# 2 Survey of Algorithmic Animation Platforms

## 2.1 Prehistory of algorithmic animation

Some authors date the origins of algorithmic animation back to the first efforts towards making the structure of computer programs easier to grasp with the help of visual aids, such as flowcharts and pretty printing [Baecker 98b]. John von Neumann is mentioned in this regard as one of the pioneers of this visual approach [Goldstein 47]. In reality, however, only when computers became widely available in the 1960s, did computer animation become a possibility. Designers interested in exploring the new technology started collaborating with computer scientists. One paradigmatic case is Ken Knowlton, a programmer and artist at Bell Labs in New Jersey, who developed "Bell Flicker" one of the first systems for producing animated movies. His interest in animation and list processing languages, led him to produce a film showing an animation of the language L6 [Knowlton 66a, 66b, 66c]. The movie explains the instruction set of the language in a visual manner. This was necessary because at that time list processing languages were not as popular as they later became. LISP, for example, had just been defined by John McCarthy in 1959, and there was a need for educational material.

After this pioneering effort, however, many years passed before there was a definite improvement in the state of the art. Notable exceptions mentioned by R. Baecker [98b] are Hopgood [74] and Booth [75]. Baecker had himself previously written a thesis about computer mediated animation, which already showed the possibilities of the medium [Baecker 69a]. He also pioneered the visualization of the execution of running programs [Baecker 73, 75]. Under the direction of Baecker, Yarwood wrote a system that could illustrate program runs on hard copy output, showing changes to monitored variables [Yarwood 74].

A system distributed as a production tool, was the visual debugger GDBX [Baskerville 85] which could animate data structures for the Unix debugger DBX. The FIELD programming environment also offered a similar viewer for data structures [Reiss 98]. GDBX ran on Sun workstations and represented records by nested boxes, and lists by linked boxes. The data in the structures could change the graphical representation, and the user could also manipulate the pointers, changing the data. However, GDBX did not develop further and quietly faded away.

## 2.2 The emergence of computer animation systems

In 1981, Ronald Baecker published his well-known film “Sorting Out Sorting” comparing several animated algorithms. The film, now a video, was immediately hailed as an excellent example of the motivation behind algorithmic animation and the high-quality visualizations that could be possible [Baecker 81]. “Sorting Out Sorting” became very influential and is probably one of the most cited examples of what algorithmic animation is all about [Backer 98a]. Figure 2.1 shows a snapshot of nine sorting algorithms processing the same data concurrently in a “race.” The cloud pattern has a well-defined relationship with the way the algorithm sorts the data. Sedgewick has included many such patterns in his algorithms book [Sedgewick 03].

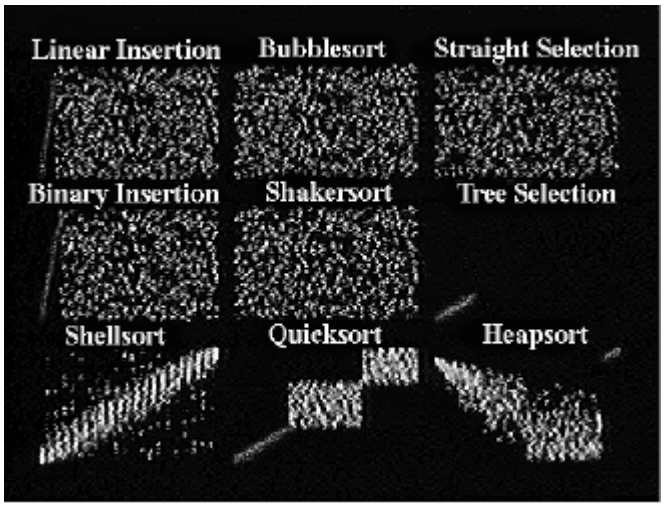


Figure 2.1 A snapshot of the video “Sorting Out Sorting” – the sorting algorithms race [Baecker 81].

Backer wrote later that the main lessons learned from the film were: a) Effective symbolism very much depends on the amount of data. It is easy to find a representation for small data sets, but large data sets are more difficult to represent; b) Illustrations and timing should be carefully selected and a narration should be added; c) Timing is the key to effective algorithm animation. Fancy pictures are not necessarily needed [Baecker 98a].

“Sorting Out Sorting” was a one-of-a-kind manually produced animation. With the availability of graphical workstations and personal computers, the time was now ripe for the emergence of algorithmic animation systems. Thus, between 1981 and 1990, two main general purpose families of algorithmic animation systems emerged: the BALSA family, whose development has been lead mainly by Marc H. Brown, and the TANGO family, whose chief designer has been John

Stasko. The BALSA family comprises the original BALSA system, BALSA II and later on, Zeus and JCAT. The TANGO family comprises the original TANGO system, XTANGO, POLKA and the front end Samba with its Java version JSamba. These two families illustrate very well how algorithmic animation started, how it grew, and which limits would eventually be reached.

## 2.3 The BALSA family

The system which is often mentioned as the predecessor of all modern algorithmic animators is BALSA (*Brown Algorithm Simulator and Animator*) developed by Marc H. Brown and Robert Sedgewick at Brown University in the early 1980s [Brown 84]. BALSA was implemented in Pascal and allowed students to instrument a program in order to visualize its behavior on a screen. The program was introduced in 1983 in one of the first electronic classrooms installed at American universities [Brown 83]. The lecturer would explain an algorithm and the students could start a script from the workstations in order to play examples of the algorithm [Brown 85a, 85b]. Figure 2.2 shows a vintage Apollo workstation running a BALSA algorithm animation in the new installed electronic classroom at Brown University. The photograph to the right shows the same classroom with second generation Apollo workstations. Students could run animations from their places, following the instructions of the lecturer [Bazik 98].

Figure 2.2 Apollo workstation running a BALSA animation (left). Panoramic view of Brown's electronic classroom (right).

### 2.3.1 BALSA and interesting events

BALSA is based on the concept of "interesting events" [Brown 98a]. The algorithm produces a list of monitored events, which are then processed and given to a renderer for visualization. Interesting events are such things as changes in the value of variables, swaps of values, permutations, etc. Multiple views of the data using different windows are possible.

BALSA provides a display (the window manager), and an interpreter for scripts and a shell to process commands. The user interface was modeled after Smalltalk, it provided multiple windows, and most of the interaction was done with the mouse. Scripts were recordings of keystrokes for algorithm replay and could be edited. This allowed the lecturers to prepare a sequence of animations that was started by the students by pressing a single key. The script was therefore, not just a sequence of interesting events in the algorithm, but also commands for the animation system on how to display the data.

Figure 2.3 is an example of an animation of binary search trees and balanced trees produced with BALSA, as well as a table of contents for several animations.

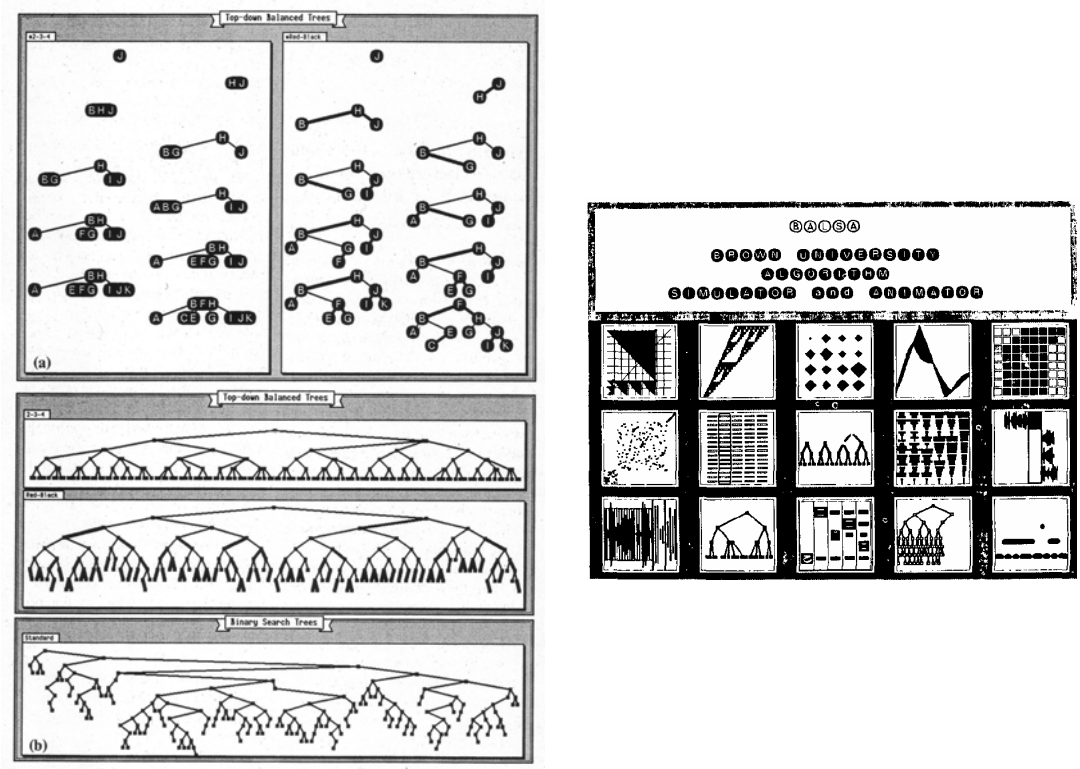


Figure 2.3 Left, a screenshot of a BALSA animation showing several views of balanced and binary trees [Brown 85a]. Right, an iconic table of contents for BALSA animations [Brown 1984].

Preparing an animation in BALSA involved a significant amount of effort. The instructors spent 15 to 25 hours writing the algorithm and its different views, and one to two hours writing a script for a “dynamic book” [Brown 85a]. Automatic animation was conceived as desirable, but not yet feasible at the time.

The main problems faced by the BALSA authors were finding an appropriate conceptualization for the data structures and algorithm (that is, a good model), designing a good sequence of examples for a script, and scaling the data in the views presented. Trees, for example, tend to grow when used with certain algo-

rithms and they have to be redrawn at every step. This irritates the viewer, who can get lost.

### 2.3.2 BALSA II

BALSA II was the successor of BALSA, which was rechristened to BALSA I. It was written by Brown as part of his doctoral dissertation [Brown 88a, 88b, 88c]. BALSA II is based, as its predecessor, on the concept of interesting events and multiple animation views.

In BALSA II, the animation model consists of an algorithm, an input data generator, and views. Views can consume events from multiple generators. The user interface allows the user to provide data and manipulate the views and parameters of components. Keystrokes of the user can be stored as "scripts" of operations (Pascal programs that can be edited) which can then be used to rerun an animation for dynamic books.

BALSA II gave the user some kind of limited interactivity, for example, allowing her to change a variable or the position of an object in the view window during a simulation. BALSA II could also show several algorithms operating on the same data. For testing algorithms, a data generator allowed the user to change the data with a few clicks and run the algorithm again.

The first step for a BALSA II animation is to instrument the program, by replacing read and write calls with calls to a module InputEvent and OutputEvent. In a sorting algorithm, for example, the length of the array to be sorted can be read with a call to

```
InputEvent.HowManyKeys(N);
```

The individual keys can be read with a call to

```
InputEvent.ReadKey(a[i]);
```

The input events are then provided by the user or by a data generator. The user interacts with the animation system to provide the data.

In bubble sort the only interesting event is the swapping of two elements at the positions $j$ and $j$+1. This can be expressed by calling

```
OutputEvent.Swap(j,j+1)
```

The routines in the modules InputEvent and OutputEvent have to be provided by the programmer, who can develop a general purpose library for many algorithms. The Event libraries take care of providing the desired functionality, but also of alerting the animation system to the event which has taken place.

Figure 2.4 shows schematically the connection between the various possible views, the instrumented algorithm, and the input generator in BALSA II. The algorithm generates output events, which are handled by the module OutputEvents, which in turn sends update events to any of the active views. The views perform

the corresponding animation actions. The view with the keyboard and mouse focus (shaded) can pass messages to a message router that transports them further to the input generator. The input data from the user is given to the algorithm, or the if the user requests automatically generated data, the input data generator produces and sends data to test the algorithm (for example, an array of random numbers to test a sorting algorithm.

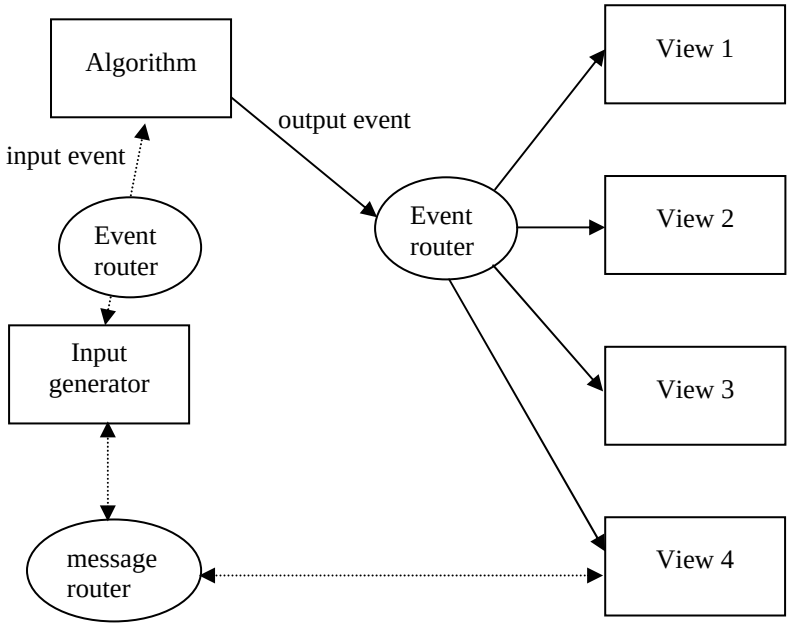


Figure 2.4 The BALSA II animation architectur

The main drawback of BALSA and BALSA II was the limited scope of the language supported, Pascal. The user had to incorporate data structures and code in their programs in a rather complicated way. There is no editor for the graphical objects – these have to be defined by the user using graphic primitives.

The BALSA family came to a close with the development of Zeus, a new algorithmic animation system, developed by Brown at the Digital Equipment Research Center [Brown 91].

### 2.3.3 Zeus and Collaborative Books

Zeus was developed to take into account the emerging object oriented programming paradigm. Unfortunately, the language chosen by Brown was Modula 3, which did not have the influence of Modula II, and not even remotely that of Pas-

cal. The Zeus system illustrates many interesting ideas but remains largely a research prototype with few users in education.

In Zeus, a program writes a list of interesting output events to a file, as in BALSA II. Events are high-level descriptions of the transformations of the data. Views are displays that are informed of the output events taking place. This leads to updates managed by a renderer. Interestingly, a view can also be a sound output, which in some way gives the user a feeling for the data and its transformations [Brown 92].

Zeus is a very flexible system and can also produce 3D effects for illustrating algorithms. The problem is the high level of involvement required from the user for producing a single animation.

The user has to define first the events for an output event file. This output event file is then processed by a pre-processor called Zume, which generates several Modula 3 files. One file, for example, is for producing the output event view, a window listing all events received by the views (a kind of console for checking the animation at run time). Another file is for the views and another for embedding the algorithm that has to be animated. The programmer continues by using these files as the general layout of the program. If a programmer wanted to animate an existing program, the class definitions would have to be inserted by hand, which presupposes an intimate acquaintance with the simulation system.

The algorithm is then embedded in the Modula 3 file wrap provided by Zeus. The algorithm makes the calls that produce the list of output events at the relevant positions in the code. Some interfaces have to be added to the code, which is then finished. A graphical control panel for starting and stepping the animation will be called at runtime. The user can also let the panel provide some of the variables for the animation, but the corresponding code has to be included in the algorithm program.

Now the user has to program his views, making use of the graphical libraries. There are different options, 2D and 3D [Brown 93], and the user has to be sure of which graphical package is better for the application. The graphics runs on top of X-Windows and was tailored initially for DEC machines. The code for the views is written in an untyped interpreted language called Obliq [Najork 94]. Using this language gives the programmer great liberty for describing the appearance of the animation, but also much more work, since Obliq has to be known and a new file with Obliq has to be written. The complete simulation consists at this point of so many files that a Makefile utility is needed. The utility handles the complexity behind the scenes, but it is probably safe to say that the programmer needs to be fully aware of the different steps, especially when a simulation fails for some reason. To reduce the complexity of writing a Zeus simulation, two packages were added later on. A packet for the display of graphical objects called GraphVBT [DeTreville 93] and an interpreted language, called GEF, which can handle Zeus events and generate views, which can coexist with Modula 3 views [Glassman 93].

One interesting aspect of Zeus is that code and data views can be connected. A code view is just a textual pseudocode representation of the algorithm running, whose rows can be highlighted at the appropriate time. This gives the viewer a good correspondence between the algorithm being executed and the animation itself.

Figure 2.5 shows a screenshot of a Zeus animation showing the proof of Pythagoras Theorem. Snapshots of the diagrams appear, below which a textual description is also simultaneously to be seen. Since an audio window can be added, an algorithm can be animated and narrated at the same time.

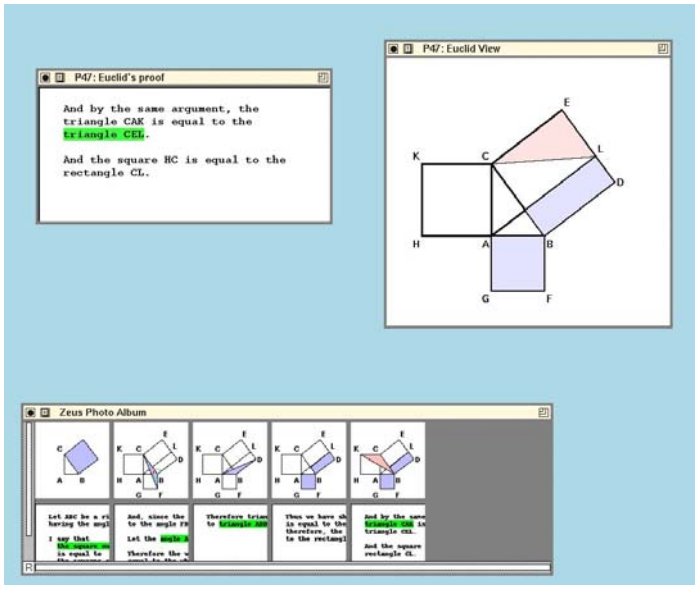


Figure 2.5 A narrated proof of Pythagoras Theorem

Collaborative books are a further development of the work done at Brown University by Brown [Brown 96, 97]. The idea of collaborative books is that text is mixed with animations, and that the reader is not confronted with a static medium. Collaborative books written in Java (with the JCAT system) include Applets and narrations that can be activated by the user. Algorithms are annotated with calls to subclasses of a view [Najork 01]. Much code is needed, but the net effect should be a more complete learning experience for the students. In JCAT the textbooks are Web based: an instructor has control over a Web page, and students connect as clients. This eliminates all need for special proprietary synchronization systems, and opens the way for teleteaching, since remote viewers can get the same information as local viewers.

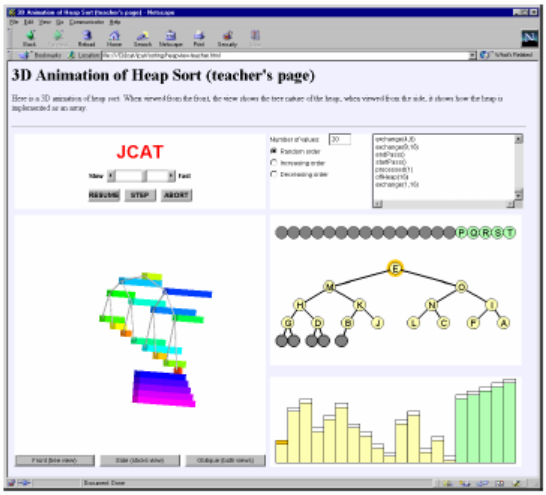


Figure 2.6 A view of a collaborative book about Heapsort

## 2.4 The TANGO family.

TANGO stands for *Transition-based Animation GeneratiOn* and was developed by John Stasko in the late 1980s and early 1990s [Stasko 89, 90b]. TANGO is similar to the BALSA family in some ways, but also has some notable differences. The name of the system emphasizes transitions in algorithmic animation. Elementary steps of an algorithm are shown in TANGO in a way that tries to maximize the usefulness of the visual channel. Variables are represented by physical objects, whose exchange is animated by smooth movements on the screen [Stasko 98a]. Interesting transitions can also be highlighted using color or other visual effects. TANGO is sometimes called the first system to implement the “path transition” paradigm [Stasko 90a].

The X-Windows version of TANGO, XTANGO, was unveiled just a few years later [Stasko 92a]. There are some minor differences between TANGO and XTANGO, the main improvement though was to use a standard graphical system to produce the animation. This made it possible to port the animation tool to more systems. The main difference between the two is that in TANGO the animation system runs as separate process from the algorithm itself. In XTANGO, the animator runs in the same binary as the main program and has to be compiled together [Hayes 90].

### 2.4.1 The XTANGO system

XTANGO was designed as a general purpose animation tool, that frees the programmer from having to write low-level graphics code. All calls to the system are kept at a high level of abstraction so that the user can concentrate on the actual

algorithmic code. There are four types of data objects in XTANGO: locations, images, paths, and transitions. A location is a point on the XWindow screen where some image will be eventually posted. Images are rectangles, circles, etc. Paths are trajectories for the movement of objects on the screen, and transitions are changes of appearance that visually highlight a particular object [Stasko 92d].

XTANGO provides nine graphical objects: circles, ellipses, rectangles, polygons, polylines, splines, bitmaps, and text. Closed shapes can be filled in different manners. Lines can be rendered with different widths and styles. When an image is created, let us say a rectangle, an XTANGO system call creates the type of graphical image and its parameter slots (size, for example). The system call returns a handle for the image that can be used to refer to it in all subsequent calls. Composite images can be also defined, that can be later loaded (for example an array of rectangles). The font of text can be specified when the text object is created.

Locations in XTANGO are defined in the real unit square. This provides a way of enlarging or shrinking the animation window, without having to rescale the animation code by hand. The animation system runs concurrently with the main program and receives calls with information about the operations to be performed. The animation controller has to be started and closed. Locations have handles, so that an algorithm can refer to positions on the screen by name, not directly by their numerical values. This makes the instrumentation of the animation easier.

Paths are lists of coordinate pairs, which determine the relative offset of an image in each successive animation frame. An image following a path moves as smoothly as defined by the number of intermediate frames. XTANGO includes some calls for starting a path. Paths can contain parameters, for example, for changing the color of the object moving along it.

Transitions in XTANGO are visual actions performed on animation objects. There are ten types of transitions: resizing, raising, lowering, grab, delete, refresh, delays, and changes in fill, color, and movement. Objects cannot “transmute” into other objects, as in other animation systems.

XTANGO provides the animator with some interactivity, mainly for defining the position and type of objects on the screen, as well as its color. The animation system returns these values to the driving program.

XTANGO makes life easier for developers by providing grids or trees of locations for positioning useful and common data structures. The user can allocate in advance an array of positions for an array, or a tree of positions for a binary tree. This frees the user from having to compute all positions later in her code.

Figure 2.7 shows an example of an XTANGO animation developed by Sami Khuri and Y. Sugono. The buttons on the left side are provided by XTANGO and

are used for scrolling to the left, to the right, upwards or downwards, as well as for zooming in and out. The mode button (below to the right) can be used to define if repaint will be called at each frame or not (which makes the animation run faster, but not as smoothly). The vertical scrollbar thumb (on the right scroll window) can be moved to slow down or speed up the animation.

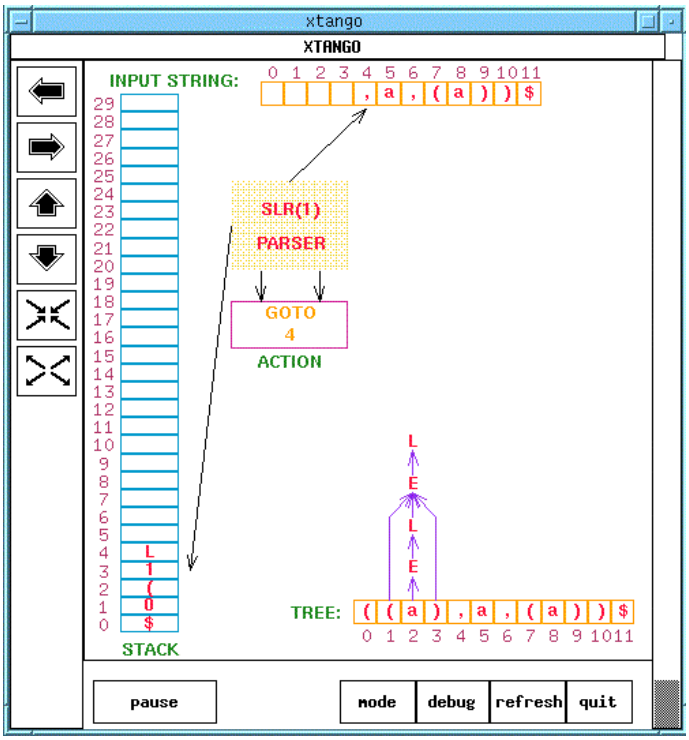


Figure 2.7 XTANGO animation of an SLR parser

Producing an animation with XTANGO is done by writing a complete program in C, or a program in C that reads a trace file of another programs execution. The animator runs as a separate thread waiting for animation commands from the main program.

### 2.4.2 Polka

The next iteration in the Stasko family of languages was Polka. Remember that TANGO started as two communicating processes, the algorithm program and the animator. The two danced together. The main motivation for Polka was making possible the visualization of parallel algorithms and this meant that the architecture of the XTANGO system had to be modified. In addition, Polka is an object oriented animation system, written in C++, which is more general and elegant that the C based XTANGO system.

One of the main differences between XTANGO and Polka is the animation paradigm. While in XTANGO a movement of an object is an atomic operation that is

performed from the beginning to the end, in Polka there is a kind of global clock, the animation frame number, which allows coordinating the transitions and movements in the animation [Stasko 95]. In Polka, the animated object performs an action at a certain frame. The frames are generated sequentially.

A Polka animation is managed by an animator object, which consumes the events generated by the algorithm's code. Polka allows the generation of multiple views of an algorithm. One of the views can show the data being sorted, another view the number of operations performed until now. As in XTANGO, the graphical commands are generated for the X-Windows system. A view keeps a list of the animated objects, in order to manage their animation. A static view, on the contrary, is just a kind of blackboard on which graphical objects can be painted and forgotten.

In Polka there are not four types of data objects (as in XTANGO) but three: locations, animation objects, and actions. Gone are the path objects, since a path is managed now directly through the global frame time. Locations are names for positions on the screen; they can be grouped in logical structures. Animations objects are graphical objects with attributes (parameters). Animations objects are associated with a specific view, have a position on it, and have a visibility type. Subclasses of the animation object class are rectangle, line, circle, ellipse, polyline, spline, polygon, pie, text, bitmap, and set, which are references to other animation objects. When an object is first added to a view, a parameter "time" specifies in which frame the object first appears. Objects can be generated in "structured layouts", as an array of objects, as a matrix, etc. This saves time when programming the animation and makes distributing objects on the screen easier.

Managing a global clock makes handling animations for sequential algorithms a little more intricate as with XTANGO. The scheme is more powerful, because it gives the programmer more control over the animation. Actions can be programmed to happen in the future at any point in the animation sequence.

An action is what was called a transition in XTANGO. An action is any change to an animation object, be it color, position, etc. Multiple actions can be turned on in an object at the same time, or a single action can be started on multiple objects at the same time. An action consists of an action type (moving, resizing, etc.) and a path, a collection of displacement offsets for the change. Since actions start at a specific time, this also coordinates the execution of a path in the subsequent frames.

Some general options allow Polka to stop or resume an animation, to change the speed at which each frame is shown.

As is obvious from the above description of Polka, the system is very much like the object oriented version of XTANGO, with a global frame time and synchronization between multiple views for parallel programs. Figure 2.8 below shows a

screenshot from the Polka Animation Gallery. It shows data being sorted, with two views. The view to the left contains the data while the other shows the data transitions. The buttons (below each window) are very similar to the XTANGO system: they allow scrolling and zooming. The global control panel (above) runs or pauses the animation, allowing the user to adjust the speed with a horizontal scroll bar.

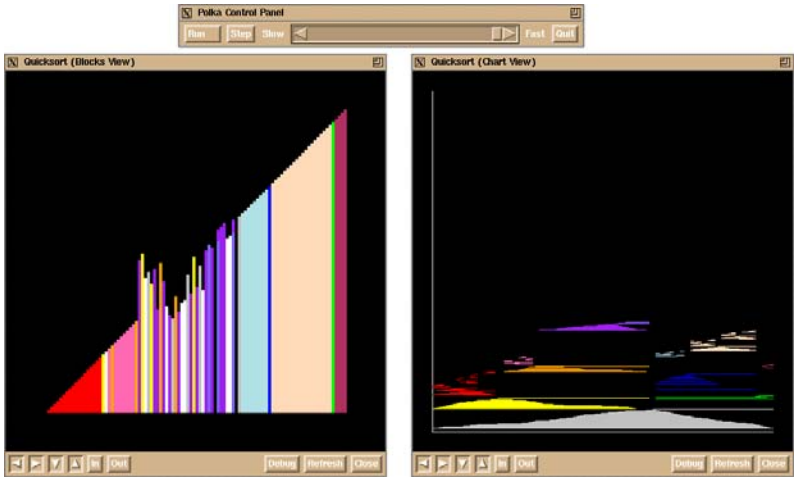


Figure 2.8 Visualization of Quicksort in Polka

Figure 2.9, also from the Polka Animation Gallery, shows multiple views of programs running concurrently and how the program calls happened.

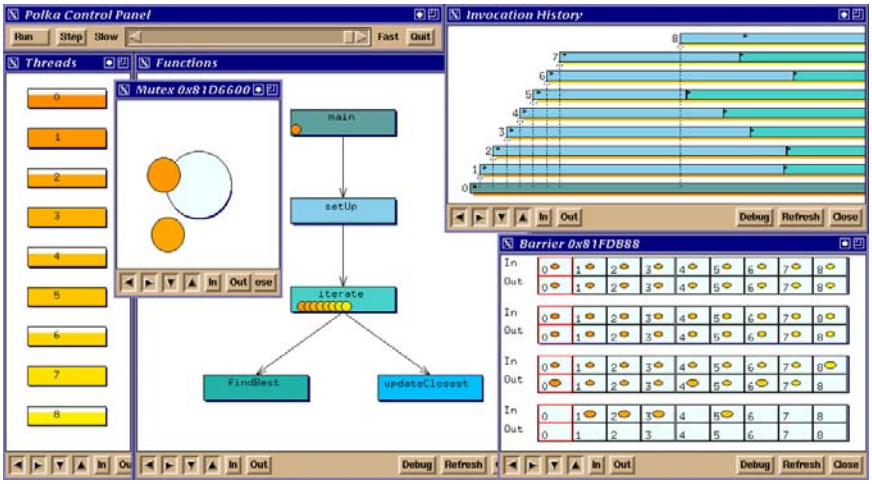


Figure 2.9 Visualization of concurrent threads with Polka

All algorithm development in the XTANGO-Polka family is done with Polka now, and there exits an MS-Windows version called PolkaW. Polka was first released in 1993, PolkaW in 1997.

### 2.4.3 Animation scripting – Samba

There exist a very user friendly front-end to the Polka system called Samba. Samba has been conceived as a set of very simple commands that can be interpreted by an animation interpreter. The user just produces an ASCII file of the appropriate commands and this provides what is needed for visualizing an algorithm.

This approach, separating the animation engine from the code of the algorithm, so that no libraries have to be compiled together with the algorithm's program, is called animation scripting. There exists an animator for the X-Windows system. Samba, in conjunction with a Java Applet that works as an animator, is called JSamba.

Samba consists of one-line commands that can be used to draw objects on the screen and displace them. The front-end accepts commands interactively or a sequence of them in a batch file. Samba commands are passed to the Polka interpreter, which performs the requested animation.

A Samba command has the following general structure

```
<command-name> <parameters>
```

The command name consists of a word; the parameters can be one or many, separated by spaces. The animation canvas is a square window with coordinates (0,0) in the lower left and (1,1) in the upper right.

A circle of radius 0.2, for example, can be drawn in the middle of the window with the command

```
circle 5 0.5 0.5 0.2 green  half
```

In this command 5 is an identifier for the object which has been created, the next three numbers are the coordinates and radius of the circle, which will be green and half-filled. The circle can now be displaced to the position (0.3,0.3) with the command

```
move 5 0.3 0.3
```

The movement is computed as a smooth transition through intermediate points, in order to create the animation illusion. Samba allows the user to open multiple windows and coordinate the movement of objects in each window so that parallel execution can also be visualized.

The main advantage of a language such as Samba is that the animation commands can be printed from the program code. The user only needs to insert some print statements at the appropriate lines and a batch file can be generated that will be interpreted by Samba. It is therefore fairly easy to generate animations without having to deal with the graphical user interface of the program itself.

Some very interesting educational experiments were conducted by John Stasko using Samba, due to its simplicity and the ease with which the animation scripts can be produced [Stasko 97].

## 2.5 Other Algorithmic Animation Systems

There are some other systems that have appeared independently from the two main algorithmic animation families. Some of them contain interesting ideas that we review in the next sections.

### 2.5.1 ANIM – A minimal system

ANIM is a set of tools for the UNIX operating systems that can be used to produce computer animations or snapshots ("stills") of an algorithm running [Bentley 91a, 91b]. The program to be animated is expanded by the inclusion of output statements which generate the commands for the visualization. The instrumented program and the visualization tools can be connected through a pipe, in order to produce almost "live" visualization.

ANIM has a very simple structure. The script language used provides four commands for drawing geometric objects: text, line, box, and circle. The general format of a command is:

*Optional_label*: *command options x y additional parts*

Options for geometric objects are the type of line and if the object is filled or not. Text is placed with the command

*Optional_label*: text *options x y string*

The label gives a name to the generated objects. The options further refine the format or position of the object. Text, for example, can be adjusted in size, justified, or placed above and below objects.

There are also four control commands, view, click, erase, and clear. The last two clear a previously defined object, or all objects defined. The command "view

name" specifies the name of the view (window) which the commands refer to thereafter. The command "click name", defines a kind of breakpoint in the program that can be used to segment later the different portions of an animation. For example, a click can be put at the end of each loop in a nested set of two loops. Later on, the user can select a snapshot that will be generated at the end of the first loop (first click) or at the end of the second loop (second click). This segmentation of the simulation is very useful for producing slides of an algorithm.

The script language is summarized by Bentley and Kernighan [91b] as follows:

```
# comment
optional_label: line options x1 y1 x2 y2
[-] -> <- <->
[solid] fat fatfat dotted dashed
optional_label: text options x y string
[center] ljust rjust above below
small [medium] big bigbig
optional_label: box options xmin ymin xmax ymax
[nofill] fill
optional_label: circle options x y radius
[nofill] fill
view name
click optional_name
erase label
clear
```

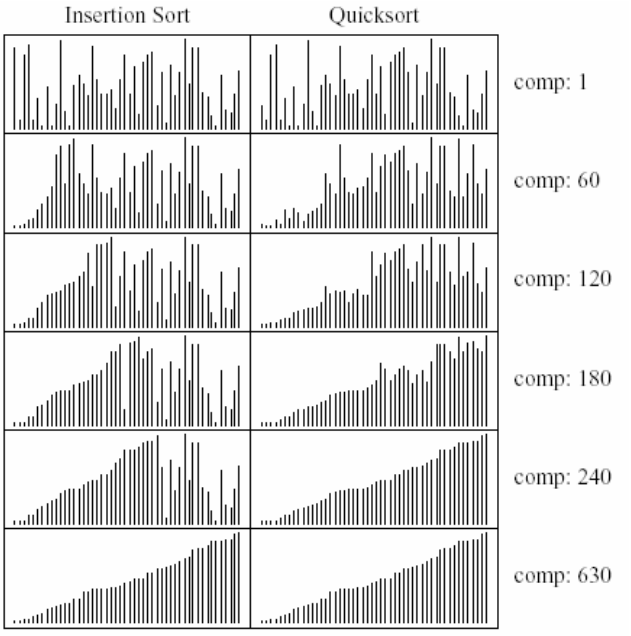


Figure 2.10 ANIM stills showing a comparison of sorting algorithms [Bentley 91a]

As can be seen, there is no command for smooth transitions. The animation effects are produced by painting and erasing objects (redrawing an object, erases it

first). The ANIM movie viewer animates a script. A menu allows the animation to be run forward, backward, faster, and slower. Stills can be generated by specifying the clicks that must be rendered. The UNIX utility TROFF takes care of the final formatting.

Figure 2.10 is an example of stills produced by ANIM comparing Quicksort and Insertion Sort at specified number of iterations.

Although ANIM never became a much used system, it is an excellent example of a minimalist implementation of an animation tool. In this thesis, I have also tried to follow a minimalist approach in the design of my own scripting language.

### 2.5.2 Mocha and Web animation

Mocha is the concept of an architecture for the distribution of computation between a server and a client rather than a specific animation system. Mocha is geared toward animation of algorithms through the Web [Baker 95]. An animation server running an algorithm repository provides the animation commands, which are executed by an Applet at the client machine. The Applet provides the GUI for the user, allows her to select options or input data. The burden of the computation remains with the server. Several animations using this architecture were produced for the Web, but the system never really got beyond the prototype stage.

### 2.5.3 Leonardo

Leonardo offers more than just an animation system. It is a full development system for C programs and visual debugging [Demetrescu 99, 01], and animations can be produced. Since it is a development system, Leonardo provides an editor and compiler. Its more interesting feature is the virtual reversible CPU [Crescenzi 00]. All computation runs on a software CPU. The CPU keeps tracks of computations and can be sent on reverse, undoing past imperative computations. Reversible execution was proposed long time ago as a desirable feature for programming languages [Zelkowitz 73] and Leonardo provides it. Leonardo does not generate executable code; programs are executed within the development environment. A predecessor, but unrelated system, which also used a virtual reversible CPU was the DYNALAB [Birch 95]. ZStep 95 is a more recent stepper for debugging, also based on reversible execution and animation [Lieberman 98].

Special commands for producing the visualization are embedded in C code as comments. The language used is called Alpha. Annotations in Leonardo are called predicates; they are like global functions known to the system.

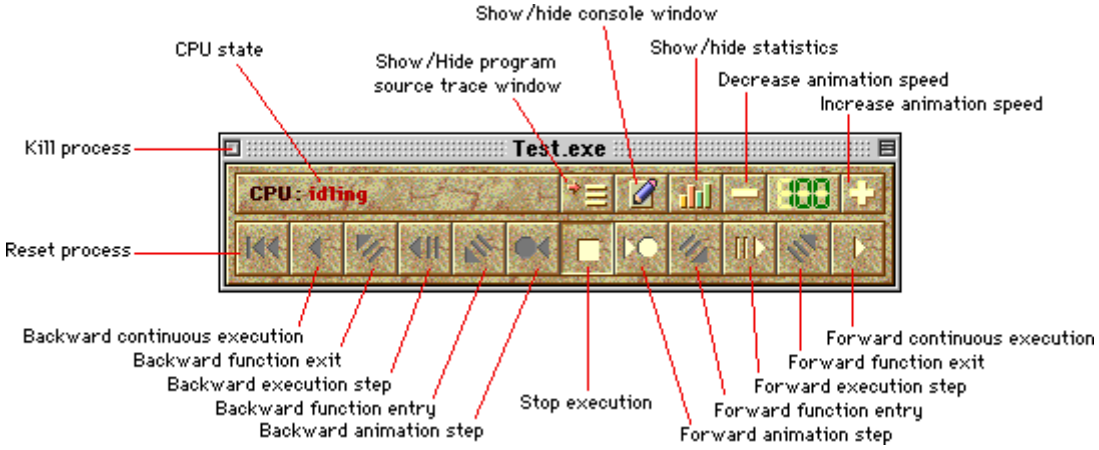


Figure 2.11 The Leonardo control panel

Predicates define graphical objects, their parameters and the correspondence of their parameters to variables in the C code itself. A variable that changes value, changes immediately the appearance of a graphical object which has this variable as one of its parameters. Standard predicates in Leonardo handle animation, the creation and parameterization of graphical objects, and their visual properties. Library predicates are sequences of standard predicates stored in libraries which can be accessed by the user. Libraries handle special data types and some conventional visualization styles.

Leonardo is clearly most useful when learning to write programs in C. There is only an implementation for the Macintosh and a prototype of a multiplatform system. The Leonardo Computing Environment is the new name of the Leonardo project. Future versions of Leonardo will provide predicates to produce smooth animations, which are not yet possible.

The Leonardo predicates are declarative. The presence of graphical objects is declared in the program, making them appear on the screen at the selected position. The linkage of parameters of the visual objects with variables produces an automatic update, which is sometimes undesirable (when swapping two numbers, for example, using a third variable as repository). Some predicates are actually imperative, they can switch the update on and off at specific parts of the code. This is also what makes smooth animations difficult in a declarative system. The level of abstraction of the data driven updates is sometimes too low and can distract from the essential properties of the algorithm.

### 2.5.4 GAWAIN – Geometric Animation

GAWAIN (Geometric Algorithms Web Based Animation) is a system developed by Alejo Hausner at Princeton, and which has a general approach similar to the MOCHA architecture. A Java Applet receives events and animates them on the

screen. GAWAIN is geared towards geometric problems, although is a general purpose simulation system [Hausner 01].

GAWAIN can animate algorithms backwards or forwards, at different speeds, and with discrete or continuous steps. The system is interactive (when an event script is not being processed) since the user can modify input data with the mouse and watch the resulting run of the algorithm. Gawain provides input generators (as in BALSA II) and support for 3D views, unfortunately with a hand crafted library without relationship to Java 3D. Multiple views of an algorithm or algorithms races are supported.

The interface to a GAWAIN animation allows the user to scroll in four directions and zoom in and out, as in XTANGO. The speed of the simulation is measured in events per second and can be adjusted. The number of events per animation step can also be adjusted, allowing the user to speed forward through an algorithm and look at it at several levels of detail.

The main problem of the Gawain system is that interesting animations are hand produced using instrumented Java classes. Although the animator Applet can animate code written in other languages (through an event stream) the bulk of the system is geared towards the seasoned Java programmer. Gawain did not achieve wide popularity.

### 2.5.5 Pavane and declarative animation

The Pavane animation system differs from all others mentioned above mainly because it is a declarative and not an imperative system [Roman 89, 92]. Code is not extended with appropriate event generators – the extension is done automatically by an interpreter which knows which variables or data structures are being monitored. Like many other algorithmic animation systems, Pavane is the name of a type of dance, namely one which originated in Italy during the Renaissance. Pavane was one of the first social dances for couples, which glided on the dance floor. Maybe it was this elegance of the dance which the authors of Pavane tried to recreate with their animation system.

Pavane was not the first declarative animation tool. Other systems, used for program run visualization and debugging, were declarative from the beginning. In PROVIDE, interactive computer graphics are used to illustrate program execution. The user specifies which variables it wants to be shown on the screen, and the system takes care of handling the graphical view [Moher 88].

In Pavane, the author of a visualization declares some logical formulas, which state when a certain object will be shown on the screen, or not. In a visualization of the game of Life, for example, the programmer needs to state only that a sphere should appear on the screen or not, according to the value of a bit in a matrix

(Figure 2.12). The animation engine takes care of updating the view according to the value of each bit in that matrix.

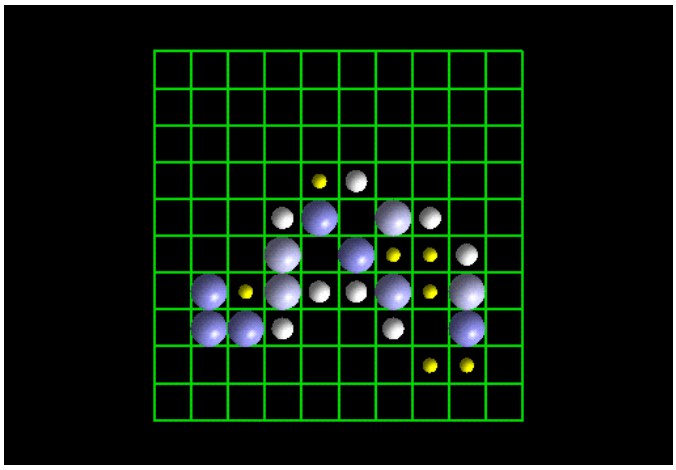

Figure 2.12 : Animation of the Game of Life in Pavane

Pavane was conceived initially for instrumenting the visualization of concurrent programs. A special language, called Swarm, was used to write concurrent programs [Cox 90]. Swarm code was then interpreted in Prolog and Prolog clauses could be added to animate the code.

The main idea in Pavane is to map program states into images. When a certain state is reached, this triggers an update of the animation. The approach is expensive when the system has to take care of identifying states which act as triggers. In Prolog it is easier to test for such program states, since states can be transformed into Prolog clauses that can be tested later.

Pavane has been extended lately to also handle programs written in C [Roman 98]. Figure 2.13 is an image of a Pavane animation for the Towers of Hanoi problem.

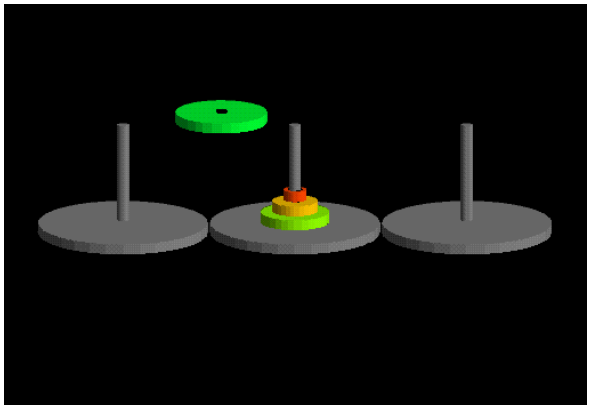

Figure 2.13 Pavane animation of the Towers of Hanoi

The C program for the algorithm has to first include the Pavane interface with:

```
#include “CtoVis.h”          /* C interface to Pavane */
```

Then at specific points in the code the “VisualUpdate()” function is called. This is not really declarative, it is an imperative command inserted in the algorithmic code. The state of the computation, represented by the relevant data, is visible to the Pavane animator. This is guaranteed by initialization code inserted before the main program start. The Pavane animator checks at each call of VisualUpdate() if something has changed in the program state and generates the necessary changes to the image.

Pavane’s approach to declarative animation is therefore sound, but expensive, if a completely declarative animation is wanted. A debugger or a development environment can do here better, because program state changes can be checked directly using a virtual machine or triggers which are activated by the variables being watched. Pavane must simulate this using the VisualUpdate function, and therefore Pavane is a kind of hybrid system: rules specify which aspects of the computation state have to be checked and which changes they generate to the image, but the monitoring of the code portions in which the checks are relevant has to be hand-coded by the programmer.

Figure 2.14 shows that Pavane can also handle 3D graphics. The illustration of Quicksort is interesting, because it demonstrates another way to create anticipation in the user and directing her attention to a compare operation. Without changing the order of the two numbers, it is evident which two are being compared. It is easy to see how the mapping from program state to image is handled here: each pointer represents part of the computation state. The elements of the array at the pointer value are slightly displaced, that is, one of their coordinates has been added a constant.

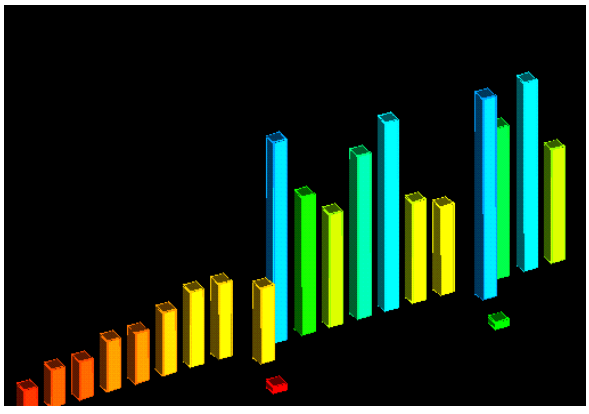

Figure 2.14 Colored animation of Quicksort with Pavane

### 2.5.6 Other approaches

There are many other algorithmic animation systems that have been developed in the past. Some of them are mentioned in what follows.

*JAWAA*

A system for the animation of algorithms using a player Applet. A script language defines the animation and the player executes it. Basic graphical primitives are provided, as well as some predefined data structures and their operations [Pierson 98]. The basic functionality is lower than Gawain, but JAWAA 2.0 includes a graphic editor for drawing some pictures and providing coordinates of animation objects [Akingbade 03].

*GANIMAL and GANIMAM*

This is a system for interactive animations geared towards virtual machines and compiler design [Diehl 97].

*JEliot*

JEliot is a Java system for algorithmic animation based on self-animated data types. Animations are produced automatically and show all state changes in the observed data structures [Haajanen 97]. JEliot animations are funny, but poor.

JEliot was derived from Eliot (from Finnish, *El*ävät ol*iot*, meaning living beings). Eliot, in turn, was derived from HALSA and HALSA+ (Helsinki Animation Library for String Algorithms). Eliot was used to animate C programs. A precompiler finds the data structures and allows positioning the animation structures. Executable code is produced and the animation is visible. JEliot applies the same idea to Java programs. The user can interact with the mouse and define the position and size of graphical elements. Other developers have embraced JEliot and there is now JEliot 2000 and JEliot 3 [Ben-Ari 02].

*ANIMAL*

The ANIMAL (*A New Interactive Model for Animations in Lectures*) system was developed by Guido Rößling, who also maintains an archive of animations produced with this tool. At the core of the system is. AnimalScript, a set of commands designed for general animations tasks [Rößling 01]. AnimalScript lines are produced by a program and can then be parsed and executed by an interpreter. AnimalScript provides graphical primitives and operations to move, rotate, show, hide, and change the color of objects. The script language can be extended by a user, to include new functionality. The player can be written in any language, but the existing implementation was developed in Java. A Java API for the animator

has also been written, which allows a Java program to control an animation directly.

Many other animation systems have been developed which are more limited in scope. *Algorithma* can couple code walkthroughs with animations [Concepcion 99]. The *Algorithm Explorer* is a front-end for Tango [McWirther 96], GAIGS (Generalized Algorithm Illustration through Graphical Software) is an object oriented animation system [Naps 94] and its Internet variation, Web GAIGS, uses a client-server architecture [Naps 97], as does its descendant JHAVE [Naps 00].

## 2.6 Sketch animation

Few authors have explored the animation of handwritten sketches produced spontaneously during a lecture. Strothotte and collaborators [98] have experimented with non-photorealistic animation. Most of the experiments which have been done in this direction are related to the animation of cartoons. The system, for example, developed by Baecker, allowed a graphics animator to sketch forms and define movement paths. Genesys would then interpolate the frames for the desired movement, not quite in the way this is done in Macromedia's Flash, but in a related manner [Baecker 69b, 70, 74].

One possible partial exception is the system SALSA (Spatial Algorithmic Language for StoryboArding) [Hundhausen 00, 01]. In SALSA, a sketch editor allows the animator to draw sketches of objects used for animations, which are then positioned by the user on the screen using a mouse and menus. The connection to the animation is done through commands in a command window (where the main algorithm also runs) which specify the changes in the position or appearance of the animation objects. The environment in which the animation runs is called ALVIS (Algorithm Visualization Storyboarder). It is rather primitive, but some educational experiments were conducted by the authors.

The main advantage of animating sketches could be the involvement of the students. There is much more to be refined and even the correctness of the correspondence between the algorithm and the animation is not guaranteed, but the authors of SALSA see this as a positive, since the meaning and value of the animation has to be negotiated between the students and the teacher. Better teaching of algorithms can be realized in an "algorithms studio" [Hundhausen 02a].

The accent of the SALSA system lays in the direct involvement of the students in the production of animations, which should be easy to produce, convey the appearance of the imperfect and unfinished, and lead therefore to stronger student participation.

## 2.7 Animation by handcrafting

Two systems which have been also used for algorithmic animation should also be mentioned here. In the form they have been used, the kind of animation possible with them can be called “animation by handcrafting”. Microsoft’s PowerPoint and Macromedia’s Flash provide ways of animating algorithms. In the case of PowerPoint they are very limited, in the case of Flash there is a real animation engine behind the whole. Hypercard for the MacIntosh has also been used for algorithmic animation [Velez-Sosa 93]. At MIT, the algorithms from [Cormen 90] were implemented with Hypercard following a systematic methodology [Gloor 92, 93]. However, Hypercard has disappeared and we will not comment further on it.

In PowerPoint the main kind of animation is making elements appear or disappear from a diagram. In the newest versions, it is also possible to give elements a movement path from slide to slide. This allows ingenious educators to illustrate algorithms with small changes in pictures. It should be also mentioned, that Microsoft Office programs accept Visual Basic as macro language. It is possible, for example, to write algorithms in Basic and visualize them using Excel [Rautama 97].

In Macromedia Flash it is easy to draw scenes, move objects on the screen and ask the system for a smooth interpolation. A Flash animation is produced by segmenting a movie in scenes, and each scene contains several layers in which objects are defined and moved. We will review the Flash animation engine in more detail in Chapter 5.

Microsoft will probably avoid further development of the animation features of PowerPoint, mainly because it makes the system more complex, but also because Flash animations can be inserted in PowerPoint presentations. That is an interesting feature for my own Flash algorithmic animations described in Chapters 3 and 5 – they can be viewed inside the main presentation tool used today.

## 2.8 Taxonomy and comparison of algorithmic animation systems

A very comprehensive taxonomy for the classification of algorithmic animation systems has been proposed by Price et al. [Price 93, 98]. Their taxonomy, however, is too large, tries to cover all possible cases and is not hierarchical. In a sense it is not even a real taxonomy, it is just a large catalog of features that an algorithmic animation system can have or not have. Myer’s early taxonomy of program visualization systems, by contrast, was too simple: it divided systems only in code and/or data visualization tools, for static and/or dynamic views [Myers 86].

For our purposes we are interested in a more useful classification. In a taxonomy, ideally, all systems described could be assigned to a leaf of a taxonomic tree. The

whole purpose of taxonomy is to bring order to the natural world, with its wealth of occurring forms.

With that in mind, I propose to look at algorithmic animation systems according to the main features described next.

*Imperative versus Declarative Instrumentation*

The first major subdivision occurs at the level of instrumentation. Is it necessary to specify in the code explicitly the places where interesting events occur? Or can this be done in a declarative manner? Almost all of the systems considered above handle interesting events in an imperative manner, only Pavane uses the declarative approach (Leonardo a virtual machine). The declarative approach seems, of course, easier to integrate in the code, but it is difficult to implement, because it can only be done by an automatic preprocessor or by a virtual machine which outputs the events when they happen.

Consider an algorithm in which only changes to a variable at the end of a loop are considered interesting. This is easy to include in an imperative instrumentation, but is difficult to specify with a declarative system. The declarative approach is easier to use only when all changes to a variable are interesting, and we only need to declare this variable. The declarative method is difficult to use when interesting changes depend on context.

It is therefore safe to say that the imperative approach handles context, the declarative approach can hardly consider context, unless the declarations themselves become so difficult to write and read as to make the system unusable.

*Separation of Instrumentation from Rendering*

When the animation is driven directly by system calls inserted into the algorithm being animated, we have a close coupling between instrumentation and rendering. When the instrumentation produces a stream of events, rendered by another thread or process, we have a clean separation between the algorithm and its animation view. In some systems, special data structures (for example a stack) provide their own animation primitives. The programmer writes her programs using these data structures from a library which can also be ordered to self-animate. This is the approach used by JEliot and GAWAIN. It can seem simpler and straightforward, but it means that the programmer will have to write programs based on this library. Preexisting code cannot be animated, unless the data structures are adapted to use the instrumented libraries.

Both the BALSA and the TANGO family work using the separation between instrumentation and rendering, because it provides a more flexible system. The instrumentation can be done in any language that provides the stream of events for

the renderer. With instrumented classes, only the language of the instrumentation library can be used.

*Single view or multiple connected views*

The first animation systems could manage a single view of the data. All current systems try to provide multiple views, which show the same algorithm from different perspectives. One view can show the data, the other the number of operations. Multiple views are especially interesting when they are used to compare different algorithms.

*Debbuging Features*

There are several visual debuggers available. They show textually changes in the variables and step through the code. The most notable example of an early visual debugger was MacPascal for the Macintosh, which probably inspired some of the algorithmic animation work started in the 1980s.

The only system described above that includes a kind of debugging capability is Leonardo, since it is a development environment in itself. Most other systems do not offer this capability, since rendering is separated from event generation.

*Development environment or strictly animation system*

The Leonardo system pioneered the idea of simulating a virtual machine that executes a program and the animation code. Having a virtual machine gives the possibility of reversing algorithms, even if the code was not instrumented this way. From all the systems considered above, only Leonardo is a development environment for C programming.

*Reversible versus non reversible animation*

BALSA II and the TANGO family do not offer any kind of reversible animation feature. Leonardo has integrated this as a core aspect of its virtual machine environment. Other systems that offer reversibility are Gawain and ANIM.

Reversibility seems to be a very important feature for algorithmic animation because it allows a student to pace through an animation and replay aspects of the animation that are difficult to understand. They give microcontrol over the animation sequence and more freedom to the viewer. Reversibility is very important in systems based on backtracking, such as Prolog [Eisenstadt 85, 88].

*Interactive versus non-interactive Animation*

An animation seems to be more educationally effective if the viewer is able to test an algorithm with her own data. This is difficult to do at the renderer level, which only displays a stream of events from the instrumented algorithm. The best world is one in which the instrumented program runs side by side with the animation, and can be stopped to change parameters. Most systems mentioned above allow the user to run the instrumented algorithm concurrently with the visualization, even if there is a separation between renderer and algorithm.

*Parallel programming*

From the system discussed above only Polka handles parallel processes. Zeus can simulate parallel processes running, but sequential code has to be specially written. Polka synchronizes events through its global clock facility.

*Standard versus non-standard animation engine*

All the systems described above use a non-standard animation engine. Animations have been especially written using a standard graphical package, such as X-Windows or MS-Windows, but the animation engine has been written and rewritten by every group by itself.

*Export function for the Web*

Most systems discussed above do not offer an export function for the Web. Only the Java based systems can let an Applet run, that interprets a script or events from the instrumented algorithm. This is the case with JSamba, JEliot, Gawain and MOCHA. However, most of the Java animations written for all these systems are not very attractive from the graphical point of view and some of them do not run in browsers anymore. Java code should be compatible across platforms and browsers, but in many cases it is not, as developers soon realize. An export function for the Web is, however, highly desirable since it can make many animations available for use outside of the one's classroom and automatically reach a global educational audience.

## 2.9 Conclusions and requirements for an algorithm animator

From our grand tour of previous algorithmic animation systems, we can extract some conclusions, which will guide the rest of this thesis. What we should demand from an animation system is the following.

- Standard graphics

Many animation systems have disappeared or become irrelevant because they do not provide support for a standard graphics engine. This is the case of the BALSA-I and BALSA-II systems. Both were geared toward special machines (Apollo and Macintosh) and were implemented with software libraries that ran only on such machines. An algorithmic animation system should provide platform independence, by using as much as possible, standard graphics or by providing alternative rendering possibilities that can be adapted fast to new machines. The transformation of TANGO in XTANGO, for example, was a step in the right direction because it standardized the graphical interface.

- Standard animation engine

The time is ripe for abandoning all efforts to provide a special animation engine for algorithmic animations. Practically all existing animation systems implement smooth animations by hard programming. Many authors have switched to Java, because of its convenience in reaching platform independence, but there is no standard animation engine for Java. The programmer has the burden of writing her own library.

Although Flash is a proprietary animation engine, its large diffusion makes it possible to adopt if for the production of algorithmic animations. In Chapter 5, I introduce the Flashdance system for the production of stand-alone algorithmic animations and for the Web.

- Web capable

Today algorithmic animations should be embeddable in electronic books. Documents in the Web should provide textual explanations but also give the reader the possibility of watching and interacting with animations. The animations should not run in special environments, that have to be installed and started, but should run in the Web browser. Currently, only two platforms provide such functionality, Java and Flash animations.

- Executable without extra software

Viewers should be able to let an animation run without needing extra software. The Leonardo system, for example, is an excellent tool for developers of C programs, but cannot be used to export teaching material. An animation should provide its own viewer, as for example PowerPoint presentations can do.

- Export function

An animation should be exportable, that is, it should be able to be shown in different ways. The animation snapshots, for example, should be printable and exportable as JPEG or GIF files. Ideally, a Flash animation, for example, should be exportable as a Java Applet.

- Export function for video

It should be possible to generate video formats from the animations, in order to support video streaming systems.

- Reversibility through a virtual machine

Some of the reviewed systems provide reversible execution. Reversible execution allows the student to stop an animation and replay the last scenes. This is equivalent to being able to go back and forth in a book or video.

- Separation of instrumentation from rendering

The most effective and versatile systems are those that separate the coding and instrumentation of an algorithm from its rendering. ANIM, for example, is a minimal system, but very flexible due to this separation. It can produce stills or animations.

Separating coding from instrumentation also allows the use of different kinds of renderers. In this thesis we show that the same scripted animation can be used to produce a high quality Flash animation, or an animation for an electronic blackboard. The same script can also be used to produce a transcript of the animation.

The separation of instrumentation from rendering, through an event file, makes the animation system language independent. Pavane, for example, is language specific, as well as Leonardo or JEliot. We would like to be able to animate any kind of computing language. Therefore we will use a script language. Instrumented classes can be derived from the script language (see Chapter 6).

- Graphical objects editor – library of graphical objects

Most Java animation systems (JEliot or JSamba) produce low quality graphical animations. This has to do with the fact that the graphical output of standard Java is not very appealing, but also with the lack of a graphical editor for animation objects. The instrumenters of the algorithm have to draw everything (as in ANIM or BALSA-II). It would be more effective if the animator could just select an object from a library of graphical objects that the user can draw directly in high quality. This functionality is provided by the graphical engine of Flash.

- Interactivity

Interactivity is desirable. The user should be able to experiment with different kinds of data and his own input.

- Audio channel

An oral explanation can reinforce the effect of the animation. In this thesis we explore the production of sketch animations coupled to an audio channel. Animations can be produced fast, with negligible cost, and can be exported to the Web.

- Multiple connected views and overlays

BALSA, TANGO or ANIM, they all support multiple views of an animation. Multiple views are convenient, but must be supplemented by something else. In this thesis we propose also to use view *overlays*, which can be switched on and off, in order to show several layers of information superimposed. None of the systems mentioned above provides this capability, which we explore further in Chapter 6 for our own system.

- Hierarchical stepping

Almost all the systems reviewed offer some kind of velocity control for the algorithmic animation. The viewer can proceed faster or more slowly. Only ANIM provides a “click” modus, which allows the viewer to advance an animation to several levels of click marks. This capability, which I will call here hierarchical stepping, is in fact not very difficult to implement, and will be adopted for the animation system that I describe in Chapter 6.

- Esthetics and ease of use

Last but not least, an algorithmic animation should be easy to use and should be esthetically pleasing. In the last few years I have started many algorithmic animations written by experts and students. When the graphics are not appealing, the result is not worth the effort. Algorithmic animation should be pleasing to the eye.

Finally, and to close this chapter about the state of the art in algorithmic animation, Figure 2.15 provides an illustrated chronology of the animation systems mentioned in the text above.

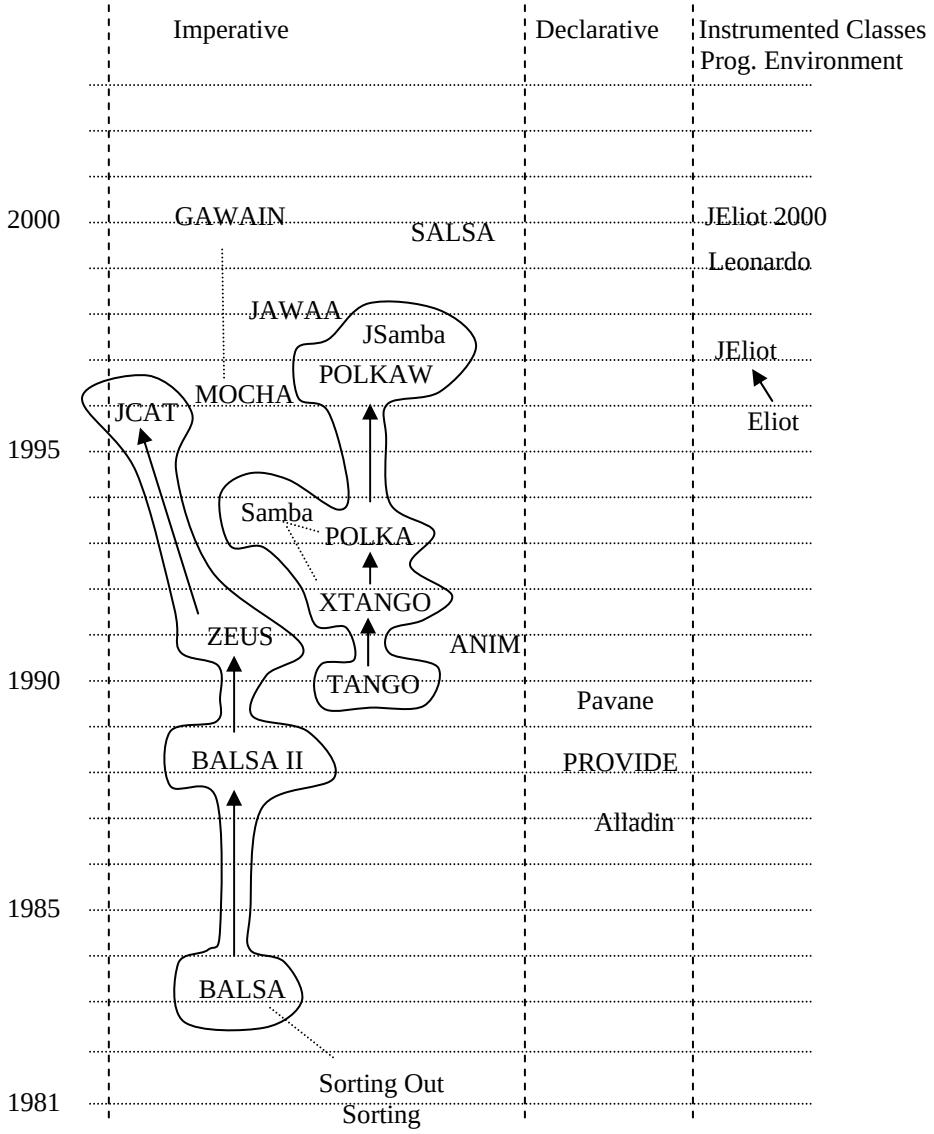


Figure 2.15 Chronology of algorithmic animation systems