GAMES103 Lab 2: Cloth Simulation

Due Date: 12/20/2021, 11:59PM

In this lab assignment, you will learn how to write cloth simulation in a numerically stable fashion by two different approaches. You can select one approach to implement and your submission will be considered to be *complete*, as long as that approach works as intended. Having said that, you are encouraged to implement both approaches and you will receive 14 points in total, if both work fine.

As before, an example package contains the basic scene and scripts for this assignment. In the scene, there are a rectangular cloth piece and a sphere. You can click and drag the mouse to adjust the camera view. When you click and drag the mouse over the sphere, you can move it around. In the `Start` function, you will find the code for resizing the mesh of the cloth piece. By default, we resize the mesh to $21 \times 21 = 441$ vertices. After that, the function builds the edges and stores every vertex index pair into an array `E[]`. For simplicity, we do not use any bending spring. The function also calculates an array `L[]` to store initial edge lengths. Please read this function for your reference.

We assume that vertex 0 and vertex 20 are the two nodes remaining fixed in simulation. To achieve this, simply ignore the updates of these two vertices whenever they occur in this document.

1 Implicit Cloth Solver

1.a. Initial Setup (1 Point) For every vertex, apply damping to the velocity: $\mathbf{v}_i* = damping$ and calculate $\tilde{\mathbf{x}}_i$ as: $\tilde{\mathbf{x}}_i = \mathbf{x}_i + \Delta t \mathbf{v}_i$. After that, set $\mathbf{x}_i$ to its initial guess: $\mathbf{x}_i = \tilde{\mathbf{x}}_i$. (Note that $\mathbf{x}_i$ is an initial guess, not a real update. The solver works regardless of how we initialize $\mathbf{x}_i$, but a bad initial guess costs the solver more iterations to converge. You may try other initial guesses to see the difference. For example, do nothing.)

1.b. Gradient Calculation (2 Points) Next, we write a `Gradient` function to compute the gradient of the objective function. The input to this function is $\mathbf{x}$, $\tilde{\mathbf{x}}_i$ and the time step Δt. The output is the gradient $\mathbf{g}$. According to the lecture, the gradient is:

$$\mathbf{g} = \frac{1}{\Delta t^2}\mathbf{M}(\mathbf{x} - \tilde{\mathbf{x}}) - \mathbf{f}(\mathbf{x}), \tag{1}$$

in which $\mathbf{f}(\mathbf{x})$ is the force vector. To calculate the first part of Eq. 1, we loop through all of the vertices: $\mathbf{g}_i \leftarrow \frac{1}{\Delta t^2}\mathbf{m}_i(\mathbf{x}_i - \tilde{\mathbf{x}}_i)$. To apply the negated spring force, we loop through every edge e connecting i and j and add to $\mathbf{g}$:

$$\begin{cases} \mathbf{g}_i \leftarrow \mathbf{g}_i + k(1 - \frac{L_e}{\|\mathbf{x}_i - \mathbf{x}_j\|})(\mathbf{x}_i - \mathbf{x}_j) \\ \mathbf{g}_j \leftarrow \mathbf{g}_j - k(1 - \frac{L_e}{\|\mathbf{x}_i - \mathbf{x}_j\|})(\mathbf{x}_i - \mathbf{x}_j) \end{cases}. \tag{2}$$

Remember to apply the negated gravity force as well. (How to do it?)

1.c. Finishing (2 Points) The lecture talks about Newton's method to solve the nonlinear optimization problem for implicit cloth integration. In reality, there are two roadblocks: 1) the Hessian matrix is complicated to construct; 2) the linear solver is difficult to implement on Unity. Instead, we choose a much simpler method by considering the Hessian as a diagonal matrix. This yields a simple update to every vertex as:

$$\mathbf{x}_i \leftarrow \mathbf{x}_i - \left(\frac{1}{\Delta t^2}m_i + 4k\right)^{-1}\mathbf{g}_i. \tag{3}$$

This method works similar to Newton's method: first we calculate the gradient $\mathbf{g}$; and then we update all of the vertices by Eq. 3. We repeat this process 32 times so that $\mathbf{x}$ can be sufficiently good. Finally, we calculate the velocity $\mathbf{v} \leftarrow \mathbf{v} + \frac{1}{\Delta t}(\mathbf{x} - \tilde{\mathbf{x}})$ and assign $\mathbf{x}$ to `mesh.vertices`. (We will come back to nonlinear optimization a few weeks later and explain why this method works.)

1.d. Chebyshev Acceleration (1 Point) The Chebyshev semi-iterative method (Lecture 5 Page 26) also applies to nonlinear optimization. Can you try it out to the above and see the outcome?

1.e. Sphere Collision (2 Points) In the `Collision_Handling` function, calculate the distance from every vertex to the sphere center and use that to detect if the vertex is in collision. To obtain the sphere center $\mathbf{c}$, find the sphere as a `GameObject` by the `Find` function and then use `sphere.transform.position`.

Once a colliding vertex is found, apply impulse-based method as follows:

$$\mathbf{v}_i \leftarrow \mathbf{v}_i + \frac{1}{\Delta t}\left(\mathbf{c} + r\frac{\mathbf{x}_i - \mathbf{c}}{\|\mathbf{x}_i - \mathbf{c}\|} - \mathbf{x}_i\right), \qquad \mathbf{x}_i \leftarrow \mathbf{c} + r\frac{\mathbf{x}_i - \mathbf{c}}{\|\mathbf{x}_i - \mathbf{c}\|}. \tag{4}$$

For simplicity, you can hardcode the radius r as: $r = 2.7$. No friction is considered.

2 Position-Based Dynamics (PBD)

2.a. Initial Setup (2 Point) In the `Update` function, set up the PBD solver as a particle system. Specifically, for every vertex, damp the velocity (as in 1.a), update the velocity by gravity, and finally update the position: $\mathbf{x}_i \leftarrow \mathbf{x}_i + \Delta t \mathbf{v}_i$.

1.b. Strain limiting (4 Points) In the `Strain_Limiting` function, implement position-based dynamics in a Jacobi fashion. The basic idea is to define two temporary arrays `sum_x`[] and `sum_n`[] to store the sums of vertex position updates and vertex count updates. At the beginning of the function, set both arrays to zeros. After that, for every edge e connecting i and j, update the arrays:

$$\begin{aligned} sum\_\mathbf{x}_i &\leftarrow sum\_\mathbf{x}_i + \tfrac{1}{2}\left(\mathbf{x}_i + \mathbf{x}_j + L_e\tfrac{\mathbf{x}_i - \mathbf{x}_j}{\|\mathbf{x}_i - \mathbf{x}_j\|}\right), \quad sum\_n_i \leftarrow sum\_n_i + 1, \\ sum\_\mathbf{x}_j &\leftarrow sum\_\mathbf{x}_j + \tfrac{1}{2}\left(\mathbf{x}_i + \mathbf{x}_j - L_e\tfrac{\mathbf{x}_i - \mathbf{x}_j}{\|\mathbf{x}_i - \mathbf{x}_j\|}\right), \quad sum\_n_j \leftarrow sum\_n_j + 1. \end{aligned} \tag{5}$$

Finally, update each vertex as:

$$\mathbf{v}_i \leftarrow \mathbf{v}_i + \frac{1}{\Delta t}\left(\frac{0.2\mathbf{x}_i + sum\_\mathbf{x}_i}{0.2 + sum\_n_i} - \mathbf{x}_i\right), \quad \mathbf{x}_i \leftarrow \frac{0.2\mathbf{x}_i + sum\_\mathbf{x}_i}{0.2 + sum\_n_i}. \tag{6}$$

Note that we should execute this function multiple times to ensure that edge lengths are well preserved. Otherwise, you may see overly stretching artifacts.

1.e. Sphere Collision (2 Points, or 0 if you already did 1.e.) As in 1.e.

3 Submission Guideline

Save all of your files, including scene and script files, and export them into a package. Submit your package by the SmartChair system.