CS229 Project 4: Dynamics and Interaction

Assignment Out:	Oct. 16th, 2000
Assignment Due:	Oct. 27rd, 2000 (11:59 pm)

1 Introduction

Physically based modeling has been an extremely active area of research in computer graphics for over a decade. Fueled by parallel research in robotics and computational physics, the field aims to model physical phenomena by replicating the underlying dynamics of motion, rather than just the kinematic results (as with most animation techniques). One large advantage of a dynamical model is that it can be interactive in real-time, yielding a wide range of believable results in response to spontaneous user input. This assignment will examine what is involved in creating an interactive physical model.

2 Assignment Overview

In this assignment you will be implementing a system for physically simulating an articulated chain of rigid bodies and interacting with it. Your formal goal is a mouse-controllable simulation of the desklamp Luxo.[1] Since we aim to create an animated physical character, we'll refer here to the simulated articulation as our 'actor'.

Since the mathematical details of the dynamics model have been covered in class and prior handouts, this document will focus on implementation details: what exactly you'll need to do and what we've given you to help. First, let's consider what's involved:

- representating the Actor, both physically and visually
- solving the equations of motion (to find out how the system is changing over time)
- state integration (to take those changes and apply them)
- contact model (to find contact forces)
- spatial algebra routines (to aid in the above)
- user input (to let the user control the actor)

[1] Inspired by Laszlo et. al.'s SIGGRAPH 2000 paper, "Interactive Control for Physically-Based Animation".

State integration, the contact model, and user input are provided in the support code. It is up to you to complete the definition of an Actor that can be both physically simulated and graphically displayed and to write the forward dynamics code. In the terms of the Spatial Vectors handout, writing the forward dynamics code means collecting external forces and joint torques to fill in $\mathbf{Q}$, and then using $\mathbf{Q}$, the current state $\mathbf{q}$ and $\dot{\mathbf{q}}$, and your inverse dynamics code to solve for $\ddot{\mathbf{q}}$.

3 What You Actually Need To Do

The flow of execution within your program will involve taking user input, figuring out internal and external forces on the actor's bones ($\mathbf{Q}$), solving the equations of motion for the accelerations ($\ddot{\mathbf{q}}$), and integrating to the next timestep. To put all this together, you'll need to implement the following:

1. **Fill in the spatial algebra routines.** The support code provides stubs for some spatial algebra routines that may be helpful. Flesh out the framework provided (`SpaceVector.java` / `SpaceTransform.java`).

2. **Create a framework for representing physical articulations.** Create spatial joint axes and inertia tensors, create and update spatial transforms between joints in the hierarchy, track external forces on each bone, and create a corresponding scenegraph (`Actor.java` / `Bone.java`). Specifically, for each joint or bone in the hierarchy, you'll need to account for the following:

   - spatial joint axis in local frame ($\hat{s}_i^l$) – the type of degree of freedom this joint provides
   - spatial transforms between neighboring joints (${}_i\hat{X}_{i-1}$, ${}_{i-1}\hat{X}_i$) and from the global frame (${}_i\hat{X}_{-1}$)
   - spatial inertia tensor in local frame ($\hat{I}_i^l$)
   - external spatial forces in global frame ($\hat{f}_i^x$)
   - corresponding Java3D scenegraph objects
   - generalized position ($\mathbf{q}_i$) and velocity ($\dot{\mathbf{q}}_i$)

3. **Evaluate the dynamics and solve the equations of motion.** Take the articulation you've built above and figure out accelerations on it given the forces applying to it.

4. **Try things out on the test Actors.** Test your dynamics code and Actor implementation using the particle and rigid body test Actors provided. You will get substantial partial credit for getting dynamics working for even the simplest test Actor, so give these a try before you launch into building Luxo.

5. **Luxo implementation.** Create the articulation for Luxo, set up a WorldBox scene, and create the PD controller for Luxo joint torques.

3.1 Code Layout

Here's a rundown of the code we've given you:

- Viewer framework – `Viewer.java`, `TrackballCamera.java`, `GroundPlane.java`, `ActorLoader.java`. Gets the window up and supplies a basic interface. The Viewer loads and instantiates an Actor-derived class specified on the command line, with the default being Luxo. These classes shouldn't require any modifications.

- spatial algebra templates – `SpaceVector.java`, `SpaceTransform.java`. Spatial algebra routine stubs for you to fill in, modelled after the idioms of javax.vecmath.

- contact model – `Plane.java`, `VolumeBox.java`, `WorldBox.java`. These classes provide the core of the penalty-method for contact forces as described in class. The approach represents bodies in the world as solid rectangular prisms, and can evaluate a contact force on one prism moving against another using the penalty method. You'll need to create a VolumeBox (or WorldBox if you'd like it to have visible geometry) for each rigid body you'll be creating, and loop through to accumulate forces on each bone.

- constants – `Constants.java`. Utility class for reading named constants from a file (`constants.txt`), to hasten tuning your simulation by not getting the compiler involved; feel free to play with the variables and add more as you see fit.

- articulated body chain framework – `Actor.java`, `Bone.java`. An Actor is a linked chain of Bones, each of which represents a single joint, possibly with attached geometry. This is the core of the simulation, handling forward dynamics and state integration. Subclasses of `Actor` are responsible for defining an actual set of Bones and joints, as well as any interaction mechanisms. These classes are probably where you'll spend most of your effort.

- example cases to help you debug your dynamics – `TestParticle1D.java`, `TestParticle2D.java`, `TestBar.java`, `TestBar2.java`. Diagnostic actors of increasing complexity meant to help you debug your dynamics. The test actors aren't interactive; they supply their own internal forces which, if everything goes according to plan, should result in the motion described in each file. Which actor gets instantiated by the viewer is specified at the command line, so feel free to try the test cases out in the demo ('`run TestParticle1D`', etc.) for a reference of what you should expect to see.

- beginnings of a controllable Luxo – `Luxo.java`. This class has the basics of an `Actor` that can respond to mouse input. There are two inner classes which accomplish this, `Interactor` and `PDController`. `Interactor` handles the nitty-gritty of responding to mouse move events, normalizing mouse coordinates to (0,0)...(1,1) and passing them to a `PDController` for each axis. The `PDControllers` are where you should implement the proportional-derivative mechanism proper.

To help you find your way around, the support code is almost fully javadoc'd. This is what the funny comments like '`/**...@param...*/`' are. Run the `doc` script to generate HTML docs in the html/ subdirectory. Feel free to maintain these javadocs (or not) as you code.

In debugging your dynamics, it will probably be handy to print out the values of various results as you recursively compute your inverse dynamics. To help with this, the support code supplies a few numerical integrators – Euler and Runge-Kutta. Runge-Kutta provides more accuracy and stability, but makes repeated calls to `Actor::calcAccel()`, which will make it harder to make sense of your debug output. Switching over to Euler (which makes a single call to `calcAccel()` per time step) may expedite debugging. See `Actor::simulate()`.

3.2 Bonus Project

First of all, if you've gotten the base spec assignment up and going, congratulations. You'll probably play around doing backflips for awhile and marvelling that all this math is actually working out... But if you want to go further, there are lots of interesting directions to head from here. Depending on the scale, some of these might make excellent final projects:

- **Extend your model to full 3D.** You've probably noticed that everything we're doing is 3D anyway: mostly what's missing is adding more DOF types to Bone.java and figuring out a control scheme.
- **Controllers.** The physical veracity of dynamic simulation is both an advantage and a drawback: control mechanisms have to be smart enough to work in a true 3D world, with all the fault-tolerance and flexibility that entails. It's not an easy task; roboticists have been working on this problem for decades. You might try creating a keyboard interface as described in Laszlo et. al. [2] This would be pretty easy, and would just need a mechanism for binding control input vectors to keys. Going even further, you could try some of the more exotic means for developing controllers, like writing a walking algorithm or even evolving one. [3]
- **Branching hierachies.** The current model only allows for linear chains of bodies. Extend this to allow branches and create a more complex character that takes advantage of a branching heirarchy.
- **A better contact model.** The penalty-method contact model is pretty clearly the weakest part of this simulation design. More intricate techniques involve subdividing time to find the exact momemt of collisions and then applying impulses or resting forces as neeeded. You might also look into preventing collisions between bodies in the articulation. This might be a good final project.

[2]This interface very closely mimics one used in the videogame *Die By The Sword*, by the way – in the game, you're a sword-wielding barbarian, and numbers on the numeric keypad correspond to quadrants where you can point your sword.

[3]See Karl Sims's incredible SIGGRAPH 94 paper "Evolving Virtual Creatures" in the course packet.

4 Handing in

Run "`/course/cs229/bin/cs229_handin dynamics`" from your project directory.

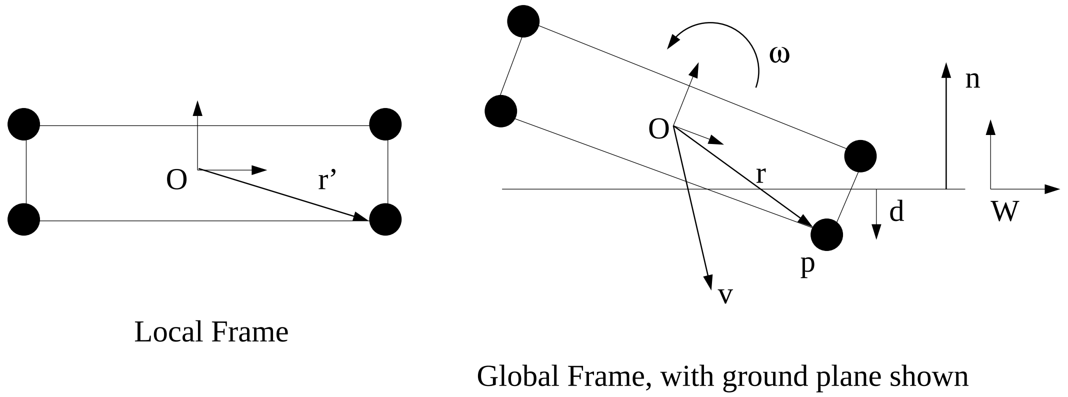

Figure 1: Parameters used in the contact model.

Appendix: The Contact Model

The contact model provided in the support code uses a simple penalty method:

- Forces are applied to any portion of the character that is below the ground plane.
- Normal forces are computed based on ground penetration distance and normal velocity.
- When tangential velocity is large, tangential (friction) forces are computed assuming the Coulomb model of sliding friction.
- When sliding velocity is small, tangential (friction) forces are computed using a damper model.

To simplify collision detection, we model every link as a collection of points. In Figure 1, point p is below the ground plane, and so a force will be applied at point p to push it back out.

The normal force treats the ground as a combination spring / damper. Given the ordinary 3x3 transformation matrix from the body to the world frame, ${}_W T_O$ and other parameters as shown in Figure 1, ground penetration distance d is computed as follows:

$$\mathbf{r} = {}_W T_O \, \mathbf{r}' \quad (1)$$
$$\mathbf{p} = \mathbf{O} + \mathbf{r} \quad (2)$$
$$d = \mathbf{p} \cdot \mathbf{n} \quad (3)$$

Parameter $\mathbf{n}$ is a unit normal, and the ground plane is assumed to pass through the origin. Negative d values are associated with points below the ground plane.

Velocity of point p in the normal direction is:

$$\mathbf{v}_p = \mathbf{v} + \omega \times \mathbf{r} \quad (4)$$
$$v_{p,n} = \mathbf{v}_p \cdot \mathbf{n} \quad (5)$$

Parameters ω and $\mathbf{v}$ might be obtained from the upper and lower components of spatial velocity $\hat{v}$ for the link, but because $\hat{v}$ is in the local link coordinate frame, these quantities would have to first be transformed into the world frame using transform ${}_W T_O$. Assuming that $\hat{v}.\omega$ and $\hat{v}.v$ access the rotational and linear velocity components of $\hat{v}$:

$$\begin{aligned} \omega &= {}_W T_O \; \hat{v}.\omega && (6) \\ \mathbf{v} &= {}_W T_O \; \hat{v}.v && (7) \\ & && (8) \end{aligned}$$

Normal force is:

$$\mathbf{f}_n = (-k \; d - b \; v_{p,n}) \; \mathbf{n} \tag{9}$$

where k is ground stiffness and b is damping. Ground stiffness has to be set fairly high to avoid excessive penetration with the ground, but not so high that it requires an excessively small integration timestep for stability.

Tangential force is always in the direction of the tangential velocity. If velocity is large, it is computed using the Coulomb model of sliding friction:

$$\begin{aligned} \mathbf{v}_{p,t} &= \mathbf{v}_p - v_{p,n} \; \mathbf{n} && (10) \\ \mathbf{f}_t &= \mu |\mathbf{f}_n| \; \frac{\mathbf{v}_{p,t}}{|\mathbf{v}_{p,t}|} && (11) \end{aligned}$$

where $|\mathbf{a}|$ represents the magnitude of a vector $\mathbf{a}$.

If velocity is small, we use a damper effect to ramp velocity slowly to zero. This avoids sudden switches in velocity direction:

$$\mathbf{f}_t = \mu |\mathbf{f}_n| \; \frac{\mathbf{v}_{p,t}}{VEL\_MAX} \tag{12}$$

Finally, the contact force must be applied to the link at point p. The easiest way to do this may be to specify an external spatial force vector in global coordinates:

$$\begin{aligned} \mathbf{f} &= \mathbf{f}_n + \mathbf{f}_t && (13) \\ \hat{f}^x &= \begin{bmatrix} \mathbf{f} \\ \mathbf{p} \times \mathbf{f} \end{bmatrix} && (14) \end{aligned}$$

A pure force $\mathbf{f}$ applied at point p is equivalent to the same force $\mathbf{f}$ applied at the origin, plus an applied torque equal to $\mathbf{p} \times \mathbf{f}$. This spatial force vector must be transformed back into the link coordinates before it can be used in the inverse dynamics equations, just as indicated in the Spatial Vectors handout.