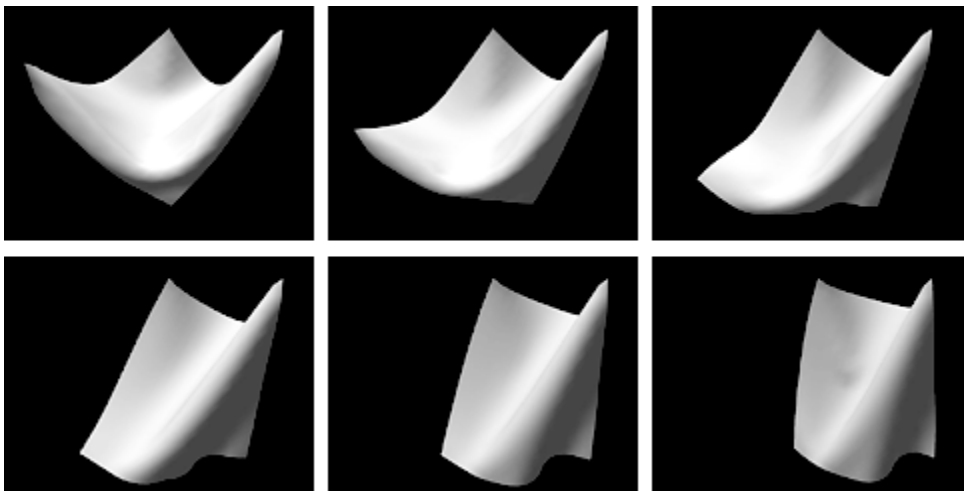
T-111.4310: Vuorovaikutteinen tietokonegrafiikka 2012

Programming Assignment 3: Physical Simulation

Due November 18th 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 be using springs to build a visually appealing simulation of cloth, as shown below.



The remainder of this document is organized as follows:

1 Getting Started

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

Run the `example.exe` file. Play around with the user interface controls to familiarize yourself with them. Your task will be to build a similar simulation.

The other files in this directory provide a simple skeleton OpenGL application. Compile the code and run it. If you change the particle system to `PendulumSystem` or to `SimpleSystem` you will notice that the program draws a single sphere. Take a look at `App::resetSystem` to see how the system is initialized. When you start implementing `PendulumSystem` or `ClothSystem` you should start simple and set the number of particles as something small in `App::resetSystem`. Once you have a system working for a simple case, you can start adding more particles.

2 Summary of Requirements

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 (`SimpleSystem`) that we covered in class.

Second, you will implement a second order system, a simple pendulum `PendulumSystem`, consisting of two particles with a spring connecting them. You should change the initialization code in `App::resetSystem` to give the number 2 as the parameter for the constructor of `PendulumSystem`. 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 (`ClothSystem`). It should be at least an eight-by-eight grid of particles. You will need to implement structural, shear, and flexion springs.

3 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 forces 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 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.

3.1 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.

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

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, we will use 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 stepsize 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.

3.2 Simple Example 40%

You will implement Euler's (`integratorEuler.h`) and the Trapezoidal Rule (`integratorTrapezoidal.h`) and simple one particle system. You should provide the implementation for the method `stepSystem` in both classes. Notice that a pointer to the particle system can be accessed with `ps`, which is declared in the parent class `Integrator`. You can get the current state of the particle system with `ParticleSystem::getState`, and set the new state once you have evaluated it with `ParticleSystem::setState`. To evaluate the derivative of the system at a state, use `ParticleSystem::evalF`. Notice that you will have to implement the `evalF` method for your particle systems.

You will test each of your implementations on the simple *first-order* ODE `SimpleSystem` similar to one we saw in class. This system has a single particle and its state is defined by its x-y-z coordinates:

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

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. You do not need to use the trick we used in class to turn Newtonian systems into first-order systems. 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.

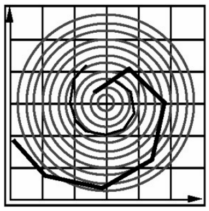
For this specific ODE, you'll implement the methods `SimpleSystem::evalF` and `EulerIntegrator::stepSystem`. `stepSystem` takes a step in your system and allows you to update the system's state.

Implement the simple system and the Euler integrator. Try different values of h (with the user interface slider) and see how the precision varies.

As seen in 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's method 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 (`TrapezoidalIntegrator::StepSystem`). Think carefully about how you are going to implement the Trapezoidal Rule. It requires that you evaluate the derivatives $f(\mathbf{X}, t)$ at a different time step at different points.

The Trapezoidal Rule is still unstable, but it diverges at a much slower rate. You should be able to compare your Euler and Trapezoidal implementations, seeing the particles diverge outwardly at different rates. 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. You can choose, which solver is used, from the user interface buttons.

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! However, you might want to think carefully about how you will store the state. One simple option is to store a big `Vec3f` array of size $2n$ where positions are stored at even indices and velocities at odd indices. It might help to write helper functions that read the position or velocity of particle i. The starter code is thought to be used this way but you may change how you store the state if you want.

4.2 Simple Pendulum 20%

You need to implement the functions `initSystem` and `evalF` in `pendulumSystem.h`. The simple pendulum should consist of one fixed particle connected to another (moving) particle with a spring.

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 think carefully about the representation of springs, as you'll need to keep track of the spring forces on each of the particles. We have suggested one method in the starter code (see the comments in `PendulumSystem::InitSystem`) but you may choose a different approach if you wish. You can store a list of springs that know the two particles they act on, the rest length and the stiffness. You can alternatively store, for each particle, what other particle it is connected to and what the stiffness and rest length are.

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 more stable, but you will still want a little viscous drag to keep the motion in check.

4.3 Multiple Particle Chain 15%

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 **75%** of the possible points.

5 Particle System Cloth 25%

You should implement the constructor of `ClothSystem`. The class inherits `PendulumSystem` and you should not have to implement an `evalF` function for this part. You may design the `ClothSystem` class from scratch if you wish.

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 a version control system. Although there is a bit of a learning curve to using a version control system, having a safety net is more than worth it.

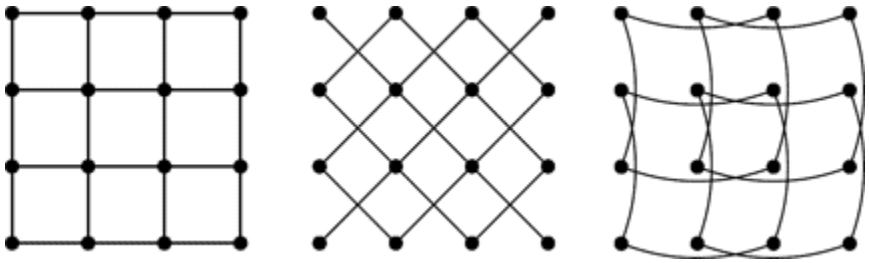


Figure 1: Left to right: structural springs, shear springs, and flex springs

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 $n \times n$ 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. 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. Again, test incrementally to avoid mistakes. Finally, add in flex springs.

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 step sizes, and you should be using the Trapezoidal solver almost exclusively.

You may also find the notes from http://www.pixar.com/companyinfo/research/pbm2001/ 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.
- 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.
- Implement a Runke-Kutta solver. See http://en.wikipedia.org/wiki/Runge-Kutta_methods

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.

6.3 Hard

- Extend your particle system to support constraints, as described in
  http://www.cs.cmu.edu/afs/cs/user/baraff/www/pbm/constraints.pdf
- Implement a more robust model of cloth, as 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.

7 Submission Instructions

IMPORTANT: Submission instructions have changed. Please include your name and student number at the beginning of the README.txt file.

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

- Your name and student number at the beginning of the document.
- 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 `.zip` archive containing:

- Your whole Visual Studio project.
- The `README.txt` file.

Submit it online in Optima by **November 18th at 23:59**.

This assignment does not require the submission of an artifact.