Easily Adding Animations to Interfaces Using Constraints

Brad A. Myers, Robert C. Miller, Rich McDaniel, and Alan Ferrency
Human Computer Interaction Institute
Carnegie Mellon University
Pittsburgh, PA 15213
{bam, rcm, richm, af1x}@cs.cmu.edu
http://www.cs.cmu.edu/~amulet

ABSTRACT

Adding animation to interfaces is a very difficult task with today's toolkits, even though there are many situations in which it would be useful and effective. The Amulet toolkit contains a new form of *animation constraint* that allows animations to be added to interfaces extremely easily without changing the logic of the application or the graphical objects themselves. An animation constraint detects changes to the value of the slot to which it is attached, and causes the slot to instead take on a series of values interpolated between the original and new values. The advantage over previous approaches is that animation constraints provide significantly better modularity and reuse. The programmer has independent control over the graphics to be animated, the start and end values of the animation, the path through value space, and the timing of the animation. Animations can be attached to any object, even existing widgets from the toolkit, and any type of value can be animated: scalars, coordinates, fonts, colors, line widths, point lists (for polygons), booleans (for visibility), etc. A library of useful animation constraints is provided in the toolkit, including support for exaggerated, cartoon-style effects such as slow-in-slow-out, anticipation, and follow-through. Because animations can be added to an existing application with only a single extra line of code, we expect that this new mechanism will make it easy for researchers and developers to investigate the use of animations in a wide variety of applications.

Keywords: Animation, Constraints, Toolkits, User Interface Development Environments, Amulet.

To appear in *UIST'96: ACM Symposium on User Interface Software and Technology*, Nov. 6-8, 1996. Seattle, WA

INTRODUCTION

By providing better modularity for the software for user interfaces, the Amulet toolkit [8] achieves increased reuse and decreased code size, and makes it easier for researchers and developers to create applications. For example, in Amulet, the interactive behavior of objects can be defined entirely independently from their graphical look by attaching "Interactor" objects to the graphics. Command objects [9] encapsulate the complete information about operations, and can be hierarchically linked so each application layer can be separately defined. We have followed this philosophy in our new support for *animations* and other time-based behaviors. The goal is to make simple animations *extremely* easy to add to an interface, and still support complex animations. This is achieved by allowing the animations to be defined independently from the actions that start the animation and the graphics that are animated. Furthermore, the animation itself is modularized: the start and end values of the animation, the path through the value space, and the timing of the animation can all be independently specified.

We are able to provide this new level of modularity by using the flexible Amulet constraint system. If an *animation constraint* is attached to a slot (instance variable) of an object, then changes to that slot trigger the animation, causing the slot's value to change smoothly from its old to its new value. For example, if a slot contains 10 and is set to be 100, an animation constraint might immediately override the value and set the slot with 11, and then set the slot with a series of values up to 100 over a period of two seconds. Since there are a wide variety of parameterized animation constraints provided in the library, an animation can be added to an existing interface *with a single line of code*. All the messy details of timing, redrawing, set-up and clean-up of animations, and interactions with other parts of the system, are automatically handled by the animation constraint mechanism.

We are particularly interested in the use of animations to enhance interaction in regular user interfaces. Although animations are widely used in games and specialized visualization software ([10] has a survey), their use in conventional interfaces is limited to a few predefined cases,

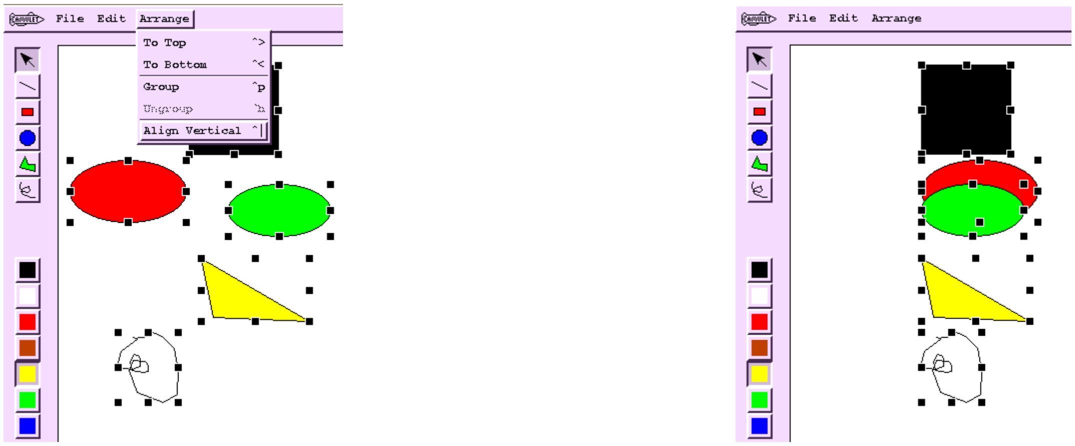


Figure 1: The objects move smoothly from their old to their new positions using an animation constraint for commands such as "Align Vertical" and when operations are undone.

such as the "expanding rectangles" effect when a file or folder is opened in the Macintosh Finder, the blinking of selected menu items on the Macintosh, and the animations in dialog boxes of Windows 95 while waiting for task completion.

If more functional animations were easier to add to interfaces, they could be used to make changes in interface objects more comprehensible, as recommended by [1]. For example, when menus or graphical objects appear, they can fade in. The effects of an Undo are often hard to detect, so animations can be used to make them more noticeable. In a multi-user interface, each user's actions can be animated so they are clearer to the other users. Amulet also has facilities that make it easy for external agents and tutors to control an interface, and animations can make their actions much easier to follow.

As a concrete example, the "Align" command of a typical graphics editor jumps the objects to their new location, which can make it hard for users to understand what happened (see Figure 1). In Amulet, by simply adding the code like the following to the position slots (`Am_LEFT` and `Am_TOP`) of the prototypes of the graphical objects (such as `Rectangle_Prototype`), the objects will now move smoothly from their old to their new positions:

```
animator = Am_Animator.Create();
Rectangle_Prototype
    .Set(Am_LEFT,Am_Animate_With(animator))
    .Set(Am_TOP,Am_Animate_With(animator));
```

Animation constraints work no matter how the new value is set into a slot; it might be due to a program setting the value, the user manipulating some object with the mouse, or a graphical constraint in the slot automatically recomputing a value. (Amulet allows multiple constraints to be in the same slot, so the animation constraint can co-exist with a "regular" graphical constraint.) Furthermore, an animation can be attached to any slot, and any *type* of slot. In addition to the obvious numerical slots like positions and sizes, animations can also interpolate colors, fonts, pictures for an icon (bitmaps), the point list of a polygon, and visibility, all without changing the graphical objects to be animated. Thus, unlike in other toolkits such as Artkit [6] and Animus [3], it is just as easy to animate pre-existing objects such as buttons from a toolkit and components from libraries as it is to animate newly created objects.

Briefly, the innovations of our approach are:

- Increased modularity since all aspects of the animations can be separately specified.
- The ability to add constraints to any type of slot without otherwise modifying the object or the interaction, including slots whose values are calculated by a graphical constraint.
- Support for animations on a wide range of *types* of values, instead of just on numbers.
- When a slot is set with a new value while it is being animated, the animation can either be interrupted when the new value arrives, or can continue smoothly through all the values.
- Built-in support for undoing of the effects of animations.
- A library of animations that can be attached to objects, including special effects such as "slow-in-slow-out," anticipation, follow-through, and various ways for objects to appear and disappear. These effects have been adapted from cartooning techniques [2] [7].

RELATED WORK

Previous systems have provided animation and temporal constraints, but none are as flexible or easy-to-use as Amulet's. The Artkit toolkit [6] provides "Transition Objects" which contain a reference to the graphical object to be animated, a trajectory along which to move the object, a time interval, and a pacing function. These are generalizations of the architecture used in [12]. For objects to be animated in Artkit, the programmer must add three special methods to the graphical objects being animated: `start_transition`, `transition_step` and `end_transition`. Amulet's animation mechanism is more general and easier to use, since it supports animations over any kind of value, not just positions, and objects can be animated without writing any new methods. Artkit also does not use a constraint mechanism to implement the animations.

A number of special animation effects were added to the Self environment [2], including motion blur, anticipation and follow-through, slow-in-slow-out, and arcing paths. Many of these are also provided by Amulet, and the others would be easy to add. However, Self implemented these effects using special-purpose code, and does not provide a way for programmers to customize or reuse them for particular applications.

The Animus system [3] pioneered *temporal constraints* in a toolkit. Animus was primarily aimed at visualizations and simulations, and did not provide any built-in mechanisms to animate user interfaces or to make it easy to animate objects. Animus provides constraints that depend on time and help with the mathematics to calculate differential equations.

TYPES OF ANIMATIONS

The next sections discuss the built-in types of animations currently provided by Amulet. It is important to emphasize, however, that the architecture makes it easy to build new types of animations as well.

Numerical Slots

The simplest type of animation constraint works on numbers, either integer or floating point. Normally, numbers will be *interpolated* between the original value of the slot and the new value. This kind of animation is used in Figure 1 to move the objects smoothly for "Align Vertical" and Undo.

In the game shown in Figure 2, the bullets shoot up from the ship at the bottom. They can be created using code like the following (0 is at the top of the window):

```
Bullet = Am_Rectangle.Create()
   .Set (Am_WIDTH, 4)
   .Set (Am_HEIGHT, 16)
   .Set (Am_FILL_STYLE, Am_Red)
   .Set (Am_LINE_STYLE, Am_No_Style)
   .Set (Am_TOP, 500)   //start at bottom of window
   .Set (Am_TOP,
      Am_Animate_With(Am_Animator.Create()
        .Set(Am_VELOCITY, 300)   // 300 pixels/sec
        .Set(Am_COMMAND, erase_bullet)));
```

This creates a `Bullet` prototype object which is a rectangle of the specified size and color located at Y = 500. Its `Am_TOP` slot has an attached animation constraint using an instance of the default animator. The program would then create instances of this prototype bullet, and

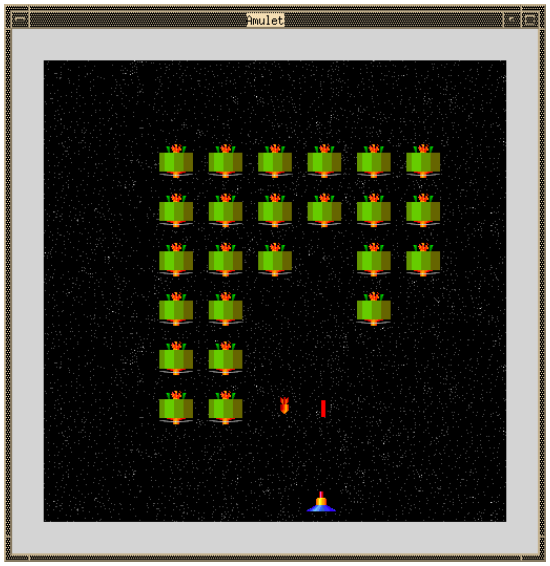


Figure 2: The ships move back and forth with a bouncing animator, and the ship, bullets going up and bombs going down are controlled by other simple moving animators.

each instance would inherit all the properties of the prototype, including the animations. To animate a bullet smoothly up the screen, the programmer just sets the `Am_TOP` to `-16`:

```
Am_Object a_bullet = Bullet.Create();
a_bullet.Set(Am_TOP, -16);
```

This triggers the animation constraint, which captures the –16 value, immediately sets the slot back to 500, and then sets the slot with a series of values to make the bullet move smoothly at 300 pixels per second towards –16. When –16 is reached, the animator executes the `erase_bullet` command (see the section on "Commands" below).

The standard numerical animator automatically calculates the frame rate and step size of the animation, in order to pass smoothly through as many intermediate values as possible. A different animator, the Stepping animator, allows the programmer to specify a frame rate and an increment amount. The Amulet scrollbar widget uses this kind of animator to support the automatic repeat when the arrow buttons are held down. A Stepping animator is attached to the scrollbar's `Am_VALUE` slot, which represents its current position. Normally this animation is disabled, so that dragging the scrollbar thumb works as usual. When the user presses one of the scrollbar's arrow buttons (or in the "trough" to specify scrolling by pages), the animator is activated and the `Am_VALUE` slot is set to the maximum or minimum value of the scroll range. The animator intercepts this change, resets the value slot back to its original position, and begins changing the value by the appropriate increment at fixed intervals, as if the user were repeatedly clicking on the arrow button. When the user releases the arrow button, the animator is disabled again. The advantage of using an animation constraint to control this behavior is that it always operates at the same rate, independent of the processor speed, so windows never scroll too quickly on fast machines.

A third numerical animator is provided to simulate behaviors typical of cartoon animation. The Exaggerated animator starts with an "anticipation" phase, taking a quick step in the opposite direction before beginning the animation. After reaching the target value, the animator does "follow-through," wiggling back and forth slightly as if to dissipate its energy. These effects, which were part of the Self user interface [2], lend realism and solidity to objects.

List of Values

Another type of animation chooses from a fixed set of values. For example, icons are often animated by cycling through a sequence of pictures. Therefore, one form of animation constraint takes a list of values and iterates through them. This might be used for an icon that should continuously change its picture (see Figure 3). Note that this animator is not limited only to bitmaps; it handles a list of any types of values. For example, the programmer might want an object to animate through a fixed set of sizes and therefore supply a list of numbers to a list animation constraint attached to the width and height slots.

The list animation constraint continuously returns the next value of the list at a specified rate. When the list animation is used on a slot which changes from an old to a new value, Amulet will try to find the old and new values in the list, and animate from the old to the new value through the list. If the old value of the slot is not found in the list, then the animator starts at the beginning of the list, and if the new value of the slot is not found, then the animator goes through the list to the end before setting the slot to the new value. The programmer can control what happens at the end of the animation, as discussed below, and options include continuously cycling through the list.

A special form of the list animator toggles between two specific values. The default values are true and false, in which case the animator might be attached to the `Am_VISIBLE` slot of an object to cause it to blink. Alternatively, the programmer might provide two colors and set this animator on the `Am_FILL_STYLE` slot, or two positions for an object to cause it to wiggle back and forth.

Amulet's novel separation of the *form* of the constraint (how the values are selected) from the specific *values* returned by the constraint is especially evident in the list animator. Also, since constraints are implemented as Amulet objects, regular constraints can be used to calculate the values used for the parameters of animations. For example, if there are different lists of pictures used for the icon of Figure 3, depending on the direction of walking, the image list passed to the animation constraint can contain a constraint that returns the correct list based on the current direction.

Figure 3: The picture for the icon can be changed using a list animation on the `Image` slot at the same time as a moving constraint controls the `Left` and `Top` slots to move the icon, which will make the figure appear to walk.

The Visible Slot

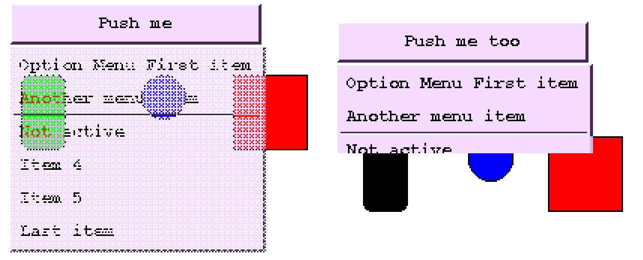
The `Am_VISIBLE` slot of a graphical object determines whether it can be seen on the screen or not. Amulet provides an animation constraint that can be attached to this slot to generate animated effects when the object appears or disappears. This animator can control any combination of properties of the object, enabling a number of interesting effects. The object does not need to be changed for these effects to work, so the Visible animator can even be attached to toolkit widgets (as shown in Figure 4) and components from libraries. Currently, we have four built-in effects for animating the object when the visible slot changes.

One effect is dissolving, shown in Figure 4a. For objects which can draw themselves with a halftone mask, the Visible animator controls the degree of halftoning to make the object gradually fade away when it disappears, and fade into existence when it appears.

Another effect is growing and shrinking, where the object shrinks from full size to zero size when it disappears, and grows back to full size when it appears. This effect might be used to make an option menu or pull-down menu unroll smoothly from its source, as shown in Figure 4b. A parameter controls where the object grows from (its center, top or some other side).

A third effect moves the object to and from a fixed location. This effect can make an object fly away to offscreen when it disappears, and fly in when it appears. We used this to add animation in a checkers game so when the pieces were captured, they fly off the screen. A constraint is used to compute the final position, so the black and white pieces fly off to the different sides. Other effects can be achieved by having the final position be onscreen. For example, one end of an arrow might "grow" out of the source object towards the destination object, as in [2].

The fourth effect takes the parts of the object and moves them away in different directions to make it look like the object is exploding. This is implemented by making a copy of the object, making the original object invisible, and then separately animating all the parts of the copy in different directions. This idea comes from the animation in the Alice system [4] for object destruction, but in Alice the animation is hard-wired and inflexible, whereas in Amulet, the programmer has complete control over what animation is used and how it appears.



(a) (b)

Figure 4: Animation constraints can be added to the `Am_VISIBLE` slot of a popup menu to make it (a) fade in using halftoning or (b) grow from the top.

Colors

Amulet provides constraints which animate through a sequence of colors by linearly interpolating between the two colors in a three-dimensional color space (RGB or HSV). Often, this will be used to fade from the current color to white or black. When the user wants to fade between other colors, different intermediate colors will appear depending on which color model is used [5]. The color animator provides a parameter to allow the programmer to choose between the RGB or HSV color models when calculating which intermediate colors to choose.

Objects in Amulet can use a filling style of "none," which means that only the border is drawn and objects underneath show through. It would not look correct for the color to animate from one color to another when the object changes from transparent to opaque, since the initial color would immediately hide any objects underneath the object, which would not be a smooth transition. Therefore, the color animator must use a halftone mask to make the new color gradually appear.

Point Lists

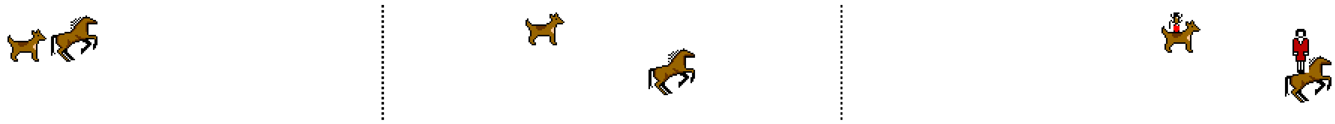
Polygons in Amulet are defined by a list of points, and animation constraints can be used for these lists as well. The point-list animator checks the differences between the new and old point lists. If either list is empty, then the object grows or shrinks to nothing from the center. If the lists are the same except for one extra point, then that point is grown out from the center of the line attaching the two adjacent points. When the point list changes dramatically, the first point in the original list is mapped to the first point in the new list, and so on, and the final point of the shorter list is repeated so both lists have the same number of points. Thus, the setup at the start of the animation ensures that the two lists being animated between have the same number of points, and the animation smoothly interpolates between each pair of corresponding points. It would be fun in the future to add a warping animator that determines an attractive way to animate between two arbitrary polygons, as in [11].

DETAILED DESIGN

The animation mechanism is implemented as a new type of constraint solver that can be attached to objects. Amulet implements a prototype-instance object model on top of C++. Each object is represented as a collection of slots (also called instance or member variables), and each slot can hold any type of value. Slots that are not set locally inherit their values from the object's prototype. Any slot of an object can hold one or more constraints, in addition to the regular value. A constraint computes the value for the slot, and will be automatically reevaluated whenever the constraint's dependencies change. Amulet supports multiple constraint solvers. Currently, there are two solvers in Amulet: a one-way constraint mechanism (called "formulas" by analogy to spreadsheet formulas) and a multi-way constraint solver. Either type of constraint can contain arbitrary C++ code, and can be attached to any slot. Amulet uses constraints ubiquitously to control the behavior of built-in objects. For example, the width of a text object is computed by a constraint dependent on the font and text of the object.

Given the normal constraint mechanisms already in Amulet, the programmer could write constraints that depend on the clock. For example the charge on a capacitor could be a function of time, as in Animus [3]. The programmer could create clock objects with a specified tick rate, and insert a constraint which depends on the clock's value into the `Charge` slot of the capacitor. The system will update the clock object at the appropriate intervals. However, this form of constraint is quite inconvenient to write, especially for the common animations and other time-based behaviors found in interactive interfaces.

To provide the commonly used animation behaviors in an easy-to-use and reusable fashion, Amulet contains a collection of *animation constraints,* implemented using a new kind of constraint solver. Any slot of any object can have an animation constraint attached to it, and the constraint will control the value of the slot. This works even if the slot's value is set by a regular constraint since slots can contain multiple constraints. For example, suppose a dog icon is constrained to be next to a horse icon using a conventional graphical constraint (see Figure 5). When the user drags the horse, the dog stays in exactly the same relative distance from the horse. If an animation constraint is placed on the dog's position slots, then the dog will lag behind the horse when the horse is dragged swiftly, and slowly catch up. This change only requires adding a single line of code to the interface, and does *not* require changing any of the existing constraints or interactions. Note that other slots that contain regular constraints are not affected, so that if a rider icon was added that was also constrained to the horse (Figure 5c), the animation constraint on the dog would not affect the rider, so the rider would stay directly on top of the horse. If an object was constrained with regular constraints to the dog,



(a) (b) (c)

Figure 5: (a) The dog is constrained to be behind the horse. (b) When an animation constraint is put on the dog, it lags behind the horse and slowly catches up if the horse is moved quickly. (c) Other objects constrained to the dog and the horse behave normally and stay attached.

such as the bug in Figure 5c, then that object would stay attached to the dog, no matter how and where the dog was moved. Thus, the animator constraints interoperate seamlessly with the regular constraints and the slot setting mechanisms.

It is often convenient to combine multiple animations. For example, the picture for the icon in Figure 3 might change as it is moved to show the legs walking. This is supported by simply putting animation constraints on multiple slots of objects. Multiple animations can even be attached to the *same* slot. For example, combining the grow-and-shrink animation and the flying-off animations on the `Am_VISIBLE` slot of a window can make the window shrink down to an icon when it becomes invisible. If the programmer specifies identical timing parameters for two animations, Amulet ensures that the constraints will both execute on the same refresh cycle, so they will stay synchronized.

Use of Objects

Each animation constraint is implemented as an *Animator object*, with its own set of slots. The Animator object contains methods for when the animation starts, for each animation step, for when the animation completes, and for when it is interrupted by a new value. Note than unlike Artkit where these methods are in every graphical object, here they are in the animators so they can be reused for a wide variety of animations. Most programmers will not need to implement new methods, but can choose from the built-in animators.

If an instance of an object is created, Amulet automatically creates instances of any constraints on that object, including any animation constraints. Therefore, the programmer can easily create a prototype object with all the required animation constraints, and then all instances will have the appropriate animation constraints.

Timing

One of the most important parameters to an animation is its timing. In Amulet, the programmer can separately control the duration of an animation, the average rate of change (speed), and the frame rate.

The duration of the animation can be either *fixed* in advance, or *derived* from the magnitude of the change to be animated. Fixed durations are useful for feedback effects, where the purpose of the animation is to illuminate a change in the interface without unduly interfering with the user's interaction. For example, the following code configures a menu to fade in using a Visibility animator with a fixed duration of 500 milliseconds:

```
menu.Set(Am_VISIBLE, Am_Animate_With
          (Am_Visible_Animator.Create()
               .Set(Am_USE_FADING, true)
               .Set(Am_DURATION, 500)));
```

Derived durations, on the other hand, are useful when the animation itself is the focus of interest, and every change should occur at a fixed rate. By default, Amulet allows the programmer to specify an average rate of change for the animation. Here is an example of an animator that moves the object `r1` in two dimensions at a fixed rate of 300 pixels per second:

```
Am_Object animator = Am_Animator.Create ()
                       .Set(Am_VELOCITY, 300);
r1.Set(Am_LEFT, Am_Animate_With(animator))
  .Set(Am_TOP, Am_Animate_With(animator));
```

Note that "velocity" applies not only to screen coordinates (where it naturally represents an object's velocity in pixels per second), but to other types of values as well. For colors, the rate of change represents the distance traveled through the color model per second; and for values chosen from a list, it represents number of list items traversed per second. If neither fixed duration nor fixed rate of change is desirable, the programmer can easily substitute a constraint formula that computes a different duration and rate of change.

The frame rate controls how often the animation stepper runs. By default, Amulet computes a frame rate that visits every intermediate value during the animation (but no faster than once every 25 milliseconds, to avoid wasting CPU cycles on frame rates beyond the limits of human perception). The scheduler attempts to schedule each animation step as close to the specified time as possible, so that simultaneous animations remain in sync. Amulet also guarantees that all animations defined on the same time step will operate before the graphics are refreshed, to avoid flicker.

Pacing and Path Functions

Like Artkit, Amulet separates the timing of an animation, called the *pacing function*, from the parameterized path it follows, called the *path function*. This separation is desirable not only for modularity, but because designers often want to tweak timing parameters independently from path parameters [7].

The pacing function maps the current elapsed time of the animation to the path parameter (a real value between 0.0 and 1.0). Pacing functions are used to accelerate and decelerate along the path. Amulet provides two built-in pacing functions: linear pacing and slow-in-slow-out pacing, which starts slowly, accelerates to full speed, and then brakes to a stop at the end of the animation. These pacing functions can be used with any type of animation, so colors can be animated using slow-in-slow-out, if desired.

The path function interpolates between the starting and ending values of the animation. It takes a parameter between 0.0 and 1.0, and maps it to a value between the starting and ending value along some parameterized path. As described above, Amulet supplies path functions that linearly interpolate scalar numbers, screen coordinates, colors, line widths, etc. In the future, we plan to add path functions for interpolating along arcs and splines in screen coordinates. Programmers can also define custom path functions which interpolate between arbitrary types of values.

Triggering Animations

Amulet provides a number of mechanisms for triggering animations. Animations can be triggered by a slot change, by an explicit procedure call, or by a user input event. Each animation can be enabled or disabled by setting the `Am_ACTIVE` slot of the animation constraint to true or false. When `Am_ACTIVE` is true (the default), any change to the animator's dependent slots starts the animation. When `Am_ACTIVE` is false, no animation occurs.

Sometimes an animation needs to be active most of the time, but disabled briefly for particular operations. For these operations, a special flag can be passed when a slot is set that prevents animations attached to the slot from triggering. For example, Amulet's Move_Grow_Interactor uses this flag so objects being dragged with the mouse do not lag behind due to animations.

Animations can be explicitly started and stopped by procedure calls, which gives the programmer direct control. This is useful for adapting animators to other purposes, such as background timers.

Finally, animations can be controlled by user input events. The `Am_START_EVENT` slot on the animator specifies the input event that should trigger the animator, and `Am_STOP_EVENT` specifies the input event that should stop it. For example, in Figure 2, the ship should move while the left or right arrow keys are held down.

Interrupting Animations

An important innovation in Amulet is that animations can be easily interrupted while they are running. By default, whenever a slot containing an animation constraint is set with a new value (either directly or indirectly through formulas), the animation is immediately interrupted, and begins animating from the current value to the newly set value. This means that the user remains in control, and can set any value at any time, even when animating. If the `Am_INTERRUPTIBLE` slot is set to false, then all the values set into the slot are queued, and the animation goes through each value in turn. For example, suppose a slot originally contains 100, and is set with 10, and then immediately set with 200. If `Am_INTERRUPTIBLE` is false, the value will smoothly go from 100 down to 10 and then up to 200. If `Am_INTERRUPTIBLE` is true, the value 10 will be overridden by the 200, and the slot will animate from 100 to 200. The programmer might find it convenient to set `Am_INTERRUPTIBLE` to false so the entire animation can be quickly specified at initialization time, by simply setting the slot with all of the desired values.

Commands on Animation Completion

Rather than using a "call-back" procedure for when a widget is operated as in most toolkits, Amulet instead allocates a "command object" and calls its `Do_Method` [9]. The advantages of command objects are that they also support various forms of Undo, the enabling and disabling (greying out) of the operation, providing help, and saving all information needed to execute or undo the command. Animation constraints also have attached commands. The programmer can write a custom `Do_Method` for the command object, or choose a command object from the Amulet library. One choice is a command to start a new animation, as in Artkit. However, the Amulet model is much more flexible. Other built in commands include:

- **Animation_Wrap_Command**, which causes the animation to start over again. This is useful for animations that should run continuously, such as the animated icons of Figure 3.
- **Animation_Bounce_Command**, which reverses the animation as if it bounced. This is useful for objects which should move around in a box, bouncing off the walls, such as the alien ships of Figure 2.
- **Delete_Object_Command**, which destroys the object being animated. A game might use this command for objects that should disappear when they leave the screen, such as the bombs and bullets of Figure 2.

A custom `Do_Method` is used in the circuit program of Figure 6. When the value of the input one-zero buttons are toggled by clicking on them, small text objects are created which move along the wires. When the animation is complete, the `Do_Method` of the attached command deletes the text objects used for the animation, and then sets the value of the wire, which causes the values for the receiving gates to change. This might, in turn, trigger subsequent animations down the next gate's wires, so the values flow correctly.

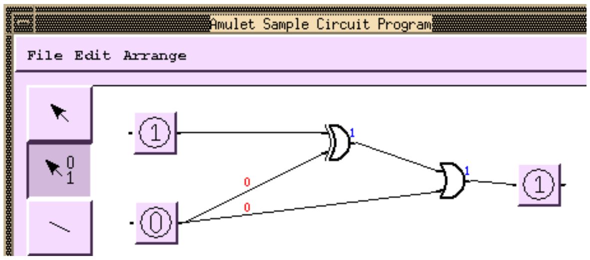


Figure 6: A circuit program that uses animations. The lower button was just changed to zero, and the small zeros are moving along the wires.

Designing New Animations

If a programmer cannot find the desired animation in the Amulet toolkit, the architecture supports several ways of creating new animation constraints. In the simplest approach, the programmer can create a new path function or pacing function and install it in a standard animator. This approach was used to implement the color animator, which differs from the standard numerical animator only in its path function. Path functions and pacing functions can have arbitrary side-effects.

For more control over an animation's operation, the programmer can override methods on the animator object. The `Start` method typically sets up the animation, and determines the initial value for the slot that the animation constraint is attached to. For example, in the Visibility animator for exploding, the `Start` method makes a copy of all the parts of the object, chooses a direction of motion for each one, and returns the value `false` for the `Am_VISIBLE` slot it is attached to, since the original object should be invisible while the animation is proceeding. The `Animation` method of the animator is called periodically to perform the animation. The default `Animation` method just calls the associated timing and path functions. There are two methods used for ending an animation: the `Stop` method for when the animation terminates normally, and the `Abort` method for when a new value arrives while the animation is in progress. These methods perform clean up activities, such as deleting the copies created by the exploding Visibility animation. The `Stop` method also returns a final value for the attached slot, which is usually the value the slot was originally set to. The method does not need to specify a value, since the animation will be restarted with the newly set value.

It is important to note that although these methods are similar to those of Artkit, in Amulet they are attached to animation objects, not graphical objects, so they can be more easily reused. Furthermore, in Amulet most programmers will not need to write these methods, whereas in Artkit, methods must be written for every animation.

UNDO OF ANIMATIONS

By default, animators are not undoable, since the most common case is that they are side effects of normal commands which are already queued for undo. For example, if the user moves an icon of a horse, and this causes the icon of a dog to animate to a new position near the horse (Figure 5), then the original move command will already be queued for undo, and undoing the move command will cause the animation to be undone automatically. An interesting property of having animations triggered by setting a slot is that Undoing an action automatically causes the animations to run, without any further programmer effort. For example, in Figure 5, the dog icon will animate back to its former position if the move of the horse is undone. Note that the animation is not run backwards. Most animations are symmetric, so it won't matter, but any anticipation and follow-through will still happen at the beginning of the animation, even during an undo.

Sometimes it might be useful for the animation constraints' actions themselves to be undoable, especially when they are not the direct consequence of a user's actions. Since the final action of an animator is to execute a Command, the standard Amulet mechanisms can be used to Undo the effect of the animation [9]. The programmer only needs to clear the "Not Undoable" flag of the animator command, or use one of the standard commands from the library, such as the `Delete_Object_Command`, and the action will be queued in the standard undo history.

DEBUGGING

Debugging animations is often quite difficult. Therefore, Amulet includes a number of features to make this easier. When an object is being animated, the programmer can point the cursor at it and hit a special key to pop up Amulet's "Inspector" window. This will temporarily pause all animators, and show which slots of the object have animators attached to them. Commands in the Inspector will resume the animation, or single-step it. The programmer can select any animator constraint and inspect which slots it uses, and its other properties. If the object does not contain any animators, but is constrained either directly or indirectly to an object which does, then the Inspector has commands to trace all the constraint dependencies of a slot. Another common problem is that the programmer sets up an animation, but it does not appear to operate. In this case, the programmer can trace that animator, and the Inspector will print a trace of all of the animator's actions, including why it did not start, or if it did start, then all the slots it sets. Often, the problem is that the animator is attached to the wrong object, or is running too fast to be seen. If it appears that a slot is being set erroneously, the programmer can trace all the animators and constraints that use the slot, or set a breakpoint for the next time the slot is set. All the parameters of the animators themselves can also be viewed and changed at run time using the Inspector.

IMPLEMENTATION

Using Constraints

The Amulet object system provides three mechanisms that make animation constraints possible. First, it allows new constraint solvers to be added into the system without changing the existing solvers. Second, slots can contain more than one constraint at a time. And third, there is a general queuing mechanism for storing procedures to be executed at a later time.

The constraint solver for animation constraints is very simple compared to other constraint solvers. When a slot changes value, the constraint sends a message to the animator object, passing the new and old values of the slot. The animator can specify a different value for the slot to be used instead of the new value. This value will replace the current value and all regular constraints dependent on the slot will be re-evaluated. This all happens before the graphical object is redrawn so the object is never seen by the user with an incorrect value.

Amulet's general queuing interface is called the *demon queue* because the procedures it stores are typically "demon" procedures which perform background chores for objects. For example, a demon is used to redraw graphical objects when their slots change. Normally, a constraint would not be able to immediately change the value of a slot it is attached to, since the value would be ignored. Therefore, the animation constraints enqueue a special demon in the demon queue. This demon is invoked later in the update cycle after the regular constraints have finished, and the slot values are valid. The demon will find the animation constraint and cause its value to become the value for the slot. Next, any required regular constraints that depend on the slot are re-evaluated. This separation in time allows the animation constraint to have the final say in determining the slot's value.

Timing Events

Amulet's main event loop was modified to include an animation scheduler, which uses a priority queue to deliver timer ticks to running animations. Since operating systems like Unix and Windows NT are prone to unpredictable delays due to swapping and network activity, timer ticks may not necessarily occur at the regular intervals specified, so our animations are designed to interpolate correctly regardless of the actual timer resolution. Under heavy loads, the animations degrade gracefully, showing fewer intermediate values.

Updating the Graphics

Other systems [2] have achieved high frame rates by colormap cycling techniques, which reserve portions of the colormap for fast rendering. In Amulet, by contrast, we animate objects simply by drawing them repeatedly. Although slower, this technique handles user-drawn graphical objects, is portable to hardware without a colormap, and avoids restricting the color space available to Amulet programmers. We use double-buffering to avoid flicker.

Performance

Animation constraints have two forms of overhead:

- Architectural overhead: the cost of using the constraint system to trigger and run animations, and the cost of separating timing functions from path functions.
- Redraw overhead: the cost of redrawing graphical objects whenever they change (rather than using colormap cycling or other techniques).

Architectural overhead was measured in isolation by animating a slot of a non-graphical object, which never has to redraw itself. On a SparcStation 20, simple linear interpolation had an architectural overhead of about 0.2 milliseconds. Redraw overhead was measured by moving a rectangle as fast as possible, without any delay between moves. On the same SparcStation 20 using a local X server, the redraw overhead was roughly 1.5 milliseconds with double-buffering turned on. The times on a Macintosh PowerMac 8500/150, the times are almost identical: 0.2 for the architectural overhead, and 1.9 milliseconds for the drawing.

Therefore, the system can in theory simultaneously animate about 20 independent objects at a high frame rate (20 to 30 frames per second) on current hardware. In our experimental tests, 10 independently moving objects works fine, but 20 causes jumpy movements. These numbers suggest that we should concentrate on the redrawing overhead, since the animation and constraint overheads do not seem to be the big problem. Note that in Figure 2, all of the alien ships are part of a single group which is being animated, so there are actually only 5 independently moving objects (the alien ships, the ship at the bottom, up to 2 bullets, and a bomb). Performance in this game is excellent.

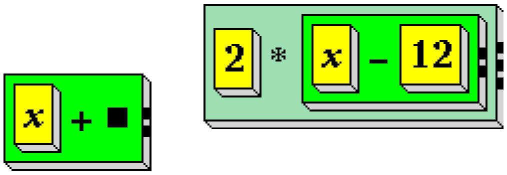
STATUS AND FUTURE WORK

The animation constraints described in this paper have been implemented in the new version of Amulet, which we expect to release for anonymous FTP by the end of the year.[1] As shown in the figures throughout this paper, we modified a few of the built-in widgets to use animations, and built a small game (Figure 2). We added animations to an existing checkers game with only 5 additional lines of code: one line to make the checkers move smoothly, and four lines to make them fly off the board in different directions when captured.

---

[1] Amulet is available for free by anonymous FTP. To get the system, see the WWW page:
`http://www.cs.cmu.edu/~amulet`
or send mail to `amulet@cs.cmu.edu`.

(a)



(b)

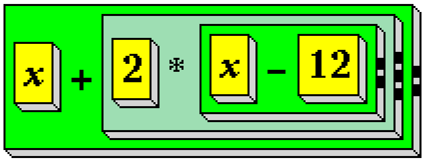


Figure 7: A program to help children learn how to factor mathematical expressions built in Amulet by Rob Miller and Neil Heffernan. (a) When the upper tile representing “2*(x-12)” is dragged to the black square in the “(x+ ■)” tile, (b) the tile grows smoothly to make room for it.

There is plenty of interesting future work that this architecture enables. We would like to expand the library of built-in animations to include squash-and-stretch of objects when they bounce. We need to make it easier to attach animations to events that are specified procedurally, such as creating and destroying objects, or adding them to a group. This requirement came up when the second author added animations to a project for a class, shown in Figure 7. We will also be working on decreasing the overhead of the animations so more animations can be run in parallel on slower machines.

Finally, we would like to create interactive tools for associating animations with objects by demonstration without writing any code at all.

CONCLUSIONS

Animation constraints make it very easy to add smooth transitions and other animations to interfaces created using Amulet. They also make it easy to create animated games. The architecture for animation constraints provides better modularization and reuse than previous toolkits because Amulet separates the graphical objects, the initial and final values for the animation, the path through the value space, and the timing function. The result is that animation methods can be easily combined and attached to existing objects and widgets. The animation constraints can support all of the flexibility provided by previous toolkits, while providing a significantly simpler interface to the programmer, which will hopefully make animations much more popular and effective in future interfaces.

ACKNOWLEDGMENTS

For help with this paper, we would like to thank Brad Vander Zanden and Bernita Myers. This research was sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326, and partially by NSF under grant number IRI-9319969. Robert Miller was supported in part by a Department of Defense NDSEG Fellowship. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

REFERENCES

1. Baecker, R. and Small, I., “Animation at the Interface,” in *The Art of Human Computer Interface Design*, B. Laurel, Editor. 1990, Addison-Wesley. p. 251-267.

2. Chang, B.-W. and Ungar, D. “Animation: From Cartoons to the User Interface.” in *UIST'93: Symposium on User Interface Software and Technology*. 1993. pp. 45-55.

3. Duisberg, R A., *Constraint-Based Animation: Temporal Constraints in the Animus System.* PhD Thesis, Dept. of Computer Science, Univerity of Washington, 1986. Also Tektronix Laboratories Technical Report CR-86-37, August, 1986.

4. Durbin, J., Gossweiler, R., and Pausch, R. “Amortizing 3D Graphics Optimization Across Multiple Frames.” in *UIST'95: Symposium on User Interface Software and Technology*. 1995. pp. 13-18.

5. Foley, J.D., *et al.*, *Computer Graphics; Principles and Practice, Second Edition*. 1990, Reading, MA: Addison-Wesley Publishing Company.

6. Hudson, S.E. and Stasko, J.T. “Animation Support in a User Interface Toolkit: Flexible, Robust, and Reusable Abstractions,” in *UIST'93: Symposium on User Interface Software and Technology*. 1993. pp. 57-67.

7. Lassiter, J. “Principles of Traditional Animation Applied to 3D Computer Animation.” in *Proceedings SIGGRAPH'87*. 1987. pp. 35-44.

8. Myers, B.A., *et al.*, *The Amulet V2.0 Reference Manual*, . 1996, Carnegie Mellon University Computer. Science Deptartment Technical Report Number CMU-CS-95-166-R1.

9. Myers, B.A. and Kosbie, D. “Reusable Hierarchical Command Objects,” in *Proceedings CHI'96: Human Factors in Computing Systems*. 1996. Vancouver, Canada. pp. 260-267.

10. Price, B.A., Baecker, R.A., and Small, I., “A Principled Taxonomy of Software Visualization,” *Journal of Visual Languages and Computing*, 1993. **4**(3): pp. 211-266.

11. Sederberg, T.W., *et al.* “2-D Shape Blending: An Intrinsic Solution to the Vertex Path Problem.” in *Proceedings SIGGRAPH'93*. 1993. pp. 15-18.

12. Stasko, J.T. “Using Direct Manipulation to Build Algorithm Animations by Demonstration,” in *Proceedings CHI'91: Human Factors in Computing Systems*. 1991. New Orleans, LA. pp. 307-314.