A System for Algorithm Animation[†]

Marc H. Brown
Robert Sedgewick

Dept. of Computer Science
Brown University
Providence, RI 02912

Abstract: A software environment is described which provides facilities at a variety of levels for "animating" algorithms: exposing properties of programs by displaying multiple dynamic views of the program and associated data structures. The system is operational on a network of graphics-based, personal workstations and has been used successfully in several applications for teaching and research in computer science and mathematics. In this paper, we outline the conceptual framework that we have developed for animating algorithms, describe the system that we have implemented, and give several examples drawn from the host of algorithms that we have animated.

Introduction

Computer programs in execution are complex objects whose properties can be difficult to fathom. Our central thesis is that it is possible to expose the fundamental characteristics of a broad variety of programs through the use of dynamic (real-time) graphic displays and that such algorithm animation has the potential to be quite useful in several contexts. In this paper, we describe a system which we have built based on this thesis and detail some of our experiences in using it over the past year.

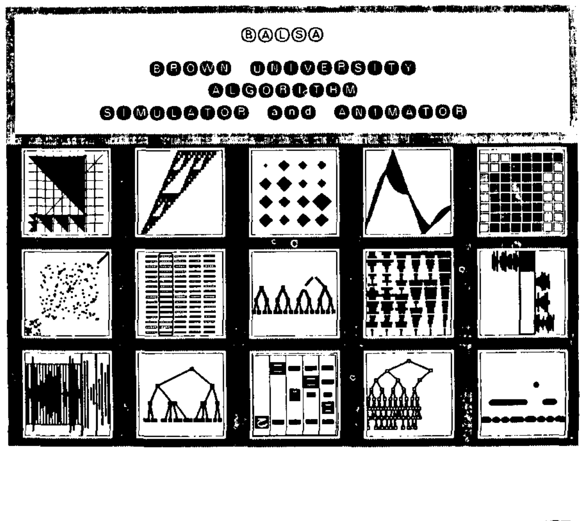
One obvious application is computer science education. At Brown University, we have a laboratory/lecture hall containing 60 high-performance scientific workstations (Apollos) with bitmap graphic displays, connected together on a high-bandwidth resource-sharing local area network. Courses in introductory programming, algorithms and data structures, differential equations, and assembly language have been taught in the lab using the software environment described in this paper as the principal medium of communication. Rather than explain a concept using a blackboard or a viewgraph projector, instructors in these courses have been able to use dynamic graphic presentations.

A second application is in research in the design and analysis of algorithms. The courseware that we have developed for teaching the algorithms and data structures course provides a firm basis to allow our software environment to be used for advanced research in a variety of areas. The ready availability of dynamic graphic displays exhibiting various properties of algorithms in execution has the poten-

[†]Support for this research was provided by the Exxon Education Foundation, and by the ONR and DARPA under Contract N00014-83-K-0146 and ARPA Order No. 4786. Equipment support was provided by NSF Grant SER80-04974 and by Apollo Computer, Inc. Support for the second author was provided in part by NSF Grant MCS-83-08806.

Figure 1. An iconic table of contents for some BALSA animations. This may be thought of as an "index" to a "dynamic book." Selecting an icon with a mouse causes a 10–15 minute dynamic simulation of the corresponding topic to be run, with pauses at key images, after which the "reader" can interact with the algorithms and images. These particular icons represent animations on mathematical algorithms (top row, left to right: Euclid's GCD Algorithm, 3/4 Recursion, Random Numbers, Curve Fitting), sorting (Insertion Sort, Quicksort, Radix Sort, Priority Queues, Mergesort, External Sorting), and searching (bottom row: Sequential Search, Balanced Trees, Hashing, Radix Searching). Several are described in more detail in following figures. The reader must bear in mind that these figures are static "snapshots" from real-time simulations, the essence of many is in their dynamic character.

One of the primary applications of the BALSA environment has been for instruction in an "Electronic Classroom" in the Dept. of Computer Science at Brown (see Fig. 10). Two exemplary courses which integrated the dynamic simulations into lectures were the first semester introductory Pascal programming course (see Fig. 3) and the third semester algorithms and data structures course (see Fig. 4).



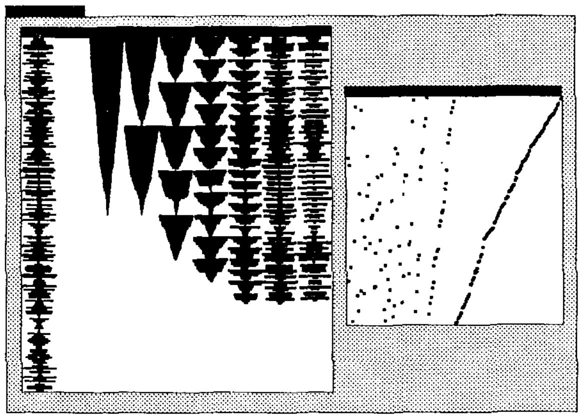
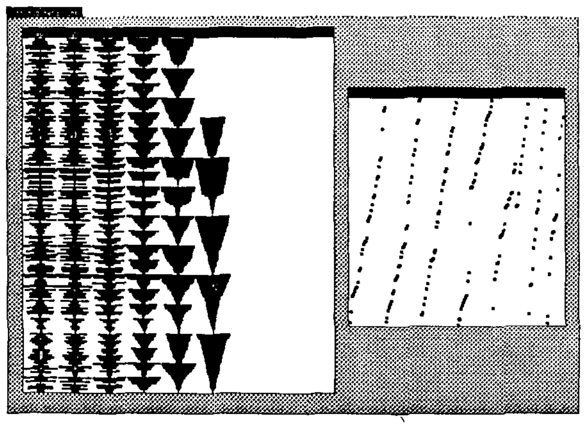


Figure 2. The image above shows top-down recursive mergesort; the image below shows bottom-up non-recursive mergesort. The large window in both images is a "horizontal bars with history" view: the array being processed is at the leftmost column (bottom to top) with lengths of bars corresponding to values of elements. The contents of the array during various stages in the algorithms are displayed in successive columns. The small window in both images is a "dots" view: each element of the array is portrayed as a dot whose x coordinate corresponds to its position in the array, and y coordinate corresponds to its value.

In the algorithms and data structures course, a typical lecture would consist of each student's workstation displaying a previously created animation script of the material from the corresponding chapter of the textbook. Rather than using a viewgraph or blackboard, diagrams such as those shown here would be presented through BALSA. Moreover, the BALSA diagrams are dynamic, which better models the true nature of the material and thus allows fuller explanations of more complex material than is available using other modes of communication. The facilities were available to students during non-class hours for completing programming assignments, replaying classroom scripts, and experimenting.



tial to significantly impact advanced research in this area.

A third application is in advanced debugging and systems programming. To date, we have done little specific work in this area, but we believe our work on techniques for visualizing properties of fundamental data structures to be of central importance in such applications. We plan to explore interfaces to performance monitors, debuggers, program development environments and other software systems in future research.

These applications illustrate that "algorithm animation" can involve a variety of different types of users and thus requires support on a variety of different levels. In this paper, we describe the technical aspects of BALSA (Brown ALgorithm Simulator and Animator), the software system we have developed to support these activities, and give examples illustrating various modes of utilization.

Certainly it is a fundamental axiom of computer graphics that visualization of abstract concepts is invaluable, and many researchers have considered the natural question of applying this principle to better understand tools of their own trade (algorithms and data structures). Some previous examples may be found in [1], [2], [3], [5], [7], [9], and [10]. Discussion of some work relating to monitoring programs in execution and to visualizing the operation of large systems programs may be found in [11] and [8], respectively. Also relevant (though not directly related) is the excellent treatment of visual displays in [14]. Many of these efforts involved considerable expense of time and money (for the use of expensive real-time systems or for the production of movies), but they do demonstrate the potential of the concept, especially [2].

The availability of high-performance scientific workstations has made it possible for us to more fully realize that potential. We have developed a software environment which makes real-time simulations of programs (as opposed to movies) using high-resolution graphics readily available to students and researchers. The BALSA system has been in production use by over 450 students and a dozen researchers since September 1983. We have gained extensive experience in actually using the system to animate scores of algorithms. Moreover, it has allowed a dynamic graphic interface to become a natural mode of interaction for a large number of students, teachers, and researchers.

The next section describes our general conceptual framework for animating algorithms. Following that, we describe in more detail what is involved in the implementations. The final section offers concluding comments and outlines some future plans. Illustrations of images from animations that we have implemented are included throughout the paper, with detailed commentary and discussion of some of the applications included in the figure captions.

User Perspective

Many of the facilities provided by BALSA are present in state-of-the-art graphics-based object-oriented programming systems such as Smalltalk [6]. The main reason that we chose to build a tailored special-purpose system is that the real-time dynamics of the programs in operation is of fundamental importance: we were not prepared to pay the performance penalties inherent in the use of a general-purpose system. Essentially, BALSA may be thought of as a laboratory for experimentation with dynamic real-time representations of algorithms. As will become apparant below, our experience with the system has uncovered a variety of fundamental issues concerning processing such objects, which we hope will be of relevance in considering the possibility of supporting BALSA-type operations with acceptable performance in general-purpose systems of the future.

The figures in this paper are only representative of the scores of algorithms that we have animated. In principle, we could make any of these animations available for any type of user at any time. A major goal of our research is to continue development of high-level facilities which might allow this as well as to integrate and assimilate generally useful views and algorithms into BALSA. We fully expect our various system "users" to be using higher level graphic and dynamic primitives as the system matures. Several different types of people can make effective use of the BALSA system, and we have found it convenient to use specific descriptive terms for each mode of use.

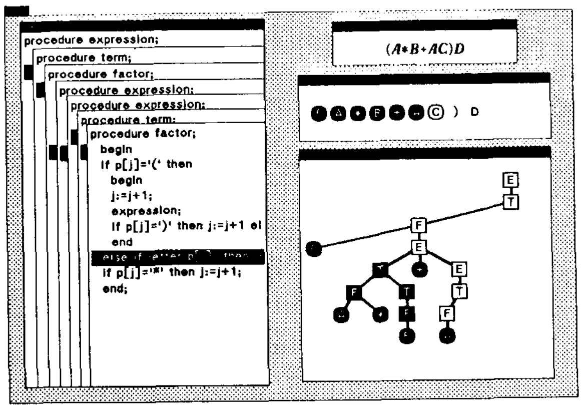
Users

We use the term *user* to describe a person who is interested in watching algorithms in execution, using BALSA's interactive facilities. This person does not write code; rather he invokes code written by others. He might be thought of as a "reader" of a dynamic book.

A *scriptwriter* is a person who prepares material for users, using BALSA's high level facilities. This person does not write code in the ordinary sense either, but he may make sophisticated use of interactive facilities and store away material for users. He might be thought of as an "author" of a dynamic book, using raw material developed by others.

An *algorithm designer* in BALSA jargon is a programmer who is interested in using BALSA's facilities to get a dynamic graphical display of his program in execution, so that it might be more easily understood. If the domain of operation of his program is close to something that has already been animated, he may use a previous implementation, and therefore not have to worry about low-level graphics. We have animated algorithms from a variety of domains, so we expect this case to be typical.

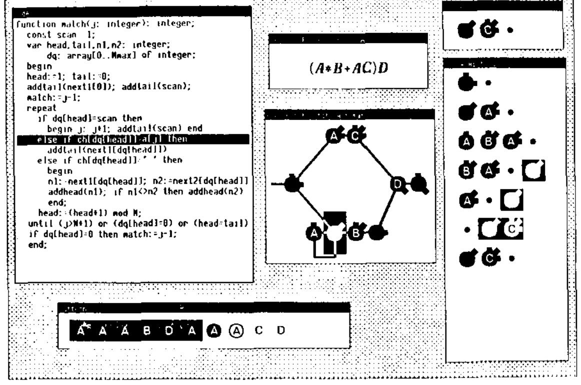
An *animator* is a person who designs and implements programs which actually display programs in execution, using BALSA's low-level facilities. This involves two types of programs: those implementing algorithms, which often come from some other source, and those involving the actual



```
procedure expression;
procedure term;
procedure factor;
procedure expression;
procedure expression;
procedure term;
procedure factor;
  begin
  if p[j]='(' then
    begin
    j:=j+1;
    expression;
    if p[j]=')' then j:=j+1 el
    end
  else if letter p[ ] then
  if p[j]='*' then j:=j+1;
  end;
```

Figure 3. These two images illustrate how a generalized regular expression pattern matcher (*grep*) works. The image above is a recursive descent compiler. In the code view at the left, the current line is highlighted and each procedure is displayed in its own overlapping subwindow. The views on the right show the regular expression, the scanning of the expression, and the parse tree. Note the correspondence between the recursive procedure invocations and the parse tree. The image below shows the workings of the non-deterministic finite state automaton built from the compiler above as it determines whether a text string can be generated by a regular expression. The views at the right show the primary data structure – a "deque." Each element in the deque is a state of the FSA, which can be identified by the tabs. The letter inside the state indicates what input character (if any) must be scanned to advance to that state.

In the introductory Pascal programming course, the principal mode of communication was for each student's machine to mimic what the instructor was doing in the BALSA environment on his machine. Students could also run programs on their own and supply data and answers in response to prompts. (Note that there are no CAI-like facilities for "response judging" in BALSA.) The animations in this course emphasized single-stepping of source code, often simultaneously with multiple levels of pseudo-code, and watching the corresponding effect on the variables and data structures. The style of overlapping subwindows for procedure invocation illustrated above proved to be a very effective method for teaching recursion.



```
function match(j: integer): integer;
  const scan 1;
  var head,tail,n1,n2: integer;
      dq: array[0..Mmax] of integer;
  begin
  head:=1; tail:=0;
  addtail(next1[0]); addtail(scan);
  match:=j-1;
  repeat
    if dq[head]=scan then
      begin j:=j+1; addtail(scan) end
    else if ch[dq[head]]=a[j] then
      addtail(next1[dq[head]])
    else if ch[dq[head]]=' ' then
      begin
      n1:=next1[dq[head]]; n2:=next2[dq[head]]
      addhead(n1); if n1<>n2 then addhead(n2)
      end;
    head:=(head+1) mod M;
  until (j>M+1) or (dq[head]=0) or (head=tail)
  if dq[head]=0 then match:=j-1;
  end;
```

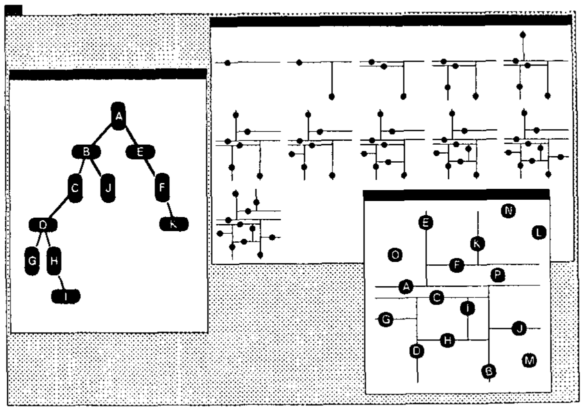
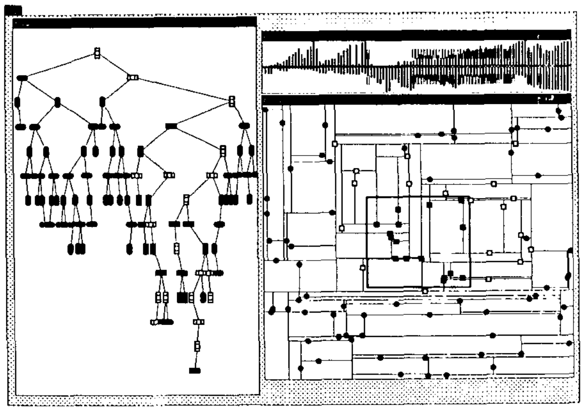


Figure 4. The image above shows the construction of a 2D tree used for range searching. The views are of the tree, the planar subdivisions induced by the tree, and a "history" of the planar subdivisions after each point is inserted. The image below shows a much larger data set (about 75 points). The dark rectangle in the planar view indicates the query range (i.e., the algorithm returns all points which fall in that area). The state of each node indicates the result of the search algorithm: circular nodes have not been accessed; hollow square nodes have been searched and found not to be in the range; filled squares are those points searched and found in the range. The view above the planar view is a 1D representation of the points: each point is drawn with a vertical bar above it corresponding to its x coordinate and the vertical bar below it corresponding to its y coordinate.

BALSA provides facilities for displaying multiple views of a data structure, all of which are updated simultaneously during program execution to give a motion picture of the program in action. Note that the representation of nodes in all views is consistent; this serves to unite the views and make a more effective total picture. Note also how the nodes are drawn in different sizes and with various levels of detail depending on both the size of the data set and the size of window in which the view is displayed. The user is able to "zoom" in or out of any level of detail, as well as "scroll" the image in the window both horizontally and vertically. For studying small cases (and introducing material to students), a "history" view proved to be very useful as did using examples directly from the textbook – with textual data. For large cases, the dynamics of the algorithm in action with abstract graphical representations of the data was the most important aspect.



graphic orders to draw pictures.

Facilities

As stated above, our general system philosophy is to provide dynamic *views* of *algorithms* in execution. To support this, the "user" of the BALSA system has three different types of capabilities: *interpretive*, for controlling the execution of the algorithms; *display*, for manipulating the presentation of the views; and *shell*, for higher-level control of BALSA primitives.

The *display* primitives of BALSA allow the user to create, size and position "algorithm windows" which contain "view windows." For example, the screen images in Fig. 4 each show one algorithm window containing three different view windows, while the second image in Fig. 8 shows two algorithm windows each containing three view windows. One possible view is a "code" view which shows the code being executed (see Fig. 3). The user has full flexibility in building his screen environment, subject only to the choice of algorithms and views left to him by the algorithm designer and animator. Commands are invoked by using a mouse on pop-up menus, and include create, delete, size, move, and other standard window operations. Zooming, panning, and overlapping windows are also supported.

The *interpretive* primitives of BALSA allow the user to start, stop, slow down, or even run an algorithm backwards. Breakpoints and stepping are supported, in units meaningful to the algorithm. It is possible to disable algorithms and views or to run several simultaneously. After an algorithm has run once, an entire history of that run is saved by the system, so it can be rerun (perhaps for comparison with another algorithm) efficiently and easily. In no sense is this part of the system intended to be a general-purpose interpreter: rather it is a set of facilities to allow control of execution of algorithms, tailored to facilitate animation.

The *shell* primitives of BALSA allow the user to save or restore window configurations and to save or invoke *scripts* consisting of sequences of BALSA primitive operations, which are typically quite long. For example, Fig. 8 shows snapshots from two scripts on the same algorithms and views: one that was developed for use in the classroom in an introductory lecture on graph algorithms, the other that was developed for use in research on graph algorithms. Both scripts use the same algorithms and views. Normally, the algorithm designer or the animator will leave a set of window configurations for the user and the scriptwriter will leave a script which loads these configurations and invokes the interpretive facilities of BALSA so as to tell the story of the algorithms. Or, the user may build and save his own window configurations and scripts for later use. Additional utilities such as screen hardcopy and communications with other BALSA users are also supported.

Animating an Algorithm

The following sequence of events would typically be involved to animate a "new" algorithm (unlike any that has ever been done before) in BALSA. First, the algorithm designer implements a "clean" version of the algorithms to be animated (for most of the examples in this paper, we started with the Pascal implementations in [13]), along with programs which provide various types of input to the algorithm. Next, the animator and algorithm designer agree on a general plan for various visualizations of the algorithms, mainly for the purpose of identifying the *interesting events* in the algorithm which should lead to changes in the image being displayed. Then the animator writes the software which maintains the image (changing it in response to interesting events) and the designer adds interesting event signals to the algorithm to pass requisite information to the graphics software. This results in a set of *algorithms* and *views* which are accessible to the user and to the scriptwriter. Then either the user could invoke the BALSA interpreter and window manager directly to create dynamic images of the type described below, or he could invoke *scripts* consisting of sequences of BALSA primitive operations previously created by the scriptwriter. More details on the creation of algorithms, views and scripts are given below, and in Figs. 6 and 7.

The Algorithm

The primary role of the *algorithm designer* is to take an algorithm and to prepare it for animation. If a similar algorithm has already been prepared for BALSA animation, he need only augment the algorithm with interesting event "signals" and inform BALSA which views and inputs are valid for the particular algorithm, using a *configuration* file. The views and inputs can be from the BALSA library, or tailored to the particular algorithm or the particular genre of algorithm (e.g., graph algorithms, or sorting algorithms, or convex hull algorithms, etc.).

At the conceptual level, the *view* paradigm is that of a "monitor" during the execution of an algorithm. As the algorithm executes and data structures are modified, the views update their graphical displays appropriately, based on information from interesting event signals. As mentioned above, we prefer this to the alternative of having the view react to general monitors on the algorithm data structures, because the needs of the view may or may not correspond directly to specific changes in the algorithm's data structures. The interesting event signals are implemented simply as procedure calls to the BALSA *IE-manager*; the parameters are the name of the interesting event followed by algorithm-specific entities. When the user causes BALSA to start normal execution of the algorithm, the algorithm will call the BALSA IE-manager for each interesting event. The IE-manager will then call all of the "active" views (i. e., those views that the user has opened on the screen). The view updates itself graphically, based on the interesting events.

Another modification that must be made to the al-

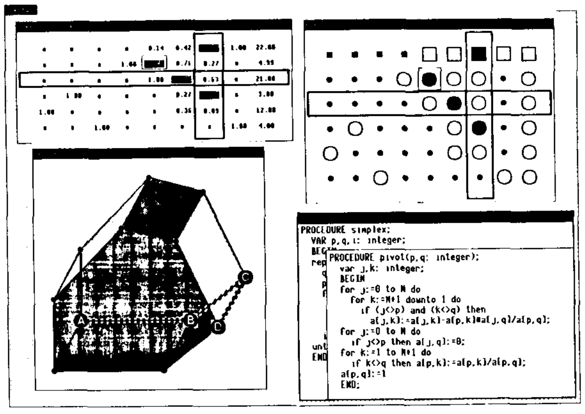
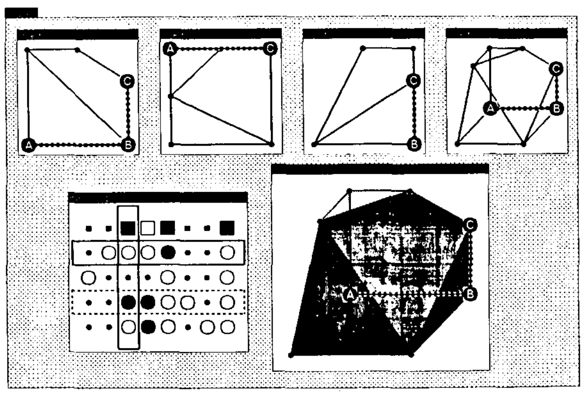

```
PROCEDURE simplex;
  VAR p,q,i: integer;
  BEGIN
  rep
  unt
  END
```
```
PROCEDURE pivot(p,q: integer);
  var j,k: integer;
  BEGIN
  for j:=0 to N do
    for k:=N+1 downto 1 do
      if (j<>p) and (k<>q) then
        a[j,k]:=a[j,k]-a[p,k]*a[j,q]/a[p,q];
  for j:=0 to N do
    if j<>p then a[j,q]:=0;
  for k:=1 to N+1 do
    if k<>q then a[p,k]:=a[p,k]/a[p,q];
  a[p,q]:=1
  END;
```

Figure 5. These images illustrate the Simplex method for solving linear programming. The image above shows the tableaux in the upper left view, an iconic version of the tableaux, the code, and the 3D object formed by plotting the system of linear equations. The algorithm is currently in the "pivot" phase, and the effect of previous pivots can be seen by tracing the object edges along the labelled vertices. The image below shows four different wire frame pictures of an object corresponding to a different set of linear constraints. The views (from left to right) are the object from the front, top, side, and finally, with some perspective (actually, it uses a "shearing" transformation). The dotted row in the iconic tableaux indicates a probe to find an appropriate row for the pivot operation, and the solid row is the current choice.

BALSA has been used in a number of non-computer disciplines to model physical experiments, as well as abstract material such as differential equations and differential geometry. These images, for example, are well-suited for a course in operations research or linear algebra. These images also illustrate a use of icons: what is important in the tableaux is not the value of the elements, but whether the element is zero (a bullet), negative (dark dot), or positive (hollow dot). Note also that the top row (which represents the condition we are solving) is displayed differently from the others (each of which represents a given constraint).



```
now:=0;
for k:=1 to V do
  begin val[k]:=unseen; dad[k]:=0 end;
pqconstruct;
repeat
  k:=pqremove;
  if val[k]=unseen then
    begin
    IE(IEaddfringe,dad[k],k,val[k]);
    val[k]:=0;
    now:=now+1;
    end
  IE(IEaddtree,dad[k],k,val[k]);
  t:=adj[k];
  while t<>z do
    begin
    if val[t↑.v]=unseen then now:=now+1;
    if onpq(t↑.v) and (val[t↑.v]>now) then
      begin
      IE(IEaddfringe,k,t↑.v,now);
      pqchange(t↑.v,now);
      dad[t↑.v]:=k
      end;
    t:=t↑.next
    end
until pqempty;
```

Figure 6. Shown above is a fragment from a typical algorithm after it has been augmented with interesting event markers (shown in italics). This algorithm was used to generate the breadth-first graph traversal images in Figs. 8 and 10, and is taken directly from the textbook.

Shown below is an excerpt from the configuration file that the algorithm designer uses to inform BALSA which views and inputs are valid for a given algorithm. The algorithm in this example is a routine called `BreadthFirst` (see excerpt above). When the user is prompted with a popup menu of possible algorithms, BALSA will use the label `BFS`. The algorithm designer has specified that this algorithm has two possible input routines and three different views.

```
ALGORITHMS =>
  BreadthFirst "BFS"
    INPUTS: GRAPHinputFile GRAPHinputRandom
    VIEWS: GRAPHviewPlane GRAPHviewFringe
```

gorithm is the I/O routines. Calls to conventional input routines (e.g., **readln** in Pascal and **scanf** in C) must be replaced by calls to a BALSA *input-manager*, which in turn calls the input module which the user has selected. Tools from the window manager/user interface package are available for the implementation of input modules, so that interaction can be arranged. However, the identity of the input module which is actually in use is transparent to the algorithm. The effect of calls to output routines (e.g., **writeln** and **printf**) are not visible in the BALSA environment per se; however, the user can see conventional, textual output by linking an "output-view" using interesting events with parameters analogous to output statements. For teaching introductory programming, this type of view (and also a view of the input stream) has proven very helpful.

BALSA can take this modified algorithm and generate a *code view*, a "pretty-printed" version (with uniform indentation, interesting event calls removed, and I/O statements restored) with special interesting event calls inserted at each line of code (see Fig. 3). These interesting events are fielded by the BALSA library code view routine, so that the user can see each line of his program highlighted as it is exectued, etc.

Views

The primary role of the *animator* is to implement the graphics commands that actually produce images, in response to interesting events signals.

Our experience has been that sophisticated views can require costly computations to update the graphics on the screen. Since many views (including multiple instantiations of the same view) frequently use the results of the costly computation, we have developed the concept of *view data structure managers* (VDSMs). A VDSM is a set of routines, frequently shared among views, that performs various computations required by the views. Thus, at each interesting event, the BALSA IE-manager calls all VDSMs associated with active views and then calls all active views. Note that computation done in one view cannot be used by another view, since the other view will only be called when the user has opened a window of the view. The shared work must be done by the VDSM.

If an algorithm is executing when the user first opens an instantiation of the view on the screen, the view must display itself corresponding to the current state of the algorithm. This could be done in one of two ways. First, BALSA could replay its saved history of interesting events and the view would update itself incrementally as if the program were executing. This method has the problem that one might not be interested in what happened in the algorithm over history; rather the current state is of interest. The second option, which is more difficult for the animator to implement, is for the view to refresh itself from the current VDSMs. (In this mode, the VDSM —if it was not already active because of another dependent view— would be called incrementally with the history of interesting events so that it would be current.)

VDSMs and views must also be able to reverse execution. Our current BALSA interpreter, when told by the user to run in reverse, will go through the history of interesting events in reverse order and call the VDSMs and views with a flag indicating that the direction is reverse. The VDSMs and views must undo the graphics associated with the interesting events. This is also another reason for VDSMs: while conventional compilers and interpreters do not run code backwards, the VDSM data structures need to be undone to some extent. Undoing the graphics for some views is simple. For example, to undo the effects of exchanging the contents of two elements from an array is usually identical to exchanging them in the first place. In contrast, undoing the insertion of a node in a balanced 2-3-4 tree is non-trivial.

The final responsibility of views is that of "inquiry." For example, if an animator writes an input module for a binary tree deletion algorithm, the user might want to specify which node to delete by pointing at it with a mouse. The view must be able to map a point on the screen into a coordinate system meaningful to the view, VDSM, and input modules.

In summary, a view can be called in one of five "modes": forward, backward, rerun (usually the same as forward), refresh, or inquiry. The VDSMs can be called in either the forward or backward mode. Most successful animators take the approach of designing for all modes, but only implementing the forward and (if the input module requires) inquiry modes to start. As the view matures, the other modes are gradually implemented. For example, animators will often not invest the time needed to make a view reverse itself until the view has become more versatile, at which time it would probably also be added the BALSA library.

Scripts

The primary job of the *scriptwriter* is to assemble algorithms and views into a coherent dynamic entity to tell a story. The mechanism currently provided in BALSA to allow this is quite rudimentary: the scriptwriter simply uses the interactive facilities of BALSA in a mode where everything that he does is saved in a file to be later played back. Some features are provided to allow different things to happen on playback: the most commonly used is the *future freeze* which is a no-op during interaction, but a "pause" (wait for the user to press a button) during playback. Also, it is possible to save complex window configurations (*scenes*) to be loaded later. Typically, the scriptwriter will create scenes or sequences of scenes consisting of several algorithms and views, then create a script to run the algorithms on a variety of inputs, with future freezes inserted at particularly interesting points.

It is possible, albeit difficult, to edit scripts: this is an area in which we plan to significantly extend the capabilities of BALSA. Also, we expect to allow various types of conditional execution of scripts (extensions to future freeze) in future versions of the system.

```
IES=>
  IEinit "Initialize" "%d %d %d %d"
     -- xmin, ymin, xmax, ymax
  IEinitvertex "Init Vertex" "%d %d %d"
     -- vertex id, xcoord, ycoord
  IEinitedge "Init Edge" "%d %d %d"
     -- vertex1, vertex2, weight
  IEaddtree "Add Vertex to Tree" "%d %d %d"
     -- father vertex, vertex, value
  IEaddfringe "Add Vertex to Fringe" "%d %d %d"
     -- father vertex, vertex, value
INPUTS=>
  GRAPHinputFile "File"
  GRAPHinputRandom "Random"
VIEWS=>
  GRAPHviewPlane "Points in Plane"
    IES: IEinit IEinitvertex IEinitedge
         IEaddtree IEaddfringe
    VDSMS: GRAPHvdsm
```

Figure 7. Shown above is an excerpt from the configuration file that the animator uses to provide BALSA with detailed information about interesting events, inputs, and views. Each interesting event registers with BALSA a control string specifying the data types of the algorithm-specific parameters, and each view lists its associated VDSMs and the interesting events to which it will respond.

Shown below is pseudo-code for the VDSM and view that displays the graph. With the VDSM as shown below, the view could not refresh itself from the current state of the data structures, nor could it execute in the "backward" direction (because the old mark-state of each node is not known). Thus, a more sophisticated VDSM would be needed, but the view would not be any more complicated than above. Note carefully that the view data structure does *not* include graph edges (they are just drawn on the screen). This view and VDSM are very versatile, and can be used for many very different graph algorithms, including those for dense graphs which are based on an adjacency matrix rather than an adjacency list.

```
GRAPHvdsm:
  switch (type of interesting event)
    case IEinitedge:
      save x and y coords of vertex
      mark all vertices as "unseen"
    case IEaddtree
      mark vertex as "tree"
    case IEaddfringe:
      mark vertex as "fringe"
  endcase

GRAPHviewPlane:
  switch (type of interesting event)
    case IEinit:
        initialize graphics window to parms
    case IEinitvertex:
        draw vertex node in its mark-state
    case IEinitedge:
    case IEaddtree:
    case IEaddfringe:
        if both vertices are "unseen" =>
          style=THIN
        else if either vertex is "fringe" =>
          style=DASHED;
        else style=THICK
        draw edge from v1 to v2 in style
        draw v1 node in its mark-state
        draw v2 node in its mark-state
  endcase
```

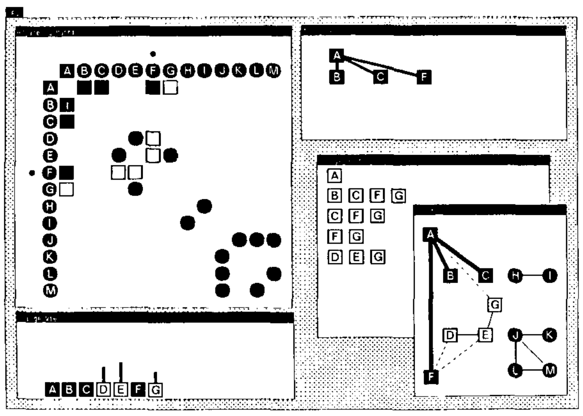
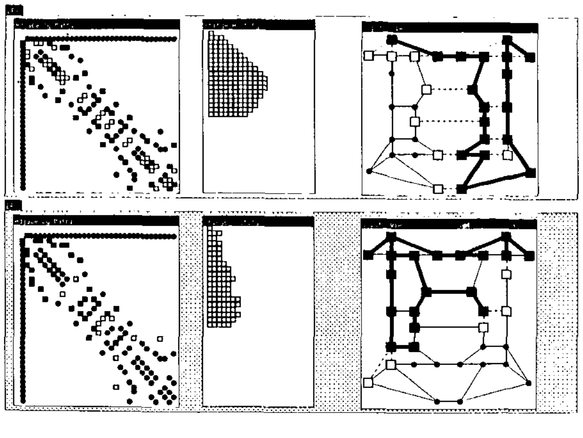


Figure 8. The image above illustrates a breadth-first traversal of a small undirected graph. The display style of each node and edge indicates its state: dark circular nodes with thin edges have not yet been visited; hollow square nodes with dotted edges are nodes on the priority queue data structure ready to be visited in the near future; and dark square nodes with thick edges are nodes which have already been visited. The large view at the left shows the adjacency matrix; the view below that shows the current contents of the priority queue (the height of the stick above each node indicates its priority); the view at the upper right shows the connected components; the view below that shows a "history" of the fringe; and finally, the view in the lower right corner shows the graph itself. The image below is a comparison of depth-first (top) and breadth-first (bottom) traversal algorithms.

BALSA has been used for research in the design of algorithms. It is especially useful when designing new variants of old algorithms, or new algorithms which operate within standard contexts. The image below illustrates the BALSA feature of executing multiple algorithms in parallel. This has proven to be a very effective means for comparing and contrasting different methods. (BALSA synchronizes the algorithms by allowing each to execute a fixed number of *interesting events*.)



Conclusion

One thing that we have learned from our experience in animating algorithms is that algorithms in execution are even more complex and intricate objects than we had anticipated, and we can make good use of further improvements in hardware technology. For example, preliminary experiments show that color is likely to play a prominent role in future animations (see Figs. 9–10). We have several plans for extending animations that we have done in black-and-white to make full use of color.

By contrast, a second thing that we have learned from our experience is that many of the animations that we finally settled on could be done on much more modest hardware. Accordingly, we are investigating the development of a BALSA that could be made widely available, say on Apple Macintoshes or IBM PCs. This does not imply that it would have been prudent to use such hardware from the beginning: most of our animations are the product of a significant amount of experimentation and development, which would not have been feasible on less powerful machines.

Our highest priority is to evaluate and assess the views that we have implemented, with a general goal of assimilating and integrating them into the system, so that more complicated animations can be easily built from them. This is likely to be quite difficult, for example, we have over a dozen animations involving trees, but each has slightly different characteristics (see, for example, Figs. 3 and 4).

Also under study is the addition of more general-purpose capabilities to the system (e.g., a syntax-directed editor), automation of some of the stages of animation (e.g., the addition of interesting events), and the implementation of BALSA-like animations on general-purpose systems (e.g., those which support monitors, such as PECAN [12]). Of prime concern here is the balance between performance (as stressed in BALSA) and functionality (as stressed in PECAN).

Another area of interest is to provide more powerful facilities for the scriptwriter. Certainly, he should be able to edit scripts, perhaps using a generalized undo-redo facility such as [15], though the extensive amount of context in BALSA makes this challenging. Yet another possibility is to consider nonlinear or conditional scripts, as in traditional computer-aided-instruction systems. Also, we have plans for providing graphical aids to the scriptwriter, allowing him to manipulate iconic representations of window configurations, algorithms, views, input modules, and scripts.

Finally, we are continually interested in extending the applicability of BALSA by animating more programs from more domains. In particular, we would like to address the problem of animating very large programs, so that BALSA could be of use in systems programming applications. For these and other applications, it is our hope that the tools that we have built to date will convince teachers and researchers that there is the potential to make a quantum step forward in the way in which they interact with computer systems.

References

[1] Baecker, Ronald, "Two System Which Produce Animated Representations of the Execution of Computer Programs," *ACM SIGCSE Bulletin* **7**, 1 (February 1975), 158-167.

[2] Baecker, Ronald, "Sorting out Sorting," 16mm color sound file, 25 minutes, 1981. (SIGGRAPH 1981, Dallas, Texas)

[3] Booth, Kellogg, "PQ Trees," 16mm color silent file, 12 minutes, 1975.

[4] Brown, Marc H. and Sedgewick, Robert, "Progress Report: Brown University Instuctional Computing Laboratory," *ACM SIGCSE Bulletin* **16**, 1 (February 1984).

[5] Dionne, Mark S. and Mackworth, Alan K., "ANTICS – A System for Animating LISP Programs," *Computer Graphics and Image Processing* **7** (1978), 105-119.

[6] Goldberg, Adele, *Smalltalk*, Addison-Wesley, Reading, MA, 1983.

[7] Guibas, Leo and Sedgewick, Robert, "A Dichromatic Framework for Balanced Trees," in *Proc. 19th Annual Symp. on Foundations of Computer Science*, October 1978, pp.8-21.

[8] Herot, Christopher F., et. al., "An Integrated Environment for Program Visualization," in *Automated Tools for Information Systems Design*, H.J. Schneider and A.I. Wasserman, Ed., North Holland Publishing Co., 1982, pp. 237-259.

[9] Knowlton, Kenneth C., "L6: Bell Telephone Laboratories Low-Level Linked List Language," two black and white sound films, 1966.

[10] Myers, Brad A., "Displaying Data Structures for Interactive Debugging," CSL-80-7, Xerox PARC, Palo Alto, CA, 1980. (Summary in SIGGRAPH 1983)

[11] Plattner, Bernhard and Nievergelt, Jurg, "Monitoring Program Execution: A Survey," *Computer* **14** (November 1981), 76-93.

[12] Reiss, Steven P., "PECAN: A Program Development System that Supports Multiple Views," , Orlando, FL, March, 1984.

[13] Sedgewick, Robert, *Algorithms*, Addison-Wesley, Reading, MA, 1983.

[14] Tufte, Edward R., *The Visual Display of Quantitative Information*, Graphics Press, Cheshire, CT, 1983.

[15] Vitter, Jeffrey S., "USeR: Undo, Skip, et Redo," , Pittsburg, PA, April, 1984.

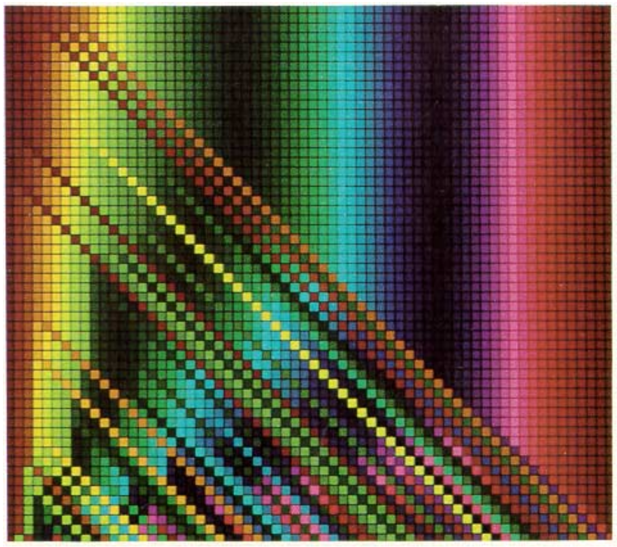
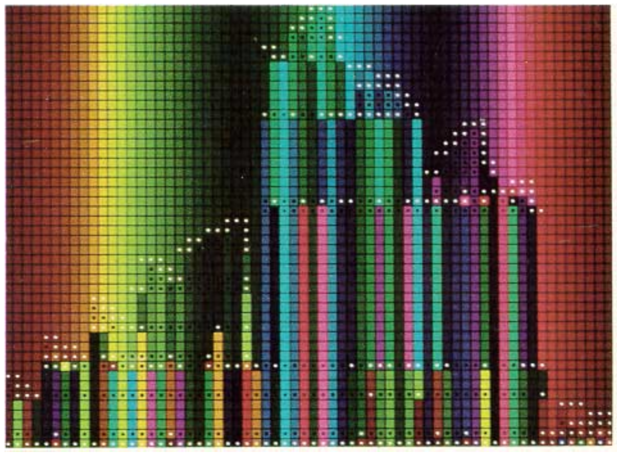


Figure 9. These two images are "chips" views of Bubble Sort (above) and Quicksort (below). The contents of the array are displayed as a row of paint chips (from left to right), with color corresponding to value. Each row (from bottom to top) shows the the array at various stages of the algorithm. It is instructive to note that the number of "stages" does not determine a fast or slow algorithm; rather, it is the amount of work that must be done during each stage. In the Quicksort image, the dot at the center of each chip indicates this "work": a white dot indicates that the element was moved, and a black dot indicates that the element was accessed but not modified. Thus, it is the sum of these dots that gives a realistic first-approximation of the algorithm running times. In a "dots" image of Bubblesort, about half of the total chips would contain dots. Note how color is used to illustrate the time dimension.



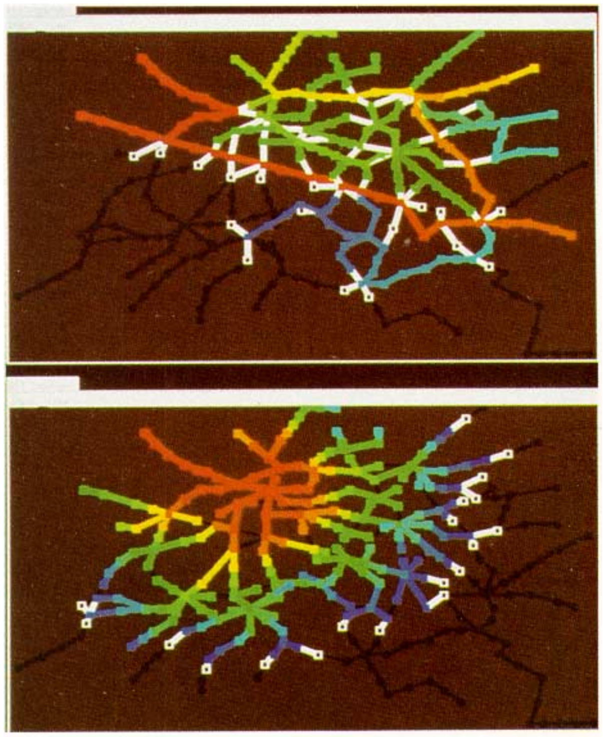


Figure 10. The image above illustrates a depth-first (top) and a breadth-first (bottom) traversal of the graph representing the Paris Metro system. Colors spanning the spectrum from red to blue are used to indicate when in time a particular node has been visited. The nodes in white are on the data structure ready to be visited; those in black have not yet been visited (see Fig. 8).

The picture below (reprinted courtesy of Bryce Flynn — Picture Group Inc.) shows the "Electronic Classroom" at Brown, a specially built auditorium/lecture hall housing 60 powerful graphics-based workstations. This picture was taken during a lecture on elementary sorting (see Fig. 9).

Acknowledgements

To date, the primary use of BALSA has been in the instructional computing laboratory at Brown. Although many people have helped to make the project successful, Andy van Dam's tireless efforts to ensure an impressive physical environment certainly must be singled out for acknowledgement. He has also contributed to the project as the instructor of the introductory programming course.

Much of the current version of BALSA was implemented by Mike Strickman. Steve Reiss, Joe Pato, and Dave Nanian wrote significant pieces of the underlying software. Tom Freeman did preliminary work for some of the color images.

As usual, Janet Incerpi's TEXpertise has been invaluable, and thanks are due to Steve Feiner for advice and support in producing the images. These two also provided detailed comments and suggestions on earlier drafts of the paper.

A prototype on which some of the graph traversal views are based was developed by the second author with Leo Guibas at Xerox PARC, using the Cedar environment on the Dorado.