Soft Body Simulation With Finite Element Method

Jingwei Xu
SIST
xujw2023@shanghaitech.edu.cn

1. Introduction

In this report, the author simulates 3D soft body with finite element method with both an explicit time integration method and an implicit time integration method. The author also compares the performance of these two methods. Thanks to the simplicity of Taichi[1], the author also compares the performance of between GPU and CPU version FEM.

2. Methods and Implementation

2.1. Kinematics Theory

Kinematics is the study of motion within continuum materials, focusing primarily on the changes in shape or deformation that occur

2.1.1. Deformation

The deformable map is a 3D grid with each cell representing a tetrahedron. The tetrahedron is defined by four vertices. The vertices are connected by edges and the edges are connected by faces. The faces are connected by tetrahedrons. The tetrahedrons are connected by the deformable map. The deformable map is a graph with tetrahedrons as nodes and faces as edges. The deformable map is used to calculate the deformation of the soft body.

The formulation of the deformable is as follows in my implementation:

$$F = D_s D_m^{-1} \tag{1}$$

$$D_s = [x_1 - x_0, x_2 - x_0, x_3 - x_0], D_m = [X_1 - X_0, X_2 - X_0, X_3 - X_0] \tag{2}$$

$\boldsymbol{x}$ is the current position of the tetrahedron, $\boldsymbol{X}$ is the initial/rest position of the tetrahedron. F is the deformation gradient.

2.1.2. Strain Energy

Previous research has already shown the required property of the strain energy definition, such as sensitivity to the deformation gradient, invariance to rigid body motion, and convexity. Here I use the Neo-Hookean model to define the strain energy in my implementation:

$$\Psi = \frac{\mu}{2}(\operatorname{tr}(FF^T) - d) - \mu \ln(J) + \frac{\lambda}{2}\ln^2(J) \tag{3}$$

μ, λ is computed from pre-defined Young's modulus and Poisson's ratio. d is the dimension of the space. J is the determinant of the deformation gradient.

2.1.3. Stress

The stress is the derivative of the strain energy with respect to the deformation gradient. The stress is used to calculate the force on the tetrahedron. The formulation of the 1st Piola-Kirchhoff stress tensor is as follows in my implementation:

$$P = \mu\left(F - F^{T^{-1}}\right) + \lambda \ln(J) F^{T^{-1}} \tag{4}$$

2.2. Time Integration

From Figure 1, we can observe the different property of these different time integration methods. I use Symplectic Euler for my explicit time integration method and I choose conjugate gradient solver for my implicit time integration method.

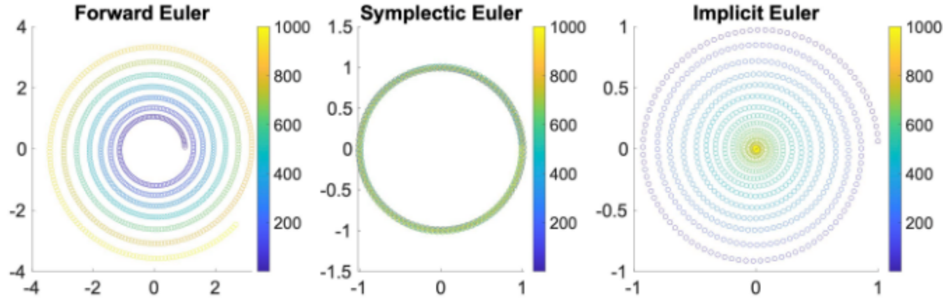

Figure 1: The forward Euler simulation eventually undergoes an unstable escalation, the Symplectic Euler closely adheres to the theoretical trajectory, and the implicit Euler, while maintaining stability, gradually brings the motion to a halt. [2]

2.2.1. Symplectic Explicit Time Integration

The formulation of the Symplectic Euler is as follows in my implementation:

- Symplectic Euler Time Integration:

$$x^{n+1} = x^n + \Delta t v^{n+1} \tag{5}$$

$$v^{n+1} = v^n + \Delta t M^{-1} f^n \tag{6}$$

After we compute the P, we can calculate the force on the tetrahedron. The force is used to calculate the acceleration of the tetrahedron. The acceleration is used to calculate the velocity of the tetrahedron. The velocity is used to calculate the position of the tetrahedron.

2.2.2. Implicit Time Integration

- Backward Euler Time Integration:

$$x^{n+1} = x^n + \Delta t v^{n+1} \tag{7}$$

$$v^{n+1} = v^n + \Delta t M^{-1} f^{n+1} \tag{8}$$

This integration process can be concluded into a linear system of equations:

$$\left(M - h\frac{\partial f}{\partial v} - h^2\frac{\partial f}{\partial x}\right)\Delta v = h\left(f + h\frac{\partial f}{\partial x}v_n\right) \tag{9}$$

with $\frac{\partial f}{\partial v}$ equals to 0. And h is the time step. $\frac{\partial f}{\partial x}$ is the stiffness matrix, which can also be seen as the Hessian of the strain energy.

The Hessian of Neo-Hookean strain energy is refered from [3].

Then the conjugate gradient solver is used to solve the linear system of equations to get the velocity of the next time step.

3. Results and Analyses

3.1. Analyses between Explicit and Implicit Time Integration

The explicit time integration method is more unstable than the implicit time integration method. The explicit time integration method eventually undergoes an unstable escalation, while the implicit time integration method maintains stability. Although symplectic explicit time integration will be conditionally stable, but the joggling of the soft body is still visible.

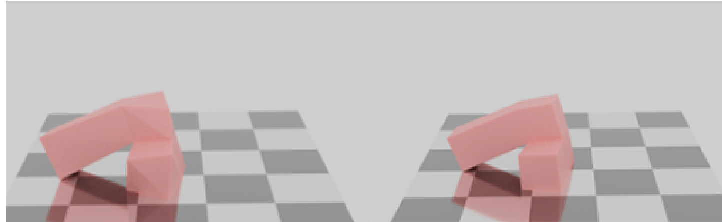

Figure 2: Left is the explicit time integration method, right is the implicit time integration method. The implicit time integration method is more stable than the explicit time integration method.

3.2. Analyses between GPU and CPU

Out of my previous expectation, the GPU version of FEM is slower than the CPU version of FEM. The CPU version is able to maintain 30 FPS, but GPU version is only able to maintain 1-2 FPS. (Both are without heavy rendering.)

Let's run the implicit method for example. From the following CPU profiler of my program:

```
=========================================================================
Kernel Profiler(count, default) @ X64
=========================================================================
[      %     total   count |      min       avg       max   ] Kernel name
-------------------------------------------------------------------------
[ 66.17%   0.042 s    240x |    0.132     0.176     0.421 ms]
assembly_c86_0_kernel_1_range_for
[  5.34%   0.003 s    240x |    0.006     0.014     0.138 ms]
CG_c88_0_kernel_0_range_for
[  5.03%   0.003 s    240x |    0.004     0.013     0.141 ms]
compute_force_c82_0_kernel_0_range_for
[  4.86%   0.003 s    240x |    0.006     0.013     0.097 ms]
assembly_c86_0_kernel_2_range_for
[  4.63%   0.003 s    240x |    0.004     0.012     0.087 ms]
assembly_c86_0_kernel_0_range_for
[  4.56%   0.003 s    240x |    0.004     0.012     0.097 ms]
time_integration_c84_0_kernel_0_range_for
[  2.59%   0.002 s    240x |    0.002     0.007     0.088 ms]
compute_force_c82_0_kernel_1_range_for
[  1.67%   0.001 s    240x |    0.002     0.004     0.046 ms]
assembly_c86_0_kernel_4_range_for
[  1.35%   0.001 s    240x |    0.000     0.004     0.068 ms]
CG_c88_0_kernel_2_range_for
[  1.23%   0.001 s    240x |    0.001     0.003     0.070 ms]
assembly_c86_0_kernel_3_range_for
[  0.98%   0.001 s    240x |    0.000     0.003     0.047 ms]
assembly_c86_0_kernel_5_range_for
[  0.37%   0.000 s    240x |    0.000     0.001     0.005 ms]
CG_c88_0_kernel_3_serial
[  0.20%   0.000 s   1728x |    0.000     0.000     0.001 ms]
snode_reader_16_kernel_0_serial
```

```
[  0.18%   0.000 s   1728x |    0.000     0.000     0.001 ms]
snode_reader_18_kernel_0_serial
[  0.17%   0.000 s   1728x |    0.000     0.000     0.001 ms]
snode_reader_17_kernel_0_serial
[  0.17%   0.000 s      8x |    0.007     0.013     0.023 ms]
projection_c80_0_kernel_0_range_for
[  0.17%   0.000 s   1152x |    0.000     0.000     0.001 ms]
snode_reader_7_kernel_0_serial
[  0.16%   0.000 s   1152x |    0.000     0.000     0.002 ms]
snode_reader_8_kernel_0_serial
[  0.14%   0.000 s   1152x |    0.000     0.000     0.001 ms]
snode_reader_9_kernel_0_serial
[  0.04%   0.000 s    240x |    0.000     0.000     0.001 ms]
CG_c88_0_kernel_1_serial
-------------------------------------------------------------------------
[100.00%] Total execution time:   0.064 s   number of results: 20
=========================================================================
```

We can observe that the main bottleneck of the program is the assembly kernel, which includes the computation of the stiffness matrix.

I have to mention that although usually $\frac{\partial f}{\partial x}$ will never be explicitly constructe, which is mentioned in [4]. But I still construct the stiffness matrix explicitly in my implementation because the test case is not so complicated.

Then I run the GPU profiler of my program:

```
=========================================================================
Kernel Profiler(count, default) @ CUDA on NVIDIA GeForce RTX 2050
=========================================================================
[      %     total   count |      min       avg       max   ] Kernel name
-------------------------------------------------------------------------
[ 31.13%   0.070 s    240x |    0.289     0.293     0.321 ms]
assembly_c86_0_kernel_1_range_for
[  8.98%   0.020 s   1728x |    0.007     0.012     0.846 ms]
snode_reader_16_kernel_0_serial
[  8.80%   0.020 s   1728x |    0.006     0.012     0.543 ms]
snode_reader_18_kernel_0_serial
[  8.49%   0.019 s   1728x |    0.007     0.011     0.067 ms]
snode_reader_17_kernel_0_serial
[  6.60%   0.015 s    240x |    0.058     0.062     0.138 ms]
CG_c88_0_kernel_3_serial
[  5.85%   0.013 s   1152x |    0.007     0.011     0.081 ms]
snode_reader_8_kernel_0_serial
[  5.84%   0.013 s   1152x |    0.007     0.011     0.047 ms]
snode_reader_7_kernel_0_serial
[  5.69%   0.013 s   1152x |    0.005     0.011     0.067 ms]
snode_reader_9_kernel_0_serial
[  3.44%   0.008 s    240x |    0.028     0.032     0.070 ms]
assembly_c86_0_kernel_2_range_for
[  1.89%   0.004 s    240x |    0.010     0.018     0.082 ms]
CG_c88_0_kernel_0_range_for
[  1.72%   0.004 s    240x |    0.009     0.016     0.065 ms]
time_integration_c84_0_kernel_0_range_for
[  1.62%   0.004 s    240x |    0.007     0.015     0.049 ms]
compute_force_c82_0_kernel_1_range_for
[  1.55%   0.004 s    240x |    0.007     0.015     0.101 ms]
```

```
assembly_c86_0_kernel_5_range_for
[  1.52%   0.003 s    240x |    0.007     0.014     0.077 ms]
compute_force_c82_0_kernel_0_range_for
[  1.52%   0.003 s    240x |    0.007     0.014     0.095 ms]
CG_c88_0_kernel_2_range_for
[  1.40%   0.003 s    240x |    0.004     0.013     0.057 ms]
assembly_c86_0_kernel_0_range_for
[  1.34%   0.003 s    240x |    0.007     0.013     0.071 ms]
assembly_c86_0_kernel_4_range_for
[  1.31%   0.003 s    240x |    0.007     0.012     0.130 ms]
CG_c88_0_kernel_1_serial
[  1.26%   0.003 s    240x |    0.006     0.012     0.117 ms]
assembly_c86_0_kernel_3_range_for
[  0.05%   0.000 s      8x |    0.008     0.014     0.032 ms]
projection_c80_0_kernel_0_range_for
-------------------------------------------------------------------------
[100.00%] Total execution time:   0.226 s   number of results: 20
=========================================================================
```

We can observe that a lot more time is taken to serial snode reader, which should be the data transfer between CPU and GPU.

Overall, since the test case is not so complicated, the data transfer between CPU and GPU is the main bottleneck of the program for the GPU version. So GPU version maybe more suitable for more complicated case for my implementation.

4. Summary

In this report, the author simulates 3D soft body with finite element method with both an explicit time integration method and an implicit time integration method. The author also compares the performance of these two methods. Thanks to the simplicity of Taichi[1], the author also compares the performance of between GPU and CPU version FEM. Easy case is more suitable for CPU version and the main bottleneck of the GPU version lies in the data transfer between CPU and GPU.

5. Acknowledgement

Besides Xiaopei Liu's lecture, my knowledge of FEM is also learned from [3], [4], [2] and Tiantian Liu's online lecture. Thanks for all these educational resources.

Bibliography

[1] Y. Hu, T.-M. Li, L. Anderson, J. Ragan-Kelley, and F. Durand, "Taichi: a language for high-performance computation on spatially sparse data structures," *ACM Transactions on Graphics (TOG)*, vol. 38, no. 6, p. 201–202, 2019.

[2] M. Li and C. Jiang, *Physics-Based Simulation (V1.0.0)*. 2024.

[3] T. Kim and D. Eberle, "Dynamic deformables: implementation and production practicalities (now with code!)," in *ACM SIGGRAPH 2022 Courses*, in SIGGRAPH '22. Vancouver, British Columbia, Canada: Association for Computing Machinery, 2022. doi: 10.1145/3532720.3535628.

[4] E. Sifakis and J. Barbic, "FEM simulation of 3D deformable solids: a practitioner's guide to theory, discretization and model reduction," in *ACM SIGGRAPH 2012 Courses*, in SIGGRAPH '12. Los Angeles, California: Association for Computing Machinery, 2012. doi: 10.1145/2343483.2343501.