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 animations to interfaces continues to be 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 new mechanisms that allow changes to graphical objects to be animated without changing the basic logic of the application or the graphical objects themselves. This is achieved by attaching animation constraints on of objects that detect when the slots are set and immediately capture the new value and cause the slot to instead take on a series of values interpolated between the original and new values. The advantage over previous approaches is that the animations are significantly more modularized: the programmer has independent control over the graphics to be animated, the start and end values of the animation, the path through the value space, and the timing of the animation. This allows animations to be attached to any object, even existing widgets from the toolkit, and on any type of value: numbers (for the coordinates of objects), fonts, colors, point-lists (for polygons), booleans (for visibility), etc. Built-in animators support special "cartoon-like" effects such as slow-in-slow-out, and exagerated motion such as anticipation and wiggling, which can also be attached to different types of animations.

1. INTRODUCTION

The philosophy of the Amulet toolkit [AmuletManual] is to modularize and separate the different aspects of the user interface. Thus, the interactive behavior of objects can be defined entirely independently from their graphical look by attaching "Interactor" objects to the graphics. Command objects [commandObjects] encapsulate the complete information about operations, and can be hierarchically linked so each layer of an application can be separately defined. We have just extended this philosophy to support *animations* and other time-based behaviors, which can be defined entirely 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 through the use of the flexible Amulet constraint system. Animations can be triggered by the changing of a slot (instance variable) of an object and can cause the slot's value to instead change smoothly from its old to its new value. For example, as shown in Figure *1*, normally, the "Align" command in a graphics editor pops the objects to their new location, which can make it hard for users to understand what happened. By simply adding the following code to the position slots of the prototypes of the graphical objects, without any further changes to the application or its interface, the objects will now move smoothly from their old to their new positions:

```
Graphical_Obj_Proto.Animate(LEFT, TOP,
    Smooth_Interpolator.Create())
```

The animation is entirely independent of the way the new value was set into the 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. Furthermore, the animations can be defined on any slot, and any type of slot. In addition to the obvious numerical slots like positions and sizes, colors, fonts, pictures for an icon (bitmap), the point-list of a polygon (to support changes of shape), and visibility can all be animated without changing the graphical objects to be animated. This means that, unlike in previous toolkits such as Artkit [**] and Animus [**], it is just as easy to animate pre-existing objects such as buttons from a toolkit and components from libraries.

The advantage of this is that user interface designers can freely use animations in interfaces to make changes more understandable. For example, when menus or graphical objects pop up, they can fade in. Affects of an Undo are often hard to detect, so animations would make them more noticable. In a multi-user interface, the other users' actions could be animated so they are clearer. Similarly, when tutors, agents, scripting mechanisms, or other external applications operate the interface, animations would help make the user aware of what happened. In addition, the animation constraints make games much easier to write.

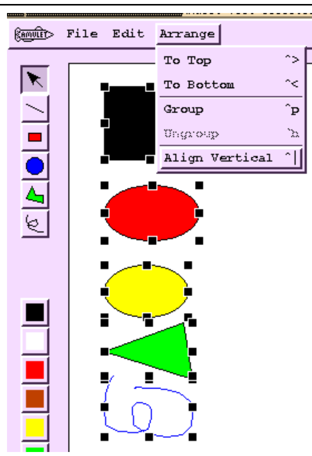


Figure 1: The objects move smoothly from their old to new positions using an animation constraint.

In summary, the innovations of our approach are:

- Increased modularity and the ability to add constraints to any type of slot without otherwise modifying the object or the interaction.
- When a slot is set with a sequence of values while the animation is in progress, the animation can either be interrupted when the new value arrives, or else can continue smoothly through all the values.
- Support for animations on all *types* of values, instead of just on numerical values.
- Undoing of the effects of animations.

Other features that are supported, which have appeared in previous systems, include:

- Support for special effects in the animation, such as "slow-in-slow-out", exagerated motion such as anticipation and wiggling. These have been adapted from cartooning techniques [**].
- Animations of object creation and destruction.
- Implementation of the animation using the constraint mechanism.

2. RELATED WORK

The Artkit toolkit [uist93**] provides "Transition Objects" which contain a reference to the graphical object to be animated, a trajectory to move the object, a time interval, and a pacing function. These are generalizations of the architecture used in Tango [stasko, jvl1,3]. For objects to be animated in Artkit, the programmer must implement three special methods: start_transition, transition_step and end_transition. Amulet's animation mechanism is much more general and easier to use, since it supports animations over any kind of value, not just positions, it allows animations to be based on a constant speed and not just a constant time, it supports arbitrary actions at the end of animations, 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 [** UIST'93**], including motion blur, anticipation and wiggling, slow-in-and-slow-out, and arcing paths. Many of these are also provided by Amulet, and the others would be easy to add. However, Self does not provide an architecture whereby programmers can use these effects for their code.

The Animus system [**duisberg**] pioneered providing *temporal constraints* in a toolkit.

Animus was primarily aimed at visualizations and simulations, and did not provide mechanisms to animate user interfaces or to make it easy to animate objects. Animus provides constraints on time, and help with the mathematics to perform differential equations.

3. TYPES OF ANIMATIONS

A unique aspect of the animation facilities in Amulet is that the they can be attached to almost any slot of any object, without any changes to the object itself. Previous systems, such as Artkit, have required that the graphical objects themselves be modified to accept special animation methods. Instead in Amulet, appropriate animation constraints can simply be attached to the slots of any object, which provides greater modularity and re-use of the animation methods. Furthermore, the *style* of animation can be separately specified from the type of the slot. The next sections discuss the built-in types of animations provided by Amulet. It is important to emphasize, however, that the architecture makes it easy to build new types of animations as well.

3.1. Number Slots

Different kinds of animation constraints are used for different types of values. The simplest type of constraint calculates 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. Similarly, to animate the bullets smoothly up the screen in Figure 2, the programmer only needs to set the top of the bullet to –10.

Alternatively, the programmer can specify an increment amount, and then the value will continuously change (at the specified rate as described in section ***). The `End_Action` slot controls what happens when the animator reaches the target value, and it can bounce, wrap or stop, as described in Section ***. The "bad" ships at the top of Figure 2 use this kind of animator to go back and forth.

A special effect that comes from cartooning is the support for "anticipation" at the beginning of the animation and "wiggling" at the end. This is provided as a option for moving objects.

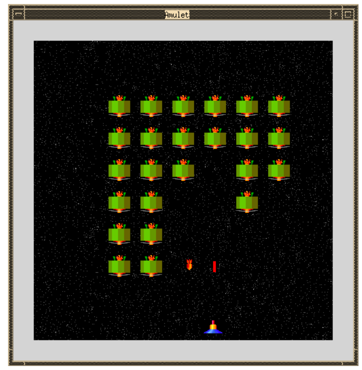


Figure 2: The ships move back and forth with a bouncing animator, and the ship, bullets and bombs are controlled by other simple moving animators.

3.2. List of Values

A frequent type of animation chooses from a fixed set of values. For example, when animating the picture in an icon or bitmap, there will be a fixed set of pictures that the programmer will want to choose from. Therefore, a form of animation constraint will take a list of values, and iterate through them. Note that this animator is not limited to only bitmaps; it will handle a list of any types of values. For example, the programmer might desire 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.

One form of the list animation constraint continuously (at a specified rate) returns the next value of the list. This might be used for an icon that should continuously change its picture. Another form is used to interpolate from a start to a final value using the values in the list as intermediate values. In this case, the animator searches for the start and final values in the list, and if found, then iterates through the items in between. If the initial value is not found, then the iterator starts at the beginning of the list, and if the final value is not found, then the animator goes through the end of the list. The `End_Action` slot again controls what happens when the animator reaches the end of the list, and it can bounce, wrap or stop.

Sometimes the values in the list should be used in a specific order, different from the order they are stored in the list. In this case, the programmer can supply a second list of indices into the main list. If the list in the

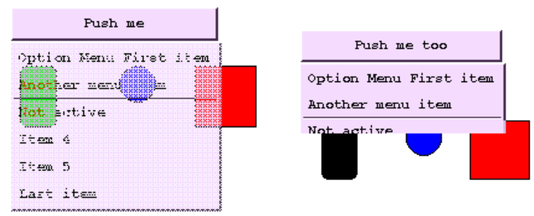
`Indices_For_Animation` slot is non-empty, then the animator will iterate through this list, using each value as an index into the first list. For example, the animation of the cup of coffee on Sun's Java page (`java.sun.com/applets`) starts off using the indices: "`1|2|3|1|2|3|4|5|...`". This slot is also useful when the main list of pictures contains values for many modes of an object, such as when it is moving in different directions. In this case, the `Indices_For_Animation` slot will often contain a constraint dependent on the mode or direction of the object, and return an appropriate list. For example, (in pseudo-code):

```
//GV means to get a slot's value, and set up
// a constraint on that slot.
if (self.GV(DYING))
   return (16,17,18,19,20,21);  //explode
else switch (self.GV(DIRECTION)) {
   case UP: return (0,1,2,3);
   case DOWN: return (4,5,6,7);
   case LEFT: return (8,9,10,11);
   case RIGHT: return (12,13,14,15);
}
```

A special form of the list animator toggles between two specific values. This is most commonly used for blinking an object. The default values are true and false, in which case the animator might be attached to the `Visible` slot of an object. Alternatively, the programmer might provide two colors and set this animator on the `Fill_Style` slot, or two positions for an object 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 are especially evident in the list animator.

3.3. The Visible Slot

The `Visible` slot of a graphical object determines whether it can be seen on the screen or not. A number of animators have been defined that can be attached to this slot, and will be automatically fired when the slot changes value. One is a fader that uses halftoning to make it appear that the object is slowly fading away, or fading back in. This is implemented by redirecting the object's drawing to an offscreen buffer, and then masking the copy back to the screen with the appropriate half-tone. The object does not need to be changed for this to work, so it can be easily applied to objects from a toolkit (as shown in Figure 3a) and components from libraries.



(a) (b)

Figure 3: Popup menu (a) fades in using halftoning or (b) grows by adding different animation constraints to the `Visible` slot.

Three other constraints have been defined for the `Visible` slot. One grows the object from zero to its current size, or back down to zero. This might be used to have an option menu or pull-down menu smoothly grow down from the source, as shown in Figure 3b. A parameter controls where the object grows from (its center, top or some other side).

Another animator moves the object from its current location to a specified point when the object becomes invisible, or back when the object becomes visible. By default, the location is offscreen, which causes the object to appear to "fly" off when it becomes invisible and fly to its location when it becomes visible. Some objects may use a different value for the location however. For example, an arrow might "grow" out of the source object towards the destination object. The final animator for the `Visible` slot takes any parts of the object and flies them in different directions to make it look like the object is breaking apart. This is based on the animation in Alice for object destruction [**], but in Alice the animation was hard-wired and inflexible, whereas in Amulet, the programmer has complete control over what animation is used and how it appears.

When an object is newly created, and first mapped onto the screen, the system checks to see if there is an animation constraint attached to the visible slot, and if so, runs the constraint as if the object changed from invisible to visible. Similarly, if an object is destroyed while it is currently visible on the screen, the animation constraint in the visible slot is run first.

3.4. Colors

Amulet provides constraints which will animate through a sequence of colors. 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 [**Foley**]. Therefore, Amulet provides a parameter to allow the programmer to choose between animating through 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 right for the color to animate from one color to another when the slot changes from none to a color, since the initial color would immediately hide any objects underneath the object, which would not be a smooth animation. Therefore, Amulet uses the fading animator for the `Visible` slot to allow the objects underneath to slowly fade away when the fill-style slot changes from none to a color.

3.5. Point-Lists

Polygons in Amulet are defined by a list of points, and animator constraints can be used for these. If a new point is added to an existing pointlist, then the new point can grow out of the position perpendicular to the line between the two points around the new point. Alternatively, the new point can "grow" out of the previous or subsequent point, which is especially useful when points are added to the beginning or end of the list. When the point-list goes from empty to a new set of points, the programmer might want to fade the new polygon in, using the animators for the `Visible` slot, as described above.

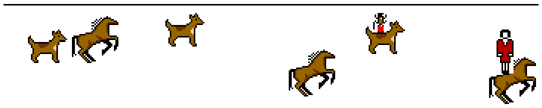
When the point list changes dramatically, an animation can map the first point in the first list to the first point in the second list, and so on, and then grows or removes any left-over points from the last point. It would be fun in the future to add a warping animator that determines an attractive way to animate between two arbitrary polygons, as in [**Sederberg, SIGGRAPH'93**].

4. DETAILED DESIGN

The new Animation mechanism is implemented as a new type of *constraint solver* that can be attaced 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 values from the object's prototype. Any slot of an object can hold a *constraint*, in addition to the regular value. A constraint is some code that computes the value for the slot, and will be automatically re-evaluated whenever anything the constraint depends on changes. Amulet supports multiple constraint *solvers* [****]. Currently, there is both a one-way constraint mechanism (called "formulas" by analogy to spreadsheet formulas) and a multi-way constraint system in Amulet. Either type of constraint can be computed using arbitrary C++ procedures, and can be attached to any slot. Amulet uses these constraints ubiquitously to control the behavior of built-in objects. For example, the width of a string is computed by a constraint dependent on the font and text of the string.

Given the normal constraint mechanisms already in Amulet, it would be easy to write a constraint that depends on the clock, for example to have the value of the charge of a transistor be a function of time, as in Animus [**]. The programmer could create clock objects with a specified tic rate, and then insert into a `Charge` slot of the transistor a constraint which depends on the clock's value. The system will schedule the clock object to be updated 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.

Therefore, we have added a collection of special *Animation Constraints* as a new kind of constraint solver, that provides all of the common animation behaviors in a very easy-to-use fashion. Any slots of any objects can have an Animator constraint attached to it, and the constraint will control the value of the slot. Typically, when the slot is set, the constraint will fire and immediately return the value to one close or equal to the original value, and then slowly animate the value to the newly set value. This works even if the slot's value is being set by a regular constraint. For example, suppose a dog icon is constrained to be next to a horse icon using a conventional graphical constraint (see Figure 4). When the user drags the horse, the dog stays in exactly the same relative distance from the horse. Now, 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 person icon was added that was also constrained to the horse, the animation constraint on the dog would not affect the person, so the person would stay directly on top of the horse. If an object was constrained with regular constraints to the dog, such as the bug of Figure 4, then that object would stay exactly attached to the dog, no matter how and where the dog was moved. Thus, the animator constraints interoperate seamlessly with the regular constraints and slot setting mechanisms.



(a) (b) (c)

Figure 4: (a) The dog is constrained to be behind the horse. (b) When an animation constraint is put on the dog, it lags behind 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.

It is often convenient to combine multiple animations. For example, the dog picture might change as it is moved to show the legs moving. This is trivially supported by simply putting animation constraints onto multiple slots of objects, for example the picture and position slots of the icon.

Each animation constraint is implemented as an *Animator object,* with its own set of slots. The default values of these slots are often correct, but if not, the programmer can easily adjust the behavior of the animator by setting the slots. Alternatively, it is sometimes useful to compute the slot values for animators using regular constraints, and as shown in some examples below.

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 (such as the prototypes from which the graphical objects in Figure 1 are created), and then all instances will have the appropriate animation constraints.

4.1. Timing Functions

One of the most important parameters to an animation is the control of the timing. In Amulet, the programmer can control the timing in a number of ways. One choice is for animations to take a fixed amount of *time*, by default, 1 second, independent of the distance. This means that the objects will change at different rates or speeds. Alternatively, the programmer can define a fixed *speed*, in which case the animations will take differing amount of time depending on the distance. Note that this distance is not necessarily a screen distance. When animating colors, the distance will be through the specified color model. The only requirement is that distances in the same model should be comparable.

Independently of these parameters, the programmer can also control how often the animation stepper runs. By default it is every 100 milleseconds, which means that the objects are updated 10 times a second, but the programmer can control this individually for each animation. As described in section ***, the scheduler tries to schedule each animation as close to the specified time as possible, and guarantees that all animations defined at the same time step happen together.

The timing function that operates over these values is also controllable, as in Artkit. The "timing-method" of each animator maps the incoming timing parameterization of 0.0 to 1.0 back to the same range. The default mapping is the identity transformation, but other effects are easily specified. Another built-in timing method implements "slow-in and slow-out," which makes the animation look more realistic for moves.

***** Talk about the general form of the timing function here*****

4.2. Path functions

*** **DEFINE HOW TO PROGRAM NEW PATH FUNCTIONS. SHOULD WE CALL THESE "PATH" FUNCTIONS SINCE THEY SEEM MORE GENERAL?**

4.3. Other parameters

`Active`: The animations have a number of other parameters that are under the control of the programmer. First, each animation can be enabled or disabled by setting the `Active` slot of the animator to true or false. Often, this slot is controlled by a constraint. As in Artkit, by default, the Interactor that moves objects and changes their size can temporarily set the `Active` slot to false for the objects and slots it directly sets, so that objects will stay attached to the mouse.

`End_Action`: When the animator completes and sets the slots to their final values, the programmer has a choice of what action to take. The default is to stop, and call any associated actions, as described in section ****. Another choice is to "wrap" and start the animation over again. This is often useful for animations that should run continuously such as changing the picture of an icon. The final option on completion is to "bounce," which means to run the values backwards (without repeating the final value). This is often useful for objects which should move around in a box, bouncing off the walls, as in the ships of Figure 2. This `End_Action` parameter applies to *all* types of animations, and therefore provides significant flexibility to the programmer.

`Interrupt`: An important innovation 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 `Interrupt` slot is set to false, however, then all the values set into the slot are queued, and the animation goes through each slot in turn. For example, if the slot originally contains 100, and then is immediately set with 10, then 200, the value will smoothly go to 10 and then to 200. If `Interrupt` is true, however, the 10 value will be overridden by the 200, and the slot will animate from 100 to 200. The programmer might find it convenient to set `Interrupt` to false so the entire animation can be quickly specified at initialization time, by simply setting the slot with all of the desired values.

`Smooth`: By default, the animations go directly through the values specified. If `Smooth` slot is set to true, however, the animators fit a spline through the specified points and travel through the new, smoother path. This also applies when the slot is set with a series of values. If `Interrupt` is true, then the value immediately curves towards the new value, and if it is false, then the value will smoothly go through all the values specified.

`Start_Event` and **`Stop_Event`:** Most animations are triggered by the setting of a slot, and should operate whenever their `Active` slot is true. However, some animations should be triggered by particular input events. For example, a bullet of Figure 2 should be created and start moving whenever the space key is hit, and the ship should move while the left or right mouse buttons are held down. Therefore, optional parameters to the animation are the start and stop events that should control them. If the start-event is omitted, then the animation is triggered by the slot being set. If the stop-event is omitted, then the animation runs to completion. This parameter is used to implement internal system functions, like the auto-repeat of scrolling for scroll bars when the user holds the mouse down over the button. An animator starts (after a certain pause) on mouse down over the button and terminates on mouse up. This makes the auto-repeat time-based, so it operates at the same speed independent of the speed of the particular computer.

5. COMMANDS ON ANIMATION COMPLETION

When an animation completes, it executes a "Command" object [***], which optionally registers the action for undo (see section ***). The programmer can override this command object with an custom action, or with a command object chosen from the Amulet library[**]. One choice is a command to start a new animation on the completion of this one, **??as in Animus, Artkit??**. However, the Amulet model is much more flexible. For example, it is often useful to execute the Delete-Object command at the end of an animation, such as for animated bullets which should be destroyed when the animator moves them off the screen. Another example is when characters die, they often first go through an animation. In this case, instead of deleting the object, the programmer can simply set the `Active` slot to true of an animator that goes through the appropriate pictures, and then executes the Delete-Object command on completion. The command object associated with an animator can even be computed by a constraint, so for the list-animator shown above in section ***, the command slot might contain the following, and then the programmer need only set the DYING boolean slot of the animator to cause the object to animate to its destruction:

```
if (self.GV(DYING))
   return Delete_Command;
else
   return NULL;
```

6. UNDO OF ANIMATIONS

By default, animators are not undoable, since the most common case is that they are side effects of normal commands that 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 4), then the original move command will already be queued for undo, and undoing that command will cause the animation to be undone automatically. An interesting property of having the animations triggered off of slot setting is that the Undoing of an action automatically causes the animations to run, without any further programmer effort. For example, on undo of the horse move, the dog icon will animate back to its original position.

In addition, sometimes it might be useful for the animation constraint's 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 [***]. The programmer only needs to clear the `Not_Undoable` slot of the animator command, or use one of the standard commands from the library, such as the Delete_Command, and the action will be queued in the standard undo history.

7. 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 the "Inspector" window. This will temporarily pause all animators, and show which slots of the object have animators attached to them. 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 so fast as to be invisible. 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 slots of the animators themselves can also be viewed and changed at run time using the Inspector.

8. IMPLEMENTATION

* Runs on all 3 architectures

* Unlike SELF, must run on all hardware, so can't use color map animation

* Uses built-in double buffering for smoothness.

Just draws the objects multiple times and relies on underlying Amulet drawing mechanisms to be fast enough.

* Uses unix select and calculates when next interrupt needed; on NT does ***, on Mac, *****

* Modified Main-event-loop to check for timing events: queued in a priority queue by start time,

as in Artkit

* Like Artkit, large delays from the OS still result in correct values, but loss of the animation

8.1. Using Constraints

**** How constraints are used to implement the animations.**

8.2. Performance Evaluation

**** How fast the constraints run; how much overhead; how many animations on Unix, on a PC.**

9. 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 this summer. As shown in the figures, we modified a few of the built-in widgets to use animations, and built a small game (Figure 2), which only required about **300** lines of C++ code. We added animations to an existing checkers game to make the checkers move smoothly and fading in and out with only 3 additional lines of code.

There is plenty of interesting future work that this architecture enables. We would like to expand the library of built-in animations to include morphing of polygons, squash-and-stretch of objects when they bounce, and motion blur. We will also be working on decreasing the overhead of the animations so more animations can be run in parallel on slower machines.

10. 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.

ACKNOWLEDGEMENTS

****

REFERENCES