Animation Support in a User Interface Toolkit: Flexible, Robust, and Reusable Abstractions

Scott E. Hudson
John T. Stasko

Graphics Visualization and Usability Center
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
E-mail: hudson@cc.gatech.edu, stasko@cc.gatech.edu

ABSTRACT

Animation can be a very effective mechanism to convey information in visualization and user interface settings. However, integrating animated presentations into user interfaces has typically been a difficult task since, to date, there has been little or no explicit support for animation in window systems or user interface toolkits. This paper describes how the Artkit user interface toolkit has been extended with new animation support abstractions designed to overcome this problem. These abstractions provide a powerful but convenient base for building a range of animations, supporting techniques such as simple motion-blur, "squash and stretch", use of arcing trajectories, anticipation and follow through, and "slow-in / slow-out" transitions. Because these abstractions are provided by the toolkit they are reusable and may be freely mixed with more conventional user interface techniques. In addition, the Artkit implementation of these abstractions is robust in the face of systems (such as the X Window System and Unix) which can be ill-behaved with respect to timing considerations.

Keywords: object-oriented user interface toolkits, window systems, animation techniques, dynamic interfaces, motion blur, real-time scheduling.

1 INTRODUCTION

Human perceptual capabilities provide a substantial ability to quickly form and understand models of the world from moving images. As a result, in a well designed display, information can often be much more easily comprehended in a moving scene than in a single static image or even a sequence of static images. For example, the "cone tree" display described in [Robe93] provides a clear illustration that the use of continuous motion can allow much more information to be presented and understood more easily.

However, even though the potential benefits of animation in user interfaces have been recognized for some time ([Baec90] for example, surveys a number of uses for animation in the interface and cites their benefits and [Stask93] reviews principles for using animation in interfaces and describes a number of systems that make extensive use of animation in an interface), explicit support for animation is rarely, if ever, found in user interface support environments. The work described in this paper is designed to overcome this problem by showing how flexible, robust, and reusable support for animation can be incorporated into a full scale object-oriented user interface toolkit. Specifically, this paper describes how the extension mechanisms of Artkit — the Advanced Reusable Toolkit (supporting interfaces in C++) [Henr90] — have been employed to smoothly integrate animation support with other user interface capabilities.

The animation abstractions provided by the Artkit system are designed to be powerful and flexible — providing basic support that can be used to build a range of sophisticated techniques such as: simple motion-blur, "squash and stretch", use of arcing trajectories, anticipation and follow through, and "slow-in / slow-out" transitions [Lass87]. Examples of several of these techniques will be given in the next section.

This work was supported in part by the National Science Foundation under grants IRI-9015407, DCA-9214947, CCR-9121607 and CCR-9109399.

In addition to providing power and flexibility, another central challenge has been to support these facilities under conventional window and operating systems such as the X window system and Unix. These systems were not designed with this sort of work in mind, and hence provide only the most limited support for time dependent activities. For example, these systems normally do not provide explicit control of scheduling, nor provide any timing guarantees, or even facilities to express timing goals. Further, in order to improve overall performance, client-server based window systems such as X, typically operate asynchronously resulting in even more unpredictable timing behavior. As a result, an animation system must cope with the possibility of arbitrary, irregular, and unpredictable delays in the

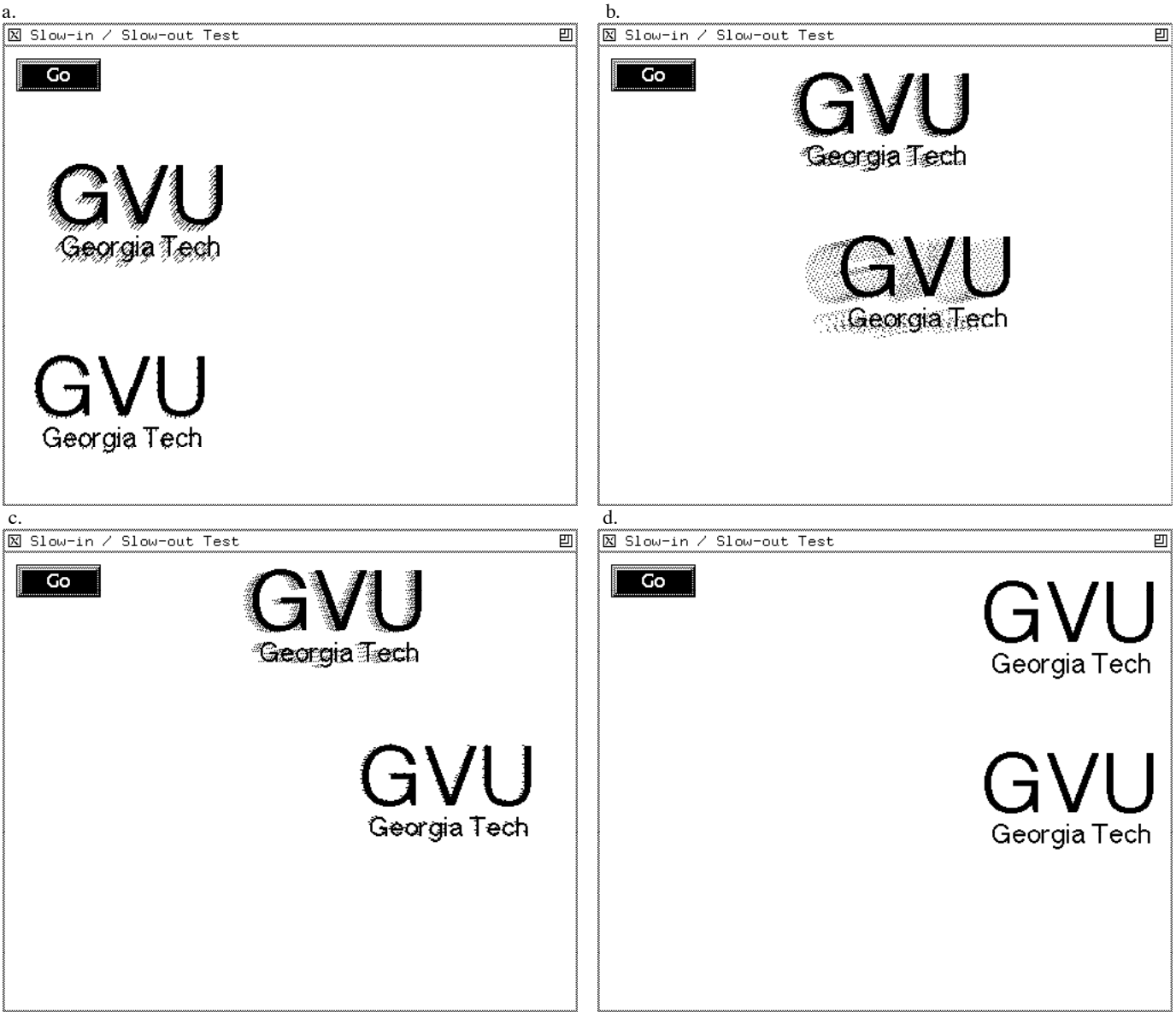

Figure 1. In this animation, two icon objects (each implementing a simple form of motion blur) move from left to right across an arcing trajectory. The lower icon employs a non-uniform ("slow-in / slow-out") pacing function — starting slowly, moving more quickly in the center of the trajectory, and finally ending its movement slowly.

delivery of display and other services. While these problems must inevitably affect the appearance of the final result, the abstractions described here are designed to be robust and degrade gracefully in the presence of these problems — providing good behavior when possible and reasonable behavior in all cases.

Finally, by providing a strong set of base facilities inside a toolkit it is possible both to integrate animated presentation with more conventional user interface techniques, and reuse the facilities for a wide range of applications that otherwise could not justify the expense and difficulty involved with starting from scratch.

The remainder of this paper is organized as follows. The next section will provide several examples that illustrate the animation techniques that can be supported by the Artkit animation abstractions. Section 3 will consider these abstractions from the interface designer's point of view, while Section 4 will consider how they are implemented. Finally, Section 5 will describe more implementation details and some experience, while Section 6 will provide a conclusion.

2. EXAMPLES

In [Lass87], Lassiter describes 11 fundamental principles of traditional animation along with some specific techniques that go with them. Some of these such as *staging*, *exaggeration*, and character *appeal* are artistic design issues that do not lend themselves well to specific toolkit support. However, the work described here attempts to support many of the others — either directly or through important enabling capabilities. These include: robust support for controlled *timing* of actions, support for "*slow-in / slow-out*" transitions, use of *arcs* in movement trajectories, the ability to support "*squash and stretch*" during animation sequences (as well as the more sophisticated technique of simple *motion blur*), and support for *anticipation* and *follow through* in movements. (Although partially supported, we leave specific facilities for the remaining techniques of *overlapping action* and use of *secondary action* , as well as complete support for keyframe (*pose-to-pose*) interpolation for future work). This section illustrates how many of these techniques can be supported by the Artkit system.

Timing: Careful timing can be critical for conveying properties of a moving object as well as properly pacing actions so that significant events are not missed by the user. Although it cannot be illustrated in static images, a central goal of the work presented here is to provide controlled timing. If an object is to move from point A to point B in 3 seconds, the interface implementor can easily state this. In addition, the system will deliver a metered set of animation steps to the object which cause it to arrive at point B as close to 3 seconds after leaving point A as possible, with as smooth a transition as the actual OS and window system performance and timing allow.

Slow-in / slow-out transitions: A slow-in / slow-out transition involves non-uniform pacing of movements. If an animated sequence is thought of as a series of poses (e.g., important points where action occurs or meaning is conveyed), then spending more time entering the pose and leaving it (and less time moving from one pose to another) can be a very effective way to get a point across. Figure 1 illustrates a slow-in / slow-out transition. Here, the top object maintains a uniform velocity while the bottom starts is trajectory slowly, moves quickly in the center, and ends slowly. This non-uniform pace is implemented using a reusable slow-in/slow-out pacing function taken from the toolkit library.

Arcs: Figure 1 also illustrates the use of an arced trajectory for movement. The animation abstractions described here support the use of any trajectory that can be described as a parameteric curve of one variable. This allows considerable flexibility and provides a uniform framework for trajectories such as splines, circular arcs, and straight lines.

Motion blur: Finally, Figure 1 also illustrates a very simple motion blur technique. Here the objects leave behind a blurred trail which varies in length depending on velocity. This technique helps prevent strobing effects for fast motions.

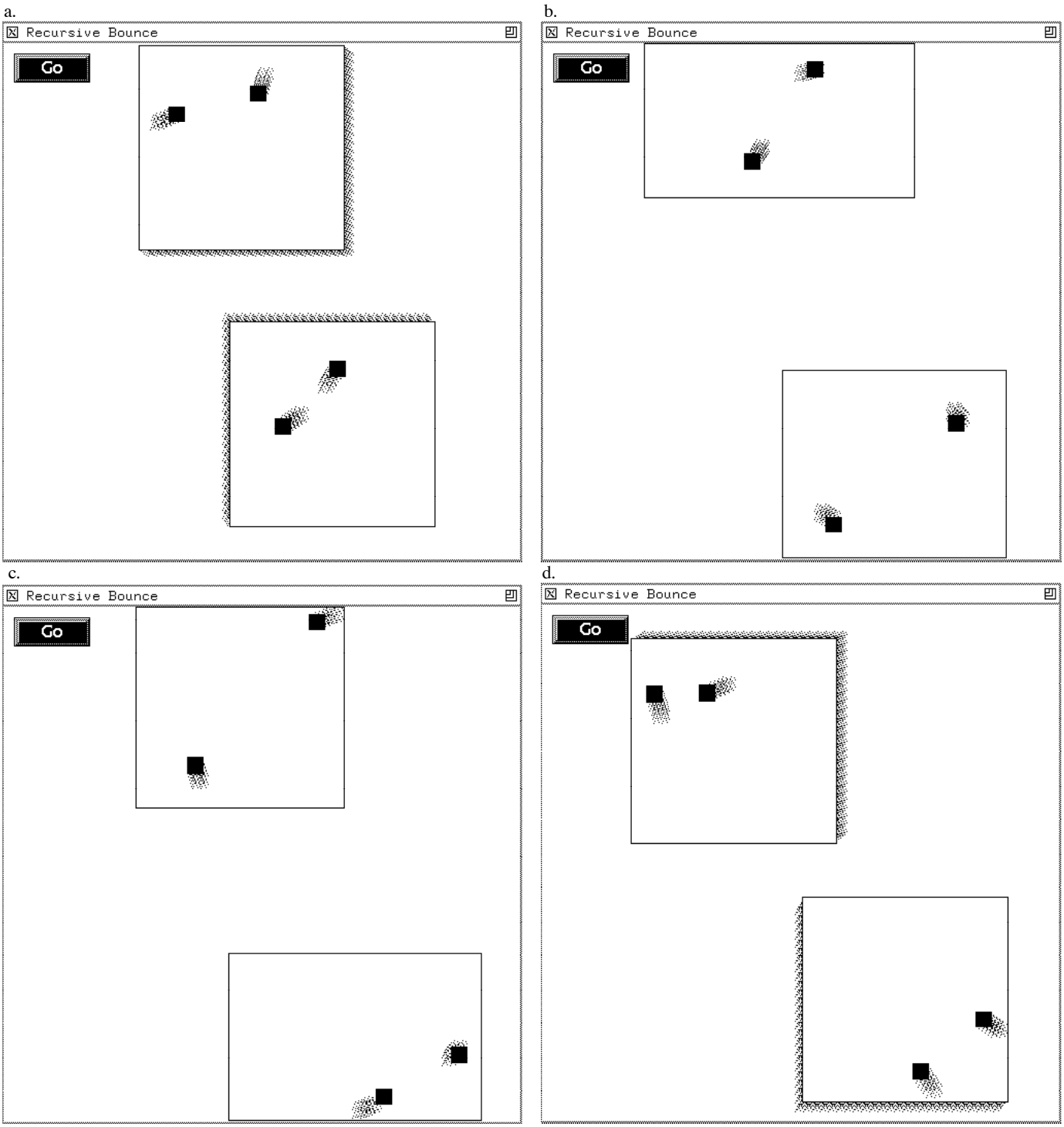

Figure 2. In this animation, two large bouncing boxes each containing two smaller bouncing "balls" illustrate squash and stretch as they impact the sides of the animation window. Trajectories for the smaller balls are applied within the local coordinate system of the box and are used to compute a position relative to the current height and width of the enclosing box.

Squash and stretch: Very few real world objects are completely rigid. More natural looking and appealing movements can be created when objects are allowed to deform somewhat as they start and stop. Figure 2 illustrates this technique. As the boxes containing the "bouncing balls" impact the sides of the window they "squash" — reducing their height or width but increasing their other dimension in order to maintain constant volume or area — then they stretch back to their normal proportions and begin movement again.

Anticipation and follow through: In traditional animation, a three part formula is often used for portraying actions. Before the action proper begins an *anticipation* action is taken to set the stage and prepare the viewer for the action that is to about to take place (for example, an exaggerated spinning of the character's feet indicating that they are about to exit the scene at high speed). Similarly, after the action proper, there is typically a termination or *follow through* movement so that the motion does not stop abruptly. This simulates or exaggerates the natural effects of inertia that we implicitly expect to see. The squash and stretch actions shown in Figure 2, are a simple example of this. Each wall to wall movement of a box has been implemented as three separate movements using the toolkit facilities: a stretch at the beginning, the linear movement from wall to wall, and a squash at the end. To support this and other effects, the toolkit provides a capability for scheduling movements relative to each other (e.g., one after the other, or with delays and/or overlaps).

The next section will consider the toolkit abstractions that each of the techniques illustrated here are built upon.

3. ABSTRACTIONS FOR ANIMATION

The central animation abstraction provided by the toolkit is the *transition*†. A transition is a movement or transformation of an object from one state to another over a period of time. Typically transitions are made in space — from one position to another — but the system can also support other kinds of transitions such as from one shape to another (as shown in the squash and stretch transitions used to implement Figure 2). In general an animated presentation will be built up of a series of transitions of various objects over time. These transitions can be individually created and scheduled by the user interface, or can be created relative to each other so that the start of one transition can be automatically triggered by the start or end of another.

The notion of a transition is directly modeled in the system with a *transition object*. As illustrated in Figure 3, each transition object is defined using three components: a reference to the *interface object* moving over the transition, the *trajectory* that the object is to take, and the *time interval* over which the transition is to occur. A trajectory is further decomposed into two components: a *curve* and an optional *pacing function* which determines how quickly various portions of the curve are traversed.

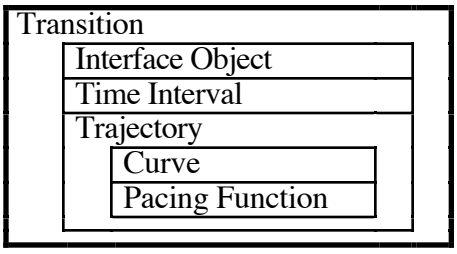

Figure 3. Parts of a Transition Object

The curve of a trajectory is implemented as a parameteric function returning a point. This point is typically in the coordinate system of the object being moved, but can also be interpreted in other ways (such as the relative trajectory of the smaller "balls" in Figure 2.). The parameter to the curve function ranges from zero to one and will be related to the passage of time. This framework allows a wide variety of curves, such as lines, arcs, splines, etc. to be expressed in a simple and compact form.

As described in Section 4, the system provides a series of parameter values that uniformly track the passage of time. If a transition with a non-uniform rate of traversal is desired, a pacing function for the trajectory can be supplied. This function non-uniformly transforms parameter values in the range zero to one back into the range zero to one. This can be used for example to implement a "slow-in / slow-out" pacing which begins slowly, runs quickly in the middle, and ends slowly, as illustrated by Figure 4. By default, the system provides a uniform pacing function which traverses the whole curve at the

† This abstraction is an extension and generalization of transition concept developed in [Stas90] which in turn was partially inspired by the *P-Curves* described in [Baec69].

same rate. By separating the pacing function from the curve of the trajectory, more complex effects are easier to manage [Lass87], and it is possible to provide a library of reusable paces which can be employed with many different actual curves or curve types.

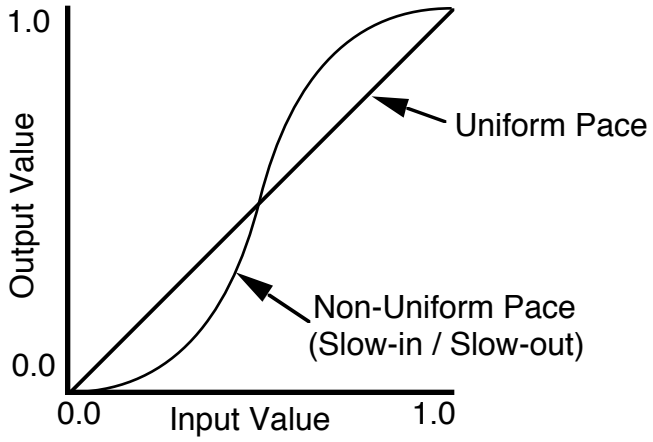

Figure 4. Two Example Pacing Functions

The next major component of a transition is the interval of time over which the transition is to occur. In its simplest form, an interval can be expressed as two specific (wall clock) times. However, this form is not typically convenient to use and a number of more flexible forms of interval expressed in relative terms can be employed as well. These include:

Interval Start Specifications	
at <time>	Start at given absolute time
in <delay>	Start <delay> from now
after-start-of <t> + <del>	Start <del> after transition <t> starts
after-end-of <t> + <del>	Start <del> after transition <t> ends

Interval End Specifications	
at <time>	End at given absolute time
in <delay>	End <delay> from now
duration <dur>	End <dur> after start

The "in" and "at" specifications each indicate relative and absolute times that can be immediately resolved by the system into a specific time for scheduling purposes. However, the "duration", "after-start-of" and "after-end-of" specifications may not be immediately resolvable into a specific time. For transitions specified with these types of intervals, the system maintains an internal system of dependencies and schedules the transition only once its full interval is resolved.

The final component of a transition is the animated object itself. This object is an Artkit *interactor* object which is placed in the normal hierarchical structure that the toolkit uses to control screen update and dispatch of translated high-level events (as a result, it can be freely intermixed with all other user interface components). If groups of objects are to be animated as a unit, they can be grouped together using normal Artkit interactor composition objects (e.g. with a simple_group object).

Using the built in extension mechanisms of the toolkit, a new high level event dispatch protocol for animation has been added to the system. Each object which wishes to respond to animation requests simply implements the following three methods:

start_transition
transition_step
end_transition

The start_transition message informs the object that it is about to begin a new transition. A series of transition_step messages then indicate how the object is to advance along the transition over time. Finally, an end_transition message is passed at the final step indicating that the transition has completed.

Each transition_step and end_transition message contains parameters that indicate both the start and end positions of the animation step being taken. In addition, the transition and trajectory objects, as well as starting and ending parameter values for the step, are provided in order to support more complex effects. Both start and end values are passed because each step corresponds not to a single point in time, but instead to a finite interval of time. The object is responsible for modifying its position and other attributes in order to create a depiction of itself suitable for that interval. Once the object has been reconfigured, the normal toolkit redraw process takes over the task of actually updating the screen.

The decomposition of animation control into the transition abstraction, and then further into a trajectory (containing a curve and pacing function), a time interval, and the animated object itself, allows the system to easily support a range of animation effects with minimal programming effort. For example, the animation illustrated in Figure 1 can be created with the following code:

```
top_trans = new transition(
    icn_obj1,
    new arc_traj(250,303, 293, 3.8,5.0),
    start_in, MSec(500), duration, Sec(4));

bottom_trans = new transition(
    icn_obj2,
    new arc_traj(250,453,293,3.8,5.0,&slowinout),
    start_after_start_of, top_trans, MSec(0),
    duration, Sec(4));

transition_agent->schedule_transition(top_trans);
```

The first statement creates a new transition object. This object controls the object pointed to by icn_obj1 (in this case a blur_anim_icon object which draws itself blurred over its current interval†). The second parameter to this transition is a pointer to a newly created trajectory object, in this case a circular arc (centered at 250, 303, with radius 293, operating from 3.8 to 5.0 radians). Since no explicit pacing function is provided, the trajectory uses the default (uniform) pacing function. The final four parameters to the transition establish its time interval. This transition is set to operate over a time interval beginning in 500 milliseconds and lasting for 4 seconds.

In general, any C++ expression can be used to compute parameter values, so more complex effects such as making transitions relative to the position of other objects are typically relatively easy to express.

The second statement sets up a similar transition for icn_obj2, this time operating on a parallel trajectory. In this case, the final parameter to the trajectory indicates an explicit (non-uniform) pacing function. This transition is scheduled to begin at the same time as the first transition and also last for 4 seconds. The final statement actually places the first transition in the scheduling queue. The second transition will be scheduled automatically when the first one starts.

The toolkit provides a library of different interactor object classes which can be animated, as well as a reusable set of trajectory and pacing function types. By mixing and matching these parameterized reusable components it is possible to support a very wide range of animation effects with a minimal amount of programming effort.

In addition, because each of the different components of the animation abstraction is isolated and performs a small well defined task, it is also easy to extend the standard library using normal object-oriented techniques.

4. ROBUST IMPLEMENTATION

The window and operating systems currently employed for user interface development (e.g., the X Window System and Unix) offer, at best, little support for applications such as animation that are time dependent. In addition, even if better support becomes available or "low-level" systems that provide more direct access to the machine are employed, there will always be cases where real-time goals cannot be met simply because too much total work needs to be done at a given point in time. As a result, a central goal of the Artkit animation support implementation has been to be robust and degrade gracefully in the presence of unexpected or variable delays.

In order to clearly demonstrate this robustness, the Artkit animation abstractions have been implemented assuming only a bare minimum of support from the underlying window and operating systems. In particular, only two capabilities are assumed (although others could be employed for better performance). First, it is assumed that the toolkit can ask what the current time is, and second, that the window system is willing to pass control back to the toolkit periodically (at least, for example, when it is idle waiting for the next input event).

This second capability is modeled inside the system by generating a series of internal *idle* events which are processed by the normal Artkit event processing framework. No specific assumption is made about how often these events occur (although clearly to produce pleasing results, they should occur at a fairly regular and

† The blur_anim_icon object implements its motion blur by drawing a fixed number of copies of itself in a speckled pattern along its trajectory at evenly spaced parameter values covering its current animation step.

rapid pace most of the time). Instead, the system is responsible for doing the best that it can under the performance situation in which it operates.

Artkit, like many other toolkits, maintains a tree structure that describes the position and appearance of all objects in the user interface. Changes to the appearance of the interface are made by modifying the interactor objects that appear in this tree. The system is then responsible for updating the actual screen appearance to correspond to these changes.

In a normal user interface, the toolkit operates in a cyclic fashion. It receives an input event, then translates and *dispatches* that event to one or more interactor objects in the form of messages. These interactor objects then respond to the (higher-level) messages by modifying their position, appearance, or state as well as invoking any application callback routines. Finally, the system updates the screen display as needed and returns to waiting for the next event. We call this the update or *redraw* cycle of the toolkit.

In order to work smoothly with other user interface components, the animation support abstraction must work within the redraw cycle. In particular, low-level idle events are received from the windowing layer of the system and translated into the start_transition, transition_step, and end_transition messages that form the protocol for the animation abstraction. In this framework we can think of animation steps as covering adjacent intervals of time roughly corresponding to redraw cycles. The goal of the system then, is to deliver animation steps with parameters that match the actually occurring intervals of time as closely as possible (this approach is conceptually similar to the *governor* used in the central animation loop of the Information Visualizer System [Robe89]). Individual objects can then respond to the intervals they are given appropriately, for example by drawing a blurred rendition of themselves spread over the interval or even by reducing their rendering detail when the interval becomes too long as described in [Robe89].

To do this, the system works as follows. Transitions are managed by the *animation dispatch agent* within the toolkit. This agent has the same form as other input handling agents in the system (such as the *drag focus* or the *snap focus* agents described in [Henr90]). As transitions are scheduled, those that have determined starting and ending times are placed in a scheduling queue (sorted by start time). As illustrated in Figure 5, when the animation agent receives control (i.e. when it receives an idle event to translate and dispatch) it asks for the current time. It then tries to make an estimate of when drawing that is started now will actually appear on the screen. This is done based on a measure of past update times as described in Section 5. Based on this estimated *redraw end time* the set of active transitions for the step is selected (i.e. the set of still active transitions that have starting times at or before the estimated end time).

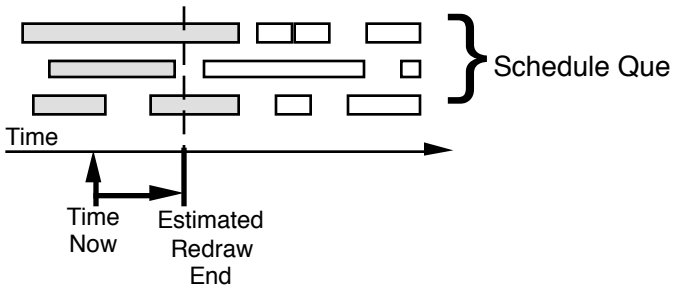

Figure 5. Active Transition Selection

For each selected transition, three sets of operations are performed (always in scheduled time order). For each transition that has not been started in a past step, the transition is marked as started, a start message is delivered to the object on the transition, and any other transitions which have requested notification through after-start-of scheduling are notified (these transitions in turn schedule themselves and may possibly be placed in the selected set and started). For each transition that intersects the estimated end time, an animation step is given as described below. Finally, all transitions that end before the estimated end time are given an end step, marked as stopped, and removed from the scheduling list. In addition each transition that has requested notification because of after-end-of scheduling is notified.

Each transition acts on animation or end steps as illustrated in Figure 6. A transition uses its start time and duration to map from the estimated end time into the corresponding position in its local parameter space (which varies from zero to one). It also maintains a record of the ending point of its previous step. These two values define the interval in local parameter space that is to be covered by this step. (The transition also insures at this point that the new estimated end is in fact after the previous step end and skips the step if it is not).

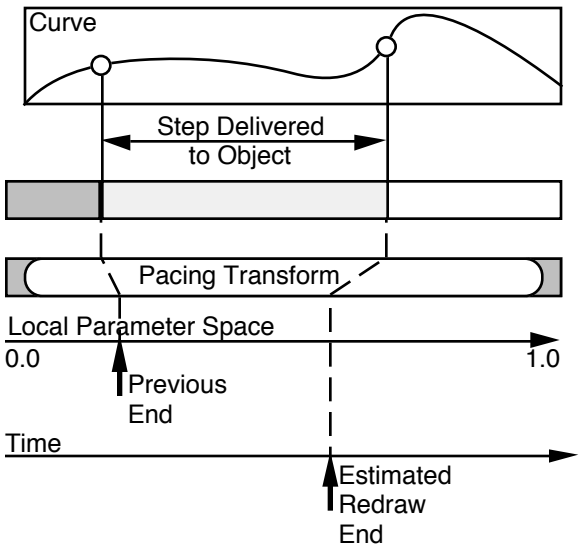

Figure 6. Translation from Time to Space

Next the parameter values are passed through the optional pacing function and mapped to screen positions using the trajectory object. Finally, if the ending screen position has changed since the last step (or this is an end step), this information is used to deliver a step message to the object.

Notice that this framework is robust with respect to unexpected or varying delays. If the animation agent receives control on a timely and regular basis, animation steps will be delivered as a set of timely and regular intervals. If delays occur, however, animation steps will still be delivered correctly, but with larger intervals. Further, even if extremely large delays occur such that whole transitions come and go during the delay, the system will still act reasonably by delivering the start_transition and end_transition messages for each of the transitions in the proper order (but without intervening transition_step messages). This will allow the semantic actions associated with these events to be carried out properly even without a graphical rendition of the transition.

Finally, note that this framework provides a form of weak synchronization guarantee. In particular, transitions which are scheduled to start (or end) at the same time will be guaranteed to start (or end) in the same redraw cycle. Again, this is consistent with the overall approach of graceful degradation and the actual accuracy will depend on the performance and timing characteristics of the underlying window and operating systems.

5. IMPLEMENTATION EXPERIENCE

In the process of implementing the system, several different forms of redraw end estimators were experimented with. The system currently measures the real-time response of the drawing portion of the redraw cycle. The current time is recorded just before the interactor tree is traversed to produce drawing updates, then, when all drawing has been completed, we force the window system to synchronize (i.e. wait until the window system server responds with a message indicating that all drawing has actually been completed) and record the time again. Although this does not capture all the delay involved (i.e. additional time is spent computing and dispatching animation steps), it seems to cover the primary source of delay that occurs when using a client-server based window system.

Several alternatives exist for using the measured drawing response time as an estimator. Which is best can depend on the particular timing goals of an application. In particular, there is a tradeoff between estimators that try to keep drawing response as close to the requested times as possible, and those that endeavor to keep the animation step sizes at an absolute minimum.

If the smallest possible step sizes are desired without regard to the timing accuracy of the step, it is possible to use a redraw end time estimator that always returns zero. This ensures that the system never over estimates redraw time, but also causes the system to deliver steps that are always one redraw cycle behind the actual time.

As a compromise solution, the system currently employs a capped exponentially decaying average. A simple exponentially decaying average gives reasonably good tracking of actual redraw times in the recent past. However, the distribution of actual delays encountered tends to include large spikes — a single delay or period of delays that is much larger than normal. To avoid having this anomalous behavior corrupt the estimator for several steps and cause a large overshoot, the system places a limit on the estimator. This approach allows the system to have a bimodal response. Under normal circumstances when redraw times are relatively small and tend to be less variable, the system uses the average estimator. However, if a single large delay occurs, the system will not overshoot and increase

the next animation step too much, and if the system response consistently falls below an acceptable threshold, this approach causes the estimator to be biased towards shorter steps rather than accurate timing.

The Artkit toolkit was originally designed to stress extensibility in its design. Our experience with this extension to the system confirmed that even large, completely unplanned for extensions like the one described here could be handled relatively easily.

Initially the toolkit had no notion of time built into it and so some modifications to the toolkit framework that went outside the normal extension mechanisms were required. In the window system interface layer of the X window version of the toolkit, 1 line of existing C++ code was changed (to force synchronization after redraw with `XSync`) and 129 lines were added (to produce idle events). In the toolkit proper, 23 lines of code had to be added to the system outside the normal extension mechanism (to measure redraw times and fix a bug that surfaced in connection with idle events). The remaining changes to the toolkit framework were done via the normal extension mechanisms. 1485 lines of code were employed in the framework proper (for the animation dispatch agent) and another 863 lines were used for a small interactor, trajectory, and pacing function class library. Finally, the "squash box" interactors used for Figure 2 were developed as subclasses of the standard library using 356 additional lines of code.

6. CONCLUSION

This paper has described how flexible, robust, and reusable animation abstractions can be smoothly integrated with the facilities of an object-oriented toolkit. The animation abstractions described are powerful and flexible, supporting techniques such as: simple motion-blur, "squash and stretch", use of arcing trajectories, anticipation and follow through, and "slow-in / slow-out" transitions. In addition, their implementation is robust — degrading gracefully in the presence of unexpected and variable delays. Finally, since these facilities are provided in the context of a full scale user interface toolkit, they are widely reusable and can be easily employed in conjunction with more conventional user interface objects.

REFERENCES

[Baec69] Baecker, R., "Picture-driven Animation", *Proceedings of the Spring Joint Computer Conference*, Vol 34, 1969, AFIPS Press, pp. 273-288.

[Baec90] Baecker, R., Small, I., "Animation at the Interface" in *The Art of Human-Computer Interface Design*, B. Laurel (ed), Addison-Wesley, 1990, pp. 251-267.

[Henr90] Henry, T.R., Hudson, S.E., Newell G.L., "Integrating Gesture and Snapping into a User Interface Toolkit", Proceedings of the *ACM Symposium on User Interface Software and Technology*, Oct. 1990, pp. 112-121.

[Lass87] Lassiter, J., "Principles of Traditional Animation Applied to 3D Computer Animation", Proceedings of SIGGRAPH '87, July 1987, pp. 35-44.

[Robe89] Robertson, G.G., Card, S.K., Mackinlay, J., "The Cognitive Coprocessor Architecture for Interactive User Interfaces", *Proceedings of the ACM Symposium on User Interface Software and Technology*, Nov. 1989, pp. 10-18.

[Robe93] Robertson, G.G., Card, S.K., Mackinlay, J., "Information Visualization Using 3D Interactive Animation", *Communications of the ACM*, Vol 36, No 4, April 1993, pp. 57-71.

[Stas90] Stasko, J.T., "The Path-Transition Paradigm: A Practical Methodology for Adding Animation to Program Interfaces", *Journal of Visual Languages and Computing*, Vol 1, No 3, Sept. 1990, pp. 213-236.

[Stas93] Stasko, J.T., "Animation in User Interfaces: Principles and Techniques" to appear in *Trends in Computing (special issue on User Interface Software)*, L. Bass and P. Dewan (eds), 1993, John Wiley.