Constraint-Based Animations

Allan Heydon Greg Nelson

Digital Systems Research Center
130 Lytton Ave., Palo Alto, CA 94301
{heydon,gnelson}@pa.dec.com

1. Introduction

This paper describes a technique for producing animations using constraints. With the increasing popularity of multimedia applications, animations are becoming more common. However, they are fairly tedious to produce. Constraints allow animation developers to easily construct constrained figures that they can manipulate and experiment with freely. As we will describe, it is then a simple matter to animate the figure by relating it to time.

Our approach is based on two principles. First, primitive animations can be constructed by drawing under-constrained figures and identifying a degree of freedom with time. Second, animations can be combined in sequence and in parallel to produce more complicated animations.

2. Composing Animations

We represent an animation as a pair `(p,dur)`, where `dur` is the animation's duration in seconds, and `p` is a procedure (actually a closure) for drawing the frames of the animation. For any `t` in the interval $[0, \texttt{dur}]$, the call `p(t)` draws the frame for time `t`. If `a` is the animation `(p,dur)`, then `Play(a)` repeatedly invokes `p(t)` for values of `t` from 0 to `dur`. The actual number of frames depends on the time required by `p` to draw each frame.

Sequential composition is provided by `Seq`: the animation `Seq(a1,a2)` has a duration equal to the sum of the durations of `a1` and `a2`, and its frames are the frames of `a1` followed by the frames of `a2`. Parallel composition is provided by `Co`: the duration of `Co(a1,a2)` is the common duration of `a1` and `a2`, and its `t`'th frame is the sequential composition of the `t`'th frame of `a1` with the `t`'th frame of `a2`. There are also operators for stretching, padding, reversing, and repeating animations, but we won't describe them here.

We've used the Juno-2 constraint-based drawing editor [3] for our animation experiments. One of the novel things about Juno-2 is that it allows constraints to be defined in a powerful declarative extension language. This language makes it easy to define two-dimensional geometric constraints, which occur frequently in interactive graphics; but it is not limited to two dimensions or to geometry. Because Juno-2's embedded language has closures, we were able to implement `Play`, `Seq`, and `Co` in Juno-2 itself.

3. An Example

Figure 1 shows some frames from an animation of a balanced binary tree rotation. To produce this animation, we began by drawing the under-constrained sketch shown in Figure 2. This shows the "skeleton" of the figure; it is easy to add the nodes x, y, and z, and the subtrees A, B, and C. One of the skeleton's degrees of freedom has been constrained to coincide with the position of the slider: the user can manipulate either the slider or the skeleton, and the other changes correspondingly.

In Juno-2, a constraint is a predicate on program variables. Solving a constraint is the action of finding values for the unknown variables that make the predicate true. Unlike one-way constraint solvers such as Delta-Blue [2] and Garnet [4], Juno-2 can solve cyclic constraint systems using numerical methods. Moreover, constraint definitions in Juno-2 are declarative, rather than imperative.

We now develop the constraints for the tree skeleton. First, we relate the tree's angle factor `ang` $\in [-1, 1]$ to the time value `t` $\in [0, 1]$. The constraint is:

```
ang + 1 = 2 * t
```

Next, we constrain the segment `xy` to be centered at `pivot`, and to have its angle constrained by the angle factor `ang`:

```
pivot = Geometry.Mid(x, y) AND
Len1 = Geometry.Dist(x, y) AND
ang * MaxAngle = Angle(x, y) AND ...
```

In order, the conjuncts constrain the `pivot` to be the midpoint of the segment `xy`, the distance between `x` and `y` to be the value of the constant `Len1`, and the angle (in radians) of the segment `xy` relative to the x-axis to be `ang` times the constant `MaxAngle` (equal to $\pi/4$ in this case).

Next, we constrain the points `top` and `bot`:

```
top = (ang,0) REL (pivot, y) AND
bot = (ang,0) REL (pivot, x) AND ...
```

The expression `(u,v) REL (p,q)` is the point with coordinates u and v in the coordinate system in which p is the origin and q is the tip of the unit x-vector. Hence, these constraints cause `top` and `bot` to be linearly interpolated between the points `x` and `y` in opposite directions.

Finally, we add constraints to position the subtree B:

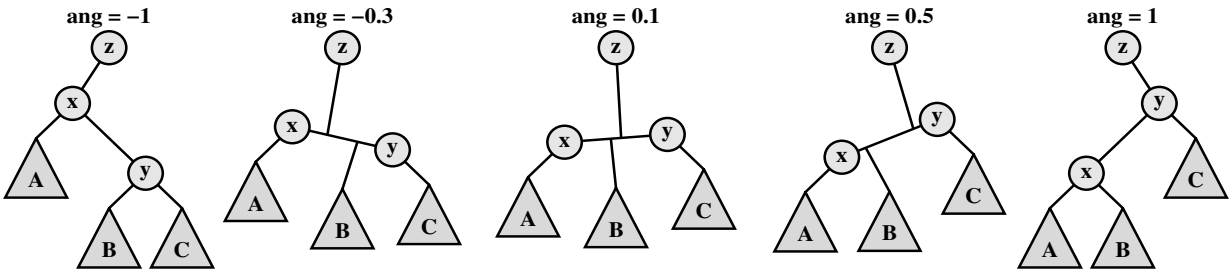

Figure 1: A simple animation of a binary tree rotation.

```
pivot VER b AND
Len2 = Geometry.Dist(bot, b) AND ...
```

The subtree B is constrained to move along the vertical line through the `pivot`, but it is also constrained to be the fixed distance `Len2` from the point `bot` as that point moves along the segment `xy`.

Up until the inclusion of these last two constraints, the other constraints could have been solved functionally. But these constraints introduce a cycle, and so they require a general purpose constraint solver. Of course, it's possible to compute the location of `b` functionally from `pivot` and `bot`, but doing so is more complicated than simply asserting the necessary constraints.

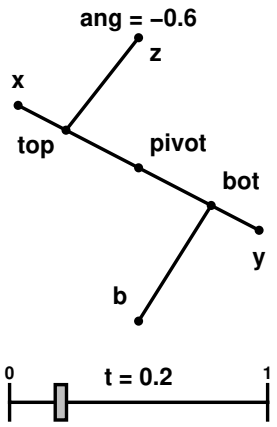

Figure 2: The constrained skeleton used to produce the animation of Figure 1.

4. Conclusions

Constraints offer several advantages. Foremost among them is that locations can be specified declaratively. This frees the user from the tedious task of computing coordinates explicitly. Our technique of relating an under-constrained figure to time saves users from having to write explicit code to set the figure in motion.

The ability to compose more complicated animations from simpler ones allows the animation developer to partition the problem and think in simpler terms. For example, to animate the man's and woman's part of a dance step, it is enough to write each half independently and then combine them to run in parallel. Similarly, each part can be subdivided into a sequence of simpler steps.

So far, we have produced animations of algorithms, of geometric theorems, and of dance steps. We have also used the Zeus algorithm animation system [1] to drive our animations. This allows us to implement the animated algorithm in another language like Modula-3 or C, and to use the constraint-based system to implement the algorithm's animated views.

We were able to produce animations using Juno-2 with minimal modifications to the system. Since Juno-2 wasn't designed for producing animations, we encountered some inconveniences in the course of our experiments. For example, entering constraints involving scalars and composing animations with `Seq` and `Co` both require some typing. However, our techniques show enough promise that it would be worthwhile to incorporate them into a more special-purpose interface for creating animations.

References

[1] Marc H. Brown. Zeus: A system for algorithm animation. In *Proc. 1991 Workshop on Visual Languages*, October, 1991. Also available as Research Report 75, Digital Systems Research Center, Feb., 1992.

[2] Bjorn N. Freeman-Benson, John Maloney, and Alan Borning. An incremental constraint solver. *Communications of the ACM*, v. 33, no. 1, pp. 54–63, Jan., 1990.

[3] Allan Heydon and Greg Nelson. The Juno-2 Constraint-Based Drawing Editor. Research Report 131a, Digital Systems Research Center, Dec., 1994.

[4] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Ed Pervin, Andrew Mickish, and Philippe Marchal. Garnet: comprehensive support for graphical, highly interactive user interfaces. *IEEE Computer*, v. 23, no. 11, pp. 71–85, Nov., 1990.