Final Report: Fast Simulation of Mass-Spring Systems

NAME: PEIJUN XU, KECHENG YE, JINXI XIAO
STUDENT NUMBER: 2021533041, 2021533151, 2021533005

INTRODUCTION

The Solver for time integration of mass-spring (classical linear (Hookean) springs) system is of significant importance. Solving the standard form of implicit Euler using Newton's method costs time, but results in accurate results. On the other hand, (semi-implicit) symplectic Euler's method (the one we used in Assignment 5) is fast but produces inaccurate results. Therefore, we adopt the method from [2] as a solution to both fast and accurate simulation process.

1 BACKGROUND AND NOTATION

We assume a mechanical system with m points in 3D with s springs, evolving through a discrete set of time samples $t_1, t_2, \dots$ with constant time step h. Let us denote the system configuration in time t_n as $\mathbf{q}_n \in \mathbb{R}^{3m}$. The system evolves in time according to Newton's laws of motion, where forces are represented by function $\mathbf{f} : \mathbb{R}^{3m} \to \mathbb{R}^{3m}$, so that $\mathbf{f}(\mathbf{q}_n)$ is the vector of forces acting on all particles at time t_n. We assume the forces are conservative, i.e., $\mathbf{f} = -\nabla E$, where $E : \mathbb{R}^{3m} \to \mathbb{R}$ is a potential function encompassing both internal and external forces. The task is to calculate all system states.

Given the diagonal (lumped) mass-matrix $M \in \mathbb{R}^{3m\times 3m}$, implicit Euler time integration results in the following update rules:

$$\mathbf{q}_{n+1} = \mathbf{q}_n + h\mathbf{v}_{n+1} \tag{1}$$

$$\mathbf{v}_{n+1} = \mathbf{v}_n + hM^{-1}\mathbf{f}(\mathbf{q}_{n+1}) \tag{2}$$

where $\mathbf{v}_n$ represents velocity at time t_n. With some simple reformulations, we can obtain

$$\mathbf{q}_{n+1} - 2\mathbf{q}_n + \mathbf{q}_{n-1} = h^2M^{-1}\mathbf{f}(\mathbf{q}_{n+1}) \tag{3}$$

and also a simplified version:

$$M(\mathbf{x} - \mathbf{y}) = h^2\mathbf{f}(\mathbf{x}) \tag{4}$$

where $\mathbf{x} \doteq \mathbf{q}_{n+1}$ and $\mathbf{y} \doteq 2\mathbf{q}_n - \mathbf{q}_{n-1}$.

The solution of Equation 4 is to find the minimum of a optimization problem:

$$\min_{\mathbf{x}} g(\mathbf{x}) = \frac{1}{2}(\mathbf{x} - \mathbf{y})^T M(\mathbf{x} - \mathbf{y}) + h^2E(\mathbf{x}) \tag{5}$$

2 METHOD

To successfully calculate the minimum of Equation 5, we need to first define the energy term $E(\mathbf{x})$. Authors come up with an idea of introducing auxiliary unknown variables to define the potential energy terms of springs, which is listed in the following Lemma:

Lemma 2.1.

$$E_{spring} = \frac{1}{2}k(||\mathbf{p}_1-\mathbf{p}_2||-r)^2 \propto (||\mathbf{p}_1-\mathbf{p}_2||-r)^2 = \min_{||\mathbf{d}||=r} ||(\mathbf{p}_1-\mathbf{p}_2)-\mathbf{d}||^2$$

where $\mathbf{p}_{\{1,2\}}$ are the endpoints of the spring, $\mathbf{d}$ is the unit direction of the spring and r is the rest length.

Therefore, we define the energy term as an optimization problem:

$$E(\mathbf{x}) = \min_{\mathbf{d}\in U} \frac{1}{2}\mathbf{x}^T L\mathbf{x} - \mathbf{x}^T J\mathbf{d} - \mathbf{x}^T\mathbf{f}_{\text{ext}} \tag{6}$$

where $L \in \mathbb{R}^{3m\times 3m}$ is a stiffness-weighted Laplacian of the mass-spring system graph, $J \in \mathbb{R}^{3m\times 3s}$ is a matrix connects mass points with springs, $U = \{(\mathbf{d}_i, \dots \mathbf{d}_s) \in \mathbb{R}^{3s} \mid ||\mathbf{d}_i|| = r_i\}$ is the set of rest-length spring directions and $\mathbf{f}_{\text{ext}} \in \mathbb{R}^{3m}$ denotes external forces.

The specific derivation is as follows, the energy term can be calculated as

$$E(x) = \frac{1}{2}\sum_{i=1}^{s} k_i||(\mathbf{p}_{i,1} - \mathbf{p}_{i,2}) - \mathbf{d}_\mathrm{i}||^2 - \mathbf{x}^T\mathbf{f}_{\text{ext}} \tag{7}$$

We first define a vector $A_i^T = [a_{i,0}, ..., a_{i,m}]$ where

$$a_{i,j} = \begin{cases} 1 & \text{if j is the first particle index of spring i} \\ -1 & \text{if j is the second particle index of spring i} \\ 0 & \text{others} \end{cases}$$

Thus, $\mathbf{p}_{i,1} - \mathbf{p}_{i,2}$ is equal to $A_i^T x$, and we have

$$E(x) = \frac{1}{2}\sum_{i=1}^{s} k_i||A_i^T x - \mathbf{d}_i||^2 - \mathbf{x}^T\mathbf{f}_{\text{ext}} \tag{8}$$

After further processing, the equation can be simplified to

$$E(x) = \frac{1}{2}\sum_{i=1}^{s} k_i x^T A_i A_i^T x - \sum_{i=1}^{s} k_i x^T A_i d_i - \mathbf{x}^T\mathbf{f}_{\text{ext}} \tag{9}$$

Second, we define another vector $S_i^T = [\theta_{i,0}, ..., \theta_{i,m}]$ where

$$\theta_{i,j} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

Thus, the direction vector $\mathbf{d}_i$ is equal to $S_i^T\mathbf{d}_i$ and the final energy equation is equal to

$$E(x) = \frac{1}{2}x^T(\sum_{i=1}^{s} k_i A_i A_i^T)x - x^T(\sum_{i=1}^{s} k_i x^T A_i S_i^T)\mathbf{d} - \mathbf{x}^T\mathbf{f}_{\text{ext}} \tag{10}$$

Obviously, we have the matrices $L \in \mathbb{R}^{3m\times 3m}, J \in \mathbb{R}^{3m\times 3s}$ are defined as follows:

$$L = \left(\sum_{i=1}^{s} k_i A_i A_i^T\right) \otimes I_3, \quad J = \left(\sum_{i=1}^{s} k_i A_i S_i^T\right) \otimes I_3 \tag{11}$$

where the matrix $I_3 \in \mathbb{R}^{3\times3}$ is the identity matrix and $\otimes$ denotes Kronecker product. Therefore, the final optimized energy equation is equal to

$$E(x) = \frac{1}{2}x^T Lx - x^T Jd - \mathbf{x}^T \mathbf{f}_{\text{ext}} \tag{12}$$

Based on Equation 5 and Equation 6, we derive the final optimization problem as

$$\min_{\mathbf{x},\mathbf{d}\in U} g(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T (M + h^2 L)\mathbf{x} - h^2\mathbf{x}^T J\mathbf{d} - \mathbf{x}^T (h^2\mathbf{f}_{\text{ext}} + M\mathbf{y}) \tag{13}$$

To minimize problem 13, we can first fix **x** and solve for **d**, which is defined as a local step. Then we fix **d** and solve for **x** with

$$(M + h^2 L)\mathbf{x} = h^2 J\mathbf{d} + h^2\mathbf{f}_{\text{ext}} + M\mathbf{y}$$

and notice that $(M + h^2 L)$ is positive semi-definite thus we can decompose it at the first place to reduce computation costs.

3 IMPLEMENTATIONS

Our framework is based on Assignment 5, and we have implemented an algorithm to solve the optimization problem 13 with Eigen [1]. To further demonstrate the efficiency of our work, we have designed a few applications, and will be discussed in the following subsections.

3.1 Scratch Point

Scratch point is designed to facilitate dynamic interaction with the cloth simulation by allowing the user to manipulate a specific point on the cloth. This functionality is particularly useful in interactive scenarios, such as real-time simulation debugging or user-driven deformation of the cloth.

To find the scratch point, we need a origin (position of the camera) and a unit direction vector. Two variables are designed, the first is the scratch point, a 3D point on the cloth and the other is the scratch distance, which is the distance between the scratch point and the camera position. The implementation can be described in three primary scenarios:

(1) **Selecting a New Scratch Point**: If no scratch point is currently selected, the function searches for the closest particle within a predefined distance threshold to the origin. The search is performed by iterating through all particles and computing the perpendicular distance between each particle and the direction vector. The particle with the smallest distance within the threshold is selected as the new scratch point. The corresponding distance along the direction vector is stored as the scratch distance.

(2) **Updating an Existing Scratch Point**: If a scratch point has already been selected, its position is updated based on the origin and direction, which points in the direction determined by the current mouse coordinates relative to the screen dimensions. The new position is calculated by extending the direction vector by the current scratch distance. Then we modify the state of the point in $\mathbf{q}_n$ and $\mathbf{q}_{n-1}$ to the newly calculated position.

(3) **Clearing the Scratch Point**: If there's no selected scratch points and the cursor is not moving, we clear the scratch point by setting it to null and resetting the scratch distance to infinity. The results are shown below.

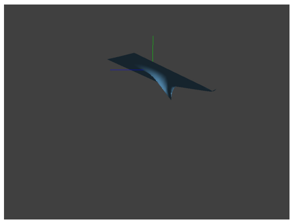
Fig. 1. Scratch 1

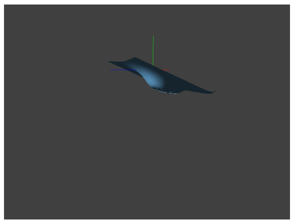
Fig. 2. Scratch 2

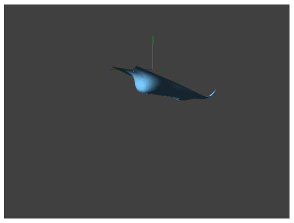
Fig. 3. Scratch 3

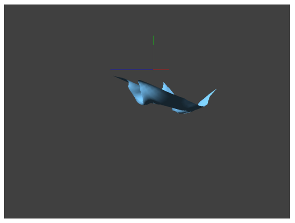
Fig. 4. Scratch 4

3.2 Cut Cloth

Cut cloth is designed to allow further interaction which can delete some part of the cloth. This functionality helps user to cut the cloth to the desired shape. If pressing key C, we will be in the cutting mode which is indicated by Bool variable "cut". Then we follow the method in 3.1 to select a scratch point. Bellow we will show the improvement process of our implementation.

(1) **Basic Implementation** The basic idea is that after the particle is selected, we invalidate the springs connected to it(totally 16 when the particle is not at boundary),so there will be no force give from this selected particle. During rendering, what we need is to delete all the triangles using this particle as a vertex, so, modify the indices vector is necessary. Using the basic idea, the hole we make is shown in Fig. 5, which is strange.

(2) **Add Triangle** The reason is that when the cloth is decomposed into triangles, the hypotenuse of a triangle is always from left-up to right-down, so a particle is used in 6 triangles in rendering, leading to asymmetrical hole. So after deleting, we add two triangles whose hypotenuse is from left-down to right-up, which can fill in the deleted parts that are inaccurate. Now the cutting hole is shown in Fig. 6. However, a new problem come up, if cutting the cloth in a special way, the cloth will not be cut totally as in Fig. 7.

(3) **Lucky Springs** The problem lies in the situation that in a square, if a left-down and right-up particle are cut, the left-up particle and right-down particle should not be connected, but the spring between them escape from the springs deleting process. So we add a new situation when deleting springs and fix this problem. We arrive at Fig. 8, which cut the cloth correctly.

3.3 Collision

Collision is a common interaction between clothing and other objects in daily life. In this project, we have implemented the collision

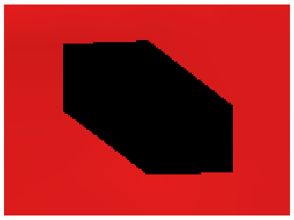
Fig. 5. basic cut cloth

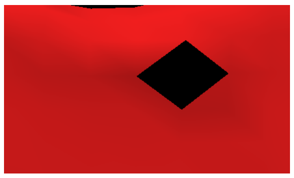
Fig. 6. add triangle

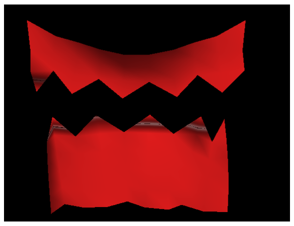
Fig. 7. cut but springs still exists

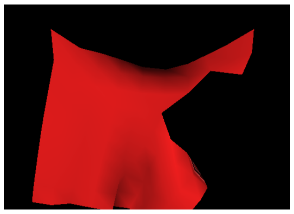
Fig. 8. finally

between the clothing and a sphere. The specific implementation details are as follows:

(1) **Mesh generation** First, we need to construct the sphere. In HW1, we built the object mesh by reading an '.object' file. However, in this project, collision detection requires explicit knowledge of the sphere's center and radius. Thus, the approach of reading '.object' files is unsuitable. Instead, we chose to directly generate the mesh using the sphere's center coordinates and radius. To begin, we need to determine the sphere's vertices. By setting a sampling rate, we sample points using spherical coordinates. Specifically, we iterate over the sphere's θ angle (ranging from 0 to π) and, for each fixed θ, iterate over the ϕ angle (ranging from 0 to 2π). This process yields a set of (θ_i, ϕ_i) coordinates, which we consider as the mesh vertices. For generating the indices, we connect points on the same θ layer to points on the adjacent layer, to form faces which enable us to create a dense mesh structure.

(2) **Collision Implementation** Next, for the collision detection between the clothing and objects, we use a simple approach to achieve realistic effects. After obtaining the updated position vector q_n of each particle in each time step using the Equation 5, we calculate the distance of each particle from the center of the sphere. For particles that are within the sphere (i.e., the distance from the center is less than the radius), we directly reset their position vectors to the closest point on the sphere's surface, ensuring that the particles do not penetrate into the sphere. The simulation then proceeds to the next time step. A pair of results are shown below.

4 CONCLUSION

This project implemented fast simulation of a mass-spring system, addressing collision detection and cloth cutting. By using optimized energy calculations, we achieved accurate simulations while maintaining computational efficiency. For cloth-object collisions, we generated the sphere mesh and adjusted particle positions to prevent penetration. The cutting functionality ensured accuracy by modifying springs and mesh faces. Dynamic interaction features enhanced realism and user experience. Future optimizations will improve stability and accuracy, supporting more complex simulations. Group members have equal contributions

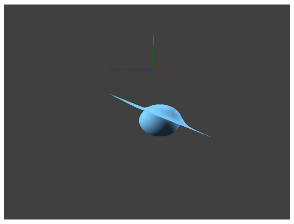
Fig. 9. Collision 1

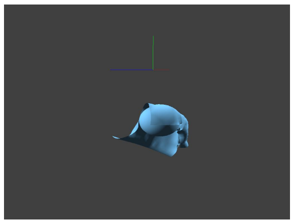
Fig. 10. Collision 2

REFERENCES

[1] Guennebaud, G., Jacob, B., et al. Eigen v3. http://eigen.tuxfamily.org, 2010.
[2] Liu, T., Bargteil, A. W., O'Brien, J. F., and Kavan, L. Fast simulation of mass-spring systems. *ACM Trans. Graph. 32*, 6 (Nov. 2013).