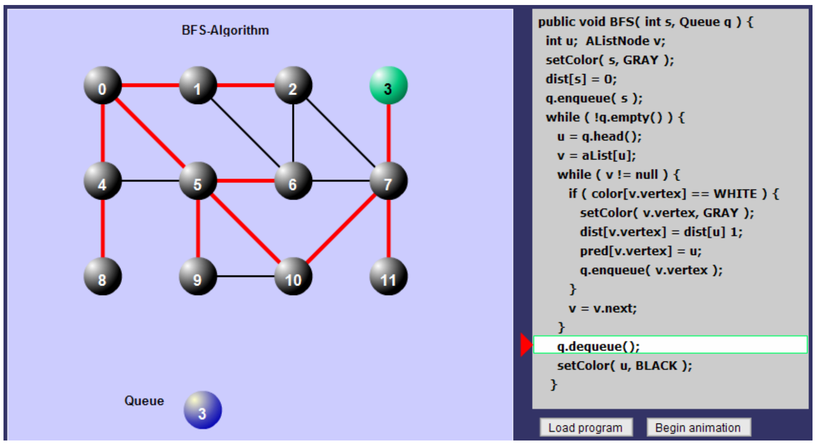
The screenshots in Figure 6.27 and Figure 6.28 show a search operation running, but now in breadth-first mode. Here, a node and all its siblings are processed first, before considering their descendants.



```
public void BFS( int s, Queue q ) {
 int u;  AListNode v;
 setColor( s, GRAY );
 dist[s] = 0;
 q.enqueue( s );
 while ( !q.empty() ) {
   u = q.head();
   v = aList[u];
   while ( v != null ) {
     if ( color[v.vertex] == WHITE ) {
       setColor( v.vertex, GRAY );
       dist[v.vertex] = dist[u] 1;
       pred[v.vertex] = u;
       q.enqueue( v.vertex );
     }
     v = v.next;
   }
   q.dequeue();
   setColor( u, BLACK );
 }
```

Figure 6.27 Breadth First Search in a graph. The queue of nodes is shown at the bottom.

In this example, an additional source of information has been added to the representation. A second view at the bottom of the animation shows the queue of nodes to be finished. Since node descendants are entered at the back of the queue, all node siblings will be processed before their descendants. The BFS is finish when the queue becomes empty.



```
public void BFS( int s, Queue q ) {
  int u;  AListNode v;
  setColor( s, GRAY );
  dist[s] = 0;
  q.enqueue( s );
  while ( !q.empty() ) {
    u = q.head();
    v = aList[u];
    while ( v != null ) {
      if ( color[v.vertex] == WHITE ) {
        setColor( v.vertex, GRAY );
        dist[v.vertex] = dist[u] 1;
        pred[v.vertex] = u;
        q.enqueue( v.vertex );
      }
      v = v.next;
    }
    q.dequeue();
    setColor( u, BLACK );
  }
```

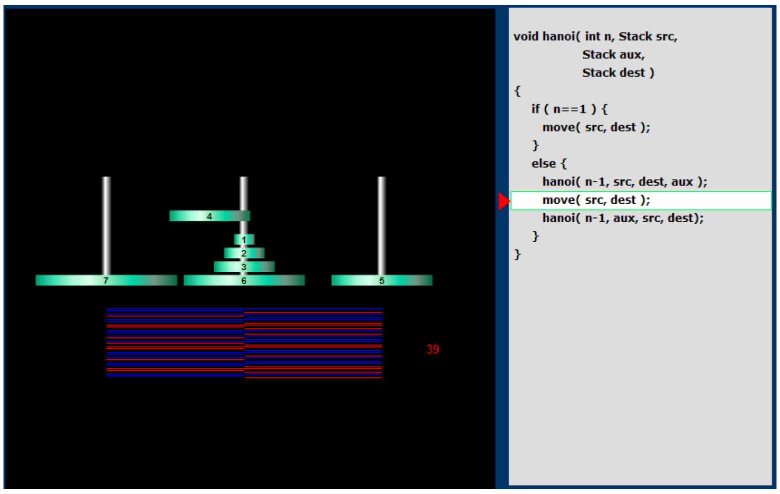
Figure 6.28 Second to Last step of the Breadth First Search algorithm.

The last screenshot (Figure 6.28) shows the final steps of the algorithm. The last node in the queue is about to be removed (node 3), making it transform from green into gray in the graph diagram.

6.9.4 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 Figure 6.29 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.



```
void hanoi( int n, Stack src,
            Stack aux,
            Stack dest )
{
   if ( n==1 ) {
     move( src, dest );
   }
   else {
     hanoi( n-1, src, dest, aux );
     move( src, dest );
     hanoi( n-1, aux, src, dest);
   }
}
```

Figure 6.29 The towers of Hanoi animation.

Now that we have seen Flashdance being used in several algorithmic animations, let us look at its implementation in greater detail.

6.10 Reversibility and Overlays

As explained in Section 2.8 and Chapter 3, reversibility is an important feature for algorithmic animations. Just a couple of the systems surveyed in Chapter 2 offer full reversibility. 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.

Reversibility was included in Chalk Animator by exploiting the existing undo/redo facility of the E-Chalk system (section 4.5). The user can backtrack or go forward in an animation by undoing or redoing a previous operation. This is also the approach I followed in Flashdance: when an animation runs, it is made reversible by generating the inverse instructions and saveing 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 on 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 6.1. All inverse instructions are generated when the program is executed.

Table 6.1 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.

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.

Figure 6.30 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.

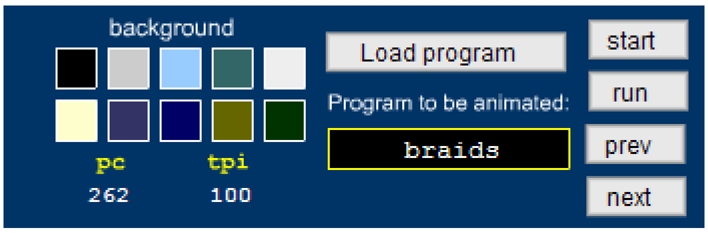


Figure 6.30 Control panel for a Flashdance animation showing the button for reversible execution.

A nice example of a reversible animation and its usefulness is this implementation of Dehornoy’s algorithm for bringing braids into a canonical form [Dehornoy 97]. A braid is a set of *n* lines starting from ordered positions 1 to *n*, which overlap in the way shown in Figure 6.31. A braid can be simplified, in order to make it comparable to other braids. Figure 6.31 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 Figure 6.31 has been executed reversibly and brings the construction process some steps back.

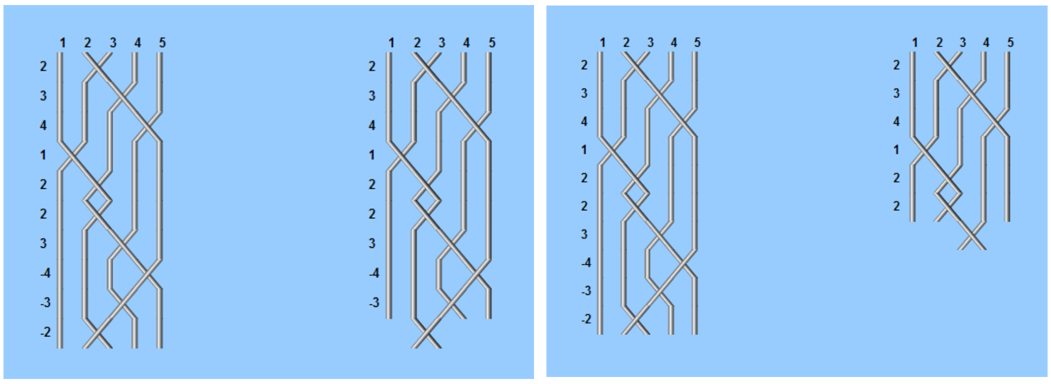


Figure 6.31 Dehornoy’s algorithm running forward (left), and running backwards (right).

Figure 6.32 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.

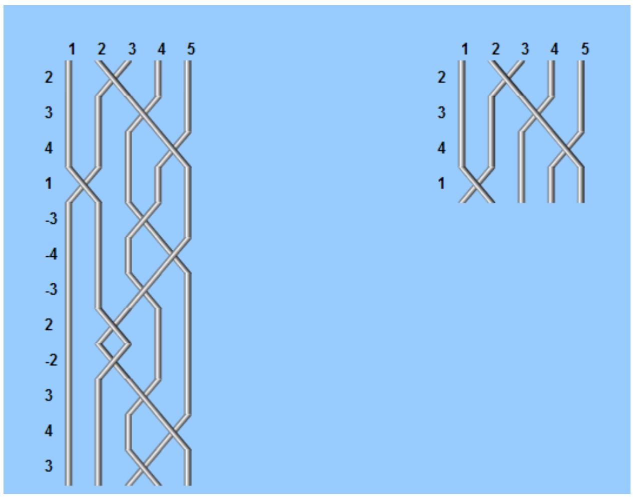


Figure 6.32 Final simplification: the braid to the left is equivalent to the braid to the right.

This animation of Dehornoy's algorithm shown in Figure 6.31 and Figure 6.32 was produced and used by a student in the Junior Mathematics Seminar at Princeton University during the fall term of 2003. It is one more of the Flashdance animations which have been used successfully in the classroom.

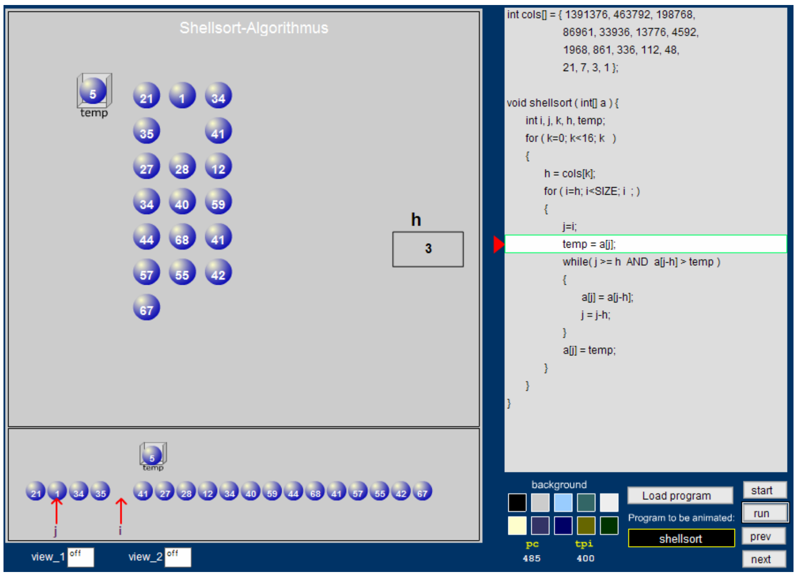
Overlays

Overlays were introduced in Chapter 3. The general idea 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. Overlays were used in the handcrafted animation of bubble sort discussed in Section 3.6.1. The trace of the displacement of a number in the array, when it is sorted, is an important piece of information which can be switched on and off to sometimes better understand the complexity of the algorithm.

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. No other algorithmic animation system, of those surveyed in Chapter 2, offers overlays as an integral part of the user interface.

Figure 6.33 shows Shellsort running, enhanced with two views defined as overlays. The upper view (on the left) shows Shellsort running with a step length of 3.

The array has been divided into three equivalence classes, each one of which is sorted with insertion sort. The lower view shows the complete array and the pointers used to make the sorting comparisons. The first view is more abstract, the second is more related to the implementation.



```
int cols[] = { 1391376, 463792, 198768,
               86961, 33936, 13776, 4592,
               1968, 861, 336, 112, 48,
               21, 7, 3, 1 };

void shellsort ( int[] a ) {
    int i, j, k, h, temp;
    for ( k=0; k<16; k   )
    {
        h = cols[k];
        for ( i=h; i<SIZE; i  ; )
        {
            j=i;
            temp = a[j];
            while( j >= h  AND  a[j-h] > temp )
            {
                a[j] = a[j-h];
                j = j-h;
            }
            a[j] = temp;
        }
    }
}
```

Figure 6.33 Shellsort algorithm with two different views.

Two toggle buttons, shown in Figure 6.34 (a magnification of part of Figure 6.33), allow the user to switch on or off any of the two views. If the button “view_1” is pressed, the view is switched off and is made transparent. The button changes to the “on” label. If it is pressed, the view is switched on again.

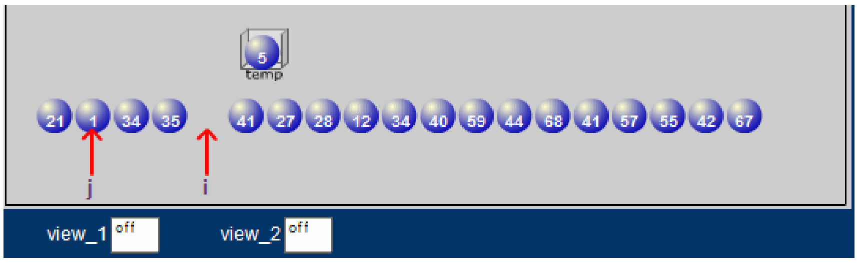
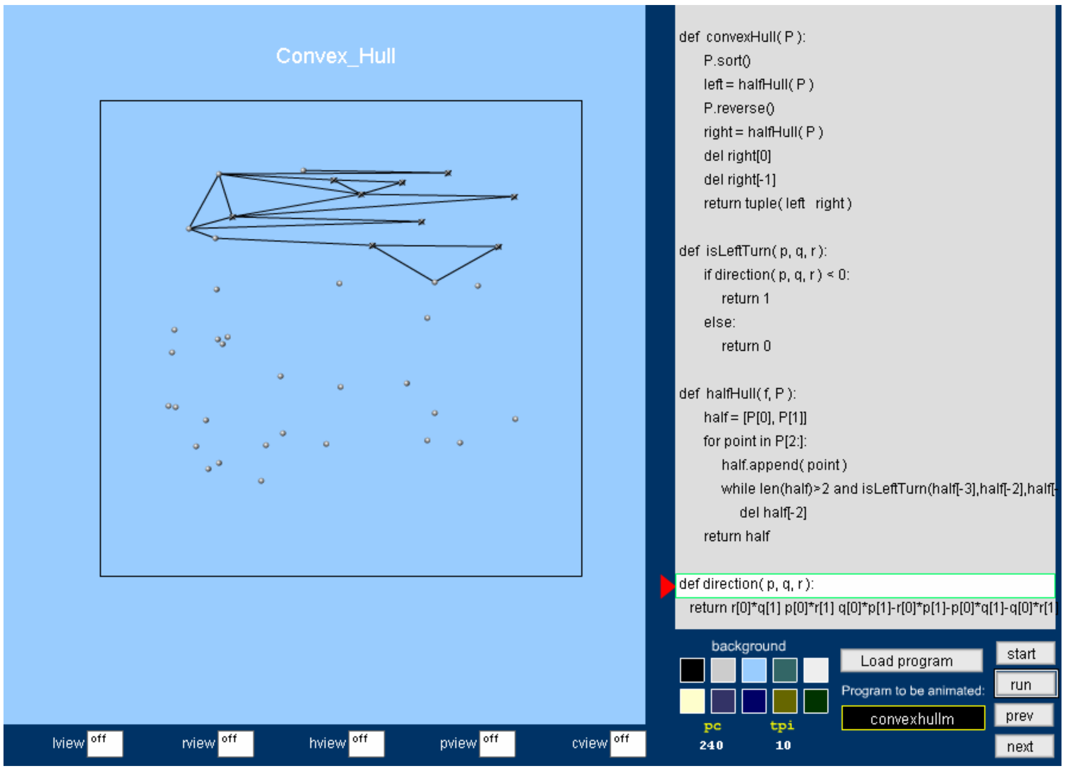


Figure 6.34 Two toggle buttons allow to switch-off view 1 and view 2.

A second example for the use of overlays is the animation of an algorithm for finding the convex hull of a set of points. Figure 6.35 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.



```
def  convexHull( P ):
     P.sort()
     left = halfHull( P )
     P.reverse()
     right = halfHull( P )
     del right[0]
     del right[-1]
     return tuple( left   right )

def  isLeftTurn( p, q, r ):
     if direction( p, q, r ) < 0:
         return 1
     else:
         return 0

def  halfHull( f, P ):
     half = [P[0], P[1]]
     for point in P[2:]:
         half.append( point )
         while len(half)>2 and isLeftTurn(half[-3],half[-2],half[-
             del half[-2]
     return half

def direction( p, q, r ):
  return r[0]*q[1] p[0]*r[1] q[0]*p[1]-r[0]*p[1]-p[0]*q[1]-q[0]*r[1]
```

Figure 6.35 The convex hull algorithm running.

Figure 6.35 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 allows you 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.

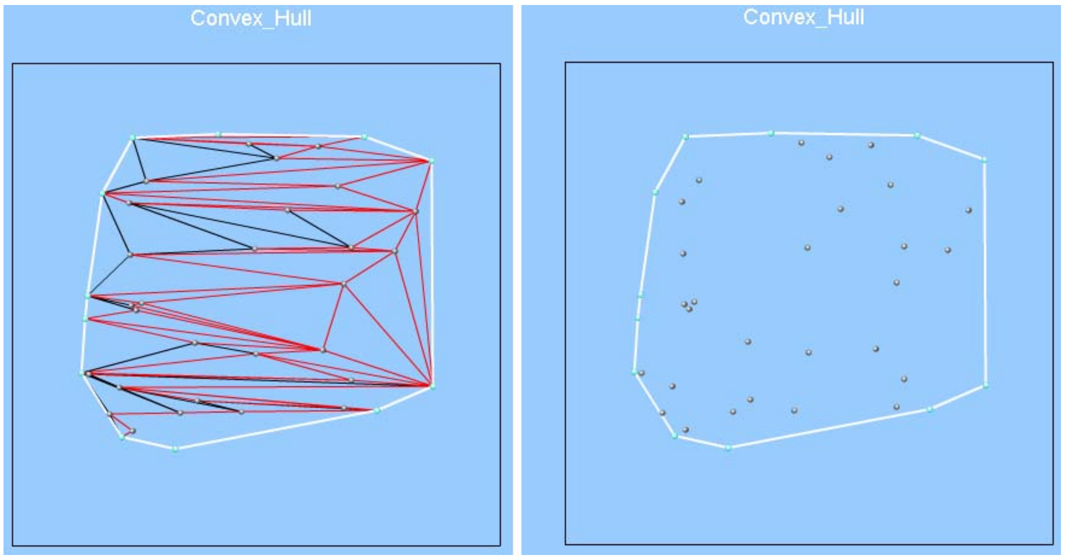


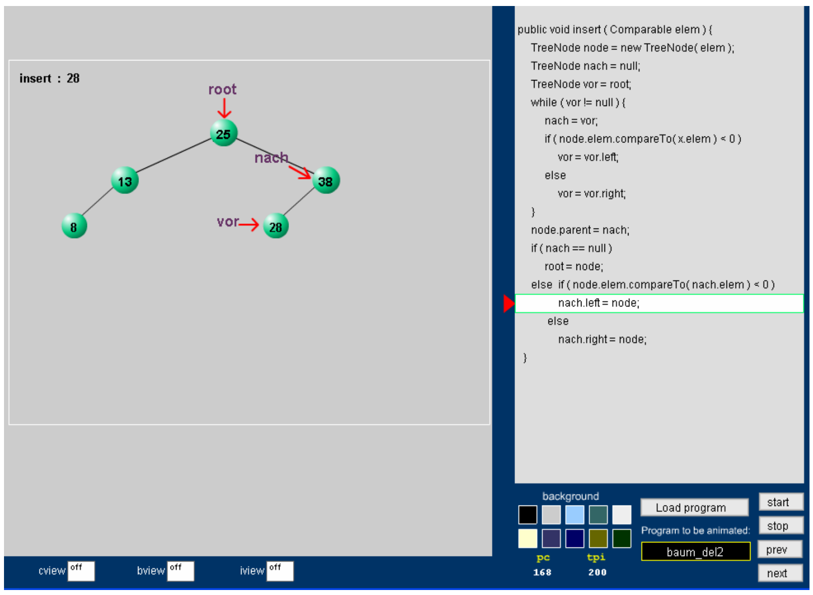
Figure 6.36 The convex hull algorithm running. On the left side we see all overlays. On the right side, the tested segments overlays for the laft and right convex hull have been switched off. This can be done while the algorithm is running.

6.11 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 [Hausner 01, Ben-Ari 02]. Instrumented classes have been used to visualize lists in Java [Dershem 2002]. 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++ [Rasala 99].

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.

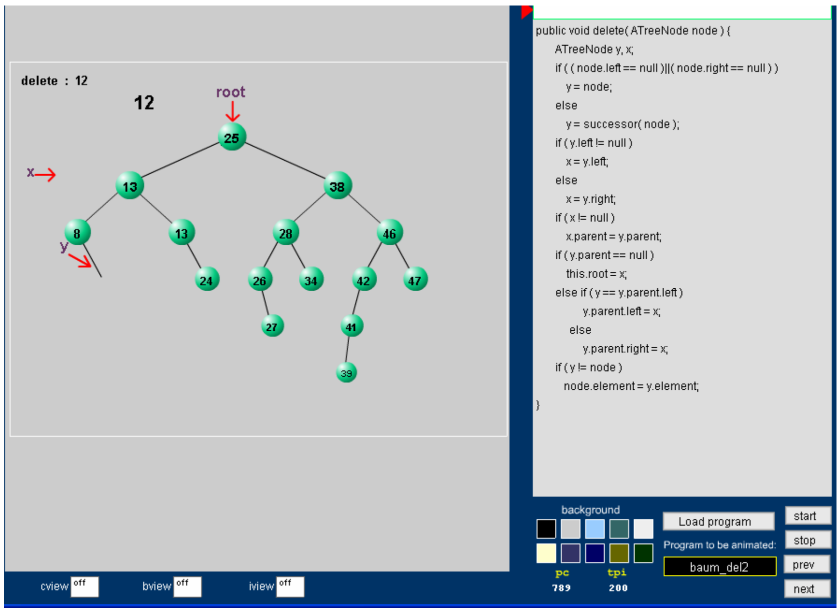
As an illustration of how instrumented Java classes can be used for animating algorithms, I instrumented a small library of Java methods for handling trees. Figure 6.37 shows a screenshot of a tree being built by repetitively inserting nodes into a tree (using the corresponding Java method). When the Java program runs, it produces the Flashdance commands which when played animate the sequential construction of the tree.



```
public void insert ( Comparable elem ) {
    TreeNode node = new TreeNode( elem );
    TreeNode nach = null;
    TreeNode vor = root;
    while ( vor != null ) {
        nach = vor;
        if ( node.elem.compareTo( x.elem ) < 0 )
            vor = vor.left;
        else
            vor = vor.right;
    }
    node.parent = nach;
    if ( nach == null )
        root = node;
    else  if ( node.elem.compareTo( nach.elem ) < 0 )
            nach.left = node;
         else
            nach.right = node;
}
```

Figure 6.37 : Construction of a tree by an instrumented Java class.

In Figure 6.38 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.



```
public void delete( ATreeNode node ) {
    ATreeNode y, x;
    if ( ( node.left == null )||( node.right == null ) )
       y = node;
    else
       y = successor( node );
    if ( y.left != null )
       x = y.left;
    else
       x = y.right;
    if ( x != null )
       x.parent = y.parent;
    if ( y.parent == null )
       this.root = x;
    else if ( y == y.parent.left )
           y.parent.left = x;
        else
           y.parent.right = x;
    if ( y != node )
      node.element = y.element;
}
```

Figure 6.38 Deleting a node from a tree by an instrumented Java class.

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.

6.12 Summary and Discussion

In this chapter I 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. I 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 chapter 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 chapter, 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. One player was written for E-Chalk, the other for Flash. I 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. Microsoft is working in this direction and the new version of Windows, to be introduced in 2006, could offer such animation scripting. Flashdance should be easy to adapt to new scripting languages, so that Flashdance animations for the Web survive more than ten years into the future.

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. I assume that other systems will emerge in the future and will follow this lead. In this chapter I 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. Flashdance thus fulfills the requirements I set out for an algorithmic animation system in Chapter 3.