# A Model for Coordinating Interacting Agents

Paul Lalonde Robert Walker Jason Harrison David Forsey

Department of Computer Science
University of British Columbia
2633 Main Mall,
Vancouver, B.C. V6T 1Z4,
e-mail: {lalonde|walker|harrison|drforsey}@cs.ubc.ca

***Abstract***

SPAM (Simulation Platform for Animating Motion) is a simulation software system designed to address synchronization issues pertaining to both animation and simulation. SPAM provides application programs with the manipulation, configuration, and synchronization tools needed when simulations are combined to create animations. It is designed to be used as the glue between applications that supply lists of the parameters to animate and the callback procedures to invoke when a user wishes to modify the parameters directly. SPAM does not impose a particular model of simulation, accommodating keyframing, physical simulation, or a variety of other models, providing they can be abstracted into a set of externally modifiable variables.

In SPAM we recognize that the important part of simulation is not the state of the system at each time step, but rather the change in states between steps. Thus SPAM uses an interval representation of time, explicitly representing the intervals over which change occurs.

In a complex animation or simulation, multiple actions will access the same resource at the same time. SPAM defines a strategy for recognizing such conflicts that increases the use and re-use of sequences.

***Résumé***

SPAM est un système de simulation crée pour adresser les problèmes de synchronisation présents dans les systèmes d'animation et de simulation. SPAM apporte aux logiciels des outils pour éffèctuer les taches de manipulation, configuration, et synchronisation qui sont nécèssaires quand différents modèles de simulation sont combinés pour produire une animation. SPAM est conçu pour servir de lien entre plusieurs constituants qui apportent chaqu'un une liste de paramètres controlables. On peut se servir de n'importe quel modèle, soit le "keyframing," la simulation physique, ou d'autres méthodes, tant qu'elles puissent exporter des variables à modifier.

SPAM témoigne du fait que l'aspect important de la simulation n'est pas l'état du système à chaque démarcation temporale mais plutôt les changements entre les moments démarqués. Par conséquent SPAM se sert d'une notation d'intervale temporale, ce qui permet une représentation explicite de l'intervale dans laquelle le système change.

Dans une scène complexe, plusieures actions accédent à une resource en même temps. SPAM incorpore une stratégie pour reconnaître ces conflits et encourager la ré-utilisation de séquences. Les résolutions des conflits peut être soit automatique ou par l'intervention de l'utilisateur.

**Keywords:** Animation, simulation, data flow, keyframing, parametric animation, intervals.

## 1 Introduction

*Animation is the art of manipulating*
*the invisible interstices that lie between frames.*
*— Norman McLaren*

Computer animation systems aid the traditional process of animation by providing keyframing, editing, sequencing, previewing, and mathematical models from the realm of simulation: procedural models, dynamics, kinematic and dynamic constraints, and inverse kinematics, to name a few. The task of integrating all of these approaches must address the problems associated with the interactions between simulators and their differing notions of time. Animation systems tend to be stand-alone monoliths, and although open animation systems have appeared that allow third-party software developers to extend the capabilities of the stock product, this approach is not useful in the highly heterogeneous and changing environment of an academic research laboratory. A research lab can generally neither afford the cost of the software, nor the staff required to create and maintain all of the packages and conversion utilities.

Much of the difficulty involved in using different models of animation and simulation arises in combining them. Simulation systems evolve from a set of initial conditions: the state at any point in the future is unknown and requires considerable computation and data, the details of both frequently unknown in advance. In contrast, animators often work towards a final state with a preconceived path. This demands much more control over the animated elements, and usually includes completely specifying all the motions of all objects. Numerical integration, especially adaptive methods that use a variable time increment, is particularly difficult to implement correctly in this context. Mixing traditional animation with simulations requires considerable thought about how the interactions of the methods should be handled. *Ad hoc* solutions will work given enough refinement of the individual animations, but a more general solution to the problems is required.

SPAM provides applications with animation, sequencing, synchronization, scripting, and coordination capabilities in a manner similar to the way that user interface toolkits such as Motif and Open Windows provide applications with window management services. The application supplies the functions and parameters to be controlled, and SPAM coordinates the animation and grouping of those parameters through a combination of its own separate user interface and any pre-existing application interface. The interfaces between SPAM and application code are called *actuators*, and define an abstract data type that embodies the particular set of system parameters to control, along with the callbacks or routines used to set or read these values.

The specification of an animation from a graphical interface is converted into a graph representation that is subsequently evaluated to update the state of the system for each time interval. Evaluation is performed in much the same way as in a traditional dataflow system [dyer90] but SPAM explicitly models time and has the added capabilities to handle discrete events, to synchronize actions, and to detect loops and conflicts that arise when two actions share a resource.

For example, for an application that provides basic forward kinematics for articulated figures, SPAM provides all the capabilities of a parametric animation system [stur84, hanr85]. To control the pose of, for example, a hand making a fist, the animator groups the degrees of freedom such that the degree to which the hand is open is controlled through the manipulation of a single parameter. Typically the animator defines multiple groups, each controlled by a single parameter and specifying some other hand posture (e.g. touching the thumb and forefinger together). Whenever these actions are combined in an animation the groups need to modify a shared resource, e.g. the angle of one of the finger joints. SPAM detects such conflicts when the animation sequence is specified and either internally resolves the conflict with a pre-defined (possibly interactive) tool, or allows the application to arbitrate using its own interface to the animator.

SPAM, however, is more than a parametric animation system; it is a system for coordinating multiple models of simulation, and is designed for use with new or existing simulation software with a minimum of additional code. We define an *agent* as an object encapsulating data and the simulation software that changes the state of the data over time, while exchanging information to and from the system as a whole as needed. SPAM treats simulations, as well as traditional animation techniques such as keyframing, as agents.

Consider a situation where several agents in the application interact. If agent $A$ and agent $B$ proceed such that agent $A$ interacts with other agents (requests or supplies data) at a schedule incompatible with the schedule of interactions of $B$, the situation will arise where the two agents attempt to modify the same resource at overlapping intervals. SPAM coordinates agents that run at different rates or with completely different notions of time by treating time as an interval and allowing the various simulations to proceed at the appropriate rates and by providing mechanisms to resolve conflicts when multiple agents access the same resource.

Many simulation and animation systems view the passage of time as a steady procession forward at a specific rate [bart89]. This view is limited and restrictive. If different simulation models are to coexist in an animation system, any incompatibility in the representation of time must be resolved and coordinated. If the only concept of time advancement is a clock tick, it becomes difficult to integrate the different simulations without aliasing artifacts, and the situation is even worse if the integrators use adaptive step sizes. The cost of imposing the smallest step size on all the components is too high.

# 2 Background

There are many examples of treating animation as a form of simulation. In particular we are interested in those approaches that address the problems of dealing with multiple simulation agents and the conflicts generated through their interactions. Rozenblat and Muntz in their Tangram Animation System [roze91] use rule and event based animation control to animate the results of their queueing network and Markov chain simulations. Their system is based on animating state changes, and as such, each rule is of the form of a script evaluated when some condition becomes true. Although useful for animation based upon events this approach is, in essence, a discrete event simulation and as such does not readily deal with continuous processes. They also do not address the problems that arise when when multiple rules could be selected by the same condition.

Kalra and Barr [kalr92] build simulation systems by composing motion behaviour rules (functions of time, differential equations of motion, or constraints) using directed graphs that specify the conditions that should cause the behaviour of the system to change. These conditions, *event units*, and the associated change in the motion behaviour can be discontinuous and asynchronous, but the general principle is that the system of equations governing the simulation changes when an event occurs. Thus the state of the system can be continuous, but the set behaviours can be discontinuous. Since the system of equations and the conditions for changing the system are under the control of a central evaluator inter-object interactions (e.g. collisions) can be handled efficiently. However, this framework does not exactly specify the behaviour of the system if multiple events occur at the same time. It is also not apparent how to extend this framework to a collection of simulation engines (*agents* in our nomenclature) where the equations of motion and the associated state variables are encapsulated within the agents.

Kazman takes a different approach, building a system based on modeling continuous dynamic simulation. HIDRA [kazm93] is structured around autonomous objects, a distributed world manager, and a collection of servers that handle object-object interactions. By centralizing certain inter-object interaction such as collisions, the servers reduce the need for each object to independently detect interactions. However, HIDRA's notion of time is expressed simply as clock cycles, and there is no structured way of dealing with the classic readers and writers problem that occurs when the state of an object is accessed by several interaction servers [cour71].

Zeleznik *et al.* [zele91] embody behaviours as *controllers* that send abstract messages to various objects in the system who in turn determine how to react to those messages. In essence, the controllers are simulation engines (our *agents*) operating on the participating objects. For example, an interactive technique is considered a controller for an object. When different controllers affect the same object at the same time, the authors propose the use of an intermediary controller to mediate the use of the shared resource. The authors did not present a solution to the difficulties that arise when controllers are running simulations at different rates.

Kühn and Müller encapsulate different simulations into independent controllers (agents)[kuhn93]. Though they acknowledge the possibility of multiple controllers affecting the state of an object during the same time step, they do not provide means for mediating such conflicts. This leaves conflict resolution in an *ad hoc* state, uncontrollable within their framework. Their notion of time is based on synchronization of local clocks in a hierarchy of object group (environments) and controller pairs. Each environment ticks forward at a specific rate, invoking its controllers at each clock tick. The controllers are responsible for advancing the state of the environment to the next state, and may in turn activate other environments. No mechanism is provided to deal with a non-hierarchical control graphs.

SPAM mediates the progression of time and coordinates communication between components in an application. The major components of an application are:

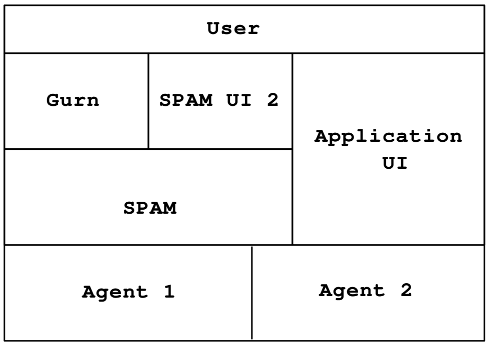


Figure 1: *The structure of an application using SPAM. The simulation agents communicate to the animation/simulation interfaces, or directly to the animator by using an interface associated with an an actuator.*

# 3 The System

**SPAM UI:** the user interface used to specify how agents interact.

**Agents:** the computational elements to coordinate. For example, a free-form surface modeller or a simulator for rigid-body dynamics.

**Agent UI:** the user interface to an agent. For example, the interactive editor for a free-form surface modeller.

**SPAM:** the graph of processes that coordinate the interaction between the agents, and its evaluator.

A separate interface, called Gurn, builds the SPAM graph using calls to library routines that build SPAM sub-graphs that implement interpolators, constraints, and synchronization operators such as those found in Fiume *et al.* [fium87]. The details of Gurn are beyond the scope of this paper.

The interface between an agent and SPAM (agents do not directly communicate with each other) is called an *actuator* and represents the parameter or group of parameters to animate. Thus for a free-form surface modeller an actuator is instantiated for each modifiable degree of freedom available in the editor. Each agent posts its actuators to SPAM and provides callback routines for access and modification of the parameters represented by the actuator. If the agent has its own user interface for setting a given parameter, then access to this interface can be included in the actuator definition. Information is transferred between an agent (via its actuators) and SPAM through an abstract data type, called a *steward*, that encapsulates sources, sinks, conflict resolvers, and forecasters (all described below). A steward administrates all access to an actuator from other components within the application.

SPAM implements a simple process control mechanism supporting small atomic sequential processes communicating over fixed, typed, communication channels. The graph, reminiscent of those generated using data flow languages [dyer90], is evaluated to advance the state of the system through a time interval.

One evaluation of the SPAM graph advances time a specified amount. At the beginning of each cycle the system is in a particular state — the actuators reflect the internal state of the agents. It is the job of SPAM to achieve a consensus about the state of the system at the end of the interval. Because agents are allowed to run at different time steps, sometimes a value for an actuator at an intermediate time is required that has not yet been computed. Stewards administrate the calculation of this intermediary information required by other agents. Once a concensus is reached, an explicit *commit* action sets the internal state of all actuator-mediated parameters in all agents, and time advances for the application as a whole.

The graph is evaluated by executing all processes not blocked awaiting input. These nodes broadcast the results of any internal computation on their output channels which triggers further computation in connected processes. Evaluation continues until all active portions of the graph have been traversed at which point the commit occurs, all the stewards write values to their actuators and each agent is invoked to deal with the change in its internal state. Evaluation of the graph is explained in more detail in Section 4.1.

## 3.1 Stewards

SPAM is designed to mediate the interaction of multiple agents by insulating an agent from the effect of any other agent's notion of how time progresses. The internal state of an agent (i.e. its actuator values) must be protected from, yet simultaneously available to, other components in the system. The internal state of the agent must be protected in those situations where constantly updating the agent's internal state is too costly or is inappropriate, and available if intermediate (i.e. requests to update the actuator value before the commit) values can be safely accomodated. SPAM deals with this by using a *steward* that controls all access to an actuator.

Consider a simulation with two agents – $A$, that proceeds at a fixed time step, and $B$ that proceeds with a variable time step and requires a value from $A$ at the beginning of each of its intervals. Forcing $A$ to run with small step sizes so that it has values available when $B$ requires them is inefficient, if not impossible. Instead some flexible strategy is needed to produce an appropriate value when needed. Depending on the nature of $A$ the agent may produce an exact value, or the steward can provide an estimate. The stewards associated with each of $A$'s actuators encapsulates this knowledge and allows the agent to choose the appropriate behaviour.

The steward also deals with those situations where requests to set an actuator occur more frequently than the agent's internal notion of time allows. Dependant upon the nature of the agent, it may be reasonable to make these intermediate values available to the rest of the system as they are written or retain the old value until the next commit phase.

## 3.2 Sources

Data is introduced into the SPAM process network via *source* nodes in the control graph. Using a time interval as its input a source returns the value that represents the actuator's value at the beginning of the interval. Each source is bound to the steward that administrates access to the actuator. In general, the value of the actuator is accurate only for the beginning of the interval, not for times within it. As the result of the steward's operation, there may be cached values that include the requested interval or bracket it, and in such cases the steward invokes an interpolation or extrapolation mechanism to provide a value for the source. Such a guess is called a *forecast*, and will be examined in further detail in Section 4.2.

## 3.3 Sinks and Caches

*Sink* nodes are used to set the value of actuators. Data sent to a sink is mediated by the steward which caches the value according to the interval for which it was calculated. At the commit stage, once all computation in the process graph is complete for the given interval, this cache is dumped to the agent. The steward may pass either the complete list to the actuator, or just the final value, depending upon the type of steward instantiated.

## 3.4 Conflict Resolution

When multiple values for the same time interval arrive at a sink node the conflict is resolved using a *conflict resolver* which utilizes one of several possible strategies ranging from a simple priority scheme which uses the most recently written value or a weighted sum of all the inputs in the overlap. Each conflict resolver has its own set of actuators that control its behaviour (e.g. the weights to apply to each input) which are also controlled via the SPAM process graph. In the GURN user interface to SPAM, the choice of conflict resolver (and values sent to its actuators) is part of the specification for an animated sequence.

Rather than always determining the final value, resolution of a conflict is deferred until a source requests a value falling within the overlapping interval. This is particularly important for computationally expensive resolution strategies, such as those involving manual intervention by a user.

## 3.5 Transformers and Synchronization Operators

The remaining nodes in a SPAM graph are called Transformers. A transformer performs an atomic stateless computation using its inputs, and passes the result to its output channel(s). A transformer may simply generate a constant value, perform interpolation, access the operating system, or partake in a complex calculation incorporating time, differential equation solvers, or constraints [glei90, haeb88, kass92]. Of particular interest are a few transformers for manipulating time and providing flow control.

A *splitter* provides a simple looping mechanism, breaking down an input interval into a stream of fixed or variable size intervals.

A *gate* clips an input interval to the interval specified at its initialization and is used to ensure that a sub-graph is evaluated only at certain times. If the clipped interval is null then no output is written.

# 4 The Control Graph

Although SPAM graphs can be very general, because they encode a particular set of operations and interactions, they typically have a great deal of structure. Degrees of freedom that do not interact form non-interacting subgraphs. Degrees of freedom that do interact typically do so in limited ways, usually simple, uni-directional dependencies. For instance, in the case where agent $A$ uses the value of an actuator in agent $B$, the dependancies are visible in the graph as a source reading $B$ in the subgraph that evaluates a result for $A$. Of course, cycles can appear in the graph. How they are dealt with is examined in greater detail in Section 4.7.

## 4.1 The Structure of the Graph

A SPAM graph consists of a starter node, with no inputs, connected to other transformers and sinks. The starter node defines the time interval over which to evaluate the graph. It is strobed with a sequence of time intervals to advance the state of the application over time.

Evaluation of the graph is a straightforward traversal that invokes any node that has all its required inputs. Not every node is evaluated since some will never receive their required inputs for a given interval because some subsection of the graph is only active for a specific period of time. (Gate nodes are used to fence off subgraphs in this manner).

## 4.2 Sources and Deferred Evaluation

Most of the nodes in the graph are simple atomic operators that provide output once all their inputs are available. Sources are an exception to this because of the behaviour of the stewards. When the value for an actuator (via a source) is requested for the beginning of an interval, the steward's actuator value corresponds to the current state in the agent, and this value is returned immediately. If some intermediate value exists in the cache, or if no current value is available, evaluation of the source node is deferred with the expectation that more information will become available as the rest of the graph is evaluated.

When all nodes in the graph are blocked awaiting input, any source node still deferred is evaluated. This causes the appropriate steward to invoke its conflict resolver which perform either interpolation or some other more sophisticated mechanism to estimate the current value of the source.

## 4.3 Cycles

Cycles appear in a SPAM graph as a result of inter-steward interactions or from constraints enforced within an agent. The former case is easier to detect and deal with because some SPAM UI in used in building the graph and thus can ensure that the graph is built properly. Cycles caused by dependencies between actuators within an agent are much more difficult. In general SPAM cannot detect these, and so they must be flagged explicitly when the actuators are instantiated. Strategies for dealing with such interactions are a topic for future research and may involve either invoking some general constraint resolution method on the affected subgraph or returning control to the agent to deal with the dependencies internally.

## 4.4 Building SPAM Graphs

Because SPAM graphs can be very general, it is important to impose structure on them to make them easy to generate. Fortunately many common actions are expressible using simple sub-graphs that have a simple interface to the rest of the process graph. These sub-graphs are encapsulated into the library routines that the SPAM UI uses to construct the graph.

Consider for example a sub-graph that performs a summation (figure 3). Its only interaction with the rest of the graph is its input, its output and choice of the source. This sub-graph is made re-useable by building a library routine that instantiates this sub-graph given the choice of source and the interval over which to evaluate the summation.

In a similar fashion, subgraphs that perform interpolations, constraint enforcement, or a complex simulation, are collected into routines whose interactions with the SPAM graph are set by parameters supplied at run-time. Assorted synchronization op-

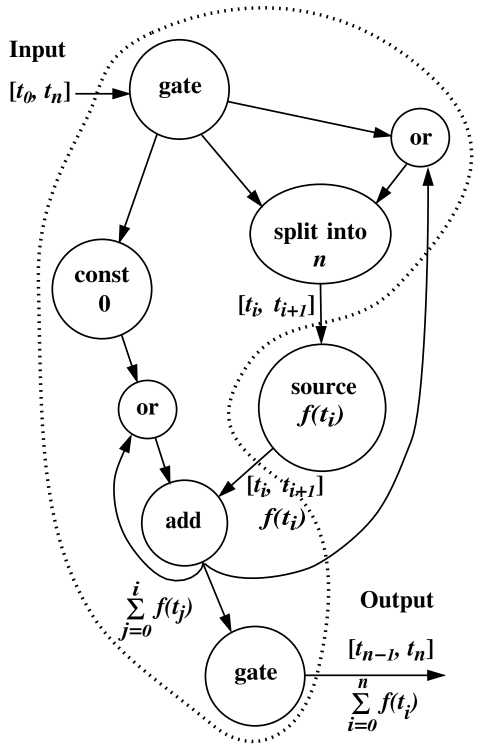


Figure 2: *A SPAM subgraph to perform a summation. The* constant *specifies the zero condition, the* or *nodes pass along one of their inputs without waiting for another, and the final gate assures that a result will only leave the sub-graph at the end of the interval.*

erators are implemented in the same way. Fiume *et al.* detail a number of useful operators [fium87] that all have isomorphic representations as SPAM graphs.

# 5 Keyframing and Dynamic Simulation

Using SPAM we built a simple application combining three agents, a display engine that displays a double pendulum, a dynamic simulation engine that performs dynamics on the pendulum, and an input agent that uses mouse input to set the pendulum's joint angles and accelerations.

The drawing agent redraws the pendulum every time a new state vector is written to its actuator. The dynamics agent, written using SD/fast, a commercial dynamic simulation package, uses a variable step size integrator to calculate the joint angles of the pendulum from one iteration to the next. The input agent is used to interactively control the joint angles and supply forces and torques applied to the pendulum.

SPAM coordinates these three agents transferring values from the input agent to set the positions or forces on the pendulum that both the dynamics agent and the drawing agent must respond to.

# 6 Conclusions and Future Work

SPAM represents time as an interval, allowing explicit control of the interval over which values are requested during the simulation. SPAM's simple data-flow like graphs, coupled with the structure imposed on the graphs make it easy to coordinate complicated agents. SPAM is sufficiently powerful to deal with the interaction of various simulation engines, be they integrators, traditional keyframing, or procedural models.

Outstanding issues remain. When cycles are encountered they are flagged for the application to respond to, but no general strategy has been developed to structure the application's response.

Although SPAM deals with mediating information transfer between simulation engines, we have not addressed the complications arising from fundamental differences in representations used by different simulation models, as would occur if one simulation used, for example, a height field representation while another used a finite volume model. This problem of differing representations would require considerable support for appropriate shared data structures and domain-specific knowledge of the representational-level interactions, and is currently beyond the scope of the toolkit approach proffered here.

## References


[bart89] Richard H. Bartels and Ines Hardtke. "Speed adjustement for key-frame interpolation". *Proceedings of Graphics Interface '89*, pp. 14–19, June 1989.

[corm90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[cour71] P. J. Courtois, F. Heymans, and D. L. Parnas. "Concurrent Control with 'Readers' and 'Writers' ". *Communications of the ACM*, Vol. 14, No. 10, pp. 667–668, October 1971.

[dyer90] D. Scott Dyer. "A Dataflow Toolkit for Visualization". *IEEE Computer Graphics and Applications*, Vol. 10, No. 4, pp. 60–69, July 1990.

[fium87] E. Fiume, D. Tsichritzis, and L. Dami. "A Temporal Scripting Language for Object-Oriented Animation". *Eurographics '87*, pp. 283–294, August 1987.

[glei90] M. Gleicher and A. Witkin. "Snap Together Mathematics". *Eurographics Workshop on Object-Oriented Graphics*, pp. 21–34, 1990.

[haeb88] Paul E. Haeberli. "ConMan: A Visual Programming Language for Interactive Graphics". *Computer Graphics (SIGGRAPH '88 Proceedings)*, Vol. 22, No. 4, pp. 103–111, August 1988.

[hanr85] Pat Hanrahan and David Sturman. "Interactive animation of parametric models". *The Visual Computer*, Vol. 1, No. 4, pp. 260–266, December 1985.

[kalr92] Devendra Kalra and Alan H. Barr. "Modeling with time and events in computer animation". *Computer Graphics Forum (EUROGRAPHICS '92 Proceedings)*, Vol. 11, No. 3, pp. 45–58, September 1992.

[kass92] Michael Kass. "CONDOR: Constraint-based dataflow". *Computer Graphics (SIGGRAPH '92 Proceedings)*, Vol. 26, No. 2, pp. 321–330, July 1992.

[kazm93] R. Kazman. "HIDRA: An Architecture for Highly Dynamic Physically Based Multi-Agent Simulations". *International Journal in Computer Simulation*, 1993.

[kuhn93] Volker Kühn and Wolfgang Müller. "Advanced Object-oriented Methods and Concepts for Simulations of Multi-body Systems". *The Journal of Visualization and Computer Animation*, Vol. 4, pp. 95–111, 1993.

[roze91] G.D. Rozenblat and R.R. Muntz. "The Tangram Simulation Animation System". *Eurographics Workshop on Animation and Simulation*, pp. 153–167, 1991.

[snyd92] John M. Snyder. "Interval analysis for computer graphics". *Computer Graphics (SIGGRAPH '92 Proceedings)*, Vol. 26, No. 2, pp. 121–130, July 1992.

[stur84] David Sturman. "Interactive keyframe animation of 3-D articulated models". *Proceedings of Graphics Interface '84*, pp. 35–40, 1984.

[zele91] Robert C. Zeleznik, D. Brookshire Conner, Matthias M. Wloka, Daniel G. Aliaga, Nathan T. Huang, Philip M. Hubbard, Brian Knep, Henry Kaufman, John F. Hughes, and Andries van Dam. "An object-oriented framework for the integration of interactive animation techniques". *Computer Graphics (SIGGRAPH '91 Proceedings)*, Vol. 25, No. 4, pp. 105–112, July 1991.