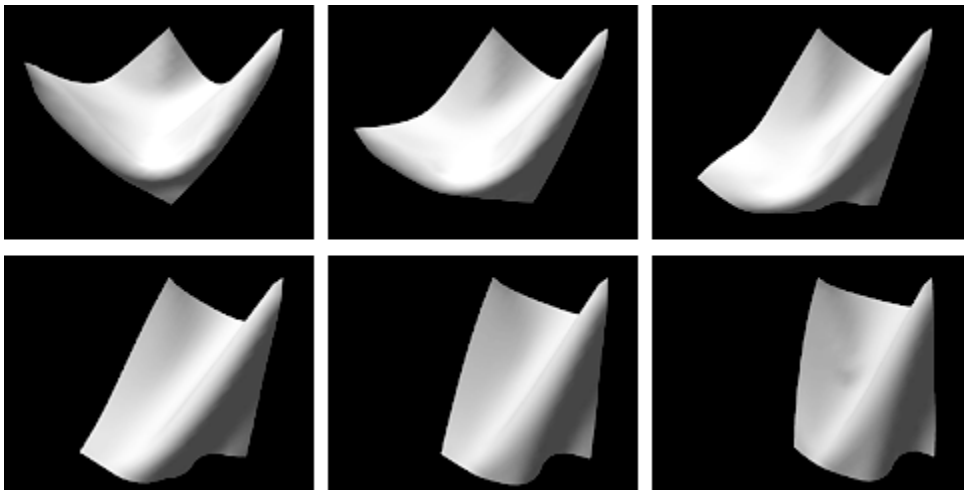
6.837: Computer Graphics Fall 2011

Programming Assignment 3: Physical Simulation

Due November 2nd at 8:00pm.

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 be using springs to build a visually appealing simulation of cloth, as shown below. Start early as this assignment will require you to think about the design of your code!



The remainder of this document is organized as follows:

1. Getting Started
2. Summary of Requirements
3. Numerical Integrators
4. Physical Simulation
5. Particle System Cloth
6. Extra Credit
7. Submission Instructions

1 Getting Started

Take a look at the sample solution for a demonstration of what you'll be implementing.

Run the `a3soln` file with no parameters and watch as the "cloth" falls. You may reset the simulation by pressing `r`, or you can flap the cloth around by pressing `s`. If you wish to toggle the wireframe view, press `w`. Your task will be to build a similar simulation.

The other files in this directory provide a simple skeleton OpenGL application. Right now, it only displays a single particle. Compile the code with `make` and run it with `a3`.

For this assignment, you will have to put thought into the design of your solution. We provide you with some starter code, although feel free to change things as you go- it is merely a way to help guide you through the assignment. In general, you should spend some time thinking about what sorts of functions or classes to write before you begin. Start early!

2 Summary of Requirements

Again, you do not need to use the provided starter code (but we recommend it). This is a challenging assignment that requires you to be a good code designer and tester. To ensure partial credit, we suggest the following steps.

First, you will begin by implementing two numerical methods for solving ordinary differential equations: Euler and the Trapezoidal Rule. You will test these on a simple first order system that we covered in class. It is important that you abstract the numerical integrator from the system. A numerical integrator should be general enough to be able to take any step for any system. Testing the simple first order system with your numerical integrators will hopefully ensure that your abstracted methods are correct before you move on to the more complicated second order systems.

Second, you will implement a second order system, a simple pendulum, consisting of two particles with a spring connecting them. This will require you to implement three types of forces: gravity, viscous drag, and springs. Each of these forces will be necessary later to create your cloth simulation.

Third, you will extend your simple pendulum to create a string of particles with four particles. This will allow you to incrementally test your spring implementation before you begin assembling the cloth.

Finally, using springs, you are to assemble a piece of cloth. It should be at least an eight-by-eight grid of particles. You will need to implement structural, shear, and flexion springs.

Your application should display a wireframe animation of the cloth. You will receive extra credit for implementing smooth-shading similar to the one shown in the sample solution. Your application should also allow the user to move the cloth in some way. It can be as simple as a keystroke that makes the cloth move back and forth, as implemented in the sample solution.

You should provide an executable called `a3` that takes two parameters. The first should be a character, either `e` or `t`, that selects the solver (Euler or Trapezoidal). The second should be the stepsize used by the solver. The third is optional for debugging, and is an integer representing an index of a particle. If specified, it should highlight the springs that are attached to that particle.

3 Numerical Integrators

It is important for you to understand the abstraction between the numerical integrators and the particle system. The numerical integrator does not know anything about the physics of the system. It can request the particle system to compute the ODE using the system's **evalF** method. Note that these forces might need to be evaluated at states other than the current state of the system. Each system has some state $\mathbf{X}$, and the numerical integrator must be able to step the system for any given state. Therefore, the particle system and the integrator communicate through their state vectors, which are represented as 1D arrays. To re-emphasize, your integrator should be modular and be able to step any system at any state.

3.1 Euler and Trapezoidal Rule

The simplest method is the explicit *Euler method*. For an Euler step, given state $\mathbf{X}$, we examine $f(\mathbf{X}, t)$ at the current state, 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.

$$\begin{aligned} t_1 =& t_0 + h \\ \mathbf{X_1} =& \mathbf{X_0} + hf(\mathbf{X_0}, t0) \end{aligned}$$

This method, while easy to implement, can be very unstable for all but the simplest particle systems. As a result, one must use very small step sizes (h) to achieve reasonable results.

There are numerous other methods that provide greater accuracy and stability. For this problem set, we will use the *Trapezoidal* method, which (at a very high level) works by taking a single euler step with stepsize h, evaluates f_1, takes a full step using f_1, and then averages the two steps. It can be stated as follows:

$$\begin{aligned} f_0 =& f(\mathbf{X_0}, t_0) \\ f_1 =& f(\mathbf{X_0} + hf_0, t_0 + h) \\ \mathbf{X}(t_0 + h) =& \mathbf{X_0} + h/2 * (f_0 + f_1) \end{aligned}$$

3.2 Simple ODE Example 20%

You will now implement Euler and the Trapezoidal Rule. We have given you an empty class, `integrator.cpp` for you to implement each of these numerical methods. `integrator.h` contains the method declarations (`eulerStep` and `trapezoidalStep`) that you should implement. Your `Integrator` methods should take in a pointer to a `ParticleSystem` and get its current state. It should then take either an euler or trapezoidal step, using the system's **evalF** method to calculate the forces $f(\mathbf{X}, t)$. It should then set the system's state to the updated values. Quickly look at the abstract class `ParticleSystem`. It provides you with methods for getting and setting the system's state. It also provides virtual methods, `stepSystem` and `evalF`, which you will implement later.

You will test each of your implementations on the simple *first-order* ODE that we saw in class:

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

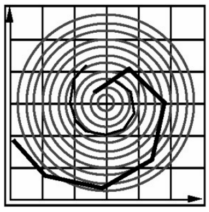
The state of this simple system is defined by its x-y coordinates:

$$\mathbf{X_t} = \begin{pmatrix} x \\ y \end{pmatrix}$$

As seen on the lecture slides, Euler's method is unstable. The exact solution is a circle with the equation

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

However, Euler causes the solution to spiral outward, no matter how small h is. After implementing Euler's method, you should see the single particle spiral outwardly in a 2D space, similar to the image below.



Next, implement the Trapezoidal Rule on another particle. The Trapezoidal Rule is still unstable, but it diverges at a much slower rate. You should be able to compare your Euler and Trapezoidal implementation side by side, seeing the particles diverge outwardly at different rates. Think carefully about how you are going to implement the Trapezoidal Rule. It requires that you evaluate the forces at a different time step at different points. *So, you should write a function that evaluates all forces given any state of the system.* Remember that your integrator functions should be separated and abstracted away from the system itself. You'll be using these integrator functions for different systems later on, so it's important that these methods are modular.

For this specific ODE, you'll implement the methods `stepSystem` and `evalF` in `SimpleSystem` which inherits from the abstract `ParticleSystem` class. `stepSystem` takes a step in your system and allows you to update the system's state by calling the specified `Integrator` function. As iterated above, you should pass the system to the `Integrator` method, who will take care of updating the system's state. In this case, the system's state is the particle's x-y position. In this case we only have a single particle, but the next problems will require you to handle multiple particles, so make sure you account for that. We have given you a hint by characterizing the state as a 1D array of `Vector3f` (review the lecture notes to see how you should be representing the state of a system). `evalF` evaluates and returns the forces $f(\mathbf{X}, t)$ given *any* state $\mathbf{X}$ of the system. Both `evalF` and your methods in `Integrator` should not be modifying individual particles within the system. `evalF` should take in a system state and return the forces associated with that state. The `Integrator` methods should atomically modify the system's state at each step.

Check that your Euler and Trapezoidal implementation work as expected, with the Euler spiraling outward and diverging, and the Trapezoidal doing the same, but at a slower rate. The command line of your application should allow the user to choose the solver and stepsize (like the sample solution). Each method implementation is worth 10%.

4 Physical Simulation

You should now have successfully verified the correctness of your numerical integrators with the simple first order system. Now, we're ready to apply these integrators to a more complicated second order system. In this section, you will implement a simple two particle pendulum and extend that to a multiple particle chain. This will require you to implement the different kinds of forces (gravity, viscous drag, and springs).

4.1 Forces

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

$$\mathbf{F}(\mathbf{x}_i, \mathbf{x}_i', m_i) = m_i \mathbf{g}$$

Or perhaps *viscous drag* (given a drag constant k):

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

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:

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

Summing over all forces yields the net force, and dividing the net force by the mass gives the acceleration $\mathbf{x}_i''$.

The motion of all the particles can be described in terms of a second-order ordinary differential equation:

$$\mathbf{x}'' = \mathbf{F}(\mathbf{x}, \mathbf{x}')$$

In this expression, $\mathbf{x}$ describes the positions of all the particles ($\mathbf{x}$ has $3n$ elements, where n is the number of particles). The function $\mathbf{f}$ sums over all forces and divides by the masses of the particles.

The typical way to solve this equation numerically is by transforming it into a first-order ordinary differential equation. We do this by introducing a variable $\mathbf{v} = \mathbf{x}'$. This yields the following:

$$\begin{bmatrix} \mathbf{x}' \\ \mathbf{v}' \end{bmatrix} = \begin{bmatrix} \mathbf{v} \\ \mathbf{f}(\mathbf{x}, \mathbf{v}) \end{bmatrix}$$

In conclusion, we can define our state $\mathbf{X}$ as the position and velocity of all of the particles in our system:

$$\mathbf{X} = \begin{pmatrix} x \\ v \end{pmatrix}$$

which then gives us:

$$\frac{d}{dt}\mathbf{X} = f(\mathbf{X}, t) = \begin{pmatrix} v \\ \mathbf{F}(x, v) \end{pmatrix}$$

Given these system characteristics, you should be able to use your numerical integrators to approximate this system. In the next sections, you will implement a simple pendulum and a multiple particle chain to test your implementation. Note that you *should not* need to modify your code for Euler or Trapezoidal. Your integrator code should be modular and abstracted enough to be able to handle any arbitrary state from any system!

4.2 Simple Pendulum 20%

You will now implement the gravity force, viscous drag force, and spring force. Test this first with a single particle connected to a fixed point by a spring (basically, a pendulum) in `pendulumSystem.cpp`. Your implementation of `evalF` should return $f(\mathbf{X}, t)$, requiring you to calculate the gravity, viscous drag, and the spring forces that now act on your particle system.

We recommend that you create some kind of data object for the springs, as you'll need to keep track of the spring forces on each of the particles. To help with debugging, make a function that allows you to see which springs are attached to a specific particle. Allow the user to specify a number i as a command line parameter that renders the springs that are connected to the particle with index i (this will be become more useful when we have many more particles).

You should make sure that the motion of this particle appears to be correct. Note that, especially with the Euler method, you will need to provide a reasonable amount of drag, or the system will explode. The Trapezoidal Rule method should be much more stable, but you will still want a little viscous drag to keep the motion in check. If you are able to demonstrate this simple example, you will receive **40%** of the available points.

4.3 Multiple Particle Chain 20%

The next step is to extend your test to multiple particles. Try connecting four particles with springs to form a chain, and fix one of the endpoints (you can fix a particle by zeroing the net force applied to it). Make sure that you can simulate the motion of this chain. As a general rule, more particles and springs will lead to more instability, so you will have to choose your parameters (spring constants, drag coefficients, step sizes) carefully to avoid explosions. If you reach this point and everything is correct, you'll get **60%** of the possible points.

5 Particle System Cloth 40%

The previous section describes how to simulate a collection of particles that are affected by gravity, drag, and springs. In this section, we describe how these forces can be combined to yield a reasonable (but not necessarily accurate) model of cloth.

Before moving on, we also recommend taking a snapshot of your code, just in case the full cloth implementation does not work out. We recommend using Git for version control (available on Athena). Although there is a bit of a learning curve to using a version control system, having a safety net is more than worth it. Alternatively, you can extend `ParticleSystem` and make your own `ClothSystem` class for the cloth simulation.

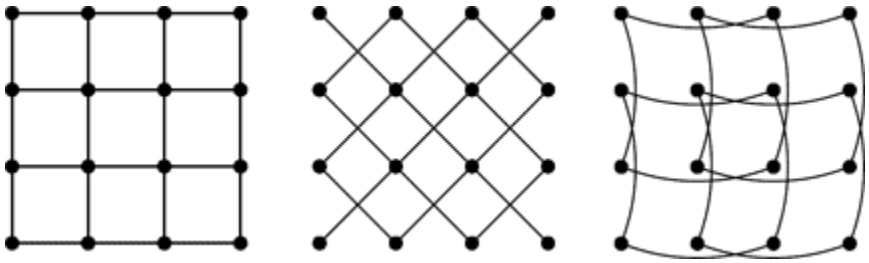


Figure 1: Left to right: structural springs, shear springs, and flex springs

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 with the same force equation given earlier.

If you have designed your code reasonably well, it shouldn't be too tough to add the necessary springs. Make sure that you use reasonable rest lengths, and start small. Write your code very carefully here; it is easy to make mistakes and connect springs to particles that don't exist. We recommend that you create a helper method `indexOf` that given index i, j into a nxn cloth, returns the linear index into our vector of particles.

First, implement structural springs. Draw the springs to make sure you've added the right ones in. Make sure it looks as you expect before moving on.

Once you've made sure your structural springs are correct, add in the shear springs. Again, test incrementally to avoid mistakes. Finally, add in flex springs.

While these springs keep the cloth in a reasonable shape, they do not alone give reasonable motion. Without some sort of dampening force, the springs will theoretically bounce around forever without losing energy. We know this is not what happens in real life: cloth, when moved, will settle down quickly to a motionless state.

There are many ways to emulate this behavior, but for this assignment you should just use viscous drag to slow down the particles. In addition to making the motion more realistic, the drag force also counters the inherent instability of explicit integrators.

To display your cloth, the simplest approach is to draw the structural springs (which you should have already done to debug your structural spring implementation!). For extra credit, you can draw it as a smooth surface like the sample solution, but this is not required. If you do choose to draw the cloth as a smooth surface, you'll need to figure out the normal for the cloth at each point.

Don't be too discouraged if your first test looks terrible, or blows up because of instability. At this point, your Euler solver will be useless for all but the smallest stepsizes, and you should be using the Trapezoidal solver almost exclusively.

If you manage to have a moving wireframe cloth, then you're at **90%**. All that's left is to add the necessary user interface elements, such rendering the cloth, moving it around, and so on. And that's **100%**.

You may also find these notes from the SIGGRAPH 2001 course on physically based modeling by Andrew Witkin and David Baraff helpful.

6 Extra Credit

The list of extra credits below is a short list of possibilities. In general, visual simulation techniques draw from numerous engineering disciplines and benefit from a wide variety of techniques in numerical analysis. Please feel free to experiment with ideas that are not listed below.

6.1 Easy

- Add a random wind force to your cloth simulation that emulates a gentle breeze. You should be able to toggle it on and off using a key.
- Rather than display the cloth as a wireframe mesh, implement smooth shading. The most challenging part of this is defining surface normals at each vertex, which you should approximate using the positions of adjacent particles.
- Implement a different object using the same techniques. For example, by extending the particle mesh to a three-dimensional grid, you might create wobbly gelatin. If you choose to implement this, please provide a different executable.
- 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.
- 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.

6.2 Medium

- Implement an adaptive solver scheme (look up adaptive Runge-Kutta-Felhberg techniques or check out the MATLAB `ode45` function).
- Implement an implicit integration scheme. Such techniques allow much greater stability for stiff systems of differential equations, such as the ones that arise from cloth simulation. An implicit Euler integration technique, for instance, is just as *inaccurate* as the explicit one that you will implement. However, the inaccuracy tends to bias the solution towards stable solutions, thus allowing far greater step sizes. The sample solution demonstrates such a technique, which can be activated with `i` on the command line.
- Extend your particle system to support constraints, as described in this document. This extra credit is actually an assignment in 6.839.

6.3 Hard

- Implement a more robust model of cloth, as described in this paper.
- Simulate rigid-body dynamics, deformable models, or fluids. 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.

7 Submission Instructions

You are to write a `README.txt` that answers the following questions:

- How do you compile and run your code? Provide instructions for Athena Linux. If your executable requires certain parameters to work well, make sure that these are specified.
- Did you collaborate with anyone in the class? If so, let us know who you talked to and what sort of help you gave or received.
- Were there any references (books, papers, websites, etc.) that you found particularly helpful for completing your assignment? Please provide a list.
- Are there any known problems with your code? If so, please provide a list and, if possible, describe what you think the cause is and how you might fix them if you had more time or motivation. This is very important, as we're much more likely to assign partial credit if you help us understand what's going on.
- Did you do any of the extra credit? If so, let us know how to use the additional features. If there was a substantial amount of work involved, describe what how you did it.
- Got any comments about this assignment that you'd like to share?

As with the previous assignment, you should create a single archive (`.tar.gz` or `.zip`) containing:

- All source code necessary to compile your assignment.
- A compiled executable named `a3`.
- The `README.txt` file.

Submit it online using the Stellar website by **November 2nd @ 8:00pm**.

This assignment does not require the submission of an artifact.