Project 1: Particle System

The Animation of Natural Phenomena

Due 10/12

In this project you will implement a particle system with constraints. You must implement at least the required features. You must also record a video artifact of your system in action. The class will vote on the best artifacts, and the top three winners will receive extra credit. Additionally, you may implement some of the listed extensions (or invent your own!) for extra credit. **Be sure to check out last year's great submissions.**

Skeleton Code:

You have been provided with some skeleton code which you may use to jump-start your coding. Basically, the only thing the code does is move three particles around randomly, and draw some (nominal) constraints and spring between them. This code does little more than implement basic window management and graphics, but this stuff is very annoying to do alone.

Required Features:

Your code implement the following features:

- **A generalized force structure.** This is described in the slides. (If you're using the skeleton code, you should replace `delete_this_dummy_spring` with a `std::vector` of forces.) You must implement two subclass forces:
  - `GravityForce`. Acts like gravity.
  - `SpringForce`. A damped spring between two particle. Skeleton rendering code is already provided.
- **A generalized constraint structure.** This is also described in the slides. (If you're using the skeleton code, you should replace `delete_this_dummy_rod` and `delete_this_dummy_wire` with a `std::vector` of forces.) You must implement at least the following two subclasses:
  - `RodConstraint`. Constrains two particles to be a fixed distance apart. (Rendering code included in the skeleton.)

    $$C(x_1, y_1, x_2, y_2) = (x_1 - x_2)^2 + (y_1 - y_2)^2 - r^2$$

  - `CircularWireConstraint`. Constrains a particle to be a fixed distance from some point:

    $$C(x, y) = (x - x_c)^2 + (y - y_c)^2 - r^2$$

- **Mouse interaction.** When the user clicks and drags the mouse, a spring force should be applied between the mouse position and the given particle to make your system interactive.

- **Several Numerical Integration Schemes (Simulators).** The integration scheme should be selectable at runtime with keystrokes or some other interaction paradigm. You will find this easiest if you implement a pluggable integration architecture as described in the slides. The minimum integration schemes are:
  - Euler
  - Runge-Kutta 2
  - Runge-Kutta 4

Optional Features:

You demo must be able to turn each of these features on and off individually so they can be verified.

- ★ **Verlet Integrator.** See here.
- ★ **Leapfrog Integrator**. Evaluates position and velocity at different times. See here for more details.
- ★ **Symplectic Integrator.** As described in class. Compute the positions explicitly and velocities implicitly. (No need for a solver.)
- ★★ **Collisions with the Walls.** Particles should bounce off the walls and floor.
- ★★ **Collisions with other Particles.** Particles bounce off each other.
- ★★★ **Angular Springs.** Pulls a triplet of particles so that their subtending angle approaches some rest angle.
- ★★★★ **Angular Constraints.** Like angular springs, but the angle is actually constrained.
- ★★★ **3D.** Implement and render this algorithm in 3D.
- ★★★ **2D Cloth.** Create a rectangular network of particles with appropriate springs holding it together. Which spring configurations work best, which don't work?
- ★★★★ **Implicit Integration.** In order to implement the linear solver, you can use the linearSolver.* code in the skeleton code. (Note that it must be added to the makefile before you can use it.) See the course notes here.
- ★★★★ **3D Cloth.**
- ★★★★★ **3D Cloth with collisions**
- ★★★★★★ **Hair with collisions.**

Deliverables:

Code. At 11:59:59 on the due date, you *must* submit zip file or tarball of the code that builds on the instructional Linux system. The code may be mailed to the TA, Jeehyung Lee at jeehyung@cs.cmu.edu.

Demo. After the due date, if you like, you may schedule a meeting with Jeehyung to demo your project to show of any special features.

Artifact. You must submit a video of your system in action. Videos can be implemented in several ways. Usually the starting point is to dump frames (by hitting 'd' in the skeleton implementation). These frames can be coalesced into a movie using several software packages (ImageMagick and ffmpeg on Linux, Quicktime Pro on Mac, and VirtualDub on Windows). Alternatively, you can use one of the new screen capture programs that are all the rage these days.

Further explanation of *J* and *∂J / ∂t* (also known as $\dot{J}$):

∂J / ∂t is the time derivative of the gradient matrix *J* of your constraints. For example if you had constraint:

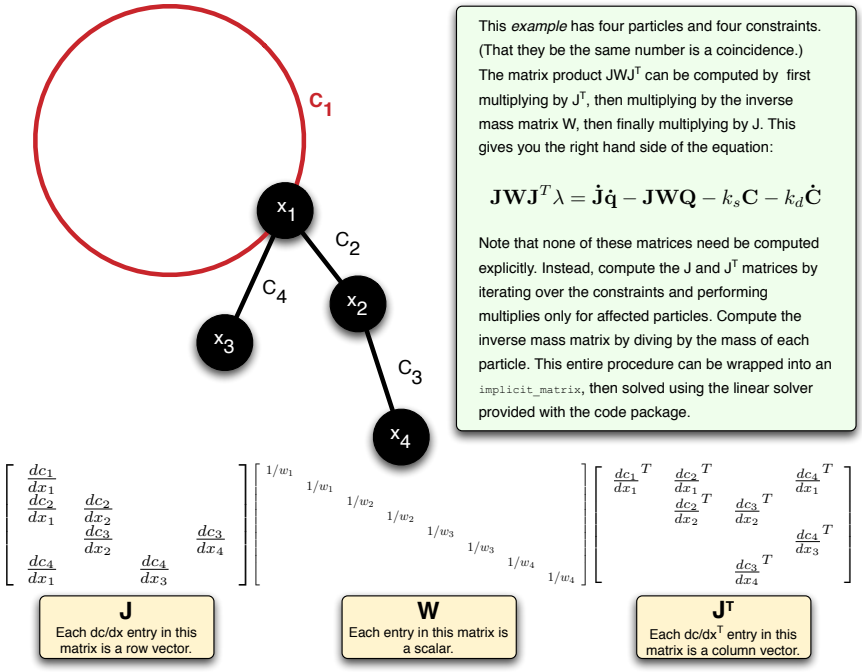
$$C(x,y) = \sin(x) + \cos(y) - 1$$

then the gradient matrix would be:

$$J = [\cos(x), -\sin(y)]$$

and its time derivative would be:

$$\partial J / \partial t = [-\sin(x)\, \partial x / \partial t, -\cos(x)\, \partial y / \partial t].$$



Analytic or Numerical Derivatives:

Derivatives (i.e. gradients) should be computed analytically, not numerically. So there's no ε step approximaton. For example, your constraint abstract base class could have an abstract function of the form:

```
gradient(double dCdX[3]) = 0;
```

You should return the exact derivative. For example if your constraint is that the particle lie on a plane C(X) = dot(x, n) - p then the derivative would be dCdX(x) = n:

```
void PlaneConstraint::gradient(double dCdX[3]) {
     dCdX[0] = this->n[0];
     dCdX[1] = this->n[1];
     dCdX[2] = this->n[2];
}
```

Note that the constraint class is computing the exact derivative. This information would then be used as in the following pseudocode:

```
double dCdX[3];
for (int constraint_index = 0 ; constraint_index <
nconstraints ; ++constraint_index) {
     OneParticleConstraint * c = constraints[constraint_index];
     c->gradient(dCdX);
     constraint_jacobian->add_gradient(
          constraint_index,
          c->get_particle_index(),
          dCdX);
}
```

Note that this is just an example, and many variations are possible, including replace the 3-length double arrays with gfx vector classes, etc.

How to “Pin” Particles:

If you are trying to "pin" a particle to be at point (a, b), you can just flag the particle rather than using constraints. Then at each timestep, set the particle's position to (a, b) and it's velocity and forces both to (0, 0). It can be shown that this is exactly equivalent to the pair of constraints:

$C_1(x) = x - a$
$C_2(x) = x - b$

You should especially not use the bead on a wire constraint with radius r=0, because this constraint will have a gradient discontinuity precisely where's it's satisfied, which is a **bad thing**, numerically.

(Note that if you do this, you can't apply any other constraints to this particle, but why would you want to?)