Connectors

Andrew Witkin
15-860 Spring 94

Implementing a constrained particle system was relatively simple because all the objects are the same. Now we have rigid bodies as well as particles, and someday we might want to add new kinds of objects. Naturally we'd like to implement things in a nice modular way.

As I said in class, we can do this by adding a new kind of entity, called a *connector*, that represents a material point on an arbitrary object. (By "material point" we mean a point that moves with the object.) The idea is that forces and constraints can act on the point without knowing anything about the object that the point lies on.

Generically, the job of a connector is to calculate the world-space position and velocity of the point it represents, and also the derivatives of position and velocity with respect to each of the object's state variables. As we'll see, these quantities are all we need to apply forces and impose constraints.

A point-on-rigid-body connector would know the constant body-space position of the point it represents, and have access to the body's orientation, θ, and center of mass $\mathbf{c}$, as well as the time derivatives $\dot{\theta}$ and $\dot{\mathbf{c}}$. It would use these quantities to compute the world-space position and velocity, and their derivatives with respect to c_x, c_y, and θ. (You should have no trouble working out the formulae.)

A particle connector would be a degenerate object whose position and velocity are just the particle's, and whose derivatives are all ones and zeros. (If you're clever about it you can use the particle itself as a connector, rather than implementing a separate object.)

The position and velocity calculations are straightforward, but let's look at exactly how we want to handle the derivatives. We can assume that each state variable of each object has been assigned a global index by the system (so a given rigid body's center of mass and orientation might occupy positions 16, 17, and 18 in the global state vector) and that the connector knows what the indices are.

Here are all the operations a connector has to handle:

- p() -> position in world space, a 2-vector
- pdot() -> velocity in world space, a 2-vector
- nvars() -> n, number of state vars on which pt depends
- idx(i) ->global index of ith var, $0 \leq i < n$ (you can think of i as the var's *local* index within the object.)

  dp(i) -> deriv of pos w.r.t. variable i (local index), a 2-vector

  dpdot(i) -> deriv of vel, as above.

First let's look at forces: the generalized force on a state variable q due to a force $\mathbf{f}$ applied at a world-space point $\mathbf{p}$ is just the dot product $\mathbf{f} \cdot \partial\mathbf{p}/\partial q$ (the derivative of a vector w.r.t. a scalar is a vector.)

So here's how to apply a force $\mathbf{f}$ to the point using the above operations, assuming that there's a global *generalized force* vector $\mathbf{Q}$, instead of local force accumulators per object:

```
for i from 0 to p.nvars()-1
        Q[p.idx(i)] += dot(f, p.dp(i))
```

(Here we're assuming that the functions are components of the struct p that represents the connector, as in C++). Simple, huh?

Now for constraints. The constraint object itself can be exactly like the constraints you implemented for the tinkertoy system, except for the way the elements of the **J** and $\dot{\mathbf{J}}$ matrices are computed. Since the constrained point's position and velocity are functions of state variables, rather than state variables themselves, we use the chain rule to calculate the derivative of the constraint function C w.r.t. each state variable on which it depends. If C depends on world-space point **p**, then C's derivative w.r.t. state variable q, through **p**, is $\partial C/\partial \mathbf{p} \cdot \partial \mathbf{p}/q$.

If C depends on more than one connector (e.g. a distance constraint) then it might depend on q through more than one of them, so the total derivative of C w.r.t. q has to be summed over all the connectors. Also, C can be nonscalar (e.g. to attach two points together we constrain the vector difference between them to be zero — really 2 constraints bundled into one.)

Here are all the methods a constraint object needs to handle:

- nout() -> n, number of outputs the constraint has (1 if it's scalar)
- idx(i) -> global index of the ith output, $0 \leq i < n$
- c(i) -> value of ith component of c.
- cdot(i) -> value of $cdot$
- nconns() -> number connectors it depends on
- conn(i) -> the ith connector
- dc(i,j) -> deriv of $c(i)$ w.r.t. $j$the connector, a 2-vector
- dcdot(i,j) -> deriv of $cdot(i)$ w.r.t. jth connector

Now here's totally generic pseudocode for calculating C's contribution to **J**:

```
for i from 0 to c.nout()-1 //  constraint outputs
  for j from 0 to c.nconns() -1 //  connectors
     for k from 0 to c.conn(k).nvars() // state vars
        J [c.idx(i)][c.conn(j).idx(k)]  // global idx of J element
           +=
          dot (c.dc(i,j), c.conn(j).dp(k)
            )
```

Here's the pseudocode for calculating $\dot{\mathbf{J}}$:

```
for i from 0 to c.nout()-1 //  constraint outputs
  for j from 0 to c.nconns() -1 //  connectors
     for k from 0 to c.conn(k).nvars() // state vars
        Jdot [c.idx(i)][c.conn(j).idx(k)]
           +=
          dot (c.dcdot(i,j), c.conn(j).dp(k)) +
          dot (c.dc(i,j), c.conn(j).dpdot(k))
```