1 Introduction

An algorithm animation environment is an "exploratorium" for investigating the dynamic behavior of programs, one that makes possible a fundamental improvement in the way we understand and think about them. It presents multiple graphical displays of an algorithm in action, exposing properties of the program that might otherwise be difficult to understand or might even remain unnoticed. All views of the algorithm are updated simultaneously in real time as the program executes; each view is displayed in a separate window on the screen, whose location and size is controlled by the end-user. The end-user can zoom into the graphical (currently, two-dimensional) image to see more detailed information, and can scroll the image horizontally and vertically. Views can also be used for specifying input to programs graphically. A specialized interpreter controls execution in units that are meaningful for the program, and allows multiple algorithms to be run simultaneously for comparing and contrasting. The end-user can control how the algorithms are synchronized by manipulating the amount of time each unit takes to execute. Programmers can implement new algorithms, graphical displays, and input generators and run them with existing libraries of algorithms, displays, and inputs.

While an algorithm animation environment is a rich environment for actively exploring algorithms, in many situations a passive, guided approach using a prepared "script" is more appropriate. For example, when dynamic material is a visual aid in a lecture or when it complements a traditional textbook or journal article, the audience is interested in viewing the "virtual videotape" as the author conceived it, not in exploring the material independently. And when an algorithm is being viewed for the first time, self-guided exploration can easily result in distorted or incorrect interpretations and leave important aspects of the algorithm undiscovered.

To a first approximation, an algorithm animation script is a record of an end-user's session that can be replayed. At one end of the spectrum, a script is merely a videotape that can be viewed passively. At the other, more interesting end of the spectrum, a script is an innovative communication medium: the viewer of a script can customize the movie interactively, and can readily switch between passively viewing it and actively exploring its contents. Scripts can be used as high-level macros, thereby extending the set of commands available to end-users of the algorithm animation system, and can also serve as the basis for broadcasting one end-user's session to other end-users on other machines. End-users can create scripts easily by instructing the system to "watch what I do." Scripts are stored as readable and editable PASCAL programs.

Systems for algorithm animation can be realized with current hardware: personal workstations—with their high-resolution displays, powerful dedicated processors, and large amounts of real and virtual memory—can support the required interactiveness and dynamic graphics. In the future, such workstations will become cheaper, faster, and more powerful, and will have better resolution. An algorithm animation environment exploits these characteristics, and can also take advantage of a number of features expected to become common in future hardware, such as color, sound, and parallel processors.

We develop here a model for creating real-time animations of algorithms with minimal intrusions into the algorithm's original source code, as well as a framework for interacting with these animations. We also describe a prototype system, BALSA–II, and its feasibility study system, BALSA–I, that realize the conceptual model. Currently, BALSA–II is being used in teaching parts of a data structures course, for research in the design and analysis of algorithms, and for technical drawings in research papers and textbooks. BALSA–I has been in production use since 1983 in Brown University's "Electronic Classroom." Appendix A documents some of our experiences using the environment as a principal mode of communication during lectures in an introductory programming course and in an algorithms and data structures course. Appendix B cites publications describing various aspects of the project.

1.1 Thesis Contributions

The primary contributions of this thesis are its models for (1) programmers creating animations, (2) end-users interacting with the animations, and (3) end-users creating, editing, and replaying dynamic documents. These models have been realized in the BALSA–I and BALSA–II systems. A secondary contribution of this research is the numerous static and dynamic graphical displays of a wide range of algorithms and data structures we have created using the prototype systems, most of which had never been displayed or even conceived previously. The domain includes sorting, searching, string processing, parsing, graphs, trees, computational geometry, mathematics, linear and dynamic programming, systolic architectures, and graphics. The systems have also be used to show innovative dynamic illustrations of fundamental concepts in procedural programming languages. The diagrams in this document are a small sampling of these images; others have been reported elsewhere [16, 17, 18].

We now elaborate on the primary contributions of this thesis.

(1) **A model for programmers creating animations.** The programmer model is independent of the contents of all algorithms, inputs and views; hence, it can be used to animate any algorithm in a systematic manner. Moreover, the model makes it easy to animate new algorithms and create new displays.

Algorithms being animated are separated into three components: the *algorithm* itself, an *input generator* that provides data for the algorithm to manipulate, and graphical *views* that present the animated pictures of the algorithm in execution. Views are built following a classical graphics *modeler–renderer* paradigm, and an *adapter* allows any particular view to be used to display aspects of many different algorithms. Modelers can be chained to provide views of views, and renderers can be based on multiple modelers. Algorithms are annotated with *interesting events* to indicate phenomena of interest that should give rise to the displays being updated; in addition, the events provide the abstraction for end-users to control the execution.

(2) **A model for end-users interacting with animated algorithms.** The end-user model, like the programmer model, is independent of the algorithms, inputs, and views; thus, end-users interact with animations

in a consistent manner. This model gives well-defined semantics for each end-user command; in fact, BALSA–I and BALSA–II can be thought of as merely two different user interfaces that manipulate these properties.

The interactive environment is characterized at any point by its *structural*, *temporal* and *presentation* properties. The structural properties are the set of algorithms currently running and the data they are processing. Information concerning the specialized interpreter, such as the program-specific units chosen for stopping and stepping points and how multiple algorithms are synchronized, are the temporal properties. The configuration of view windows on the screen are considered the presentation properties. In addition to providing data for algorithms to process, end-users can manipulate the underlying algorithms, input generators and views through the concept of *parameters* for each component. That is, end-users can select among parameters preset by the programmers; end-users cannot create new variants at runtime.

(3) **Model for end-users creating, editing, and replaying dynamic documents.** Dynamic documents, called *scripts*, are created by having the system watch what the end-user does. However, a semantic interpretation of the actions is maintained in a textual file—an executable PASCAL program—not a command or keystroke history. Scripts form a basis for passively watching the dynamic material like a videotape, or actively interacting with the material. The script model is mostly independent of an algorithm animation system: the principals can be applied to virtually any system with well-defined structural, temporal and presentation properties.

1.2 Applications of Algorithm Animation

An obvious application of an algorithm animation environment is computer science *instruction*, particularly courses dealing with algorithms and data structures, e.g., compilers, graphics, databases, algorithms, programming. Rather than using a chalkboard or viewgraph to show static diagrams, instructors can present simulations of algorithms and programming concepts on workstations. Moreover, students can try out the programs on their own data, at their own pace, and with different displays (from a library of existing displays) from those the instructor chose. Non-naive students can code their own algorithms and utilize the same set of displays used by the instructor in demonstration programs. As mentioned earlier, the appendices describe how BALSA–I was used in computer science instruction.

Another proven application of an algorithm animation environment is as a tool for *research* in algorithm design and analysis. Human beings' ability to quickly process large amounts of visual information is well documented, and animated displays of algorithms provide intricate details in a format that allows us to exploit our visual capabilities. For instance, experimenting with an animation of Knuth's dynamic Huffman trees [43] revealed strange behavior of the tree dynamics with a particular set of input. This lead to a new, improved algorithm for dynamic Huffman trees [70]. A variation of Shellsort was discovered in conjunction with static color displays of Bubblesort, Cocktail-Shaker Sort, and standard Shellsort [41]. An early version of BALSA–I was used to help understand and analyze a newly discovered stable Mergesort [36].

Animations developed for instruction can be used for research, and vice versa. For example, animations for an algorithms course were used with minor changes to investigate shortest-path algorithms in Euclidean graphs [61]. Conversely, displays developed in conjunction with research on pairing heaps [29] were later incorporated into classroom lectures on priority queues.

Another application for an algorithm animation environment is as a testbed for *technical drawings* of data structures. It allows interactive experimentation with input data and algorithm parameters to produce a picture that best illustrates the desired properties. Furthermore, the drawings produced are always accurate, even ones which would tax the best of draftsmen. For example, it is laborious indeed for a draftsman to take a set of points and prepare the sequence in Figure 1.1 showing the construction of a Voronoi diagram and its dual. As more and more researchers begin to typeset their own papers and books, this application will become increasingly important.

A prime but so far unexplored application area for algorithm animation is in *programming environments.* Pioneering environments on graphics-based workstations, such as Cedar [65], Interlisp [64], and especially Smalltalk [31], are, by and large, text-oriented. Recent workstation-based program development environments incorporate graphical views of the program structure and code, not data, and consequently have had limited success in giving additional insight into the programs: "The experience we have had with

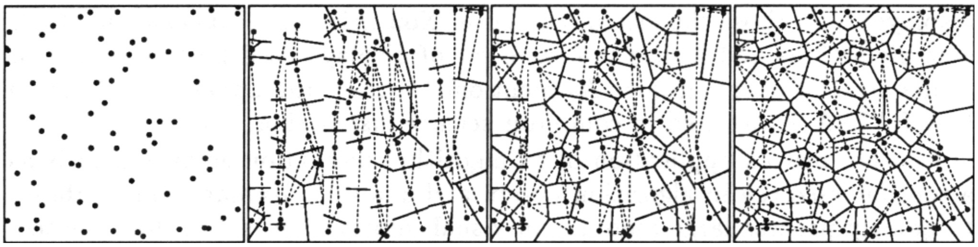


Figure 1.1: *Construction of a Voronoi diagram.*

PECAN, however, has shown that such graphical views are limited in their power and usefulness when they are tied to syntax. The syntactic basis forces the user to treat these two-dimensional representations in a one-dimensional way, and the graphics do not provide any significant advantage over text" [57]. These systems could be greatly enhanced by the display capabilities of an algorithm animation environment.

Algorithm animation has also been used for *performance tuning* [24], and has the potential to be helpful in *documenting programs* [47] and in *systems modeling*, especially for multi-threaded applications.

1.3 Conceptual Model

Algorithm animation involves two types of users: *end-users* and *client-programmers*. The end-users watch and interact with the animations on a workstation, whereas the programmers implement the algorithms, displays, and input generators that the end-users see and manipulate. An algorithm animation system itself is domain-independent: the system does not know whether an algorithm sorts numbers or produces random numbers, or whether a view shows a tree or a table. It does not attempt to decide what phenomena are interesting in a program, or what styles of input or visual representations are appropriate. Rather, it provides tools so that a large variety can be easily implemented and end-users can watch and interact with them in a consistent manner.

We will use the terms *algorithm animation environment* and *algorithm animation system* to reflect the two types of users. The algorithm animation system is the code with which client-programmers interface, and the algorithm animation environment is the runtime environment that end-users see. It is the result of compiling the code that client-programmers implement with the algorithm animation system.

For end-users, the main goal of the algorithm animation environment is to provide a consistent manner in which to interact with animations, independent of who happened to prepare the animation and what domain the animation happens to be from. Once an end-user has used the system for one algorithm, he should know how to use it for any and all algorithms.

For client-programmers, the main goal of an algorithm animation system is to provide all of the ancillary functions needed to make an interactive animation. Each programmer should not need to reinvent and reimplement facilities common to all views, such as zooming into displays. A second important goal is to provide a model whereby the animation code is separated from the algorithm. Moreover, the code relating to the animation (and to preparing input for an algorithm) should be shareable by many algorithms. Thus, a programmer implementing an algorithm should be able to concentrate primarily on the algorithm, independent of input generators and displays and the window configuration selected by the end-user. Conversely, a programmer implementing a display should do so without concern for the algorithm, input generators or the end-users.

The program being animated must be split into various pieces so that the algorithm animation system, as well as the end-users, can manipulate them systematically. Programs are separated into three components: the *algorithm* itself, the various *input generators* that provide data for the algorithm to manipulate, and the various *graphical displays*, or *views*, that present the animated pictures of the algorithm in execution.

The remainder of this section presents a high-level overview of the model an algorithm animation systems gives to its two types of users. The descriptions of the models here are necessarily brief and incomplete; Chapters 3 and 4 are devoted to end-users, and Chapters 5 and 6 to programmers.

End-User's Model

The end-user of an algorithm animation environment is always in a "setup-run" loop:

Setup: The end-user arranges the screen and decides which algorithms to run, which input generator and views to use, and what the values of any parameters to each of these should be. Each algorithm has a default setup that can be designated and changed by the end-user.

Run: The end-user runs the algorithms and watches them in the view windows on the screen. While the algorithms are running, the end-user can suspend them to change the ensemble of views on the screen as well as the program's speed and breakpoints.

Changing or creating the content of an algorithm, view, and input generator is done, strictly speaking, not by an end-user but by a programmer. Such editing is done outside of the algorithm animation environment, using the standard editors and compilers. If the machine supports multiple processes as well as dynamic loading and unloading, the algorithm animation environment does not have to be exited.

Because the notion of parameters to algorithms, input generators, and views is rather unconventional, we now elaborate.

Algorithms, input generators, and views can all be tuned directly by the end-user. Just what the parameters mean for any particular algorithm, input generator, or view depends on how it was implemented. Thus, it is the programmer, not the end-user, who decides what the parameters are—whether the particular component will even have any parameters, what the user interface will be that controls how they are set, what their default values are, and so forth. The user interface management tools and guidelines of the underlying workstation environment promote consistency in the user interface for manipulating parameters across many domains.

Algorithm parameters affect the algorithm, not the data that the algorithm manipulates. For example, should the lexical analyzer in an animated compiler use a hash table of size 119 or 2001? Or should it use a binary tree (or some specific type of balanced tree) rather than a hash table? *Input parameters* affect the input generators. For example, the input parameter to a generator that reads numbers from a file for sorting algorithms would be the name of the file. Another generator for the same sorting algorithm might produce random numbers; the parameters for this generator would be how many numbers to produce and a seed for a random number generator. Of course, an input parameter will affect the data indirectly, which in turn will affect the algorithm. *View parameters* affect how information is displayed in a particular view window; they do not affect the algorithm or input generator. For example, should a node in a graph be displayed as a circle or as a square? Should text inside the node scale with the size of the node, or stay a fixed size? The end-user's display preferences are not relevant to the algorithm or to the input generator.

Using parameters to interact with programs is a novel approach. It provides a consistent framework for allowing end-users to specify various types of information when they want, not when the program wants it. In general, algorithm and input generator parameters are specified before an algorithm runs; view parameters can be changed while a program is running or even after it has finished. At any time, the end-user can observe the current values of any of the parameters. Unfortunately, this strategy does not work 100% of the time. There are situations in which information cannot be specified before a program runs—for example, picking a node to delete in a binary tree—because they are based to some extent on the current runtime state of the algorithm. Such information is called *runtime-specifics.*

Programmer's Model

While algorithms, input generators, and views are highly interrelated, they must also be independent and must conform to rigid interface specifications in order to work with one another and with the algorithm animation system. Each algorithm has two parts: code to implement the algorithm and code to manipulate end-user parameters. Similarly, each input generator and each view has a separate part to handle its end-user parameters. As will be described in later chapters, each component has additional parts to handle other specific functions. An overview of the relationship among components is shown in Figure 1.2.

Algorithms are coded in a high-level language such as PASCAL and are annotated with markers called *interesting events* or *algorithm events*, indicating the phenomena that will be of interest when the program runs. One type of algorithm event, an *output event*, corresponds to the points at which a *Writeln* might have been placed for debugging, tracing or generating output of the algorithm in a non-graphical environment. When the program runs within the algorithm animation environment, all views on the screen are notified whenever an output event occurs and each view updates itself as appropriate to reflect the event. The other type of algorithm event, an *input event*, corresponds to the place at which a *Readln* might have appeared in

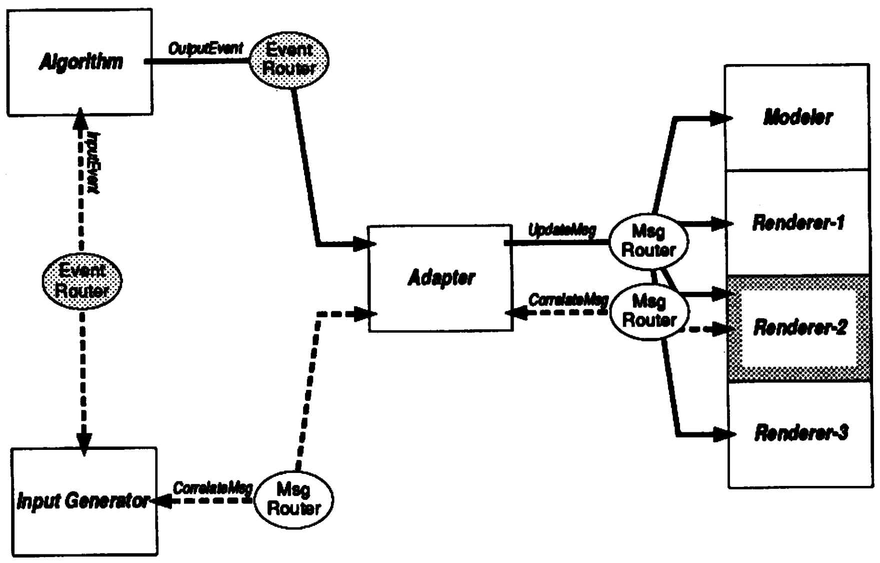


Figure 1.2: *Relationship among components that a client-programmer implements to animate algorithms. Boxes represent components implemented by the programmer, and ovals represent parts of the algorithm animation system that are responsible for routing information among components. Solid arrows indicate unidirectional flow of information, and dashed arrows indicate bidirectional flow. The renderer associated with the view window having the keyboard focus is highlighted.*

a conventional implementation of the algorithm. During execution, the current input generator is notified of each input event and responds by returning some data to the algorithm.

Events are an important aspect of the conceptual model. Essentially, an event is a "code atom" which serves the following purposes: (1) it gives a name to a segment of code so end-users can refer to it; (2) the end-user can associate a cost to computing the segment; (3) the end-user can single-step and set breakpoints based on events; and finally (4a) an output event is a signal for the views to update themselves and (4b) an input event is a signal for the input generator to provide the algorithm with data.

Algorithm events are the software analogue of an oscilloscope: the output events correspond to points on a circuit where the probe leads are attached to observe signals of interest, and input events correspond to connecting a circuit's input port to power, ground, or some other circuit. The end-user is aware of events because they provide the abstraction through which the algorithm's execution is controlled.

A view embodies a synthetic, dynamic, graphical entity. The image displayed on the screen is the result of *update messages* the view receives; a view does not access an algorithm's data structures. A view is internally structured into a *modeler* and a *renderer*: the modeler maintains the model which the renderer displays on the screen in a view window. Multiple renderers can be simultaneously displaying (in different ways) the same model (in separate windows). An *adapter* converts algorithm output events into update messages understood by the modelers and renderers. Consequently, views can be used without change in a variety of algorithms; in addition, as we shall see, views can be chained together to display aspects of themselves.

Views can also be used to allow the end-user to provide input graphically in response to an algorithm input event by pointing in a view window. Because only the renderer knows how the model is displayed on the screen, the input generator must query the renderer to correlate a point on the screen to the model. The adapter is also used to convert the queries and responses between the input generator and the renderer associated with the view window that has the current keyboard focus; the queries are called *correlate messages.* Syntactic error-checking can be done in the renderer, but semantic checking must be done either by the input generator or the algorithm. Specifying information graphically is not always meaningful in all views.

Cast of Characters

Algorithm animation involves a variety of activities, each of which draws on the skills of a particular group of people. Because a primary focus of this thesis is to consider just what tools are needed to animate programs, we first need to identify this cast of characters. These classifications are just for the sake of analysis; a single person often plays multiple roles.

The *end-users* actually use the environment interactively to explore algorithms in action. They use a specialized interpreter to control the execution of the available algorithms, and a specialized user interface to manipulate windows in which to view the execution via any of the available graphical displays. They can run the algorithms using a wide assortment of input generators and various instantiations of any particular input generator. Because there are many possible ways to connect algorithms, views, and inputs, end-users can generate displays never seen by the client-programmers who implemented the pieces.

End-users are called *script authors* and *script readers* when they use the system as an electronic medium for communication by creating and replaying scripts. In an educational setting, the script authors are often the instructors and the readers are the students. Often, a person coding a new algorithm or view will also prepare a script to illustrate properties of the algorithm or view. When an end-user is devising a new script or browsing through an existing script, there are additional commands that are not normally available, e.g., commands that control what type of information is saved or restored, command that traverse the dynamic document, and so on.

There are two types of client-programmers. The first type, *algorithmaticians*, take the actual algorithms being animated, often from a textbook, journal paper, or even existing applications, and augment them with markers indicating interesting phenomena that should give rise to some type of display. The algorithmaticians usually also implement the input generators and the adapters. In an educational setting, the course instructor and teaching assistants are usually the algorithmaticians. In a research environment, the researchers themselves are usually the algorithmaticians. This process must require only a nominal amount of effort in order for an algorithm animation system to be successful.

The second type of client-programmer, *animators*, design and implement the view code which actually displays algorithms in execution, i.e., renderers and modelers. They code in a high-level language, such as PASCAL or C, and use a library of graphical primitives provided through the algorithm animation system. Ideally, these people should have training in graphic design; in reality, however, these people are usually computer scientists without formal artistic training. They should have access to graphic designers with whom effective displays can be jointly developed. Achieving real-time animation often requires low-level system-dependent coding.

The central figure in any software environment is the *systems guru*, the person responsible for implementing, maintaining, enhancing and debugging the system. Systems gurus must be very competent systems programmers; they serve as the interface between the people who use the system interactively (end-users in general, and script authors and script readers in particular) and those who use the algorithm animation system as programmers (algorithmaticians and animators). As an algorithm animation system matures, the need for a systems guru diminishes.

1.4 Perspective on Graphics in Programming

Algorithm animation is a form of *program visualization*, "the use of the technology of interactive graphics and the crafts of graphic design, typography, animation, and cinematography to enhance the presentation and understanding of computer programs. Program visualization is related to but distinct from the discipline of *visual programming* which is the use of various two-dimensional or diagrammatic notations in the programming process" [8]. Visual programming also includes those programming-by-example, by-demonstration or by-constraints systems that use graphical objects as fundamental computational entities. A number of surveys about visual programming have appeared recently [20, 21, 32, 55]. Displays of the execution of visual languages used in visual programming can be easily confused with program visualization. More to the point, they *should* be entirely similar! Systems such as GARDEN [58] strive to unify these two disciplines; that approach, however, is not our concern here.

Program visualization systems can be classified by whether they illustrate code or data, and whether displays are dynamic or static [53]. In addition, dynamic displays are either interactive or passive, such as a videotape. Algorithm animation displays are dynamic displays showing a program's fundamental *operations*; operations embody both transformations and accesses to data and to a lesser extent, flow-of-control.

Typical static displays of program code are flowcharts, Nassi-Shneiderman diagrams, scoping diagrams, and module interconnections, as well as text itself when enhanced through formatting and typography. Numerous systems have been developed to display one or more such diagrams automatically from programs coded in high-level procedural languages, and to use the diagrams for editing the underlying program. Static displays of program structure can be animated automatically by highlighting the appropriate parts as the code runs.

Static displays of program data are more difficult to create automatically than static displays of code. One problem is that a given data structure can be implemented in many different ways, and a second problem is that a data structure has many different representations. Typically the more informative displays are not the canonical displays that can be created automatically, but are those discovered only through experimentation. Even canonical displays are difficult to construct for arbitrarily linked structures; they tend to look like rat's nests and lose meaning. This thesis presents no new layout algorithms for static displays of data structures. We assume that either such a package is available (Chapter 2 cites some packages) or the user is interested in a customized display to illustrate particular features of an algorithm.

Creating dynamic displays of program data has all of the problems that creating static displays has and more. In particular, dynamic displays must decide when the display should be updated and how this should be done incrementally to look effective. The precise definition of what constitutes an "effective" display is beyond our scope here. It is subjective, and involves many complex, interacting and competing factors, such as the viewer's visual vocabulary, the speed of the changes, the techniques used for highlighting, and so on.

Algorithm animation displays can be thought of a dynamic displays of an algorithm's operations, not merely its data or structure. We now explore the nature of these displays in depth.

Algorithm Animation Displays

Algorithm animation displays can be described using three dimensions, as shown in Figure 1.3. The *content* of the displays ranges from direct representations of the program's data to synthetic images information not necessarily inside the program. The *persistence* dimension ranges from displays that show only the current state of information to those that show a complete history of each change in the information. The *transformation* dimension ranges from displays that show changes in the pictures discretely to those that show incremental and continuous changes.

Readers are encouraged to pause at this point to scan through the screen images in Chapter 3. We will refer to those images in the remainder of this section to illustrate the various types of algorithm animation displays. In addition, Figures 1.4 and 1.5 show typical displays in the BALSA–I algorithm animation environment. Figure 1.4 shown First-Fit Binpacking operating on

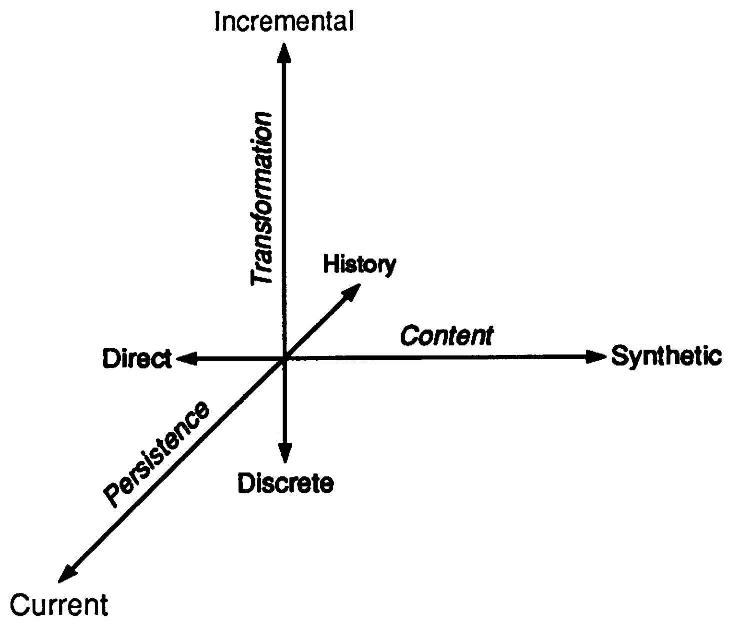
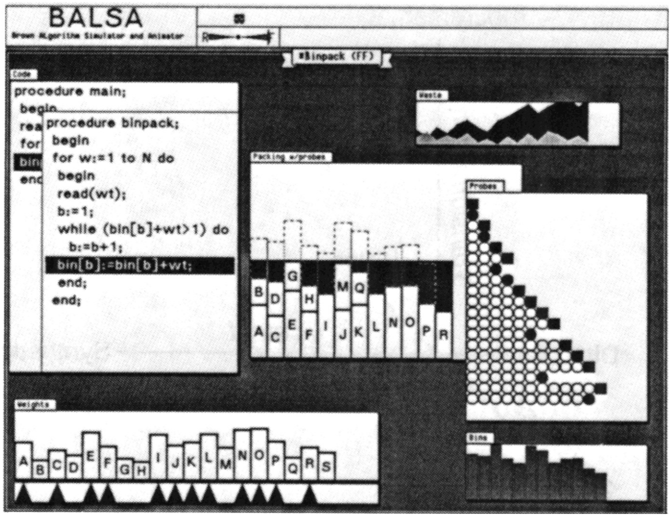


Figure 1.3: *Attributes of dynamic algorithm animation displays along three axes. Displays classified in the rear upper right corner (synthetic, history, and incremental) are usually the most intricate to implement, and those in the front lower left (direct, current, and discrete) the easiest.*

a relatively small amount of data; Figure 1.5 shows four different binpacking algorithms operating on a much larger amount of data.

First, we will look at the content axis. *Direct* displays are pictures that are isomorphic to one or more data structures in the program. At a given instant, the data structure(s) could be constructed from the display, and the display could be constructed from the data structure(s). No additional information is needed. For example, the Bins view (Figure 1.4) in a direct view of the array *bins*. The Dots view (Figure 3.3) is a direct view of the array of numbers being sorted.

Synthetic displays, on the other hand, do not have a mapping to any program variables. They can show the operations causing changes in the data, or can be abstractions of the data. The Waste view (Figures 1.4 and 1.5)



```
procedure main;
 begin
 rea
 for
 bin
 end
```

```
procedure binpack;
 begin
 for w:=1 to N do
  begin
  read(wt);
  b:=1;
  while (bin[b]+wt>1) do
    b:=b+1;
  bin[b]:=bin[b]+wt;
  end;
 end;
```

Figure 1.4: *First-fit binpacking algorithm.*

is a good example of a synthetic view: after each weight is inserted in the bin, the graphs are continued at the right edge to show how much space is wasted. The top graph shows the wasted space and the bottom graph shows the lower bound of this quantity. The concept of wasted space is not in the program. The Probes view (Figure 1.4) is also a synthetic view. Each row corresponds to searching for a bin in which to insert a new weight. A hollow icon indicates a bin did not have enough room for the weight, whereas a filled icon indicates that it did. A square icon indicates that the weight is the first one in a bin; the circle indicates a bin that has been started already.

Many displays are composites of direct and synthetic components. For example, the Compare-Exchange view (Figure 3.1) shows some values of the array; additional information (showing the results of comparisons or exchanges) is encoded as the color of the elements. A proper reading of the picture would enable the current contents of the array to be reconstructed, although this property is not invertible: the picture could not be reconstructed just by knowing the contents of the array at an instant of time. This view shows more than simply changes to the algorithms data structure; it shows both the fundamental operations the algorithm performs, and the

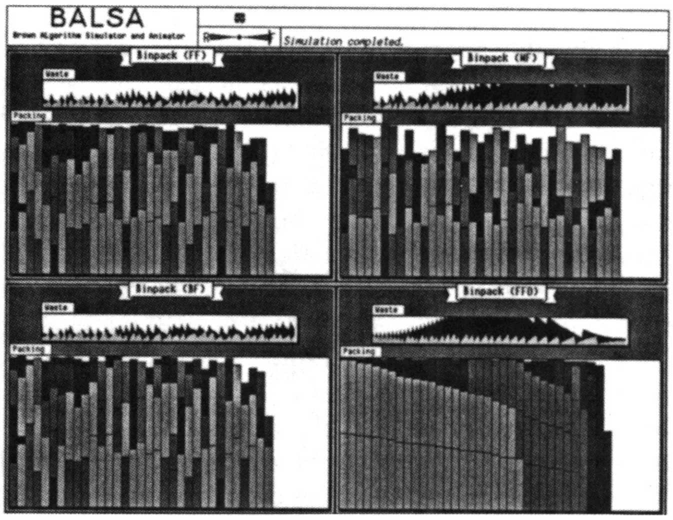


Figure 1.5: *Four binpacking algorithms.*

flow of control: each iteration of the algorithm's main loop is displayed on a separate line.

A second criterion for classifying displays is whether a display shows *current* information or illustrates a *history* of what has happened. In Figure 1.4, all views except for the Bins view shows some history. The Weights view is a history of each weight inserted into a bin, the variable *wt*. A triangle is drawn beneath a weight when that weight causes a new bin to be started; thus, it is a history of a synthetic entity. In the Waste view, only the right edge shows the current waste; the part of the graph leading up to the right edge is a history of the waste as each weight is processed. The Probes view shows a history of each and every bin that was tested to see whether or not it could support the new weight. Even the Code view shows history!

The third and final criterion by which we can classify animations is based on the nature of the transitions from the old displays to the new ones. *Incremental* transformations show a smooth transition. For example, in the Packing w/probes view in Figure 1.4, the dotted box showing the attempt to put a new weight into each of the bins, advances smoothly from one bin to the next, while also keeping a trace of itself. *Discrete* transitions are just that: the old value is erased and the new value is drawn. All of the views in Figure 1.4, with the above mentioned exception, are discrete transitions.

Discrete transformations are perceived as incremental when the difference between the new and old pictures is "small enough" in relationship to the complexity of the data. Genuinely discrete transformations tend to be most useful on large sets of data. Incremental transitions are most effective when users are examining an algorithm running on a small set of data; in fact, incremental transformations tend to hinder the display of large data because they slow down the animation significantly and contain too much low-level detail. Unless a good animation package is provided, incremental transitions are often tedious and difficult to program.

Another issue that incremental transformations must address is how much time they should consume. For example, changing a single pointer in a tree might cause a very large subtree to move a large distance (which might also necessitate repositioning all nodes in the tree). How fast should the subtree movement be? Should it be at the same speed as if it moved a small amount? Or should the total movement consume a constant amount of time, so that it moves quickly if it has a large distance to cover?

1.5 Automatic Algorithm Animations

What is it about algorithm animation that is more than simply monitoring a variable? That is, why could we not simply associate a procedure with a variable that would be called each time the variable is modified or accessed, with information about which part of it was modified or accessed? In particular, annotating the algorithm with interesting event markers and creating the specialized displays are time-consuming and error-prone activities. Can these activities be automated or eliminated?

Algorithm animation displays cannot be created automatically because they are essentially monitors of the algorithm's fundamental *operations*; an algorithm's operations cannot be deduced from an arbitrary algorithm automatically but must be denoted by a person with knowledge of the operations performed by the algorithm. In addition, there are problems relating to real-time *performance* and to *informative displays.* We shall return to these problems below.

Even if we assume that an algorithm's operations have been identified, automatically creating pictures to depict the operations is not currently feasible. Although graphic designers have worked on a semiology of display techniques for static information [12, 13, 67], except in a few highly specialized application domains, they have not had the tools to create dynamic displays and therefore have not looked at the issues involved in their semiology. Moreover, many data structures and algorithms are highly specialized and require one-of-a-kind displays to make aspects of their properties understandable. Some knowledge-based systems have been built that present meaningful pictures and even animations, but in restricted domains [25, 48]. More experience is needed with animated algorithm displays before we can hope to automate the task, though libraries of standard displays are possible.

Automatic Program Visualization Displays

We now examine in more detail the nature of program visualization displays that can be created automatically. Dynamic displays require two types of information: the *entity* to be displayed and a *delta* describing the change in the entity. Static displays need only information regarding the entity.

Dynamic and static displays of static or even executing code or program structures can be created automatically because the set of entities and deltas is well-defined and can be accessed directly by a display routine. The entities are derived from the source code, and the deltas from the changes in the program counter. Examples of code entities are procedures, statements, files, and blocks; examples of deltas are advancing to the next line of code, entering or exiting a procedure, and entering or exiting a block.

Dynamic and static displays of data can also be created automatically. The entities are the data structure(s) to be displayed, and the deltas can be inferred by examining the data each time it is accessed or modified. A routine to display a static picture of the data structure automatically would access the data through the runtime environment. It would need to be told how the data is represented in the program (i.e., a mapping into a canonical representation) and what type of display is desired (i.e., a display technique for the canonical representation). Although canonical displays can be created without modifying the algorithm, they are not always very informative, especially for non-trivial data structures.

Moreover, dynamic displays created automatically do not show the delta in the way the change is conceptualized. This problem is because only the "before" and "after" conditions of the data are known and the display can do nothing more than interpolate between the two states. Consequently, views that are direct views of data structures and are updated discretely can be created automatically, subject to the inherent problems with displaying data mentioned above, because they do not attempt to display the deltas.

Difficulties of Algorithm Animation Displays

We now examine in detail the problems with creating algorithm animation displays automatically.

Problem 1. Capturing operations. Algorithm operations do not necessarily correspond to each access or modification of the algorithm's data structures. In particular, (a) accessing a particular variable has different meanings at different locations in the algorithm, and (b) an arbitrary number of accesses and modifications (including zero) results in a single operation.

The following fragment of Quicksort illustrates the first of these problems:

```
if r - l ≤ M then
  for i := 2 to N do
    begin
    v := a[i]; j := i;
    while a[j - 1] > v do
      begin a[j] := a[j - 1]; j := j - 1; end;
    a[j] := v
    end
else
  begin
  v := a[r]; i := l - 1; j := r;
  repeat
    repeat i := i + 1 until a[i] ≥ v;
    repeat j := j - 1 until a[j] ≤ v;
    t := a[i]; a[i] := a[j]; a[j] := t;
  until j ≤ i;
  a[j] := a[i]; a[i] := a[r]; a[r] := t
  end
```

If the subfile being sorted (the elements between l and r) is small enough, Insertion sort is used to sort the subfile. Two fundamental operations are being performed on the array: "set value" (in the **then** part) and "exchange" (in the **else** part). A client-programmer would like to illustrate these operations differently.

A monitor cannot not know which accesses to array a constitute a "set value," and which an "exchange." Even if this problem were eliminated, say by re-coding the Insertion sort phase using exchanges, a second, more difficult, problem remains: how can a monitor infer which accesses comprise an exchange? It is the result of a variable number of modifications to the array; it is not the result of every other modification. Of course, one could always change the Quicksort code fragment to guarantee that every exchange is the result of exactly two modifications. We assert, however, that to do so would be just as difficult and error-prone—if not more so—than adding the algorithm event annotations that our model requires. Moreover, it would be counter to one of our primary goals: minimal intrusions into the original algorithm's original source code.

The second aspect of the problem, that of an arbitrary number of accesses and modifications resulting in a single operation, is seen in the Partition-Tree view in Figure 3.5. The shape of the tree is determined by the algorithm operation, "ElementInPlace." To simplify the discussion, assume that Insertion sort is not used for the small subfiles. That is, remove all code in Quicksort fragment above except the body of the **else** statement. After the modified fragment has been executed, the value stored in *a[i]* is finalized. This abstract operation cannot be inferred by simply monitoring variables: it is a conceptual operation that is essentially triggered by the control flow.

Another example of this second aspect of the problem is illustrated in the following fragment of code from an implementation of Pairing Heaps [29] (the pairing heap is stored using right-sibling and left-child links, with back pointers to each node's parent):

```
...
back[0] := 0;
if (info[x] < info[y])
  then begin rbro[x] := rbro[y]; back[rbro[y]] := x; end
  else begin
    back[y] := back[x];
    if (rbro[back[x]] = x)
      then rbro[back[x]] := y
      else lson[back[x]] := y;
    t := x; x := y; y := t;
    end;
rbro[y] := lson[x]; back[lson[x]] := y;
lson[x] := y; back[y] := x;
...
```

This rather complex sequence of pointer manipulations performs a single (and rather simple) "link" operation of subtrees *x* and *y*. Linking two subtrees involves making the subtree with the larger root node the leftmost son of the other subtree. A monitor would need to coordinate accesses and modifications to four arrays (*info*, *back*, *rbro*, and *lson*). Note that displaying the data at each operation would result in pictures that are misleading and would obscure the essence of the link operation.

Problem 2. Real-time performance. In general, the data being accessed or modified may be costly to identify, resulting in unacceptable performance. For example, knowing which node in a tree is being modified may lead to costly computation to determine its parent, siblings, and children. Just because a task is expensive does not mean that it is impossible; however, for the real-time interactive systems that we address in this thesis, performance is a real issue. One cannot just hope for faster hardware or clever display algorithms, since for every increase in hardware or algorithm speed, the complexity of the algorithms and the size of the data that one will wish to animate is bound to also increase.

Problem 3. Informative displays. Many displays necessitate detailed knowledge about the algorithm's runtime behavior and the specific data upon which the algorithm will be run. In fact, the optimal size and layout parameters for some displays require two passes. For example, because the Partition-Tree view knows that the height of the tree will be about equal to the binary logarithm of the number of elements in the corresponding array, it can allocate that much vertical space from the start. However, it cannot know the exact height until the algorithm runs.

Towards Automatic Algorithm Animation Displays

Algorithm operations must be identified by a programmer. Languages that support abstract data types are particularly well-suited to this approach. For example, in Smalltalk, entities and their deltas are defined by objects and the messages they react to. Given a properly modularized Smalltalk program, one just needs to specify how the objects and messages map into entities and deltas. This approach is not limited to Smalltalk, object-oriented programming languages, or data abstraction languages. In conventional programming languages, such as PASCAL or C, one could encapsulate the operations in procedure calls (there would be a one-to-one mapping between the procedure calls and the Smalltalk messages), which could then be monitored automatically.

It is tempting to believe that such a strategy is a panacea. However, algorithms from textbooks and journals are given in "straight-line" code; they are not broken into procedures. It is impossible to take straight-line code and to infer automatically the correct abstractions to form the encapsulations.

Consider, for instance, the fragment of Quicksort from above. After the algorithm operations have been encapsulated (by hand) into procedure calls, it becomes:

```
...
repeat
  repeat $i := i + 1$; until $Compare(a[i], v, `\geq')$;
  repeat $j := j - 1$; until $Compare(a[j], v, `\leq')$;
  $Exchange(a[i], a[j])$;
until $j \leq i$;
$Exchange(a[i], a[j])$; $Exchange(a[i], a[r])$;
$ElementInPlace(i)$;
...
```

The procedure *ElementInPlace* does nothing; it has been added strictly for animation purposes. The difficult issue of annotating an algorithm is one of identifying the phenomena of interest in the program; the appropriate syntax for enunciating the abstractions may or may not be directly supported in the implementation language.

The approach we have taken in BALSA–I and BALSA–II is to annotate algorithms with "events" rather than forcing the algorithmatician to radically proceduralize his algorithm to encapsulate each meaningful operation. This approach minimizes the changes to the algorithm, since the algorithm is augmented, not transformed. Of course, if an algorithmatician is willing to procedurize the algorithm, events can be inferred "automatically" by a rather simple preprocessor that inserts an annotation as the first statement of each procedure. The parameters of the event would be the name of the procedure, followed by the arguments of the procedure.

Events also help to solve the second and third problems mentioned above. Data that may be costly to identify automatically can often be identified by the algorithm and associated with the event. Other events can broadcast information of interest concerning the characteristics of the algorithms; displays can adjust parameters appropriately.

There are numerous additional pleasant side-effects of having annotations in the algorithm which we shall discuss briefly here and explore more fully in later chapters. The end-user can specify events for setting breakpoints, setting granularity of single stepping, and marking how much time each event should take to execute. The animator can debug a view independent of the algorithm, by feeding it a stream of events generated by hand (or even randomly). The algorithm need not be implemented in any specific language as long as it is callable by the algorithm animation system, nor are there any restrictions on the data structures used in the algorithm. The systems guru can give the end-user the illusion of an interpreter—one that is language-independent! Annotations give flow of control back to the algorithm animation system, which can then poll the end-user to see whether execution should be paused.

1.6 Disclaimers

This thesis does not address the issue of what makes for the most effective displays of operations, data, or code; we have noted some preliminary observations elsewhere [18]. Fortunately, the pictures that concern us do not need to be realistic images. They can have "jaggies" and do not need texture, shadows, reflections, or refractions. They exist to communicate information, not as objects of art—although many images, especially those involving color, are quite attractive. In fact, it is unlikely that one could ever claim that one particular display is the best. Each display highlights particular features of the program, and thus is more or less desirable depending on its intended use. Moreover, a given picture can mean several different things to each viewer, and the meaning will change depending on many factors, such as what other images are simultaneously begin displayed, how developed one's "visual vocabulary" is, and so on. Consequently, an algorithm animation system should not impose a rigid set of displays for programs; rather, it should make it easy to create new displays and use existing displays to explore the runtime nature of a wide variety of programs.

Another aspect we do not address here is how fast the animation should take place, or with what granularity. It is essential, however, that an algorithm animation system allow users to control the speed. Not surprisingly, it has been shown that information is lost if an animation is either too fast or too slow [49]. The optimum speed depends on the viewer and the purpose of the animation. In practice, we have found that, as one would expect, viewers can get a high-level intuition for the dynamics of an algorithm when relatively fast speeds are used. To understand details of the behavior, however, relatively slow speeds are required. The absolute speed also depends on the complete ensemble of views on the screen; complex views or multiple views often require slower speeds to let the user digest all of the information on the screen. Displays showing representations that are unfamiliar to viewers also require slow speeds—at least until the displays are incorporated into the viewer's visual vocabulary.

Our environment for algorithm animation is oriented to sequential programming in the small. It has been tuned for "algorithms" such as those found in a typical textbook or journal article. These are usually less than a page or two of high-level procedural code, and are self-contained with simplified data assumptions. For example, a priority queue "algorithm" might operate on small integers, whereas in practice the priority queue "program" might be embedded in a database system and would operate on names and manufacturers of automobiles. "Programming in the small" does not, by any means, imply trivial or toy programs. Experience has shown that relatively little is understood mathematically or otherwise about many small (in size), well-known, fundamental algorithms. There is a huge world of small programs to be explored, many of which form the basis of large real-world systems. While many aspects of an algorithm animation environment scale to large systems, however, programming in the large needs additional tools that are beyond the scope of this research.

1.7 Thesis Outline

In the next chapter, we review previous work on animation of programs. We limit ourselves to displays of data, not program structure or code, both static and dynamic. This chapter provides background reading and is self-contained. In Chapter 3, we present a tour through the interactive environment of our prototype system, BALSA–II. Here we discuss not how one goes about animating a program, but rather how one uses the environment for exploring algorithms. We also present a formal description of the interactive environment. If you read only one chapter, read this one. If you read more than one other chapter, also read this one; the remaining chapters assume familiarity with the interactive nature of the prototype. Chapter 4 discusses using the interactive environment for creating dynamic documents. It describes our model of dynamic documents and the user interface presented to script writers and script readers, and discusses various implementation aspects.

In Chapter 5, we leave the realm of the end-user and enter that of the programmer. We present in detail our model of how client-programmers go about animating their algorithms, implementing the necessary input generators, and building graphical displays. In Chapter 6, we present an overview of how BALSA–II is implemented. The system is the glue that binds the algorithms, input generators, and views, to the end-users, script writers, and script viewers. We conclude in Chapter 7 with a discussion of areas for future research, both short-term and long-term.