ISTD 50.017 Home Page

height=

Assignment 3 : Physical Simulation

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!

1. Getting Started

Take a look at the sample solution a3soln.exe for a demonstration of what you'll be implementing.

Run the a3soln file with parameters 'e', 't' or 'r' 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. Compile the code with make and run it with **a3**. Remember, incremental debugging is your friend! We recommend, for debugging purposes, trying to get the code to display a single particle, and then moving on to implementing the integrators.

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 all of 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 time integrator from the system. A time integrator should be general enough to be able to take any step for any system. This step is independent of the next steps as we will provide you with a reasonable integrator (RK4).

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, **e**, **t** or **r** that selects the solver (Euler , Trapezoidal or RK4). The second is an optional stepsize used by the solver. Your application should have a key 't' to togle between showing the cloth, the simple system, and the pendulum. Alternatively, if you are lazy, show all of them simultaneously.

3. Time Integrators (TimeStepper.xpp)

It is important for you to understand the abstraction between time integrators and particle systems. The time integrator does not know anything about the physics of the system. It can request the particle system to compute the derivatives 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 derivative vector **for this particular state**, which are both represented as 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 **X**, but the time integrator might request the derivatives 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 derivatives. Make sure you understand the difference between this internal state of the system and that requested by the integrator.

To re-emphasize, your integrator should be modular and be able to step any system, while your particle system should be very careful to compute forces at the requested state, which is potentially different from its current state. Do not be misled by the Euler integrator where both states are the same, or you will suffer miserably when implementing the trapezoidal rule.

3.1. Refresher on Euler and Trapezoidal Rule

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

$$f_0 = f(\mathbf{X}, t)$$

$$f_1 = f(\mathbf{X} + hf_0, t + h)$$

$$\mathbf{X}(t+h) = \mathbf{X} + h/2(f_0 + f_1)$$

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 20% (simpleSystem.cpp)

You will implement Euler's and the Trapezoidal Rule. It should use the a particle system's **evalF** method to calculate the derivative at the appropriate state. It should set the system's state to the updated values. Look at the abstract class **ParticleSystem**. It provides you with methods for getting and setting the system's state.

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

And the right-hand side is

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 are of length 3. The z coordinate does not do anything interesting but we kept it to make the various systems in this problem set more consistent and use **Vector3f**.

Implement **evalF** in **SimpleSystem** which inherits from the abstract **ParticleSystem** class. As iterated above, you should pass the system to the takeStep method, who will take care of updating the system's state.

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 $f(\mathbf{X}, t)$ (which is the velocity in this case) 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 derivatives associated with that state. The **Integrator** methods should atomically modify the system's state at each step. Implement the simple system and the Euler integrator. Try different values of h 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

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. 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. *So, you should write a function that evaluates all derivatives 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 they are modular.

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. The command line of your application should allow the user to choose the solver and stepsize. Each method implementation is worth 10%.

4. Physical Simulation

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). We have provided you with a fourth order Runge-Kutta (RK4) integrator, since the integrators you implemented are unstable.

4.1. Forces

The core component of particle system simulations are forces. Suppose we are given a particle's position xi, velocity x, and mass mi. We can then express forces such as gravity:

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

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:

Summing over all forces yields the net force, and dividing the net force by the mass gives the acceleration x''_i.

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

In this expression, **x** describes the positions of all the particles (**x** has $3n$ elements, where *n* is the number of particles). The function **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 orginary differntial equation. We do this by introducing a variable **v** = **x**'. This yields the following:

In conclusion, we can define our state **X** as the position and velocity of all of the particles in our system:

which then gives us:

Given these systesm characteristics, you should be able to use your time 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 **Vector3f** 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*.

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(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. 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.

(Optional) To help with debugging, make a fucntion 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 which 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 more stable, but you will still want a little viscous drag to keep the motion in check. If you have everything correct so far, you will already 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.

You should extend **ParticleSystem** or **pendulumSystem** and make your own **ClothSystem** class for the cloth simulation.

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 **index0f** that given index *i*, *j* into a *n x 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 x 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.

To display your cloth, the simplest approach is to draw a grid or 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 step sizes, 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 on physically based modeling David Baraff helpful, particularly particle system dynamics.

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. Also make sure that your code for previous sections still works after implementing extra credit.

6.1 Easy

[bell]   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.

[bell]   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.

[bell]   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.

[bell]   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.

[bell]   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.

[bell]   Implement 4th order Runge-Kutta time stepper. This is the same one we provided you with.

6.2 Medium

[bell]   Implement an adaptive solver scheme (look up adaptive Runge-Kutta-Felhberg techniques or check out the MATLAB ode45 function).

[bell]   Extend your particle system to support constraints, as described in this document.

6.3 Hard

[bell]   [bell]   Implement an implicit integration scheme as described in this paper: Large Steps in Cloth Simulation (Baraff and Witkin, SIGGRAPH 98). 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.

[bell]   [bell]   Implement a more robust model of cloth, as described in this paper: Robust Treatment of Collisions, Contact and Friction for Cloth Animation (Bridson, Fedkiw and Anderson, SIGGRAPH 02).

[bell]   [bell]   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:

- 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 extra credit? If so, let us know how to use the additional features. If there was a substantial amount of work involved, describe what and how you did it.

- Got any comments about this assignment that you'd like to share? Was it too long? Too hard? were the requirements unclear? Did you have fun, or did you hate it? Did you learn something, or was it a total waste of your time? Feel free to be brutally honest; we promise we won't take it personally.

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

- All source code necessary to compile your assignment.

- A compiled executable built from your code name a3.

- The aforementioned **README.txt** file.

Submit it online

This assignment does not require the submission of an artifact.