CS533B Assignment 2 *Second Order Particle Systems* **Due Jan. 29, 2004**

In this assignment, you will run a second order particle system (i.e. particles with mass that accelerate due to forces) with collisions. The goal is to simulate snow (or hail, or ash, or sparks, or golf-balls, or...) falling on a sculpture.

The three parts of the simulation are initial conditions, the force fields, and the collisions.

For initial conditions, every frame ($1/24$ of a second) seed a few particles with uniformly distributed positions on a square just above the scene, at height $3m$ and some specified side length. (Exactly how many will be an input parameter to the simulation.) Each particle's radius is randomly chosen from an interval, also specified in the input. The density is also given in the input; the mass of a particle is then just the density times its volume, $4/3\pi r^3$.

The particles fall under the influence of gravity and wind forces. That is, the force on particle i is:

$$F_i = m_i g + D r_i^2 (v_{\text{wind}}(x_i, t) - v_i)$$

where m_i is the particle mass, g is the acceleration due to gravity ($9.8m/s^2$ in the negative y direction), D is a drag coefficient (specified in the input), r_i is the particle radius, $v_{\text{wind}}(x, t)$ is the wind velocity at position x and time t, x_i is the position of the particle, and v_i is the velocity of the particle.

The velocity field of the wind will be a simple random noise model of an incompressible fluid:

$$v_{\text{wind}}(x, t) = V + \sum_{j=0}^{9} A_j \sin(K_j \cdot x + \omega_j t)$$

Here V is an underlying wind vector (an input parameter to the simulation). A_j and K_j are vectors and ω_j is a scalar, Derive a condition on A_j and K_j that will ensure the resulting velocity field v is incompressible, a.k.a. divergence-free ($\nabla \cdot v = 0$). Recall that the divergence $\nabla \cdot v$ of a vector field $v = (v_1, v_2, v_3)$ is just $\partial v_1/\partial x + \partial v_2/\partial y + \partial v_3/\partial z$. Include this derivation in your write-up. After implementing this condition on A_j and K_j you still have a lot of freedom to play with this model: fix your own values for A_j, K_j, and ω_j as you like. The goal is of course to make the motion of the particles look like they're being blown around realistically.

Advance the particles forward in time using Symplectic Euler, stopping to output particle positions every frame. Note that there is a stability limit for this method: you will need to write a function which gets a conservative upper-bound on that limit and implement the algorithm in class for making sure we land on exactly the frame time.

Our model test equations from class were $a = -Kx$ and $a = -Dv$. Each gave a limit on the time step Δt in terms of the scalars K and D. Figure out upper-bounds on the eigenvalues of the Jacobians $\partial a/\partial x$ and $\partial a/\partial v$ that hold for all x and t. (One useful fact: the eigenvalues of a matrix are bounded by a norm of the matrix, such as the the 1-norm or 2-norm). This doesn't have to be a particularly tight bound, so you could, for example, bound the norm of the sum of matrices with the sum of their norms.

In any case, once you can compute upper bounds on K and D, use those to get lower-bounds on the two separate stability limits for Δt. A reasonable, though not rigourous, stability limit is then the minimum of the two. You may want to multiply this by some fudge factor (try 0.5 or 0.9, for example) to be safe.

I will provide you with a text file containing a level set of a sculpture (the Happy Buddha model from the Stanford Graphics Lab data repository). Your program should use this sculpture, which is about $2m$ tall[1] and $0.4m$ wide, along with the ground (y=0) for collisions.

Process collisions with the (last) algorithm discussed in Jan. 13's class that could handle elastic bounces as well as contact robustly. More specifically, at the end of every frame, detect if any particle is below the ground or inside the sculpture level set, and if so, process the collision, repeating until the particle is outside again. Note that the particles have a radius, so you should be detecting interference between a sphere and the objects, not just a point. The coefficient of restitution ϵ (how elastic the bounces should be) and dry friction coefficient μ should be more input parameters to the simulation.

[1] That is, the level set is about $2m$ tall; the actual sculpture that was original scanned in is only a dozen or so centimetres tall.

1 File Formats

I'm providing you with example code in C as before, as well as some data files.

The file `buddha.levelset` contains the level set information for the sculpture. The first line has 3 integers, the dimensions of the grid (in the x, y, and z directions). The second line has 1 floating point number, the grid spacing (how far apart are grid points in each direction). The third line 3 floating point numbers, the corner of the grid with the smallest coordinates (at grid index (0,0,0)). The rest of the file has the values of phi, stored in order along z, then along y, then along x.

The file `buddha.trimesh` contains a triangle mesh of the same geometry (approximately) for use in the OpenGL renderer. The first line has the number of vertices, followed by that number of lines (with 3 floating point numbers per line, for each vertex's position). Then there is an integer saying how many triangles there are, and that many lines each with 3 integers (the indices of the corner vertices for a triangle). Note that the indices are assumed to start at 0.

As input to the simulation, instead of command-line arguments, your code should read in from a parameter text file the following numbers, in order. (See the example `parameters` file)

- number of seconds to simulate (integer)
- number of new particles per frame (integer)
- minimum particle radius (floating point, in metres)
- maximum particle radius (floating point, in metres)
- particle density (floating point, in kg/m^3)
- drag coefficient (floating point, in Newtons per square metre per metre per second, i.e. $kg/(m^2 s)$)
- base wind vector V (three floating point numbers, in m/s)
- coefficient of restitution ϵ (floating point, dimensionless)
- friction coefficient μ (floating point, dimensionless)
- the width of the area in which particles appear (floating point, metres)

The output for each frame should be in the same format as in assignment 1: a text file with the number of particles, followed by a line for each particle with the particle's position, its RGB colour (which should be white: 1, 1, 1), and its radius.

2 What to hand in

Your source code for this, along with a README explaining how to use it. Also include an example parameters file which you think gives the nicest looking output.

The written part should include the derivation of the divergence-free condition on the parameters in the wind field, and your derivation of a bound on the maximum stable time step.

3 Optional

I'm always interested in feedback on the course: what you wish I would do differently, or what you wish I would do more of, or less of, or not at all. Feel free to talk to me about it or send email, or write comments at the end of your assignments, or if you want to be more anonymous you can leave notes in my mailbox in the department office.

4 Final Project Ideas

This assignment could serve as the basis for a good final project. For example, you could extend the simulation with something from this list (or something else—just run it by me). Note also these are just suggestions, not exact specifications of what you need to do.

- When particles stick due to static friction, grow the level set by unioning the analytical level set of the particle (just a sphere) and the sculpture (or the ground). Once the particles have stuck, you can "turn them off" and not update their position anymore so they stay stuck (and you don't have problems with them intersecting the level set).
- Also track orientation and angular velocity for each particle, and then render the particles as snowflakes (or leaves, or other more interesting objects) instead of just spheres.
- Put more geometry in the scene using triangle meshes, and collide the particles with that.
- Put more options in: using wet friction instead of dry friction (e.g. for rain), using an implicit solver for more speed, turning off simulation of particles once they have stopped moving (for speed again). If it's fast enough, you could combine with the OpenGL renderer and get a nice demo.