Animated Graphical Interfaces

using Temporal Constraints

by Robert A. Duisberg[1]

Computer Research Laboratory, Tektronix, Inc.

P.O.Box 500, MS 50-662, Beaverton, OR 97077

Abstract: Algorithm animation has an acknowledged and growing role in computer aided algorithm design, as well as in documentation and technology transfer,since the medium of interactive graphics is a broader, richer channel than text by which to communicate information. Since an animation constitutes the interface between a user and an algorithm, a kit that facilitates the construction of such has all the basic elements of a User Interface Management System. Constraint languages are useful in constructing such an interface construction kit, whereby consistency is maintained among the elements of a structure and among those of a view of that structure presented to the user. But constraints specify only static state in current implementations. To specify the evolution of structures and views by discrete time increments, as in animation, requires a extension to current constraint languages to allow expression of specifications of temporal behavior.

1. Motivations

Animation is a powerful means to visualize the inner workings of a process in action. Data structures are almost invariably represented by two dimensional diagrams in texts and documentation (e.g. in [Standish 80, Sedgewick 83]) with the clear implication that the concepts and structures are more easily comprehended and understood in this spatial, visual form, than, say, as a textual description or an algebraic specification. By extension, the often obscure algorithms operating on data structures should be more easily understood through dynamic motion in such diagrams and images, rather than through a linear string of pseudo-Algol commands. In keeping with the dictum that "software must be designed to optimize not the way the machines work, but the way that people think," [Birnbaum 85] we seek to present to a person dealing with a program appropriate dynamic imagery by which to understand the program and through which to interact with it.

[1]This research was supported in part by the NSF, grant number DCR82-02520, and in part by the Computer Research Laboratory, Tektronix, Inc.

Animation has many applications from algorithm design, documentation and debugging [London and Duisberg 85], to education [Brown and Sedgewick 84], skill development [Chi 84], and performance monitoring. Indeed animation may be considered an integral part of an interactive interface, with the emphasis shifted so that changes in the system are driven not only by user actions but by autonomous actions of the animated object itself.

Much of the appeal of animation is in being able to simulate the behavior of an experimental design and "see how it works." Surely this appeal is diminished if an animation is laborious to construct, which it is if done "by hand." Experience has shown that better than 80% of the code in an animation is involved in just doing the graphics, even in a system like Smalltalk which offers a powerful high-level graphics interface.

But the difficulties are further magnified if one insists, as we do, that an animation must be more than just a movie showing a process through passive bits on the screen. Rather, we strive to create the illusion that the pictures *are* the things they represent. This means that the animation must be designed as a fully interactive interface and that the user should be able to gain access, through their pictures, to the actively executing objects which drive the animation. Ideally he should be able to point to a part of the picture as the animation is in progress and effectively say, "What is that?", or "Do this to that over there." Clearly the overhead and mechanism to provide such interaction is much more involved than for just transfering bits to the screen.

2. Background

The application of computers to assist in creation of animated movies has a long history [Meltzer 69] with many recent advances driven by the commercial success of computer generated cinematic effects [Booth 83]. But the emphasis in much of this work is to create realistic, quasi-photographic images, which is beside the point in the task of algorithm animation where the images are of a schematic nature.

The color, sound film, *Sorting Out Sorting* by Ronald Baecker [Baecker 81] has served as an acknowledged motivation and inspiration for both the Brown University ALgorithm Simulation and Animation (BALSA) system [Brown and Sedgewick 84], and our work at Tektronix [London and Duisberg 85]. The emphasis of the BALSA project has been instructional, an effort to create "a dynamic book in an electronic classroom," whereas in the industrial environment animation is regarded as a design tool for prototype simula-

tion. The movie was quite costly to produce, and the building of new animations, even with BALSA's extensive library of routines, is arduous [Brown 85], and thus a need is generally felt for tools that can simplify the process.

Especially in applications to algorithm design and development, it is almost essential that it be relatively easy to construct new animations. Since the algorithm is the principal item of interest, it is unacceptable if much more effort must be spent bringing up an animation than the algorithm itself that the animation is supposed to test or provide insights about. This has led to the notion of building an animation kit to allow the construction of new animations by assembling and connecting components with graphical and temporal behavior built into them. This is the essential motivation for the creation of Animus, which is intended to be a system in which the construction of a new animation is facilitated by providing the animator with a kit of composable parts. The fundamental problem for the system is to organize its responses in view of the interactions of the parts so composed.

One may think of animation as the creation of a picture whose state is constrained to bear some specified relation to the state of the object being animated. The internal state of the object should be self-consistent in keeping with the real-world laws governing the thing being simulated, and the picture of the object should consistently show changing state, according to the rules of some representational scheme.

The requirement of consistency of state and representation suggests the use of constraints to implement an animation kit, since constraint languages allow the programmer to state relations that the system will be guaranteed to satisfy.

A number of constraint systems have proven their utility for specifying and maintaining consistency in structures [Borning 79, Gosling 83, Leler 85, Steele 80, Sutherland 63]. Techniques used by these systems to insure constraint satisfaction range from propagation of known states and degrees of freedom to algebraic manipulation and relaxation, in numerous variations and combinations including rewrite rules, both graphic and textual, alternate redundant views, user query and retraction. Clearly constraint satisfaction is a hard problem in general. Fortunately many useful applications are not as hard as the general problem, and in designing a constraint based animation in particular the kinds of constraint networks that appear to arise have the desirable features of low branching factor and a general absence of loops. (For example a constraint that a view be consistent with the thing it represents may always be satisfied by updating the view which rarely in turn would cause a change back in the model.) ThingLab was chosen as the base system upon which to build a constraint-based animation kit for a number of reasons, not the least of which is the excellent performance afforded by its incremental compilation feature; constraint networks tend to be solved just once at "compile time" when the system composes new methods to respond consistently to changes and compiles these into the structure. Thus after an initial pause, execution proceeds swiftly.

3. Constraints Involving Time

None of the constraint systems mentioned above explicitly models time evolution or allows expression of constraints on how an object evolves in time. This is not to say that these systems are not dynamic or interactive. But the animation-like dynamic responses of a system like ThingLab are user driven in response to the demand, "Satisfy constraints with these values now." There can be no history-dependent specifications; there is no history.[2]

Time requires special treatment and cannot simply be inserted as another variable in a constraint relation. For one thing, time is a distinguished variable in that it may be thought of as driving the animation, and the user expects it to behave reasonably, advancing steadily and monotonically. Furthermore one must deal with the disjunction between the inherently discrete frame by frame, time slice character of computation, and the kinds of continuum statements one would like to make in describing rates of change, e.g. $v = dx/dt$. Consider for example the problems that arise from a straight-forward expansion of this relation into a set of methods a constraint might use to satisfy itself. This might yield:

$$v \leftarrow (x - \text{old } x) / (t - \text{old } t).$$
$$x \leftarrow v * (t - \text{old } t) + \text{old } x.$$
$$t \leftarrow ((x - \text{old } x / v) + \text{old } t.$$

where the temporary variable 'old' is bound to a copy of the moving object before the update. Several difficulties become apparent. First there is an inherent circularity in the constraint network so described, since the value of the velocity v depends on the value of the position x and vice versa. Thus the constraint satisfier can only construct a method which employs costly relaxation. The third equation above is also problematic for initial conditions; if the velocity happens to be initially zero, which it usually is, this leads to a division by zero. This method also causes the bizarre behavior of setting the clock back to a negative number in response to a user request to adjust the initial position of the object. Whereas the first equation above is a reasonable approximation to $v = dx/dt$ its subsequent inversions are problematic; d/dt is an *operator* on $x(t)$ and the differentials, dx and dt may not properly be manipulated as separate algebraic entities as implied in inverting the approximation.

Clearly the kind of equations that the system should generate would reflect the approximation that one must make in going from the continuous to the discrete representation, that is, something of the form

$$x \leftarrow \text{old } v * (t - \text{old } t) + \text{old } x$$

where the approximation is embodied in the use of the old value of the velocity as though it were valid throughout the time interval. Essentially the stated constraint should *not* be satisfied during the transitions between consistent states and the constraint satisfier must be held in abeyance. It is suggestive that we refer to the velocity as the *derivative* of the position, in the sense that it is a derived, secondary value and so it is appropriate to essentially suspend the constraint on the velocity's value for the duration of the time interval and only alter its value later to agree with other constraints after the first approximation to the new state has been obtained.

A sort of inverse of the problem described above can arise in the case in which the evolution is explicitly discrete, as in

[2] Steele's system keeps track of the order in which constraints have been satisfied in the history of the session, but this only as an aid in retracting variable bindings when a contradiction arises. It does not reflect the history of the object from consistent state to consistent state.

an animation of a finite state process like most algorithms. In this case it is possible to discretely update the display with each significant change of state (InterestingEvent). This is the approach taken in the Brown BALSA animations and it is appropriate in some situations (e.g. highlighting a line of currently executing code), but it has the disadvantage of "flashing" from one state to the next. This can be both annoying visually and downright confusing to a viewer who is trying to understand what is going on and who cannot see how things "got from there to here."

Thus one would often like to represent evolution that is explicitly discrete with as much visual smoothness and continuity as possible. Intermediate graphical states for display must be interpolated which in fact have nothing to do with any actual state of the thing itself. This led to the use of smoothly moving "data lozenges" to display data motion and pointers in many animations described in [London and Duisberg 85]. In the current implementation, the animation kit provides a class Response whose sublasses are such things as Trajectories, Flashers, Sliders, etc. These classes represent abstractions or prototypes of responses, which, when instantiated, generate streams of explicitly time-stamped events corresponding to the intermediate states for smooth motion.

4. Implementation

The animation kit, Animus, is being implemented on a Tektronix 4404 artificial intelligence workstation, in Smalltalk, as an extension to ThingLab [Borning 79]. At the heart of the Animus implementation of time constraints and responses is a Simula-style event queue [Birtwistle et al. 79] where an event is essentially a time-stamped Smalltalk message. A message in Smalltalk semantics consists of an object designated the receiver, a symbol serving as the message selector, followed by an array of arguments. The time-stamp serves to defer the action until the simulation clock has a value greater than or equal to the time-stamp.

Since events are explicitly time-stamped, the management of all events and responses in one time-ordered queue obviates the need for synchronization primitives. An arbitrary degree of concurrency may be implicitly handled by the queue since any number of events may be stamped to occur at the same time. In particular, the interleaving of complex responses consisting of sequences of events occurs "for free." This was impossible with techniques used heretofore in animating programs in Smalltalk [London and Duisberg 85] due to features inherent in the Smalltalk Model-View-Controller paradigm (MVC). In MVC, if a number of independent views are open on a single object (e.g. the value of a single index is to be shown separately in the animation as a pointer to a position in a string, a pointer into a row of an array and as a number in a little window), then the object may only broadcast the fact that it has changed, and the system then goes through the list of dependent views and updates them one at a time. Thus what is in fact one event has the appearance of a sequence of many.

In Animus interleaving of simultaneous responses occurs automatically since the times at which responses are instantiated is given explicitly. If two responses, such as "Trajectories" of icons moving across the screen, consisting of series of single-step events are specified to begin at specific times with durations such that they overlap, the constituent events will be inserted time-ordered in the queue and executed in that order, thus interleaved.

As in ThingLab, constraints are expressed locally in the components which are to exhibit the specified behavior, which are then combined so that the constraints interact in a network. But unlike ThingLab, where constraints are satisfied locally, in Animus, time is considered global and the single clock is managed by a single object, the anima, which is an ancestor to all parts of the animated structure. Therefore at the time of initialization the anima must traverse the animated object causing all temporal constraints to migrate up to the anima, which must "own" them since time is an instance variable of the anima. Thus any message that would increment time becomes interpreted considering the constraints on time and a new method is compiled to respond to that message which in addition to incrementing time guarantees that the constraints are satisfied.

A prerequisite to the implementation of temporal constraints is a language by which temporal relations can be specified. A simple equational language is not adequate since temporal relations may involve complex responses, delays, causality, etc. (Time differential constraints, expressible simply as, say, $v = dx/dt$, will be seen as a special case below.) Of languages developed for temporal specification, the most attractive are purely functional with an event-driven data-flow mode of execution, since a functional style maps readily into constraint methodologies. Dannenberg has developed a language for real-time control [Dannenberg 84] motivated by the observation that for "programs that sample inputs, perform operations and produce sampled outputs, the appropriate model of computation is a data-flow graph rather than a collection of processes." In his Arctic language all variables are implicit functions of time and thus concurrently varying values are handled succinctly, obviating the need for explicit synchronization primitives. Expressions in Arctic are "prototypes" which become instantiated in time (at a particular offset and scaling in time) in response to "events." Sequential and concurrent instantiation is provided for by special operators in prototype expressions. In a similar vein, CSP [Hoare 85] provides functional expressions to describe processes, and the instantiation of processes so described is "guarded" by events. In either case instantiation of a response or a process effectively amounts to the creation of a new stream of events.

Following this approach a temporal constraint may be viewed most generally as a relation that is required to hold between the existence of a stimulus event and a response in the form of a stream of new events. The statement of the constraint must include a description of the stimulus event and some abstract response prototype from which the response stream can be generated, perhaps differently depending on computational circumstances. A time differential constraint is a special case in which the stimulus is an increment of the clock and the response is a single event scheduled in the *next* major clock cycle to increment the differential variable by the specified amount (e.g. $v * \Delta t$). The fact that the response occurs at a subsequent time breaks the circularity (x depends on v depends on x. . .) and builds causality into the system in that responses follow stimuli.

As an illustration one may build dynamic circuit simulations from components such as capacitors and inductors with explicit time dependent behavior. In Figure 4-1 for example the capacitor owns the constraint $i = dq/dt$, and the inductor owns the constraint $\Delta V/L = di/dt$. Likewise the sweep of the oscilloscope trace is constrained to track with time modulo the width of the scope.

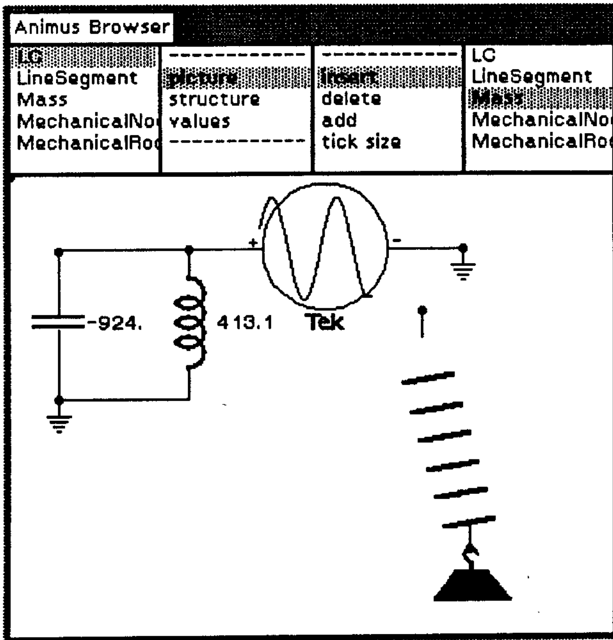

Figure 4-1:

Oscillating animations. The numbers indicate the charge on the capacitor, and the current through the inductor. The mass on the spring oscillates in two dimensions.

When the clock is sent the message "incrementby: dt," this becomes compiled into a method which enqueues two events, one which adjusts the charge on the capacitor and one which adjusts the current in the inductor. When these events occur (i.e. the messages are sent) other constraints fire off which adjust the voltage on the capacitor ($V = q/C$), propagate voltages and currents along the wires, update the text, etc. The sinusoidal behavior seen on the scope thus occurs automatically as though naturally; no sines are calculated! Rather the system has automatically generated code that implements a standard finite-difference approximation for the simulation. Note that since the increment to the charge calculated at $t = t_o$, namely $\Delta q = i * \Delta t$, is assigned at $t = t_o + \Delta t$, this effectively performs the desired approximation described above; the "derivative" value, the current, is considered good for the duration of the time interval in order to get a first approximation to the new primary variable, the charge, after which other values are adjusted according to the constraints for static consistency. The mechanical oscillator performs analogously, except that the constraints, $v = dx/dt$ and $F = m * dv/dt$ are satisfied for vector quantities.

More generally than time differential constraints, the system implements triggered constraints which are like guarded commands. The constraint is owned by a particular object and with a particular selector as the trigger, so that when the object receives a message with that selector the response specified in the constraint is incorporated into the object's normal response to the message. For purposes of animation of algorithms this allows greater specificity than the Active Values of LOOPS [Stefik 83] where *any* access to a variable triggers a daemon, and is less intrusive and more transparent than broadcasting Interesting Events [Brown and Sedgewick 84, London and Duisberg 85] where explicit broadcast statements must be inserted in the algorithmic code to notify the display routines.

Triggered constraints are naturally applied in animating discrete processes such as algorithms, where processes are like streams of events (the *trace* of a process in CSP terms). Consider for example a standard selection sorting animation as in Figure 4-2. Using triggered constraints one may effectively animate the sort using the unanimated sorting code *absolutely unchanged*.

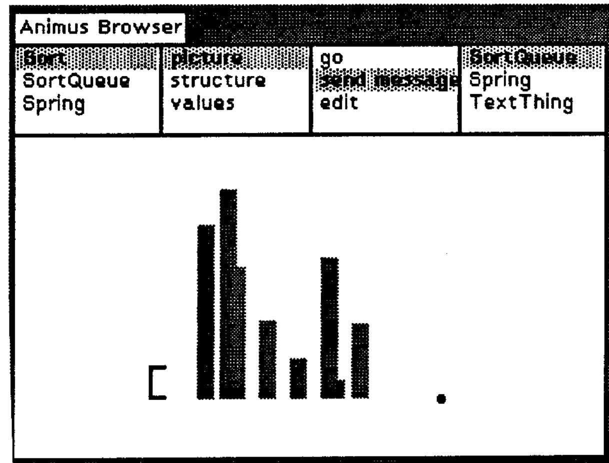

Figure 4-2:

An animation of a selection sort caught in mid-swap.

One writes a trigger constraint owned by the list being sorted so that when the list receives the trigger event, which is a message of the form "swap: i with: least", the specified response is instantiated, in this case two Trajectory objects which show the responses they represent by placing a sequence of graphical events, icon moves, into the eventQueue. Notice that standard animation techniques (e.g. subroutine calls) typically show the swap as a pair of sequential moves [London and Duisberg 85] but the technique described here shows the moves simultaneously, due to the automatic interleaving in the eventQueue described above, giving the right *conceptual* appearance of the swap as a single event (swap:with: is a single method selector). In order to animate the comparisons as well as the swaps one need only set a trigger constraint on the selector "compare:with:" with a response such as flashing the bars as they are compared.

5. An Industrial Strength Example

In the case of a simple sorting algorithm it may not seem significant that one should be able to animate unaltered algorithmic code by stating trigger constraints externally. However, the technique takes on much greater importance when the system or algorithm being animated grows to be large. Then this style may provide considerable reduction in code size since a single declaration of the trigger constraint may serve to bug many uses of the trigger selector. Also in large applications the animator and algorithm designer are probably two different people, and it is well if their jobs are clearly separable. (Even if the algorithm author is also the animator, these two roles should nevertheless still remain separate.) In particular, the animator should not be required

to develop a deep understanding of the algorithm in order to animate it, but rather should be able to take large chunks of code directly from the algorithm's author, with some decisions about what occurrences are of interest, that is, what selectors should be bugged, and these chunks of code incorporated wholesale into the animation system.

This approach has been successfully applied to aid in the design and "tuning" of a real-time operating system before it goes into a Tektronix instrument under development. The system implementor, Kenneth Dickey, has made animation his "method of choice" to test the operating system design, and to experiment with the effects of altering time slice size, process priorities and programs, etc. since he can *see* bottlenecks and process starvation much more readily than he can infer them by studying execution traces and statistics. This author, as the animator, was simply given some 12 Kbytes of simulated operating system code, which, using the techniques described here, he successfully animated with, fortunately, very few changes indeed to that code.

An important feature of the animation interface, essential for the support of the experimental, exploratory aspects of the design process, is the ability to edit and inspect objects on the fly. For example, selecting the item "edit" in the third pane of the browser and pointing to the icon representing a process in the operating system causes a new "inspector" window to open displaying the internal state of the process. Each process's program, priority, program counter, name, position, etc. can be edited and execution immediately resumed. Such a capability demonstrates how an animation

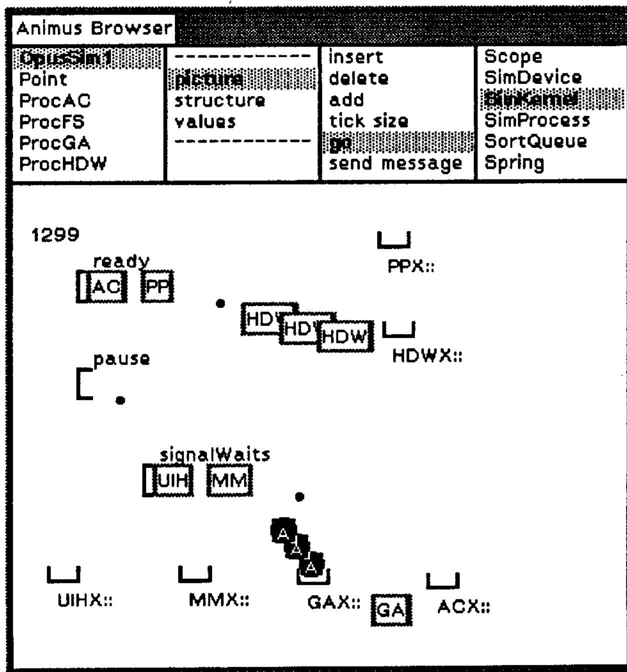

Figure 5-1:

An operating system animation. A Hardware process is proceeding from the ready list, to wait on its message exchange, and a message is being sent to the process GA (General Accounting)

can provide a flexible, interactive interface to processes in action.

Further directions for this research, being pursued in this author's dissertation in preparation [Duisberg 86], involve identification of a useful set of animation constraints, development of a library of modular animation components, and numerous interface issues concerning how the user may express the relations he desires to see which are tantamount to language issues.

6. Acknowledgements

Much of this work would have been impossible without the foundations provided by ThingLab, and many thanks are due to Alan Borning for his guidance and encouragement in helping to bring up this extension. I am also grateful for the support, interest and encouragement of the Smalltalk community at the Tektronix Computer Research Labs, in particular, Ralph London, Ward Cunningham and Roxie Rochat.

References

[Baecker 81] Baecker, R.
Sorting Out Sorting.
16 mm color sound film, 25 minutes, 1981.

[Birnbaum 85] Birnbaum, Joel S.
Toward the Domestication of Microelectronics.
Communications of the ACM 28(11):1225-1235, November, 1985.

[Birtwistle et al. 79]
Birtwistle, G.M., Dahl, O., Myhrhaug, B., Nygaard, K.
Simula Begin.
Van Nostrand Reinhold, N.Y., 1979.

[Booth 83] Booth, K.S. and Kochanek, D.H.
Computers Animate Films and Video.
IEEE Spectrum 20(2):44-51, Feb, 1983.

[Borning 79] Borning, A.H.
ThingLab -- A Constraint-Oriented Simulation Laboratory.
PhD thesis, Stanford, March, 1979.
A revised version is available as Xerox Palo Alto Research Center Report SSL-79-3 (July 1979).

[Brown 85] Brown, Marc H.
personal communication.
Tektronix , Feb. 1985.

[Brown and Sedgewick 84]
Brown, M. H., and Sedgewick, R.
A System for Algorithm Animation.
Computer Graphics 18(3):177-186, July 1984.

[Chi 84] Chi, U. H.
Formal Specification of User Interfaces.
Technical Report 84-05-01, Computer Science Dept., University of Washington, 1984.

[Dannenberg 84]Dannenberg, Roger B.
Arctic: A Functional Language for Real-Time Control.
In *Symposium on LISP and Functional Programming,* pages 96-103. ACM, 1984.

[Duisberg 86] Duisberg, R.A.
The Design and Implementation of Animus: A Constraint-Based Animation Kit (working title).
PhD thesis, University of Washington, in preparation, 1986.

[Gosling 83] Gosling, J.
Algebraic Constraints.
PhD thesis, Carnegie-Mellon University, May, 1983.
Available as CMU Computer Science Department tech report CMU-CS-83-132.

[Hoare 85] Hoare, C. A. R.
Communicating Sequential Processes.
Prentice-Hall International, London, 1985.
ISBN 0-13-153271-5.

[Leler 85] Leler, Wm.
Bertrand, A General Purpose Constraint Language.
Technical Report Draft, Computer Research Laboratory, Tektronix, Inc., Feb, 1985.

[London and Duisberg 85]
London, R. L., and Duisberg, R. A.
Animating Programs Using Smalltalk.
IEEE Computer 18(8):61-71, Aug, 1985.

[Meltzer 69] Meltzer, J.
Computer Animation: A Literature Survey.
Technical Report 403-8, NYU Dept. of EE, October, 1969.

[Sedgewick 83] Sedgewick, Robert.
Algorithms Book.
Addison-Wesley, 1983.

[Standish 80] Standish, Thomas A.
Data Structure Techniques.
Addison-Wesley, 1980.

[Steele 80] Steele, G.L.
The Definition and Implementation of a Computer Programming Language Based on Constraints.
PhD thesis, MIT, August, 1980.
Available as MIT-AI TR 595, August 1980.

[Stefik 83] Stefik, M., Bobrow, D.G., Mittal, S., and Conway, L.
Knowledge Programming in LOOPS: Report on an Experimental Course.
AI Magazine 4(3):3-13, 1983.

[Sutherland 63] Sutherland, I.
Sketchpad: A Man-Machine Graphical Communication System.
PhD thesis, MIT, 1963.