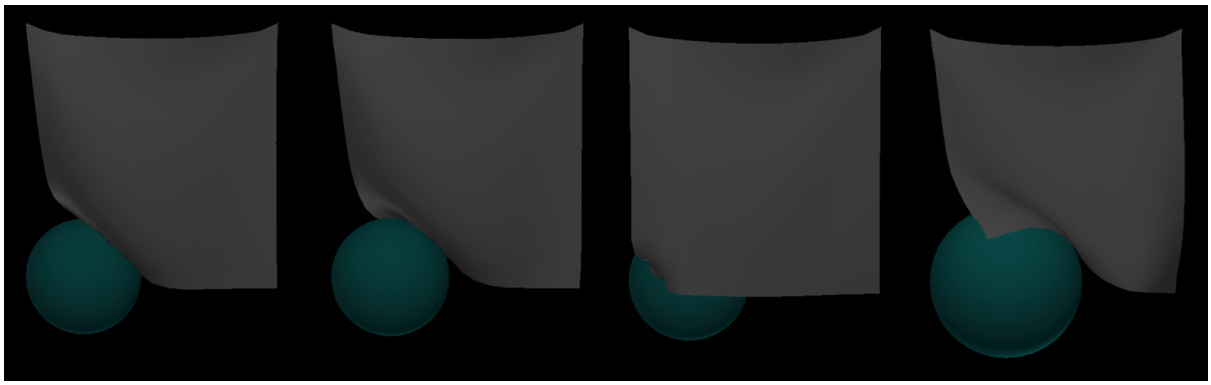
Programming Assignment 3: Physically-Based Simulation

MIT 6.837 Computer Graphics

Fall 2020

Due October 21 at 8pm, Boston Time

Physically-based simulation is widely 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

The sample solution is included in the starter code distribution. Look at the `sample_solution` directory for Linux, Mac, and Windows binaries. You may need to change the file mask on the Linux/Mac binaries, e.g., by executing the following command in your terminal:

```
chmod a+x sample_solution/linux/assignment3
```

To run the sample solution, execute

```
sample_solution/linux/assignment3 r 0.005
```

The first parameter is the integrator type that should be a character `e`, `t` or `r` (corresponding to **E**uler, **T**rapezoidal or **R**K4). The second parameter is the (time) step size used by the integrator.

You will see three examples of simulation that you will implement:

- On the left is a sphere with a simple circular motion.
- In the middle is a chain of particles that emulates a pendulum.
- On the right is a sphere swinging back and forth that interacts with a piece of cloth.

Your task will be to build a similar simulation scene with **all three examples**.

The relevant starter code for this assignment is in the `assignment_code/assignment3` directory. To build the starter code, follow the same steps as in previous assignments. For instance, on MacOS or Linux, execute the following:

```
mkdir build
cd build
cmake ..
make
```

If you run the starter code now, a window of an empty scene will pop up. We recommend, for debugging purposes, trying to get the code to display a single particle (as a sphere), and then moving on to implementing the integrators. Start early!

2 Summary of Requirements

This is a challenging assignment that requires you to be a good code designer and tester. The starter code has provided one particular design that separates time integrators and particle systems (see Section 3 for more details). You do not need to use all of the provided starter code (but we recommend it). Here is a summary of the recommended steps to approach this assignment.

(a) **Euler, Trapezoid, and Simple System.** Firstly, you will begin by implementing two numerical methods for solving ordinary differential equations: (forward) Euler integration and the trapezoidal rule. You will test these on a simple first-order system that we covered in class (this corresponds to the sphere in circular motion in the sample solution). It is important that you abstract the time integrator from the particle system (the starter code described in Section 3 has done this for you). A time integrator should be general enough to be able to take any time step (regardless of numerical stability) for any first-order ODE.

(b) **Multi-Pendulum and RK4.** Secondly, you will implement a second-order system, a pendulum consisting of multiple particles in a chain of springs, as shownin the middle of the sample solution. This will require you to implement three types of forces: gravity force, viscous drag force, and spring force. Each of these forces will be necessary later to create your cloth simulation. This will allow you to test your spring implementation before assembling the cloth. At this point, you will implement the last integrator, Runge-Kutta 4 (RK4), which will allow you to use larger time steps while remaining stable and accurate.

(c) **Cloth.** Finally, you will simulate a piece of cloth. Your cloth should be a mass-spring with an 8×8 grid of particles (or bigger). You will need to implement structural, shear, and flexion springs.

Cloth simulation will quickly blow up when using the Euler integrator. Trapezoidal integrator is okay with small-enough steps (e.g. 1ms, or `h=0.001`), and RK4 allows you to take larger steps (about 5ms, or `h=0.005`).

Your application should display an animation of the cloth, represented using a wireframe (i.e. 3D line segments) or shaded triangles. Your simulator should also interact with the cloth in some way. This can be as simple as pressing a key (say, `R`) that resets the system to its initial state and then the cloth naturally falls down due to gravity. You can also use a moving sphere that interact with the cloth constantly, as done in the sample solution.

You will receive extra credit for implementing smooth shading similar to the one shown in the sample solution.

3 Interface Design

In this section we describe the design choice of the starter code. Again feel free to come up with your own design. It is important for you to understand the abstraction between time integrators and particle systems. A time integrator does not know anything about the physics of the system, while a particle system only has the knowledge of the physical law (as a first-order ODE system) without knowing how to solve the system.

3.1 ParticleState

The `struct ParticleState` describes a state (snapshot) of the particle system. In C++, a `struct` is like a `class` but all its fields are public by default. The struct `ParticleState` stores the positions and the velocities of the particles. Specifically it has two `std::vector<glm::vec3>`, one for positions and the other one for velocities, and these two `std::vector` arrays share the same size, which is equal to the number of particles in the system.

We provide overloaded operators, `+` and `*`. Adding two `ParticleState` with `operator+` results in component-wise addition for both positions and velocities arrays. Multiplying a `ParticleState` with a `float k` scales both positions and velocities arrays by a scalar constant `k`.

3.2 ParticleSystemBase

A particle system describes the law of the physics among the particles. Concretely, it is the function f in the ODE

$$\frac{\mathrm{d}\mathbf{X}}{\mathrm{d}t}(t) = f(\mathbf{X}(t), t) \tag{1}$$

where $\mathbf{X}(t)$ is the vector of positions and velocities concatenated together at time t. In particular the particle system has no information of the current particle state or the current time. We provide a base class `ParticleSystemBase` that contains one virtual method `ComputeTimeDerivative` with signature

```
ParticleState ComputeTimeDerivative(const ParticleState& state, float time);
```

The `ComputeTimeDerivative` method takes a particle state $\mathbf{X}$ and a time t, and returns $f(\mathbf{X}, t)$ which also has type `ParticleState` (it mathematically has the type of the gradient of a state, but we use the same type as the state so that you can do arithmetics on it using the overloaded operators `+` and `*`). You will need to inherit `ParticleSystemBase` and implement `ComputeTimeDerivative` for each simulation type (i.e., pendulum and cloth).

3.3 IntegratorBase

A time integrator should be implemented in a generic way for any type of physical system whose law is governed by a first-order ODE. Even more generally, its implementation should not depend on the internal format of the state. For example, the state can have different representations for a rigid body or for a collection of particles. There are some minimal assumptions that need to be made. For instance, we need to be able to add two states and multiple a state by a scalar in order to solve Eqn. 1 numerically.

In C++ one can use *template classes* to impose these mild assumptions. This is the approach adopted in the starter code. The base class of the time integrators is `IntegratorBase<TSystem, TState>` where `TSystem` and `TState` are template parameters for the physical system and the physical state. You should find this type of syntax familiar from STL examples like `std::vector<T>`. The base class `IntegratorBase` contains one virtual method `Integrate` with signature

```
TState Integrate(const TSystem& system, const TState& state,
                 float start_time, float dt);
```

This function will integrate the provided state according to the law of `system` from time `start_time` to `start_time + dt`. You should assume that `TSystem` is a base class of `ParticleSystemBase`. More precisely, it contains a method named `ComputeTimeDerivative(state, time)` where `state` has type `TState`. You should also assume that `TState` has overloaded operators `+` and `*` in the same way that `ParticleState` has. In this assignment it is okay to assume `TState` is `ParticleState` (other for the extra credit), but your implementation only needs to be based on the assumption that `TState` can be added (via `+`) and multiplied with a scalar (via `*`).

To implement your own integrators, you will need to inherit `IntegratorBase`. Your derived classes will also be template classes. An example is provided in `ForwardEulerIntegrator` for the forward Euler integrator (note this class is not completed! you will still need to implement the `Integrate` method). For template classes, you will need to provide the definitions of the methods in the header file (you will likely not need a `.cpp` file). It is recommended to check out online tutorials on C++ template classes to see some examples before you start implementing integrators.

3.4 `IntegratorFactory`

Finally, you will need to be able to create integrators based on their types. This can be done by implementing the `CreateIntegrator<TSystem, TState>` static method of `IntegratorFactory`. The method takes a enum of type `IntegratorType`. For instance, `IntegratorType::RK4` corresponds to the RK4 integrator. Note that `CreateIntegrator<TSystem, TState>` is a template method that creates a specialized integrator to solve physical system `TSystem` with state representation type `TState`.

4 Requirements

4.1 Refresher on Euler and Trapezoidal Rule

The simplest integrator is the explicit *(forward) 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 picking a step size h and taking the following step based on $f(\mathbf{X}, t)$, which depends on our system.

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

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 first use the *trapezoidal* approach, which uses the average of the force f_0 at the current state and the force f_1 after an Euler step of step size h:

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

4.2 Simple Example (20% of grade)

As a first step, you will implement Euler's and the Trapezoidal Rule. You will test each of your implementations on a simple ODE similar to the one we saw in class. This system has a single particle and its state is its xyz position (velocity is not part of the state in this example). The physical system is

$$f(\mathbf{X}, t) = f\left[\begin{pmatrix} x \\ y \\ z \end{pmatrix}, t\right] = \begin{pmatrix} -y \\ x \\ 0 \end{pmatrix}. \tag{2}$$

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.

You have two options for representing the state. You can either directly use `ParticleState` but with one particle (i.e. the position and velocity arrays have size 1, but we won't need to use the velocity array; note the position and velocity arrays need to have the same size), or you can just use a `glm::vec3` for the x, y, z states of that single particle. If you choose to do the former, you should inherit `ParticleSystemBase` and implement the physical law Eqn. 2 while ignoring the velocity array. If you choose to do the latter, you will need to create your own physical system class that has a method `ComputeTimeDerivative(state, time)` where `state` is of type `glm::vec3`; this is needed for the template class of your integrators to compile. Note that the z coordinate is constantly 0. You can keep it there to make things more consistent (e.g. you will need to provide a `glm::vec3` when you try to set the position of the particle during the animation).

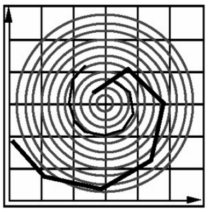
Implement this simple system and the forward Euler integrator. The starter code has created the source file for the Euler integrator for you (`ForwardEulerIntegrator.cpp`) and you need to implement its `Integrate` method. To put everything together, you will need to create a derived class of `SceneNode` and override the `Update` method to update the state and the visualization. More concretely, you will want to keep your state, your integrator, and your physical system somewhere in your new `SceneNode` subclass. They need to be initialized properly in the constructor. When implementing `Update` which

takes a float indicating elapsed time since the last frame, you will need to advance the state using your integrator for this amount of elapsed time for multiple steps – the number of integration steps depends on the step size passed in from the command line. Try different values of the integrator step size and see how the precision of the simulation varies.

As seen in the lecture slides, Euler's method is neither stable nor accurate. The exact solution is a circle with the equation

$$\mathbf{X}(t) = \begin{pmatrix} r\cos(t+k) \\ r\sin(t+k) \end{pmatrix}.$$

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. It requires that you evaluate the derivatives $f(\mathbf{X}, t)$ at a different time step. This should be straightforward once you have your Euler integrator working. The trapezoidal rule is relatively more stable and accurate: it tolerates larger time step sizes and 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.

4.3 Pendulum System (40% of grade)

In this section, you will implement a multi-pendulum system with multiple particles in a chain connected by springs. This requires you to implement three different kinds of forces: gravity force, viscous drag force, and spring force.

4.3.1 Forces

Forces form a important component of particle system. Suppose we are given a particle's position $\mathbf{x}_i$, velocity $\mathbf{x}_i'$, and mass m_i. Then the *gravity force* is determined as

$$G_i = m_i\mathbf{g}.$$

The *viscous drag force* (given a drag constant k) is

$$D_i(\mathbf{x}_i') = -k\mathbf{x}_i'.$$

We can also express forces that involve other particles as well. For instance, if we connected particles i and j with an undamped spring of rest length r_{ij} and spring constant k_{ij}, this would result in a *spring force* on particle i as

$$S_{ij}(\mathbf{x}_i, \mathbf{x}_j) = -k_{ij}(||\mathbf{d}|| - r_{ij})\frac{\mathbf{d}}{||\mathbf{d}||}, \text{ where } \mathbf{d} = \mathbf{x}_i - \mathbf{x}_j.$$

The pendulum system we are considering will have these three types of forces. Summing over all forces yields the net force, and dividing the net force by the mass gives the acceleration $\mathbf{x}_i''$ for particle i. We let $\mathbf{x}$ be the positions of all the particles ($\mathbf{x}$ has $3n$ real components, where n is the number of particles, and the factor 3 comes from the fact that the particles are in 3D). Similarly we use $\mathbf{x}'$ to denote the velocities of all particles. We define a function $\mathbf{A}(\mathbf{x}, \mathbf{x}')$ to be the accelerations of all particles subject to the three types of forces described. That is, $\mathbf{A}(\mathbf{x}, \mathbf{x}')$ has $3n$ real components, and the $3i$, $3i+1$, $3i+2$

components (0-based) form a vector representing the acceleration of the ith particle (0-based) defined by

$$\mathbf{A}(\mathbf{x}, \mathbf{x}')[3i \ldots 3i+2] = \frac{1}{m_i} \left(G_i + D_i(\mathbf{x}_i') + \sum_{j \text{ is connected to } i \text{ with a spring}} S_{ij}(\mathbf{x}_i, \mathbf{x}_j) \right).$$

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

$$\mathbf{x}''(t) = \mathbf{A}(\mathbf{x}(t), \mathbf{x}'(t)).$$

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

$$\begin{bmatrix} \mathbf{x}'(t) \\ \mathbf{v}'(t) \end{bmatrix} = \begin{bmatrix} \mathbf{v}(t) \\ \mathbf{A}(\mathbf{x}(t), \mathbf{v}(t)) \end{bmatrix}.$$

To put this in the form of Eqn. 1, we define our state $\mathbf{X}$ as the position and velocity of all of the particles in our system:

$$\mathbf{X}(t) = \begin{pmatrix} \mathbf{x}(t) \\ \mathbf{v}(t) \end{pmatrix},$$

which then gives us

$$\frac{\mathrm{d}\mathbf{X}}{\mathrm{d}t}(t) = f(\mathbf{X}, t) = \begin{pmatrix} \mathbf{v}(t) \\ \mathbf{A}(\mathbf{x}(t), \mathbf{v}(t)) \end{pmatrix}.$$

Given a description of the pendulum system (i.e. the number of particles, their mass, and the springs connecting them), you should be able to use your time integrators to simulate the system. In the next sections, you will implement a simple two-particle pendulum and a multiple-particle pendulum forming a chain. Note that you *should not* need to modify your code for the Euler or trapezoidal integration.

4.3.2 Pendulum System

You will now implement the gravity force, viscous drag force, and spring force, and then create a four-particle pendulum system.

You will want to create a new derived class `PendulumSystem` of `ParticleSystemBase` for pendulums. Your implementation of the `ComputeTimeDerivative` method for `PendulumSystem` class should calculate the gravity force, the viscous drag force, and the spring forces that now act on your particle system. Additionally, You will need to provide public methods for `PendulumSystem` that allows adding a new particle of a certain mass value and adding a spring between two particles. You might also want to provide a public method to "fix" a particle - i.e., a method that sets a flag for a particle so that its position will be fixed throughout the simulation (one implementation is to set zero derivatives in `ComputeTimeDerivative`). Otherwise your particles will just fall out of the screen thanks to gravity.

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 (indices of the) two particles they act on, the rest length and the stiffness. You can alternatively store, for each particle, what other particles it is connected to and what the stiffness and rest length are.

It is recommended to test this first with a single particle connected to a fixed point by a spring. 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 should be more stable, but you will still want a little bit of viscous drag to keep the motion in check. *If you get this simple pendulum to work but not the multi-pendulum, you can still get at least 20% of the grade (i.e. half of the grade for the pendulum system part of the assignment).*

The next step is to extend to multiple particles. For submission purpose, four particles are enough. Connect the particles with springs to form a chain, and fix one of the endpoints. Make sure that you can simulate the motion of this chain correctly. 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.

At this point, implement the RK4 integrator. This will allow you to use more particles and larger step sizes than the trapezoidal integrator. *Finishing both the multiple-particle pendulum and the RK4 integrator gives you the remaining 20% of the grade of this part of the assignment.*

Optionally, to help with debugging, make a function that allows you to see (e.g., using `std::cout`) 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 don't need to visualize the spring for your submission - just the particles like in the sample solution are fine.

4.4 Cloth Simulation (40% of grade)

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 recommend taking a snapshot of your code (also known as "version control"), just in case the full cloth implementation does not work out and you need to revert the changes. For version control, we recommend Git.

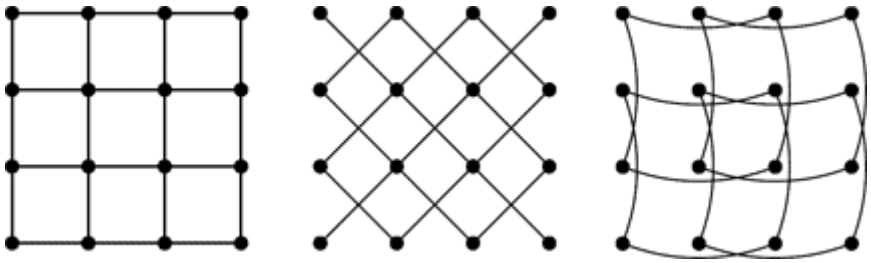


Figure 1: Left to right: structural springs, shear springs, and flex springs

We begin with a uniform grid of particles and connect them to their vertical and horizonal neighbors with springs (Figure 1). 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 *flex springs* to prevent the cloth from folding over onto itself. Notice that the flex springs are only drawn as curves in Figure 1 to make it clear that they skip a particle – they are still "straight" springs with the same force equation given earlier.

If you've designed your `PendulumSystem` class carefully from the previous section, you can reuse it for the cloth without creating another derived class of `ParticleSystemBase`. Then it shouldn't be too tough to add the necessary springs. Make sure that you use reasonable rest lengths. Write your code very carefully here; it is easy to make mistakes and connect springs to particles that don't exist, especially at boundaries of your cloth. We recommend that you create a helper method `IndexOf` that given index i, j from an $n \times n$ cloth, returns the index of the (i, j)th element in the list of particles (the particles should have indices from 0 to the number of particles minus 1).

First, implement structural springs. Draw the springs to make sure you've added the right ones before moving on. Run the simulation and you should obtain something that looks like a net. As usual, a reasonable amount of viscous drag helps prevent explosions. For faster debugging, use a small net of particles, e.g., 3×3. 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 the structural springs (which you should have already done to debug your structural spring implementation!). As a reminder from assignment 1, in GLOO you can draw line segments by changing the draw mode of the `RenderingComponent` (`component` reference variable here) associated with a node:

```
component.SetDrawMode(DrawMode::Lines);
```

Like in assignment 1, you will need to populate the position and index arrays of the corresponding `VertexObject` appropriately to draw the wireframe of the cloth as line segments (e.g. two consecutive indices describe the two endpoints of a segment). 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. This can be done similar to the smooth shading you hopefully implemented from skeletal subspace deformation in assignment 2.

Don't be too discouraged if your first result doesn't look good, or the simulator 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 or RK4 solver instead.

If you manage to have a moving wireframe cloth, then you're almost done. Make sure that resetting the the system with `R` works, unless you have implemented a fancier way to inject dynamics to your system (e.g. a moving sphere like in the sample solution). Adding more interesting interactions, such as collision with an object, will be considered for extra credit.

You may also find these notes on physically based modeling by David Baraff helpful, particularly the one on particle system dynamics.

5 Extra Credit

The list of extra credits below is a short list of possibilities. In general, physical 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.

5.1 Easy (3% for each)

- 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.
- 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, as done in the sample solution. 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. Think about how to deal with particle velocities when colliding with the moving sphere.

5.2 Medium (6% for each)

- Implement an adaptive solver scheme (look up adaptive Runge-Kutta-Felhberg techniques or check out the MATLAB `ode45` function).
- Implement an implicit integration scheme as described in this paper. 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.

5.3 Hard (9% for each)

- 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.

6 Submission Instructions

You are to include a `README.txt` file or PDF report that answers the following questions:

- How do you compile and run your code? Specify which OS you have tested on.
- 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. In particular, mention if you borrowed the model(s) used as your artifact from somewhere.
- 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 how you did it.
- Do you have any comments about this assignment that you'd like to share? We know this was a tough one, but did you learn a lot from it? Or was it overwhelming?

You should submit your entire project folder including your **source code** and **executable (at the root directory of your project)**, but **excluding** `external/` **and** `build/` since they take too much space. **Make sure your code can be built successfully, since we will compile it from scratch for grading.**

If you are on Windows, make sure the paths in your `#include "..."` pre-processing commands in `cpp/hpp` files use "/" instead of "\" as file separators. The later will not work on other operating systems.

To sum up, your submission should be your entire project folder, plus:

- The `README.txt` file or PDF report answering the questions above. Leave this file at the root directory of the project folder.
- Images of three simulation systems (`SimpleSystem`, `Multiple Particle Chain` and `Cloth`) in PNG or JPEG format. Make sure you submit multiple images of each system so that TAs can judge whether the particles are moving naturally from the images. You can either embed them in your PDF report or leave them at the root directory of the project folder.
- The executable file at the root directory of the project folder.

Please compress your project folder into a `.zip` file and submit using Canvas.
We will follow the late day policy explained in Lecture 1.