A Temporal Scripting Language for Object-Oriented Animation

E. Fiume[1]
D.Tsichritzis
L. Dami

Centre Universitaire d'Informatique
Université de Genève
12 rue du Lac
CH-1207, Genève, Switzerland

Abstract

Object orientation and concurrency are inherent to computer animation. Since the pieces of an animation can come from various media such as computer-generated imagery, video, and sound, the case for object orientation is all the stronger. However, languages for expressing the temporal co-ordination of animated objects have been slow in coming. We present such a language in this paper. Since the movements that an animated object can perform are also encapsulated as objects in our system, the scripting language can also be used to specify motion co-ordination. Such "motion objects" can be applied to any animated object. The syntax, semantics, and implementation of this language will be described, and we shall show how to specify device-independent computer animation.

[1]The financial assistance of an FNRS grant from the Swiss Federal Government is gratefully acknowledged. The first author also wishes to acknowledge the financial support of the Natural Sciences and Engineering Research Council of Canada. Address of first author as of 1 September, 1987: Department of Computer Science, University of Toronto, 10 King's College Road, Toronto, M5S 1A4, Canada.

1 Introduction

The term "object" has become a very popular buzzword. Interpretations vary as to what precisely constitutes an object-oriented system. We view an object as being an encapsulation of activities and data. Its basic properties are inherited from a prototype, of which the object is an instance. An object executes independently of other objects, and communicates with them according to a (message-passing) protocol. Rather than viewing an application as a single large piece of code, the object-oriented approach favours viewing it as a set of communicating capsules of activity, perhaps executing concurrently. The general belief, with which we concur, is that this approach facilitates structuring a system in a manner that clearly reflects its conceptual design. Applications from several areas of computer science, including database systems, office information systems, and simulation, have been successfully modelled by object-oriented approaches. Surprisingly, computer graphics has yielded comparatively slowly to object orientation. In this paper, we shall show how one application, computer animation, benefits from object orientation. In particular, we outline an environment in which animated objects are fashioned into complex animations using a concise, directly-executable specification language. The uniform encapsulation as objects of activities in animation allows one to extend both the expressive range of the language, and the repertoire of output media on which animation can be depicted.

Computer animation is inherently object-oriented. This view is not new [Reyn82; Berg83; MaTF85]. Reynolds' well-known ASAS language, which is an extension of a Lisp-based Actor system, is essentially object-oriented [Reyn82]. Moreover, languages such as BGRAF2 have been proposed which attempt to capture the temporal aspects of computer animation [BeKa76,77]. In BGRAF2, time is a quantity that can be directly sampled, and can thus be used to drive an animation. In this manner, it is possible to separate the static representation of a graphic object from how it is animated. However, it can be difficult to comprehend the global temporal behaviour of a system from many local views of time, or from a set of asynchronous events. Our view is to specify a global temporal behaviour, and to constrain local temporal behaviour to meet the specification.

To construct an animation in our system, an animator begins with a library of animated objects. Each object has its own autonomous spatiotemporal be-

haviour. One can then globally co-ordinate several such activities by means of a temporal scripting language. A parser/scheduler interprets temporal expressions and, according to their semantics, causes the animated objects to generate a sequence of graphics commands which satisfy the specification. This approach differs from others in that it is primarily concerned with global temporal behaviour, it treats animated objects and motion uniformly, thereby making it easy to reuse objects, and in that the language has a well-defined semantics, allowing one to do temporal reasoning about a system. It has the additional benefit of being extensible to specifying the co-ordination of animation encapsulated by different media. Our strategy in developing such a language is motivated by several potential applications:

- To specify (and transmit) device-independent computer animation. Since objects are active, they can customise their behaviour to the specific environment in which they are to be viewed. The notion of "information hiding" is helpful for separating essential temporal aspects of object activities from device-specific aspects of their representation or display.

- To specify the co-ordination of both animated and sonic objects into so-called *multimedia electronic messages* [FiTs87]. We wish to be able to construct easily electronic messages which are composed of computer animation, sound, and video segments.

- To understand the issues underlying the specification and enforcement of real-time constraints in general object-oriented applications. This problem is extremely difficult, and we hope that working on a nontrivial subproblem such as computer animation will help us to gain some insight into the general problem.

- To depict the operation of concurrent systems. In the object-oriented environments we are constructing, an application may be composed of many objects which execute concurrently. [Nier85; TFGN87]. While we believe this will result in extremely powerful modelling tools, it can also be difficult to understand the behaviour of applications constructed in this manner. Consequently, it will be helpful to have the ability to animate the behaviour of objects [Baec81; FoMa86; Myer83].

To accommodate these applications, we have developed an object-oriented animation testbed. The testbed contains two major software components: an animated object-creation environment, and a scripting environment.

The object-creation environment provides a means of interactively defining new animated objects and their motion. Once defined, animated objects are placed in a library for use as basic elements for constructing computer animations. The library embodies an open-ended store of object prototypes that can be instantiated as desired within a script, in the same way as the data type **int** (or **integer**) can be instantiated within a C (or PASCAL) program. Moreover, the motion of objects can be defined separately and also placed in a library. Such basic motions are then available to animate any graphic object. We are currently developing a keyframe animation module for this environment [KoBa84; Reev81]. Moreover, we hope to extend this environment to objects which produce sound rather than pictures. The use of an object-oriented approach is important for being able to deal with dynamic activities which can operate over a diverse set of output media.

The focus of this paper is on the scripting environment, in which a complex animation may be fashioned, using temporal operators, from a set of animated objects. Many animated objects can execute concurrently. We have built a parser for the language in which expressions are directly executable. This facilitates quick prototyping and editing. Moreover, animation can be triggered by the execution of other objects. Thus, for example, objects will be capable of depicting their behaviour graphically. Motion specification is also encapsulated in terms of objects. The scripting language can therefore be used to co-ordinate complex motion, as well as co-ordinating object execution. We find that this homogeneity facilitates concise specifications, re-usability, and open-ended design. Since the language is based on a formal syntax and semantics, it provides us with the ability to specify, experiment with, and evaluate new temporal connectives. We shall discuss the current syntax, semantics, and implementation of our language, as well as presenting several examples.

2 Language Description

An expression in our scripting language states a temporal relationship among instances of animated objects. An *animated object* is simply an active entity which knows its duration in abstract units of time called *ticks*, and knows its behaviour from tick to tick. It can report this information when a *message* is sent to it. Our animated objects are actually defined over a continuous time model. Consequently, when asked for its behaviour at a specific time, an object may interpret the request as "what are you doing now?", or it may interpret it as "what have you done since I last called?". Moreover, a continuous internal time model makes it easy to scale the speed of an object's activities. In general, when asked for its behaviour for a particular time, an object produces a set of graphic commands resulting from internally sampling its temporal behaviour. Ultimately, an animation scheduler will "tick" regularly every 1/30 seconds. However, as will be seen, objects can be sampled more or less frequently.

More formally, an animated object O responds to the messages "$O.t$" and *"$O.duration$"*, defined as follows.[2]

$$O.duration \in \Re \cup \{\infty\}$$

$$O.t = \begin{cases} \{\text{graphics commands for time } t\} & \text{if } t \in [0, O.duration) \\ & \text{otherwise} \end{cases}$$

The duration of an object can be infinite. We shall now define the semantics of temporal operators in terms of defining new animated objects. That is, if $\odot$ is a temporal operator, and E_1 and E_2 are scripting expressions (i.e. are themselves objects), then $E_1 \odot E_2$ specifies an object whose semantics is synthesised from that of E_1, E_2, and $\odot$. Throughout the following discussion, let E_1 and E_2 be arbitrary, well-formed expressions in the scripting language. We shall occasionally abbreviate "$.duration$" by "$.d$".

Chronological Sequencing.
The statement $E_1; E_2$ expresses the well-known idea that E_1 is *followed by* E_2. Formally,

$$(E_1; E_2).duration = E_1.duration + E_2.duration$$

[2]We plan to generalise the system to a full message-passing model. At present these are the only two message types permitted in the system.

$$(E_1; E_2).t = \begin{cases} E_1.t & \text{if } t \in [0, E_1.duration) \\ E_2.(t - E_1.duration) & \text{otherwise} \end{cases}$$

Observe that $E_2.t =$ for any $t \notin [0, E_2.duration)$, so it is not necessary to define explicitly the semantics of $E_1; E_2$ outside the range $t \in [0, E_1.duration + E_2.duration)$. If either expression denotes an animation of infinite duration, then the standard rules apply:

$$i + \infty = \infty + i = \infty + \infty = \infty.$$

It is easy to show that ";" is associative. That is, $E_1; (E_2; E_3) \equiv (E_1; E_2); E_3$. However, ";" is not commutative.

Simultaneous Activation.
The expression $E_1 \& E_2$ denotes the intuitive notion that E_1 and E_2 are to *commence simultaneously*. Formally,

$$(E_1 \& E_2).duration = \max\{E_1.duration, E_2.duration\}$$

$$(E_1 \& E_2).t = E_1.t \cup E_2.t.$$

It is easily seen that "&" is associative and commutative.

Simultaneous Termination
A surprisingly useful operator is the converse of "&": $E_1 \wr E_2$ indicates that E_1 and E_2 must *terminate simultaneously*. Its semantics is slightly more complicated than simultaneous activation:

$$(E_1 \wr E_2).d = \max\{E_1.d, E_2.d\}$$

$$(E_1 \wr E_2).t = \begin{cases} E_1.t \cup E_2.(t - (E_1.d - E_2.d)) & \text{if } E_1.d \geq E_2.d \\ E_2.t \cup E_1.(t - (E_2.d - E_1.d)) & \text{otherwise} \end{cases}$$

As with simultaneous activation, "$\wr$" is associative and commutative.

Delayed Activation.
It is sometimes desirable to specify a pause in an animation or to delay the activity of one object relative to another. The expression **delay** n specifies a object with the following semantics.

$$(\textbf{delay } n).duration = n$$

$$\forall t \in \Re.(\textbf{delay } n).t =$$

The value of a **delay** can be an arbitrary arithmetic expression.

Some Simple Examples.

1. "A & (**delay** 5; B)" causes B to be activated 5 ticks after A.
2. "A ≀ (B; **delay** 5)" causes B to terminate 5 ticks before A.
3. "A & **delay** 5 & B" is equivalent to "A & B" if A and B both have duration greater than 5.
4. if A.d = B.d then A≀B ≡ A&B.
5. **delay** n; (A & B) ≡ (**delay** n; A) & (**delay** n; B).

A General Synchronisation Operator
All binary temporal synchronisation operators can be expressed in terms of a general synchronisation operator. It is very useful for specifying precise synchronisation conditions, and it is important to our implementation, since it allows us to reduce all temporal expressions to expressions involving just this operator. However, in general, the above operators have a simpler, more intuitive semantics. The general synchronisation operator is of the following form: $A[t_a] \asymp B[t_b]$. This expression states that A's notion of the time t_a is to be synchronised with B's notion of time t_b. Its semantics is somewhat complicated.

$$(E_1[t_1] \asymp E_2[t_2]).d = \max\{E_1.d, E_2.d, E_1.d + t_2 - t_1, E_2.d + t_1 - t_2\}$$

$$(E_1[t_1] \asymp E_2[t_2]).t = \begin{cases} E_2.t \cup E_1.(t - \Delta t)) & \text{if } t_1 \leq t_2 \\ E_1.t \cup E_2.(t - \Delta t)) & \text{otherwise} \end{cases}$$

Observe that t_1 and t_2 need not specify times during which E_1 or E_2 are active. For example, if A and B are objects each with a duration of 100 ticks, then $A[5000] \asymp B[200]$ specifies that A is to begin 4800 ticks before B.

Arbitrary arithmetic expressions may appear within "[ ]". The symbol "$" stands for the duration of the expression to the left of the brackets. For example,

$$(E_1[\$/2] \asymp E_2[\$/2])[\$/4] \asymp E_3[0]$$

states that the halfway points of animations E_1 and E_2 must coincide, one-quarter of the way into which the animation E_3 must begin.

Equivalences.
We stated above that the general synchronisation operator can be used to rewrite other binary temporal operators. The reader may wish to verify the following propositions:

1. $E_1; E_2 \equiv E_2[0] \asymp E_1[\$]$.
2. $E_1 \& E_2 \equiv E_1[0] \asymp E_2[0]$.
3. $E_1 \wr E_2 \equiv E_1[\$] \asymp E_2[\$]$.

The general operator is commutative but not associative (because expressions of the form $(A[t_a] \asymp B[t_b]) \asymp C[t_c]$ are not defined–that is, a bracketed expression should itself have a synchronisation time as in the example above).

Repetition.
The **repeat** operation provides a straightforward way to specify repetition of an animated expression. Its semantics is straightforward:

$$\textbf{repeat } n \ E_1 = \begin{cases} \textbf{delay } 0 & \text{if } n = 0 \\ \textbf{repeat } (n-1) \ E_1; E_1 & \text{if } n > 0 \end{cases}$$

Asynchronous Execution.
It is sometimes desirable to specify animations independently, and introduce them into another animation in an asynchronous manner. This is especially useful if some animations must occur at some specific "absolute" time point in an animation. The **at** operation is used to express this idea. We define its semantics structurally as follows.

$$\textbf{at } t \ E_1 \ \mathcal{S} \equiv (\textbf{delay } t; E_1) \& \mathcal{S},$$

where $\mathcal{S}$ refers to the remainder of the script. Observe that the specified time point is truly absolute only if the statement occurs at the beginning of the script before any synchronous operations are specified.

Temporal Scaling.
It is useful to have the ability to redefine the length of time between an object's "ticks". This is particularly handy when attempting to describe motion co-ordination (see below). The following operation changes an object's duration and scales its temporal behaviour in proportion to the change.

$$E_1\langle n \rangle.d = n \quad (n \geq 0)$$

$$E_1\langle n \rangle.t = E_1.\left(\frac{tn}{E_1.d}\right)$$

Note that we allow temporal behaviour to be defined over $\Re$. One may instead wish to change the semantics to round to the nearest natural number, for example.

The next two operations are somewhat more experimental than the others, in that we wish to explore some issues of nondeterminism and randomness in specifying computer animation (and object activities in a more general setting).

Nondeterminism

To add variety to an animation, one could specify that one expression or another is to be chosen *randomly* to execute. The expression $E_1 \mid E_2$ has the following semantics:

$$(E_1 \mid E_2).duration = E_1.duration \ \wedge \ (E_1 \mid E_2).t = E_1.t$$

$$\oplus$$

$$(E_1 \mid E_2).duration = E_2.duration \ \wedge \ (E_1 \mid E_2).t = E_2.t.$$

We have chosen to view the alternatives as being mutually exclusive. However, it would be useful to specify a desired distribution of choices, and moreover to know *which* alternative is chosen. The latter is particularly useful if one wishes to define an animation based on a nondeterministic "shuffling" of several animated sequences (such as video clips). However, this appears to require saving a state, and then formulating a boolean condition on such a state.[3] We are beginning to experiment with notations for these functions, but introducing state variables and conditions complicate matters, since one may then have to consider issues such as mutual exclusion–in short, one gets a full parallel programming language.

Temporal Overlap

Another way to add variety is to specify simply that the execution of two expressions must overlap in time, but that their duration of overlap, and their order of execution is left unspecified. This is captured by the notation $E_1 \star E_2$, which has the following semantics:

$$E_1 \star E_2 \equiv (E_2 \& (\textbf{delay } \xi_2; E_1)) \ \mid \ (E_1 \& (\textbf{delay } \xi_1; E_2))$$

[3]Clearly, to solve the problem at hand, it suffices to define an n-ary shuffle operator, but this begs the basic question of the desirability of states and conditions.

where $\xi_1 \in [0, E_1.duration)$ and $\xi_2 \in [0, E_2.duration)$ are uniformly distributed random variables over their respective ranges.

The following is a basic property of temporal overlap that the reader may wish to prove:

$$(A \star B) \star C \Rightarrow (A \star C) \vee (B \star C).$$

It states simply that if two overlapping activities overlap with a third, then one of the first two activities must also overlap with the third. The converse is not necessarily true.

3 Instantiation, Motion, and Relative Time

Animated objects can be instantiated from prototypes in a library. Prototypes typically encapsulate a graphic object (defined in terms of polygons and free-form surfaces) and a default dynamic behaviour.[4] However, if permitted by the prototype, much of the default behaviour can be changed when instances of the object are created. The general form of an instantiation command is:

InstanceName: ObjectPrototype(parameters) {DynamicsScript}.

If, for example, **Teapot** is a prototype of an animated teapot, then a script may contain some of the following declarations:

tea1: **Teapot**
tea2: **Teapot**("origin=(7,4,3)")
tea3: **Teapot**("origin=(7,4,3)"){DynamicsScript}

In the first case, *tea1* inherits the default teapot behaviour. In the second case, *tea2* inherits the dynamics of **Teapot**, but the default origin is overridden.[5] In the last case, *tea3* redefines the dynamics of **Teapot**. We have been careful to employ an object-oriented approach to motion specification in the following sense: a particular motion (such as a trajectory) is encapsulated as a *motion object*, and can be used by any graphic object. Since motion

[4] A reasonable default dynamic behaviour would be to not move at all, that is, to be a prop.

[5] This does not necessarily mean that the animation will now function correctly, however. The prototype is not obliged to accept the new behaviour, nor to ensure consistency if it does accept it.

objects have a duration and temporal behaviour as do animated objects,[6] the scripting language can be used to build complex motion behaviour from these objects, in a manner that is identical to that of building complex animations from animated objects. Consider the following example.

rot: **RotateY**(-180)
trajectory: **FollowTrajectory** $(p_0, p_1, \cdots, p_n)$
cup1: **Cup**("origin=5,0,0") {(**repeat** 2 trajectory)$\langle 30 \rangle$ & rot$\langle 30 \rangle$ }
cup2: **Cup**("origin=3,0,0") {trajectory$\langle 45 \rangle$ & rot$\langle 60 \rangle$ }
cup3: **Cup**("origin=1,0,0") {trajectory$\langle 30 \rangle$ & (**repeat** 2 rot)$\langle 30 \rangle$ }
tea: **Teapot** { **repeat** 3 (**RotateY**$(360)\langle 30 \rangle$) }
eye: **Eye**("origin=(0,0,-5)", "lookAtPoint=(0,0,0)", "up=(0,1,0)")

Two implementations of the scripting language and environment have been made on a network of Sun3 computers running UNIX 4.2 BSD.[7] The compiler-writing tools Lex and Yacc have been extremely helpful in allowing us to modify in fairly arbitrary ways the syntax and semantics of temporal operators. We shall briefly outline both implementations, which are intended to be complementary.

In the first implementation, all graphic and motion objects are realised as shell scripts or C programs in the UNIX C-shell environment. This has the advantage of being fairly portable (to most other UNIX systems, in any case), while still remaining object oriented. A scheduler is built into the parser so that scripting expressions can be directly executed. Animated objects are invoked by the scheduler, which in turn produce device-independent graphics commands. These commands are pipelined (or "piped" to use a familiar UNIX expression) into a 3-D rendering system, which produces a sequence of images corresponding to the animation. With simple changes to the animated objects, the same animations can be ray-traced.

The above prototype system works well, but is somewhat inefficient, because each object is essentially a UNIX process. We have therefore made a second implementation of the scripting environment. The "second-generation"

[6]Indeed, a motion object may be viewed as an animated object composed of a null set of graphic primitives, or alternatively as an animated object with the static graphical model as a free variable.

[7]UNIX is a trademark of AT&T Bell Laboratories.

implementation is somewhat more programmer-oriented, in that scripts are invoked from within a C program. Objects co-exist as separate processes controlled by the scheduler. We have built a process-creation facility using locally-developed software that manipulates run-time stacks, allowing one to make many copies of object prototypes (which are now implemented as sets of C procedures) at run-time. A profusion of pseudo-processes can therefore run within the same UNIX process, with very little overhead. This has resulted in much faster execution of the scripting expressions, at the cost of more verbose animation specifications. The rendering of graphic images uses the same pipeline as that described above. The next generation system will go back to directly-executed scripting expressions, and will again look like the expressions presented in this paper.

We shall briefly discuss how the scheduler works for both implementations. When a well-formed scripting expression involving a number of temporal operators is scanned, the temporal operators are rewritten in terms of the general synchronisation operator. At this point, the required temporal scaling operations are performed, so that the entire expression is defined relative to regular ticks occurring at the desired sampling rate of 1/30 seconds.[8] Expressions in this form are then placed in a binary expression tree, the nodes of which indicate the displacement in ticks between the left subtree and the right subtree. One exception is that the **repeat** operation is also encoded as a node in the expression tree. Each **at** expression encountered results in the creation of an independent expression tree which exists for the duration of its execution (which may exceed that of any current synchronous expression). The overall internal representation of the current "scene" in the animation is that of a forest of expression trees, namely one synchronous expression tree, together with all active asynchronous expression trees. The scheduler then begins ticking for the duration of the expression (which is synthesised from the semantics of the temporal operators). For each tick, the expression forest is "aged", and the timing information percolates down each tree, causing messages to be sent to the affected objects. These objects then generate their animated behaviours for time instant specified by the scheduler.

[8]The choice of this rate is not fixed. It refers to the standard interlaced video rate for North American television. Certainly, other durations such as the standard European video rate of 1/25 seconds could just as easily be employed.

5 Extensions: Event-Driven Animation

Our experience with several implemented versions of the scripting language has indicated to us the need for increased responsiveness of animated objects to their environment. One particularly appropriate mechanism for doing so within an object-oriented environment is to permit conditional or triggered execution on the basis of events that may occur in a system. In this section we briefly discuss our plans for extending the animation language in this direction. We have not as yet implemented the notation, although we have developed an implementation model based on the "overseer" concept [BBBF82], and we are working on a formal semantics for event-driven animation. We believe the implementation of the notation will be a fairly straightforward extension of the process-management software we have already developed.

Suppose an object A has publicised the existence of events $E_1, E_2, \cdots, E_n$. The occurrence of an event is given by a predicate over time. Examples of events include: pushing a button, a timeout, an external interrupt, the completion of an activity, and so on. We extend the scripting notation to make use of these events.

Triggered Expressions. The **at** command was used before to introduce a new scripting expression into an otherwise synchronous set of expressions. The following construct

$$[*]\text{A.E} \rightarrow \textit{Expr}$$

means that the scripting expression *Expr* is to be invoked when event E within object A next occurs. The "*" is optional, and states that this trigger is to be enforced for all occurrences of E. Note that A is not itself required to be an animated object. It must only be able to inform a requesting object whether or not an event has occurred.

Repetition. The **repeat** operation is augmented as follows.

repeat until A.E *Expr*

which simply states that *Expr* is to be repeatedly invoked until the event E occurs. Although the event may occur in mid-execution, the expression always receives an integral number of executions.

Asynchronous Termination. An arbitrary expression (including the expressions introduced above) can be terminated asynchronously upon the occurrence of an event. We denote this by the following.

$$Expr \leftarrow \text{A.E}$$

To say, for example, that the first button push on a mouse is to invoke an expression, and the second is to terminate it, one could write

$$\text{Mouse.ButtonDown} \rightarrow (Expr \leftarrow \text{Mouse.ButtonDown})$$

The use of parentheses is important, since

$$(E_1 \rightarrow Expr) \leftarrow E_2$$

states that the trigger $E_1 \rightarrow Expr$ is to stay in effect until the occurrence of E_2.

6 Conclusions

We have presented a new, object-oriented approach to the temporal coordination of computer animation. Individual objects encapsulate basic animation activities, and a temporal scripting language is used to synthesise an overall animation from these parts. The language also extends to motion coordination, since motion is also given an object-oriented encapsulation. This approach has led to the ability to specify complex animation and motion in a concise, device-independent manner.

There are several directions in which we would like to take this research:

- To implement event-driven animation. The advent of personal workstations with powerful processors is beginning to make feasible the generation of animation in quasi-real time. Moreover, events can be immediately applied to triggering precomputed animation.
- A general message-passing model for objects. Currently, animated objects only respond to two kinds of messages. This imposes a regular interface for all objects, but it is not flexible.

- Extending our approach to incorporate "sound objects" as well as animated objects. Preliminary investigation indicate that the scripting language extends nicely to such objects. However, it is easy to change the internal clock of an animated object. It is not as simple to tell a piece of music that it must be sped up or slowed down, although there are signal-theoretic mechanisms to do so.

- Applying the scripting expression language to triggering video clips from an optical disk. This would be a very straightforward application, and one to which our expressions are perfectly suited.

- Adding more intelligent sampling behaviour. Animated objects can make decisions regarding their rendering. For example, since they know what their duration is, and since they are defined on a continuous time model, they may wish to produce representations of themselves which take into account problems such as temporal aliasing [KoBa83]. However, it is not clear that traditional global techniques can be applied.

References

[Baec81] Baecker, R., *et al.*, "Sorting Out Sorting", 16mm. film shown at SIGGRAPH 1981, available from R. Baecker, Computer Systems Research Institute, University of Toronto, 10 King's College Road, Toronto, Canada, M5S 1A4.

[BBBF82] Beach, R.J., J.C. Beatty, K.S. Booth, E.L. Fiume, and D.A. Plebon, "The message is the medium: multiprocess structuring of an interactive paint program", *ACM SIGGRAPH 1982 Conference Proceedings*, also published as *ACM Computer Graphics 16*, 3 (July 1982), 277-287.

[Berg83] Bergeron, P., "A structured motion specification in 3D computer animation", *Proceedings of Graphics Interface '83* (May 1983), 215-222.

[BeKa76] Bergman, S., and A. Kaufman, "BGRAF2: A real-time graphics language with modular objects and implicit dynamics" *ACM SIGGRAPH 1976 Conference Proceedings* (July 1976), 133-138.

[BeKa77] Bergman, S., and A. Kaufman, "Association of graphic images and dynamic attributes", *ACM SIGGRAPH 1977 Conference Proceedings* (July 1977), 18-23.

[FiTs87] Fiume, E., and D. Tsichritzis, "Multimedia objects", *IEEE Office Knowledge Engineering Newsletter* (Feb. 1987).

[FoMa86] Foley, J.D., and C.F. McMath, "Dynamic Process Visualization", *IEEE Computer Graphics and Applications 6*, 2 (March 1986), 16-25.

[KoBa84] Kochanek, D.H., and R.H. Bartels, "Interpolating splines with local tension, continuity, and bias control", *ACM SIGGRAPH 1984 Conference Proceedings*, also published as *ACM Computer Graphics 18,* 3 (July 1984), 33-41.

[KoBa83] Korein, J.D., and N.I. Badler, "Temporal anti-aliasing in computer generated animation", *ACM SIGGRAPH 1983 Conference Proceedings*, also published as *ACM Computer Graphics 17,* 3 (July 1983), 377-388.

[MaTF85] Magnenat-Thalmann, N., D. Thalmann, and M. Fortin, "Miranim: An extensible director-oriented system for the animation of realistic images", *IEEE Computer Graphics and Applications 4*, 3 (March 1985).

[Myer83] Myers, B., "Incense: a system for displaying data structures", *ACM SIGGRAPH 1983 Conference Proceedings* also published as *ACM Computer Graphics 17,* 3 (July 1983), (July 1983), 115-125.

[Nier85] Nierstrasz, O.M., "Hybrid: a unified object-oriented system", *IEEE Database Engineering 8*, 4 (Dec. 1985), 49-57.

[Reev81] Reeves, W.T., "Inbetweening for computer animation utilizing moving point constraints", *ACM SIGGRAPH 1981 Conference Proceedings* also published as *ACM Computer Graphics 15,* 3 (Aug., 1981), 263-269.

[Reyn82] Reynolds, C., "Computer Animation with scripts and actors", *ACM SIGGRAPH 1982 Conference Proceedings* also published as *ACM Computer Graphics 16,* 3 (July 1982).

[TFGN87] Tsichritzis, D., E. Fiume, S. Gibbs, O. Nierstrasz, "KNOs: Knowledge acquisition, dissemination, and manipulation objects" *ACM Transactions on Office Information Systems 5,* 2 (April 1987).