CS-C3100 Computer Graphics, Fall 2021

Lehtinen / Aho, Kaskela, Kynkäänniemi

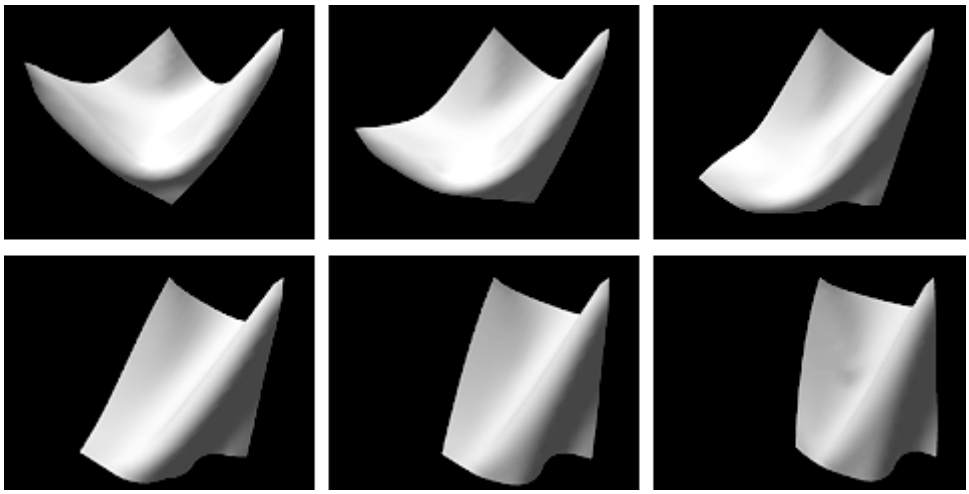
Programming Assignment 4: Physical Simulation

Due Sun Nov 21th at 23:59.

Physical simulation is used in movies and video games to animate a variety of phenomena: explosions, car crashes, water, cloth, and so on. Such animations are very difficult to keyframe, but relatively easy to simulate given the physical laws that govern their motion. In this assignment, you will build simulations of increasing complexity, ending up with a cloth model built from a large number of particles connected to each other with springs.

Requirements (maximum 10p) ***on top of which you can do extra credit***

1. **Euler integrator (1p)**
2. **Spring system (2p)**
3. **Trapezoid integrator (2p)**
4. **Pendulum system (2p)**
5. **Cloth system (3p)**



1 Getting Started

In this assignment, you'll be both solving ordinary differential equations (ODEs) numerically by writing different kinds of integrators, and formulate dynamical systems of your own (springs, cloth, etc.) that are driven by forces such as gravity, springs, and viscosity.

1.1 Simple Newtonian Mechanics and Differential Equations

The motion of many physically-based systems is governed by Newton's laws of motion, which are second order differential equations that, as you know, *relate forces to accelerations*. For a single point-like particle at position $\mathbf{x}$, this means

$$m\frac{\mathrm{d}^2\mathbf{x}}{\mathrm{d}t^2} = \mathbf{F}, \tag{1}$$

i.e., the familiar "force equals mass times acceleration". The force $\mathbf{F}$ is a vector sum over all the different forces acting upon the particle. The forces can vary with time, and also other particles may have an effect. We'll talk more about forces below.

As we saw in class, it pays off to reduce higher-order differential equations to first order, so that we can write a solver once and use it for many different kinds of problems. For our simple point dynamics, this means adding an auxiliary velocity variable $\mathbf{v}$ to each particle, and writing

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix}\mathbf{x}\\ \mathbf{v}\end{pmatrix} = \begin{pmatrix}\mathbf{v}\\ \mathbf{F}/m\end{pmatrix}. \tag{2}$$

This equation is of first order — i.e., it only involves the first time derivatives of the system state that consists of the position and velocity of the particle — and it says the same thing as the one above. Solving this differential equations means that starting from a given initial state $(\mathbf{x}_0, \mathbf{v}_0)$, we trace out the curve determined by the infinitesimal displacements

$$\mathrm{d}\begin{pmatrix}\mathbf{x}\\ \mathbf{v}\end{pmatrix} = \mathrm{d}t\begin{pmatrix}\mathbf{v}\\ \mathbf{F}/m\end{pmatrix}$$

that say "in the infinitely small time step $\mathrm{d}t$, my current position changes by $\mathrm{d}t$ times my current velocity, and my velocity changes by $\mathrm{d}t$ times the current forces divided by my mass".

When we have a larger number $n > 1$ of point-like particles in our system, we write $\mathbf{x}_i$ for the position and $\mathbf{v}_i$ for the velocity of the ith particle. Because the above equations of motion apply to all particles separately (the acceleration for a single particle is determined by the forces acting on that particle), we can stack all the positions and velocities in a single big vector $\mathbf{X}(t)$

$$\mathbf{X}(t) = \begin{pmatrix}\mathbf{x}_1(t)\\ \mathbf{v}_1(t)\\ \mathbf{x}_2(t)\\ \vdots\\ \mathbf{x}_n(t)\\ \mathbf{v}_n(t)\end{pmatrix}, \tag{3}$$

and following the above, write this bigger particle system abstractly as

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{X} = \begin{pmatrix} \mathbf{v}_1 \\ F_1(\mathbf{X},t)/m_1 \\ \mathbf{v}_2 \\ \vdots \\ F_n(\mathbf{X},t)/m_n \end{pmatrix} \stackrel{\text{def}}{=} f(\mathbf{X},t). \tag{4}$$

The vector $\mathbf{X}$ has length $6n$ for n particles (all positions are 3D, as are all velocities), and it lives in what is called *phase space*. We denote the force acting on the ith particle by — you guessed it — $\mathbf{F}_i(\mathbf{X},t)$. The derivative $\mathrm{d}/\mathrm{d}t\mathbf{X}$ depends potentially on time — for example, a changing wind force — and the current state — e.g. spring forces are determined by the current positions of the particles — so we explicitly write it out as a function $f(\mathbf{X},t)$ that returns the derivative vector given the state vector $\mathbf{X}$ and time t.

1.2 Forces

The core component of particle system simulations are forces. Suppose we are given a particle's position $\mathbf{x}_i$, velocity $\mathbf{v}_i$, and mass m_i. We can then express forces such as gravity

$$\mathbf{F}(\mathbf{x}_i,\mathbf{v}_i,m_i) = m_i\mathbf{g}, \quad \text{where } \mathbf{g} = \begin{pmatrix} 0 & -g & 0 \end{pmatrix}^{\mathrm{T}}$$

or perhaps *viscous drag* that opposes motion (given a drag constant k):

$$\mathbf{F}(\mathbf{x}_i,\mathbf{v}_i,m_i) = -k\mathbf{v}_i.$$

We can also express forces that involve other particles as well. For instance, if we connected particles i and j with an with an undamped spring of rest length r and spring constant k, it would yield a force of:

$$\mathbf{F}(\mathbf{x}_i,\mathbf{v}_i,m_i) = -k(||\mathbf{d}|| - r)\frac{\mathbf{d}}{||\mathbf{d}||}, \text{ where } \mathbf{d} = \mathbf{x}_i - \mathbf{x}_j.$$

Think back to your high school physics: what is the expression of the gravitational force between two massive bodies that are so far away from each other that they can be considered point-like?

Summing over all forces yields the net force, and dividing the net force by the mass gives the acceleration $\mathbf{a}_i = \frac{\mathrm{d}}{\mathrm{d}t}\mathbf{v}_i$.

2 Numerical integrators

It is important for you to understand the abstraction between numerical integrators and particle systems. The numerical integrator does not know anything about the physics of the system. It can request the particle system to compute the derivative $f(\mathbf{X},t)$ using

the system's **evalF** method. This function is the critical communication channel between a system and an integrator. It takes as input a state vector and returns a force vector *for this particular input state*, which are both represented as 1D arrays regardless of the precise type of particle system (only the size of the array varies). This allows integrators to be general and reusable.

A particle system stores its current state $\mathbf{X}$, but the numerical integrator might request the forces for a different state, in particular for the trapezoidal method. It is critical that the particle system uses the correct state, the one requested by the call to **evalF**, to compute the forces. Make sure you understand the difference between this internal state of the system and that requested by the integrator.

2.1 Refresher on Euler and Trapezoidal Rule

The simplest integrator is the explicit *Euler method*. For an Euler step, given state $\mathbf{X}$, we examine $f(\mathbf{X}, t)$ at $\mathbf{X}$, then step to the new state value. This requires to pick a step size h, and we take the following step based on $f(\mathbf{X}, t)$, which depends on our system.

$$\mathbf{X}(t+h) = \mathbf{X} + hf(\mathbf{X}, t)$$

Look back to Equation (4) and see if you can spot a similarity; we've sort of replaced dt with a small but finite number and multiplied it over to the other side. This technique, while easy to implement, can be unstable for all but the simplest particle systems. As a result, one must use small step sizes (h) to achieve reasonable results.

There are numerous other methods that provide greater accuracy and stability. For this problem set, you will implement the *Trapezoidal* approach, which works by using the average of the force f_0 at the current state and the force f_1 after an Euler step of step size h:

$$\begin{aligned} f_0 &= f(\mathbf{X}, t) \\ f_1 &= f(\mathbf{X} + hf_0, t + h) \\ \mathbf{X}(t+h) &= \mathbf{X} + \frac{h}{2}(f_0 + f_1) \end{aligned}$$

This method makes it critical for the particle system to be able to evaluate the forces at a state $\mathbf{X} + hf_0$ other than its current state $\mathbf{X}$. The state of the particle system should not be updated until reaching the last equation.

2.2 Simple example system

We have supplied you with a *first-order* ODE similar to one we saw in class (`SimpleSystem`). This system has a single particle and its state is defined by its x-y-z coordinates:

$$\mathbf{X} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

and the right-hand side is

$$f(\mathbf{X}, t) = \begin{pmatrix} -y \\ x \\ 0 \end{pmatrix}.$$

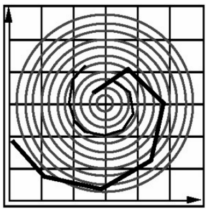
This is only a first-order system and the right-hand side of the ODE does not describe physical forces. (Using pen and paper, take a moment to plot some points and their associated derivatives on paper to understand what is going on. It is not hard!)

The arrays passed between the system and the integrator consist of a single `Vec3f`. The z coordinate does not do anything interesting but we kept it to make the various systems in this problem set more consistent.

The exact solution of the ODE is a circle with the equation

$$X(t) = \begin{pmatrix} rcos(t+k) \\ rsin(t+k) \end{pmatrix}$$

but that is not quite what we get from our implementation. Test this with `example.exe`. Pick the simple particle system, Euler integrator and a large step size. You'll see the particle spiral outwardly in a 2D space, similar to the image below.



While large step size allows you to see the divergence faster, it will eventually occur regardless of how small you make the step size. The other integrators - midpoint, trapezoidal and Runge-Kutta - are also unstable, but diverge at a much slower rate.

3 Detailed instructions

Since we have supplied you with one working particle system (the simple system) and one working integrator (midpoint), you can test your work regardless of in which order you complete the requirements. Still, you'll probably want to do R2, R4 and R5 in that order since they are progressively more difficult applications of the same techniques.

R1 Euler integrator (1p)

Implement an Euler integrator into the `eulerStep` function within `integrators.cpp`. Once it is correct, you'll see the particle systems moving when you select the Euler integrator in the application.

R2 Spring system (2p)

For this requirement, you'll implement a simple physically modeled particle system where one particle is fixed in place and another hangs from it attached with a spring. Unlike the simple system, this equation involves a real spring force and is hence a 2nd-order system that needs to be reduced to 1st order by introducing the velocity variable like we saw above.

Head over to `particle_systems.cpp` to fill in the missing parts of `SpringSystem` class implementation. First, tackle the `reset` function which sets the initial state of the particle system. You essentially just have to set the positions of the particles and store the data about the spring in the system. You need to pay attention to the indices where the components of position and velocity go in the state vector.

The real work happens in the `evalF` function which computes the forces affecting the state given as a parameter. You can start by applying only gravity; you should see one particle staying put at origin and the other one falling. Then add the drag and spring forces. There are helper functions for the forces at the beginning of `particle_systems.cpp`; `fGravity` is already implemented, while the other two, `fDrag` and `fSpring` are up to you.

Fixing a particle in place is just a matter of setting the net forces affecting it to zero.

R3 Trapezoid integrator (2p)

Just like R1, only this time you implement a trapezoid integrator within the `trapezoidStep` function. The trapezoid integrator is considerably better than the Euler integrator, and should give results roughly equivalent to the midpoint integrator we supplied you with.

R4 Pendulum system (2p)

Here you'll repeat what you did in R2 but in a more general form, building a particle system where an arbitrary amount of particles are connected with springs in a chain and suspended from the topmost one. Appropriate rendering could make such a system look like a length of rope.

You again need to fill in the `reset` and `evalF` functions. It is substantially no different from R2, but there are more particles (note that `PendulumSystem`'s constructor takes the particle count as a parameter) and it's easier to make mistakes while indexing into the state array. You might find it useful to define small helper functions, e.g. `pos` and `vel`, which take the index of a particle and return the index at which that particle's position and velocity are stored in the state array.

At this point it's also useful to consider forming the proper springs in the `PendulumSystem::reset` function. After this, in `PendulumSystem::evalF` you can loop first over the particles to compute independent forces (gravity) and then loop over the springs, adding their resultant force to both affected particles. This makes the implementation simpler; you

only have to think about the particle indices once instead of figuring out the neighbors of each particle separately – just be careful to only add each spring once into the list. Furthermore, with this approach your `evalF` can be essentially the same in R5.

R5 Cloth system (3p)

What you have learned in R2 and R4 is sufficient to implement a simulation of cloth in `ClothSystem`. We merely need to build a more elaborate configuration of particles and springs than we did before.

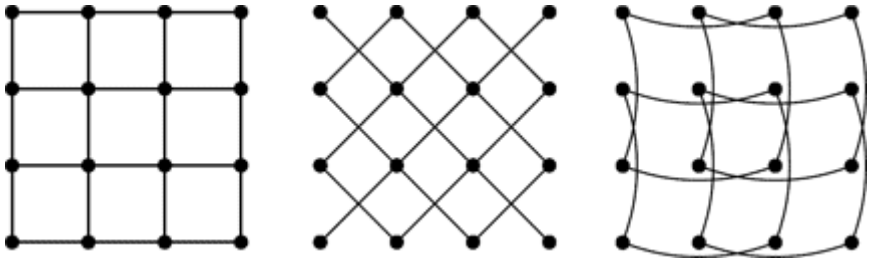


Figure 1: Left to right: structural springs, shear springs, and flex springs

We begin with a uniform grid of particles and connect them to their vertical and horizonal neighbors with springs. These springs keep the particle mesh together and are known as *structural springs*. Then, we add additional *shear springs* to prevent the cloth from collapsing diagonally. Finally, we add *flexion springs* to prevent the cloth from folding over onto itself. Notice that the flex springs are only drawn as curves to make it clear that they skip a particle—they are still "straight" springs.

Write your code very carefully here; it is easy to make mistake and connect springs to particles that don't exist. Again we recommend that you create helper methods, like the previously mentioned `pos` and `vel`, but this time taking the x and y grid coordinates of the particle and returning indices in the state vector belonging to that particle. As already suggested for R4, here it's extremely useful to generate the springs in `reset` and simply evaluate their result in `evalF` instead of figuring which neighboring particles exist for which particles.

First, implement structural springs. Make sure it looks as you expect before moving on. Run the simulation and you should obtain something that looks like a net. As usual, viscous drag helps prevent explosions. For faster debugging, use small meshes of e.g. 3×3 particles.

Once you've made sure your structural springs are correct, add in the shear springs, and then the flex springs. When inserting a new set of springs you can temporarily remove the previous ones to see whether the initial shape of the new ones looks correct.

Don't be too discouraged if your first tests blow up because of instability. Use the most stable integrator you have, stick to low step sizes, and use the example binary as a yardstick for how low the step sizes need to be. Doing the easy Runge-Kutta extra credit

or one of the harder extras will help a lot to keep the cloth together.

4 Extra Credit

As always, you are free to do other extra credit work than what's listed here — just make sure to describe what you did in your README. We'll be fair with points, but if you want to attempt something grand, better talk to us beforehand. On some items, you have the choice to implement more or less features; we'll give you points based on how much you did.

4.1 Recommended

- **Easy: Implement RK4 Runge-Kutta integrator (2p)**
  A RK4 integrator is nearly as easy to implement as the trapezoid integrator; it's just more code to write. RK4 allows using substantially larger step sizes without your system blowing up. See http://en.wikipedia.org/wiki/Runge-Kutta_methods

- **Easy: Spray/waterfall/sprinkler system (1-3p)**
  Create your own particle system that keeps spraying out particles which disappear after a while (think "water hose" or "rising smoke"). For full points, also have the particles bounce off a plane or other object (you don't have to render the object). If you'd like to also render your particles using semi-transparent textures or other cool effects, there's an extra for that in the Medium section.

4.2 Easy

- **Wind (2p)**
  Add a random wind force with an on/off toggle to your cloth simulation.

- **Mouse drag/poke UI (2-3p)**
  Provide a mouse-based interface for users to interact with the cloth. You may, for instance, allow the user to click on certain parts of the cloth and drag parts around.

- **Frictionless collisions (2p)**
  Implement frictionless collisions of cloth with a simple primitive such as a sphere. This is simpler than it may sound at first: just check whether a particle is "inside" the sphere; if so, just project the point back to the surface. (You don't need to draw the primitive.)

- **Cloth tearing (1+p)** Simulate the cloth breaking up by removing springs that have stretched beyond some threshold. Simply doing this with an uniform treshold is worth one point, and additional features like non-uniform tresholds (anisotropic or spatially varying) and generating a reasonable mesh with the boundaries handled nicely are worth more.

4.3 Medium

- **Particle editor (2-3p)**
  Give the user a mouse UI to modify some particle parameters using spline curves — for instance, the emission rate and dispersal of a geyser, or wind strength and direction over time. Provide a way to save and load these parameters from files.

- **Particle rendering (1-4p)**
  Create a cool rendering mode for some particle system. You could make your extra credit sprinkler/waterfall system emit particles that look like fluid/fire/smoke, do a visualization of kinetic and potential energy or forces in one of your spring-based systems, or render the cloth mesh. A crude cloth mesh as in the example binary is worth 1p, but you can apply better shading or texture mapping for more points. We award points based on your *best* system, so invest in quality rather than quantity.

- **Adaptive solver scheme**
  Implement an adaptive solver scheme (look up adaptive Runge-Kutta-Felhberg techniques or check out the MATLAB `ode45` function).

- **GPU simulation (4+p)**
  Transfer the computational load of the simulation to run on the graphics processor using OpenGL compute shaders or other suitable API (OpenCL, CUDA) – we have provided starter code for OpenGL so it might be the easiest especially if you have no former experience in GPU computing. The classroom computers have GL compute support in case your own computer doesn't. You might want to see the slides for the Programming parallel computers course, lectures 4 and 5 are relevant. Although the slides focus on CUDA, many concepts apply to GL compute as well, just with different names. This OpenGL Wiki page gives a good overall description of GL compute shaders as well.
  To get started, first you'll have to uncomment the line including ComputeCloth.hpp from the top of App.hpp. You'll have to fill in function bodies to `BxcPlusD.glsl`, `evalF.glsl` and `reset.glsl`, as well as the body of the CPU-side method `Advance` in `ComputeCloth.cpp`. Each file contains some helpful comments and starter code that should help you get started.
  Points are awarded based on which systems, integrators and additional features (like wind and tearing) you implement. You'll get four points for the basic cloth system with no extra features.

4.4 Hard

- **Implicit integration (8-10p)**
  Implement an implicit integration scheme, such as implicit Euler. Such techniques allow much greater stability for stiff systems of differential equations, such as the ones that arise from cloth simulation. (Even though implicit techniques suffer from numerical error just like explicit ones, the inaccuracy tends to bias the solution towards stable solutions, thus allowing far greater step sizes.) This will require

the ability to evaluate the Jacobian of the force vector with respect to its own components, and a Newton solver. We recommend you use the Eigen C++ linear algebra template library for solving the large linear systems that arise. Be sure to look at the Baraff & Witkin notes. You get more points for implementing more than one kind of implicit solver (e.g., implicit midpoint rule).

- **Implicit optimization (4p)**
  The Jacobian matrix of the force function is required for implicit integration. While the numerical values of the matrix change for each iteration in the Newton method, the *sparsity pattern* stays the same; only certain entries of the matrix are ever non-zero. See if you can use Eigen's "sparse direct solvers" to perform a symbolic factorization of the matrix beforehand. Then, given the numerical values in the current step, this allows you to quickly compute a matrix factorization that lets you solve for the Newton step quickly.

- **Constraints (8p)**
  Extend your particle system to support constraints, as described here. This means that you remove the springs, and using so-called Lagrange multipliers, project the remaining forces (gravity, etc.) onto a subspace that does not violate the constraints.

- **Robust model of cloth (15p)**
  Implement the technique described in
  http://graphics.stanford.edu/papers/cloth-sig02/

- **Simulate rigid-body dynamics**
  http://www.cs.cmu.edu/ baraff/sigcourse/notesd1.pdf or
  deformable models:
  http://graphics.cs.cmu.edu/projects/stvk/ or
  fluids:
  http://physbam.stanford.edu/~fedkiw/
  In theory, particle systems can be used to achieve similar effects. However, greater accuracy and efficiency can be achieved through more complex physical and mathematical models. You will receive credit according to how large and hard your implementation is.

5 Submission

- Make sure your code compiles and runs both in Release and Debug modes on Visual Studio 2019, preferably in the VDI environment. Comment out any functionality that is so buggy it would prevent us seeing the good parts. Check that your `README.txt` (which you hopefully have been updating throughout your work) accurately describes the final state of your code.
- Fill in whatever else is missing, including any feedback you want to share. We were not kidding when we said we prefer brutally honest feedback.
- Package all the code, project and solution files required to build your submission, the `README.txt` and any screenshots, logs or other files you want to share into a ZIP archive. Note: the solution file itself does not contain any code, make sure the contents of `src/` are present in the final submission.
- Sanity check: look inside your ZIP archive. Are the files there? (Better yet, unpack the archive into another folder, and see if you can still open the solution, compile the code and run.)

Submit your archive in MyCourses folder:
"Assignment 4: Physical Simulation".