致谢

时光荏苒，岁月如梭，在求是园探寻计算机科学奥秘的四年时光告一段落了。每一年，我都有新的收获、新的成长，这离不开学校和互联网提供的丰富而优质的资源。感谢有幸相遇的同学和老师们，以及我遭遇的那些挫折和走的“弯路”——你们丰富了我的生活，增长了我的知识，开拓了我的视野。如今，我可以自信地宣称，这四年时光，我过得十分充实，完全没有浪费，更完全不会后悔。

我要特别感谢那些指引我走向计算机图形学研究的那些“因素”。每一部精美的影片，每一款优质的游戏，背后起支撑作用的技术原理和美术工艺令人心向往之。感谢闫老师为我打开了图形学的大门并在渲染上小试牛刀，感谢靓仔让我得以一窥基于数据的动画系统的全貌，感谢王锐老师把我引向了基于物理的动画这一迷人的领域，并为本毕业设计提供了高配置的开发环境。在旅途中，我已小有收获，仍需继续前进。

物理仿真，是呈现一个世界的规则支柱，是创造一个宇宙的技术基石。在这一浪漫的领域内，路漫漫其修远兮，吾将上下而求索。

最后，感谢行业，感谢国家，感谢时代。这个世界还在进步，未来仍然充满了希望，而我也大有可为。

摘要

当前，图形学社区针对布料、橡胶和软体等可变形体的仿真提出了许多新颖的算法。这些算法在性能和表现上可胜过工业界倾向于沿用的，通过基于位置的动力学（position based dynamics，PBD）仿真框架实现的较成熟的方法。在计算并行化成为高性能计算的一大发展趋势的当下，整合现有的高性能算法，利用现代图形处理器（GPU）的强并行计算能力，设计一个高性能弹性体仿真器已成为工业引擎的迫切需要。

本毕业设计基于投射动力学（projective dynamics，PD）仿真框架和 Nvidia CUDA GPU 计算平台，设计并实现了一款新颖的弹性体仿真器。该仿真器支持在 GPU 上并行完成局部约束求解和全局线性系统求解这两大计算流程，从而高效地实现了复杂场景或高分辨率模型的交互式速率甚至实时仿真。通过精简但稳健的离散碰撞检测，程序实现了复杂弹性体模型与简单凸几何体的实时碰撞检测与处理。仿真器提供了友好的图形用户界面（GUI），使用户可以方便地创建自己的沙盘模型进行仿真或调整参数。本文的实验部分（节 7）提供了大量交互式速率或实时仿真实例供参考，并与著名的工业界物理引擎 PhysX（NvCloth）和 Bullet 就弹性体仿真方面进行了对比。结果显示，本仿真器在布料仿真上的效率不逊于 PhysX（NvCloth）和 Bullet，而在视觉和物理表现上基本一致。在具体积弹性体的仿真上，本仿真器较 PhysX 使用了更高精度的四面体模型，效率低于 PhysX，但更多地还原了局部细节；在设置 Bullet 仿真的四面体模型与本仿真器一致后，本仿真器在效率和视觉表现上均优于 Bullet。

关键词：基于物理的模拟；弹性体；迭代求解器；GPU

Abstract

Recently, for deformable meshes such as cloths, rubbers and soft bodies, the graphics community has proposed various novel algorithms for simulation. These algorithms may surpass traditional and developed methods based on position-based dynamics (PBD) that the industries continue to use. While parallel computing has become a major trend in high-performance computing, there is a need for industrial engine to assemble existing efficient algorithms, utilize the strong parallel computing power of modern GPUs, and design a high-performance deformable mesh simulator.

In this project, a novel deformable mesh simulator based on projective dynamics (PD) and Nvidia CUDA GPU computing platform is designed and developed. The simulator can perform local constraint solving (local solve) and global linear system solving (global solve) on the GPU parallel, making it be able to simulate complex scenes or high-resolution models at an interactive rate or even in real time. By reduced but robust discrete collision detection, the simulator can perform real time collision detection and resolution between complex deformable mesh and simple convex geometry. Friendly GUI is offered in the simulator, helping user to create sandbox on their own and control the simulation or tune the parameters. Various instances of simulation are provided in Section 7 for reference purpose, and compare the simulator to the famous industrial physics engines PhysX (NvCloth) and Bullet. As a result, for cloth simulation, the simulator is not worse in efficiency than PhysX (NvCloth) and Bullet; for visual and physical behavior, the difference is negligible. For volumetric mesh simulation, the designed simulator uses a tetrahedron mesh with higher resolution than PhysX's, thus has a lower efficiency but shows more local details. After setting the simulated tetrahedron mesh in Bullet the same as the designed simulator, the simulator shows better result both in efficiency and visual behavior compared with Bullet.

Key words: Physics-based simulation; Deformable mesh; Iterative solver; GPU

目录

1 绪论

1.1 背景

生活中的弹性体包括布料、橡胶和可塑性软体等，它们不同于刚体和流体，既具有一定的延展性，又具有显著的整体性。对诸如此类的弹性体进行物理仿真，是当前图形学的热门话题。随着电影、游戏、设计和虚拟现实工业的迅速发展，以及现代 GPU 并行计算能力的大幅提升，如何高效地设计针对弹性体的仿真算法，成为了学术界和工业界长期共同的挑战。

在图形学产业充分发展之前，制造业就在使用力学上的有限元方法[1]（FEM）进行弹性体的仿真。对于实时交互型应用而言，这些方法的计算量往往过高，无法满足实际需求。1998 年，Baraff 提出仿真布料的隐式欧拉积分法[2]，这是第一个实时仿真弹性体的有效方法。随后，基于隐式欧拉积分的 shape matching[3]算法被提出，其具有极佳的运行时效率，但由于算法设计并非出于物理定律，因此可能导致一些不符物理规律的结果。基于位置的动力学[4]（position-based dynamics，PBD）在 shape matching 的基础上，将弹性体用约束建模，迭代投影约束，实现了效率与表现上的平衡，成为了工业界许多求解器实现的基本算法。投射动力学[5]（projective dynamics，PD）则在 PBD 的基础上，定义了有物理意义的势能项，将约束投影拆分成最小化约束流形和求解线性系统两个步骤，解决了 PBD 方法的若干问题，但在仿真速率上不具优势。后续的许多研究成果，基本上都以这两个方法之一为框架进行改进，如提出新的加速方法[6][7][8]，或是修复这些方法在某些方面的局限性[9][10]，抑或证明这些方法在某些配置下与其他论文方法的等价性[11][12]。可见，PBD 和 PD 都是弹性体模拟的通行算法框架，而且在实际应用中仍具备许多挖掘和改进价值，各类科研成果方兴未艾。

鉴于 PD 的提出时间较晚，实现较复杂，且效率相对不及 PBD，工业界仍倾向于沿用基于 PBD 的更加成熟的方法，但这些方法在表现和性能上均存在许多可改进之处。基于 PBD 的方法在物理上的理论依据不强，仅仅将模型作为粒子系统整体进行运动学模拟，与弹性相关上的约束处理则基于一个简单的非物理

的模型；而 PD 框架在设计之初就已经考虑到了这一点，使用各类约束来模拟弹性势能，以正确表现剪切、拉伸、收缩和体积/长度变化的效果。由于在设计上定义了约束势能，PD 在物理正确性上更胜 PBD 一筹。在实现上，PBD 的这一问题则表现为被模拟物体的弹性与网格的分辨率和迭代次数有关，在迭代次数过高时，往往产生 locking issue[13]，物体的弹性表现大大减弱，这使得迭代次数这一参数难以调节。Nvidia 推出的物理引擎，如 PhysX 和 FleX，均采用了 CUDA[14]在 GPU 上对 PBD 进行并行计算，但 PD 同样具有良好的可并行性[15][7][8]，通过实现高吞吐率的 GPU 算法，可以实现高性能的 PD 求解，这一特性可弥补 PBD 和 PD 在串行计算性能上的差距，从而达成更好的物理仿真结果。

整合现有的高性能算法，利用现代 GPU 的强并行计算能力，设计一个高性能弹性体仿真器已成为生产实际的迫切需要。基于以上分析，本毕业设计基于 PD 仿真框架和 Nvidia CUDA GPU 计算平台，设计并实现了一款新颖的弹性体仿真器，同时与工业界流行的仿真器（物理引擎）PhysX（NvCloth）和 Bullet 进行了比较。在各方面上，该仿真器的表现均可圈可点，之后将预期应用到国产工业引擎 RaysEngine 中，为实际生产实践做贡献。

1.2 同类产品研发情况

目前已有许多工业界的仿真器使用 PBD 作为算法框架。Nvidia PhysX 和 Nvidia FleX 均使用 PBD 模拟弹性体，但实现细节不同，前者同时也支持缓慢但物理精确的 FEM。自 PhysX 4.0 后，NvCloth 从 PhysX 中独立出来，成为 Nvidia 长期开发维护的工业级布料仿真器，算法实现基于 PBD。Bullet 也是知名的开源物理引擎，对弹性体的模拟方法同样基于 PBD。在 Unity 引擎广泛使用的布料仿真器 DynamicBone 基于 PBD，而其替代品 Automatic-DynamicBone 则基于快速质点-弹簧系统[11]，可看作 PBD 的某种特例。Unreal Engine 5 在弹性体仿真上集成了 Chaos 物理引擎，使用 XPBD[9]（一种 PBD 的改进）作为算法框架。虽然根据迄今为止的调研，基于 PD 的工业界仿真器仍未问世，但这不能否定 PD 相对 PBD 具有的独特优势——目前学术界在基于 PD 的方法上发表的文章相对 PBD

更多，也佐证了这一点。

本毕业设计在规模上仅包含约 5000 行代码，虽不能与这些专业开发的仿真器（物理引擎）在功能的广度上相比，但在弹性体仿真方面可谓五脏俱全，能够仿真所有超弹性模型[16]。在易用性上，本毕业设计可提供的约束类型均具有直观含义，且可供调节的参数不多，易于用户理解使用，在规模大小之外同工业界仿真器相比并无劣势。

1.3 本文结构

节 2 描绘了算法的数学物理原理与仿真器的技术路线和整体架构。节 3 、节 4 和节 5 分别探讨了局部约束求解、全局线性系统求解和离散碰撞检测这三个仿真器的重要算法系统的实现，包括数学原理和算法描述。为展现仿真器的 UI 实现及其易用性，节 6 介绍了其模型管理和图形用户界面（GUI）系统。节 7 展现了该仿真器对弹性体进行仿真的若干实验情况，并与工业界著名的 PhysX 和 Bullet 物理引擎就弹性体仿真方面进行了对比。最后，节 8 总结了本毕业设计的工作，并提出进一步的展望。

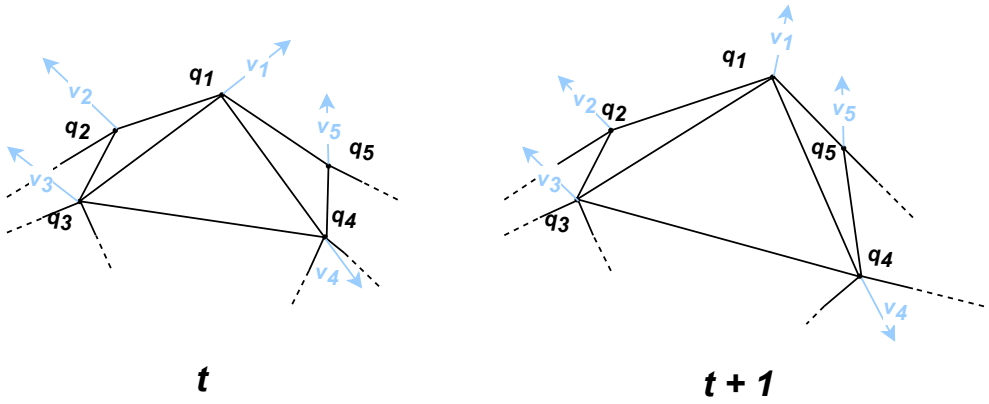


图 2.1 动力系统示意。图中仅展示 n 顶点模型在顶点集 $\{v_1, v_2, \ldots, v_5\}$ 的局部动力系统。$\boldsymbol{q}_i \in \mathbb{R}^3$，$\boldsymbol{v}_i \in \mathbb{R}^3$，$\boldsymbol{q} = (\boldsymbol{q}_i)^T$，$\boldsymbol{v} = (\boldsymbol{v}_i)^T$，$i \in \{1, 2, \ldots, n\}$。

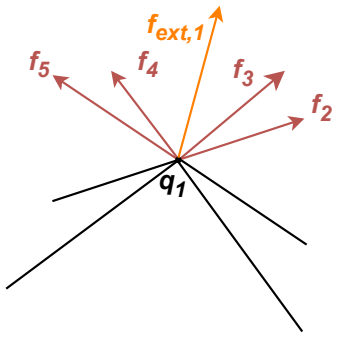


图 2.2 作用于顶点的力示意。图中 $\boldsymbol{f}_{ext,i} \in \mathbb{R}^3$，$\boldsymbol{f}_{ext} = (\boldsymbol{f}_{ext,i})^T$，$i \in \{1, 2, \ldots, n\}$。当为所有边建立弹簧（边张力约束）时，各边为两端顶点施加弹力，故 $\boldsymbol{f}_{int,1} = \boldsymbol{f}_2 + \boldsymbol{f}_3 + \boldsymbol{f}_4 + \boldsymbol{f}_5$。$\boldsymbol{f}_{int}$ 的一般形式与具体约束有关，故其定义为约束势能的负梯度之和 $-\sum_c \nabla_{\boldsymbol{q}} w_c(\boldsymbol{q})$。

2 实施方案与整体架构

为使本文叙述完备，接下来将形式化 PD 框架的计算流程，介绍程序采用的技术栈，并给出其整体架构。

2.1 数学约定与算法描述

给定一个顶点数为 n 的三维动力系统，记位置向量 $\boldsymbol{q} \in \mathbb{R}^{3n}$，速度向量 $\boldsymbol{v} \in \mathbb{R}^{3n}$，作用于各顶点的外力 $\boldsymbol{f}_{ext} \in \mathbb{R}^{3n}$，内力 $\boldsymbol{f}_{int}(\boldsymbol{q}) = -\sum_c \nabla_{\boldsymbol{q}} w_c(\boldsymbol{q})$ 为一关于 $\boldsymbol{q}$ 的函数，其中 $w_c : \mathbb{R}^{3n} \to [0, +\infty)$ 为约束 c 的势能函数。运用牛顿第二定

律，如图 2.1、图 2.2 所示，对时刻 t 到时刻 $t+1$ 进行隐式积分得到：

$$\begin{aligned}\boldsymbol{q}_{t+1} &= \boldsymbol{q}_t + h\boldsymbol{v}_{t+1},\\ \boldsymbol{v}_{t+1} &= \boldsymbol{v}_t + h\boldsymbol{M}^{-1}(\boldsymbol{f}_{int}(\boldsymbol{q}_{t+1}) + \boldsymbol{f}_{ext}),\end{aligned} \tag{2-1}$$

其中 $\boldsymbol{M} \in \mathbb{R}^{3n\times 3n}$ 为质量矩阵，h 为时间步长。将速度 $\boldsymbol{v}_{t+1}$ 代回，得到

$$\begin{aligned}&\boldsymbol{M}(\boldsymbol{q}_{t+1} - \boldsymbol{q}_t - h\boldsymbol{v}_t) = h^2(\boldsymbol{f}_{int}(\boldsymbol{q}_{t+1}) + \boldsymbol{f}_{ext}),\\ \rightarrow&\boldsymbol{M}(\boldsymbol{q}_{t+1} - \boldsymbol{q}_t - h\boldsymbol{v}_t - h^2\boldsymbol{M}^{-1}\boldsymbol{f}_{ext}) = h^2\boldsymbol{f}_{int}(\boldsymbol{q}_{t+1}).\end{aligned}$$

令

$$\boldsymbol{s}_t = \boldsymbol{q}_t + h\boldsymbol{v}_t + h^2\boldsymbol{M}^{-1}\boldsymbol{f}_{ext}, \tag{2-2}$$

它的物理意义为不考虑内力、碰撞等其他因素，顶点在当前状态下继续移动一时间步长后的位置。上式化为

$$\boldsymbol{M}(\boldsymbol{q}_{t+1} - \boldsymbol{s}_t) = h^2\boldsymbol{f}_{int}(\boldsymbol{q}_{t+1}) \tag{2-3}$$

式 2-3 传统上可使用牛顿迭代法[2]求解：

$$\boldsymbol{M}(\boldsymbol{q}^{(k+1)} - \boldsymbol{s}_t) = h^2(\boldsymbol{f}_{int}(\boldsymbol{q}^{(k)}) + \boldsymbol{K}(\boldsymbol{q}^{(k)})(\boldsymbol{q}^{(k+1)} - \boldsymbol{q}^{(k)})) \tag{2-4}$$

其中，迭代初值 $\boldsymbol{q}^{(0)} = \boldsymbol{q}_{t+1}$，$\boldsymbol{K}(\boldsymbol{q}^{(k)}) = \frac{\partial \boldsymbol{f}_{int}}{\partial \boldsymbol{q}}$ 为在 $\boldsymbol{q}^{(k)}$ 处的 Hessian（物理上是刚性矩阵）。这一方法收敛较快，但计算 Hessian 的代价很大，现已不再通行。PD 重新考虑了式 2-3，将其看作一个优化问题：

$$\arg\min_{\boldsymbol{q}_{t+1}} \frac{1}{2h^2}||\boldsymbol{M}^{1/2}(\boldsymbol{q}_{t+1} - \boldsymbol{s}_t)||^2 + \sum_c w_c(\boldsymbol{q}_{t+1}) \tag{2-5}$$

可对式 2-5 求梯度验证其在解上与式 2-3 的等价性。可见，$||\boldsymbol{M}^{1/2}(\boldsymbol{q}_{t+1} - \boldsymbol{s}_t)||^2$ 一项希望最小化 $\boldsymbol{q}_{t+1}$ 与 $\boldsymbol{s}_t$ 的“距离”，而 $\sum_c w_c(\boldsymbol{q}_{t+1})$ 则希望最小化约束势能。

PD 进一步基于超弹性假设指出[5]，势能项 $w_c(\boldsymbol{q}_{t+1})$ 具有二次形式

$$\sum_c w_c(\boldsymbol{q}_{t+1}) = \sum_c \frac{\omega_c}{2}||\boldsymbol{A}_c\boldsymbol{q} - \boldsymbol{A}'_c\boldsymbol{p}_c||^2, \tag{2-6}$$

其中 ω_c 是约束 c 的正权重，$\boldsymbol{p}_c$ 是 $\boldsymbol{q}$ 在约束 c 上的使势能为零的投影，$\boldsymbol{A}_c$ 和 $\boldsymbol{A}'_c$ 为两个与 c 的类型有关的常量矩阵，描述 $\boldsymbol{q}$ 与 $\boldsymbol{p}_c$ 的关系。例如，若 c 是一个弹簧，则式 2-6 中的 $\boldsymbol{A}_c\boldsymbol{q} \in \mathbb{R}^3$ 表示模拟时，弹簧两端顶点之间的位移向量，而 $\boldsymbol{A}'_c\boldsymbol{p}_c \in \mathbb{R}^3$ 则表示弹簧处于原长时，两端顶点之间的位移向量，如此，模拟时弹簧的弹性势能可由 $w_c(\boldsymbol{q}_{t+1})$ 描述。基于该模型，PD 将主要求解过程分成两步：

1. **局部约束求解（local solve, local step）**。这一步将当前位置 $\boldsymbol{q}$ 根据各约束 c 投影到 $\boldsymbol{p}_c$，求解优化问题

$$\min_{\boldsymbol{p}_c} \frac{\omega_c}{2}||\boldsymbol{A}_c\boldsymbol{q} - \boldsymbol{A}'_c\boldsymbol{p}_c||^2. \tag{2-7}$$

2. **全局线性系统求解（global solve, global step）**。这一步利用 local step 获取的各约束投影 $\boldsymbol{p}_c$ 求解一个线性方程组。具体地，通过令式 2-5 的梯度为零，可以得到如下的线性方程组

$$\left(\frac{\boldsymbol{M}}{h^2} + \sum_c \omega_c \boldsymbol{A}_c^T\boldsymbol{A}_c\right)\boldsymbol{x} = \frac{\boldsymbol{M}}{h^2}\boldsymbol{s}_t + \sum_c \omega_c \boldsymbol{A}_c^T\boldsymbol{A}'_c\boldsymbol{p}_c. \tag{2-8}$$

其中，系数矩阵 $\boldsymbol{A} = \frac{\boldsymbol{M}}{h^2} + \sum_c \omega_c \boldsymbol{A}_c^T\boldsymbol{A}_c$ 是常量实对称矩阵，因此可预计算。原始的 PD 框架即使用基于 Cholesky 分解的直接法求解该线性方程组，而本仿真器则支持 Cholesky 分解直接法和 Jacobi 迭代法两种算法进行求解。

求解 $\boldsymbol{x}$ 后，PD 的一次迭代即完成，令 $\boldsymbol{q}_{t+1} = \boldsymbol{x}$，即进入下一次迭代。迭代达到最大次数后即完成，更新各顶点的位置和速度。

以上的原始 PD 的主要求解过程可由算法 1 描述。可见，算法 1 第 5 行描述的即为局部约束求解（local step，式 2-7），而第 7 行描述的即为全局线性系统求解（global step，式 2-8）。记算法 1 第 3 行至第 5 行进行的迭代为 local-global 迭

算法 1: 原始 Projective Dynamics 求解框架

$\boldsymbol{s}_t = \boldsymbol{q}_t + h\boldsymbol{v}_t + h^2\boldsymbol{M}^{-1}\boldsymbol{f}_{ext}$;
$\boldsymbol{q}_{t+1} = \boldsymbol{s}_t$;
repeat
  forall constraint c **do**
    $\boldsymbol{p}_c = \texttt{ProjectOnConstraintSet}(c, \boldsymbol{q}_{t+1})$;
  end
  $\boldsymbol{q}_{t+1} = \texttt{SolveLinearSystem}(\boldsymbol{s}_t, \boldsymbol{p}_1, \boldsymbol{p}_2, \dots)$;
until reaches max #PDIteration;
$\boldsymbol{v}_{t+1} = (\boldsymbol{q}_{t+1} - \boldsymbol{q}_t)/h$;

代，简称 L-G 迭代。L-G 迭代的次数 `#PDIteration` 是 PD 算法的一个核心参数，一般设置为 2 到 10 即可满足绝大多数场景的仿真需求，该参数设置过大将导致计算缓慢，但计算结果不会有大的变化。PD 算法框架中，需要仿真的弹性体均包含若干约束，因此需要在仿真之前对这些约束进行预计算，以高效地完成每次 local-global 迭代。因此，本仿真器的求解过程将主要包括如下两部分：

- **预计算**，包括局部约束的预计算以及全局线性方程组系数矩阵的预计算，这一步在工业软件中也常被称为“烘焙”。在预计算流程中，首先将根据模型的 $\boldsymbol{M}$，不同约束的 $\omega_c\boldsymbol{A}_c^T\boldsymbol{A}_c$ 预构建系数矩阵 $\boldsymbol{A}$，并预计算 global step 中的线性求解器。同时，若用户要求使用 GPU 进行 local step，则求解器还需要将所有约束拷贝至 GPU 内存。

- **PD 仿真**，即算法 1 的流程。本仿真器对 PD 框架的实际计算流程与之略有不同，如算法 2 所示。在该计算流程中，主要加入了两次碰撞检测与处理函数 `ResolveCollision()`，输入参数 S_{co} 表示场景中可能与被仿真对象产生碰撞的物体的集合。此外，该流程根据实际情况调整了 L-G 迭代求解方法的传参，提高了求解效率。

算法核心流程,即 `Project()`、`SolveLinearSystem()` 和 `ResolveCollision()` 的具体实现，将在接下来的若干节详细介绍。

算法 2: 本仿真器实现的 Projective Dynamics 求解框架

$\boldsymbol{s}_t = \boldsymbol{q}_t + h\boldsymbol{v}_t + h^2\boldsymbol{M}^{-1}\boldsymbol{f}_{ext}$;
$\boldsymbol{s}_t = \texttt{ResolveCollision}(S_{co}, \boldsymbol{s}_t)$;
$\boldsymbol{q}_{t+1} = \boldsymbol{s}_t$;
repeat
  $\boldsymbol{b} = (\boldsymbol{M}/h^2)\boldsymbol{s}_t$;
  forall constraint c **do**
    $\omega_c \boldsymbol{A}_c^T \boldsymbol{A}_c' \boldsymbol{p}_c = \texttt{Project}(c, \boldsymbol{q}_{t+1})$;
    $\boldsymbol{b} = \boldsymbol{b} + \omega_c \boldsymbol{A}_c^T \boldsymbol{A}_c' \boldsymbol{p}_c$;
  end
  $\boldsymbol{q}_{t+1} = \texttt{SolveLinearSystem}(\boldsymbol{b})$;
until reaches max #PDIteration;
$\boldsymbol{q}_{t+1} = \texttt{ResolveCollision}(S_{co}, \boldsymbol{q}_{t+1})$;
$\boldsymbol{v}_{t+1} = (\boldsymbol{q}_{t+1} - \boldsymbol{q}_t)/h$;

2.2 技术栈简介

本仿真器是使用高效的 C++ 编程语言实现的跨平台程序，使用 CMake 进行构建，在 Arch Linux 上开发并测试，同时采用了如下关键技术栈，它们均是跨平台的，因此本毕业设计也可以方便地在 Windows 或 macOS 上部署运行。

- **OpenGL** 是用于渲染 3D 图形的跨语言、跨平台的应用程序编程接口（API），在本毕业设计中用于构建窗口，进行仿真场景的渲染。由于其应用广泛，社区丰富，对于图形学研究应用而言，OpenGL 基本上是首选的图形 API。

- **Dear ImGui** 是基于 C++，支持多种图形 API 的即时模式 GUI，在提供丰富的 UI 部件库的同时提供高效的渲染，是开发图形学实验应用的首选。

- **Eigen**[17]是基于 C++ 的开源数学库，内置的向量、矩阵数学结构为工程和科研人员提供了基础数学运算上的便捷。在本毕业设计中，Eigen 将被用于在 CPU 上表达大多数的向量、矩阵数学结构，并进行稀疏矩阵的构建和奇异值分解。

- **Nvidia CUDA**[14]是 Nvidia 推出的通用 GPU 计算平台，在持有相应设备的情况下，是开发高效并行计算程序的利器。本毕业设计聚焦于实现基于 GPU

的并行计算，故使用 CUDA 完成并行计算的开发。

- **libigl**[18]是图形学社区维护的高效 OpenGL 几何处理算法库和框架，因其方便性和实用性也常用于物理仿真的实验性算法开发。本毕业设计采用 libigl 精简了渲染和模型管理相关的代码，利用其 API 实现了各类模型文件的导入功能，以集中于物理仿真相关的实现。

- **TetGen**[19]和 **TetWild**[20]都是具有一定流行性的高效四面体网格生成算法，配套有开源代码库；一般而言，后者相对前者而言的算法稳健性更高。在具体积弹性体的仿真上，本毕业设计采用 TetGen 生成四面体网格，并在 TetGen 无法完成工作时转而使用 TetWild 生成。

2.3 整体架构

本毕业设计的整体架构如图 2.3 所示。其中各模块描述如下：

1. 用户能够接触到的程序表层模块为 GUI 模块和渲染模块。用户可操控 GUI 进行模型的增、删、改、查，控制算法参数，对指定顶点施加外力，等等。本毕业设计的渲染器通过 OpenGL 实现，实现了计算量较低的 Blinn-Phong 光照模型、基于 shadow mapping 的软阴影和简单的材质捕获（MatCap）技术。渲染器同时支持简单的调试功能，如显示线框、显示顶点的邻接情况等等。在模型的几何结构较复杂、约束数目较多时，渲染的计算代价相对物理仿真过程可忽略不计，因此不影响性能测试的进行。用户可通过系统反馈的渲染情况，一目了然地获知物理仿真结果。

2. 程序的模型管理（mesh manager）模块管理了仿真过程中各模型的类型、位置、速度和约束情况等信息，包括弹性体和可碰撞刚体的状态。用户可为这些模型分配不同的着色方式，设置不同的仿真约束等，以呈现不同的仿真效果。

3. 物理仿真算法模块包括 CPU 算法和 GPU 算法，其中预计算主要在 CPU 上进行，而 PD 仿真则可选择在 CPU 上或 GPU 上进行。在 PD 仿真阶段中，

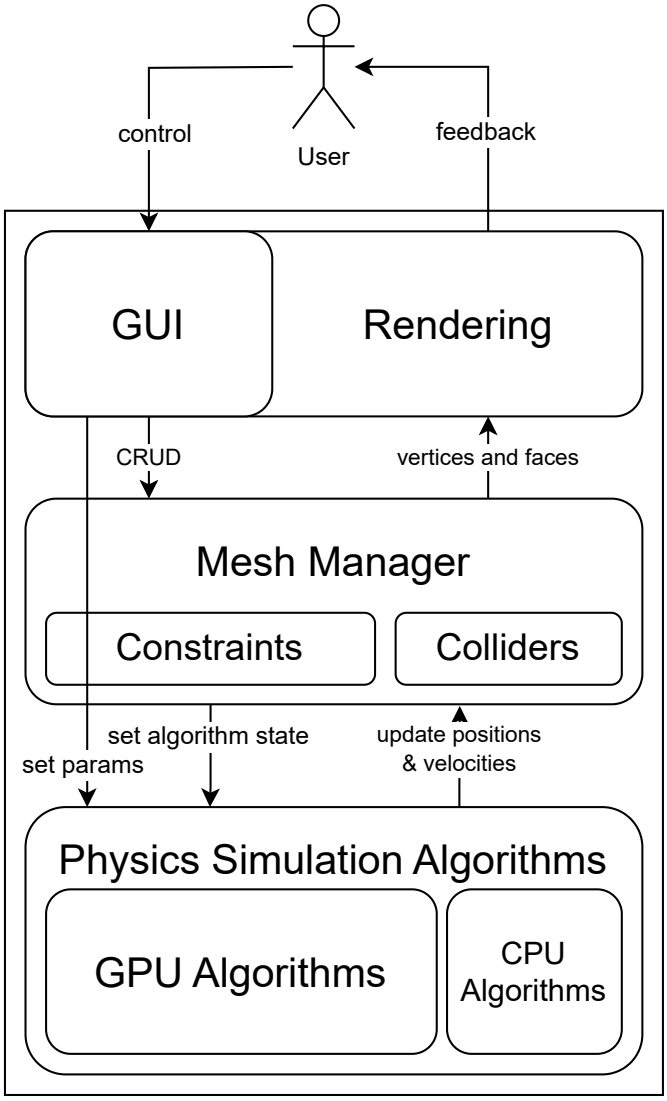


图 2.3 毕业设计的整体架构

local step 和 global step 这两个计算过程中的 CPU 和 GPU 算法将被用于性能对比，为 GPU 算法的高效性给出充分论证。该模块通过接收各模型的物理仿真信息和算法参数进行仿真计算，并在每一帧的物理仿真结束后，将所有仿真结果通过设置各模型的位置和速度，传递给模型管理模块；随后渲染模块对各模型进行渲染，即可将结果反馈至用户。

本毕业设计中，除技术栈中介绍的第三方库之外，不再依赖其他环境。下文将继续介绍本文的重点——高效的物理仿真算法的实现。

3 局部约束求解的实现

局部约束求解（local step，算法 2 第 7 行）是 PD 框架中易于并行化的一部分。为实现这一部分，首先，需要对局部约束进行数学建模，提取其抽象形式编程实现，并在实验中优化算法。

3.1 局部约束的建模

一般地，一个 PD 模拟过程的局部约束 c，是指一个 4 元组 $c = (\omega_c, V_c, \boldsymbol{A}_c, \boldsymbol{A}'_c)$，其中 ω_c 表示该约束的权重，V_c 表示该约束涉及的顶点集合，$\boldsymbol{A}_c$ 和 $\boldsymbol{A}'_c$ 为式 2-7使用到的投影矩阵，其中：

- $\boldsymbol{A}_c$ 的作用是选择当前顶点位置向量 $\boldsymbol{q}$ 中存在于 V_c 的部分，并通过与约束相关的计算流程，将其表达成一定的状态 $\boldsymbol{A}_c\boldsymbol{q}$。

- $\boldsymbol{A}'_c$ 的作用是在约束顶点集 V_c 中构造满足约束的状态。具体而言，记约束顶点集的大小 $|V_c| = k$，$\boldsymbol{p}_c = (\boldsymbol{p}_1, \ldots, \boldsymbol{p}_k)^T$ 为某一满足约束的顶点位置状态，则 $\boldsymbol{A}'_c\boldsymbol{p}_c$ 将这一顶点位置通过与约束相关的计算流程，表达成与 $\boldsymbol{A}_c\boldsymbol{q}$ 同一维度的形式。

可见，构造 $\boldsymbol{A}_c$ 和 $\boldsymbol{A}'_c$ 的目的就是计算式 2-6的二次势能形式，其中 $\boldsymbol{A}_c\boldsymbol{q}$ 表示“当前状态”，它不一定满足约束；$\boldsymbol{A}'_c\boldsymbol{p}_c$ 表示“某个满足约束的状态”，$\boldsymbol{p}_c$ 通常为需

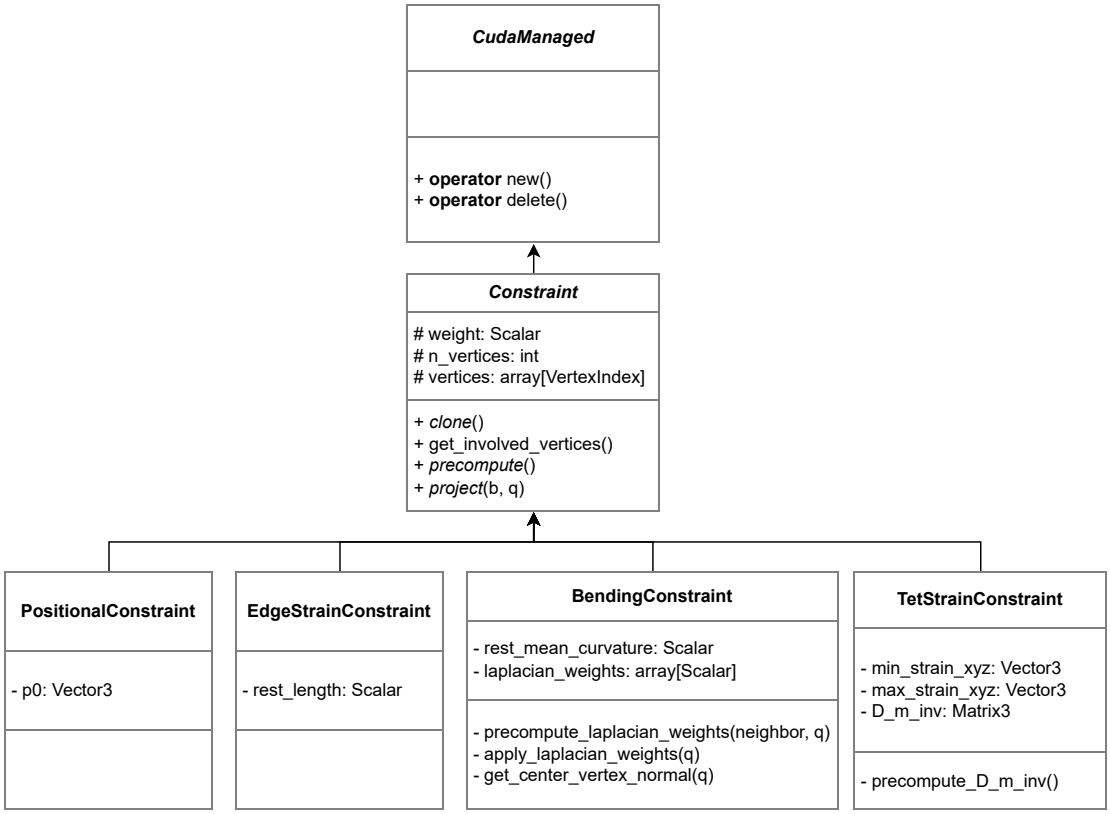


图 3.1 表达约束的 UML 类图

要求解的未知元。该势能项即最小化两者之间的距离，从而使系统在代价最小的情况下满足约束，在数学上可称“投影到约束流形（constraint manifold）”。

在预计算阶段，所有约束需要计算 $\omega_c \boldsymbol{A}_c^T \boldsymbol{A}_c$ 的值以在系数矩阵上累加；在 PD 仿真阶段，局部约束的求解结果 $\omega_c \boldsymbol{A}_c^T \boldsymbol{A}'_c \boldsymbol{p}_c$ 将加入 global step 线性方程组的常数项 $\boldsymbol{b}$ 中（算法 2，第 8 行）。

根据以上对约束计算的描述，容易想到使用 C++ 的多态实现约束求解。本毕业设计共实现了 4 种约束，包括位置约束（positional constraint）、边张力约束（edge strain constraint）、弯曲约束（bending constraint）和四面体张力约束（tetrahedron strain constraint）。约束相关的类图如图 3.1 所示。图中的 `Constraint` 为所有约束的共同父抽象类，包含若干用斜体表示的纯虚函数，如用于预计算的 `precompute()` 和用于 local step 的 `project()` 函数。`CudaManaged` 为定义了在 CUDA 统一内存（unified memory）上进行内存分配和释放的 `new` 和 `delete` 运算符的抽象类，用于所有约束类型，以方便在 GPU 上的约束求解。对每一个

具体约束，父类定义的纯虚函数都得到了实现，并使用了尽可能少的空间表达四元组 $(\omega_c, V_c, \boldsymbol{A}_c, \boldsymbol{A}'_c)$。

小节 3.6 将会介绍 GPU 并行求解的实现细节。接下来将具体介绍每种约束在仿真中的作用，明确其四元组表达，以阐述其对父类虚函数的实现方式。

3.2 位置约束

3.2.1 约束原理

位置约束用于将弹性体的顶点限制在某一位置，可用于仿真固定弹性体的指定顶点于某一指定位置的场景，如衣物的悬挂。改变该约束的权重可以影响该约束对顶点位置的控制程度，故较小权重的位置约束可用于模拟阻尼。

本仿真器仅设计了涉及单个顶点的位置约束，若要对多个顶点进行同一位置约束，则需要为每个顶点分别创建约束。当用户为弹性体的第 i 个顶点 v_i 设置位置约束时，被约束位置即为 v_i 此时的位置 $\boldsymbol{q}_i$。约束的描述如下：

- $V_c = \{v_i\}$。
- $\boldsymbol{A}_c$ 为 $3 \times 3n$ 分块矩阵，每块为 3×3，且第 i 块为 3×3 恒等矩阵 $\boldsymbol{I}_3$，其他块均为零矩阵，即 $\boldsymbol{A}_c = \begin{bmatrix} \boldsymbol{O}_3 & \dots & \boldsymbol{I}_3 & \dots & \boldsymbol{O}_3 \end{bmatrix}$。
- $\boldsymbol{A}'_c = \boldsymbol{I}_3$。

对于每次 local step，满足位置约束的顶点位置是唯一的，即 $\boldsymbol{p}_c = \boldsymbol{q}_i$，其中 $\boldsymbol{q}_i$ 为首次设置约束时，希望顶点固定于的位置。可见，该约束投影 $\boldsymbol{p}_c$ 与仿真时的 $\boldsymbol{q}$ 无关，甚至不需要进行求解。

3.2.2 约束实现

由于该约束仅与顶点 v_i 有关，故 $\omega_c \boldsymbol{A}_c^T \boldsymbol{A}_c$ 仅影响系数矩阵 $\boldsymbol{A}$ 的第 i 个 3×3 对角块的值，且 $\omega_c \boldsymbol{A}_c^T \boldsymbol{A}'_c \boldsymbol{p}_c$ 仅影响 $\boldsymbol{b}$ 的第 i 个 3×1 块的值，因此位置约束的实现是简单的，此处不再赘述。

3.3 边张力约束

3.3.1 约束原理

边张力约束在三角形网格的指定边添加指定弹性系数的弹簧，实现弹性体（如布料）整体具有的延展性效果，应用十分广泛。改变该约束的权重可影响弹性体的“刚度”。在布料中，仅建立边张力约束进行仿真与经典的“质点-弹簧模型”在数学上等价。

本仿真器支持对三角形网格整体添加权重一定的边张力约束，暂不支持非均匀约束的添加，而支持这一功能需要设计更友好的权重笔刷，不属于本毕业设计的核心内容。设弹性体顶点 v_i 和 v_j 之间恰有一条边 ij 相连，当用户为该边设置边张力约束时，记被约束边此时的长度，即原长（rest length）为 r_c。约束的描述如下：

- $V_c = \{v_i, v_j\}$。
- $\boldsymbol{A}_c$ 仍为 $3 \times 3n$ 分块矩阵，且第 i 块为 $\boldsymbol{I}_3$，第 j 块为 3×3 取反矩阵 $-\boldsymbol{I}_3$，其他块均为零矩阵，即 $\boldsymbol{A}_c = \begin{bmatrix} \boldsymbol{O}_3 & \dots & \boldsymbol{I}_3 & \dots & -\boldsymbol{I}_3 & \dots & \boldsymbol{O}_3 \end{bmatrix}$。
- $\boldsymbol{A}_c' = \begin{bmatrix} \boldsymbol{I}_3 & -\boldsymbol{I}_3 \end{bmatrix}$。

满足位置约束的顶点位置 $\boldsymbol{p}_c = (\boldsymbol{p}_i, \boldsymbol{p}_j)^T$ 仅需满足它们之间的距离 $||\boldsymbol{p}_i - \boldsymbol{p}_j|| = r_c$，使弹簧恢复原长，或等价地，最小化弹性势能。易见，此时式 2-7 对应的优化问题有唯一解

$$\boldsymbol{A}_c' \boldsymbol{p}_c = r_c \frac{\boldsymbol{q}_i - \boldsymbol{q}_j}{||\boldsymbol{q}_i - \boldsymbol{q}_j||}, \tag{3-1}$$

其中 $\boldsymbol{q}_i$ 和 $\boldsymbol{q}_j$ 分别对应仿真时顶点 v_i 和 v_j 的位置向量。

3.3.2 约束实现

由于边张力约束仅涉及两个顶点，实现边张力约束也并无难度，$\omega_c \boldsymbol{A}_c^T \boldsymbol{A}_c$ 仅影响系数矩阵 $\boldsymbol{A}$ 的 4 个 3×3 对角块的值，且 $\omega_c \boldsymbol{A}_c^T \boldsymbol{A}_c' \boldsymbol{p}_c$ 仅影响 $\boldsymbol{b}$ 的第 i 个和第 j 个 3×1 块的值，在 GPU 上计算式 3-1 也是简单的，此处不再赘述。

3.4 弯曲约束

3.4.1 约束原理

在弹性体仿真中，引入弯曲约束的设计是自然的——对于裙摆、软管等非原始平整网格而言，它们具有一定的“保弯曲”程度，使得它们在模拟过程中具有抵抗重力自然“展平”的能力。在微分几何领域，平均曲率（mean curvature）衡量了曲面的弯曲程度。于是，可以通过构建弯曲约束，获取该曲面在某一位置 v_i 的原始局部平均曲率 H_m；而在仿真过程中，可以衡量该位置在变形后的局部平均曲率 H_s，定义其与 H_m 共同决定的弯曲势能（bending energy）[21]，给出最小势能的投影位置，即可尽量保证局部平均曲率的不变性。

改变弯曲约束的权重可影响弹性体的“保弯曲”程度。该约束与边张力约束同时使用，可仿真几乎所有的可变形曲面，赋予不同的权重之比可获取不同的仿真效果。

如何计算离散化的三角形网格曲面在某一位置的平均曲率，并实现稳健性高的弯曲约束，是本毕业设计的难点之一，接下来将给出相应的数学推导。

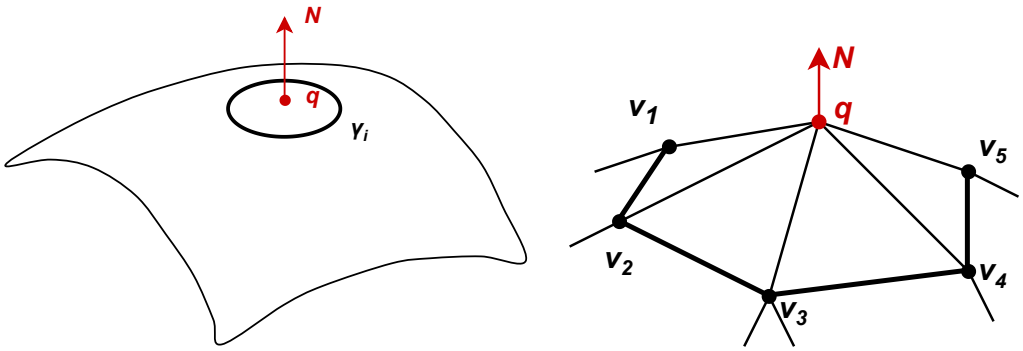
3.4.2 离散化 Laplace-Beltrami 算子计算平均曲率

考虑二维流形 S 在三维欧氏空间 $\mathbb{R}^3$ 的嵌入，设 $p \in S$ 为流形上任意一点，$\boldsymbol{q} \in \mathbb{R}^3$，定义坐标函数（coordinate function）$c : S \to \mathbb{R}^3$ 满足

$$c^{(i)}(p) = q^{(i)} \tag{3-2}$$

对分量 $i = 1, 2, 3$ 成立。微分几何领域的 Laplace-Beltrami 算子 Δ 是一个定义于 Banach 空间的映射，可以证明[22]，$\Delta c : S \to \mathbb{R}^3$ 满足

$$\Delta c(p) = H\boldsymbol{N}, \tag{3-3}$$



(a) 连续情形的 L-B 算子（式 3-5） **(b) 离散情形的 L-B 算子（式 3-6）**

图 3.2 L-B 算子对局部平均曲率向量的刻画

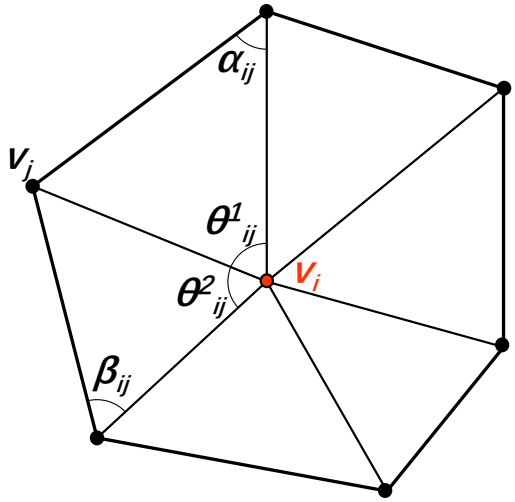


图 3.3 余切公式（式 3-7）和平均值公式（式 3-8）的符号示意

其中 H 表示流形 S 在 $c(p)$ 处的平均曲率，$\boldsymbol{N}$ 为 $c(p)$ 处的单位法向量。$H\boldsymbol{N}$ 在微分几何中也被称为**平均曲率向量**，其模长为平均曲率，方向与法向量相同。

在离散微分几何领域，应用于坐标函数的 Laplace-Beltrami 算子具有广泛应用，因此也常将 $\Delta c(p)$ 称作定义于 S 上的（局部）Laplace-Beltrami 算子（后简称 L-B 算子）。

L-B 算子的离散化是一个在计算几何领域得到广泛研究的问题[21][23][24][25]。作为一系列离散方法的启发的，是一个曲线积分

$$\frac{1}{|\gamma_i|}\int_{\boldsymbol{v}\in\gamma_i}(\boldsymbol{q}-\boldsymbol{v})\,\mathrm{d}l(\boldsymbol{v}), \tag{3-4}$$

其中 γ_i 为任意嵌入于 S，环绕 $\boldsymbol{q}$ 的闭合曲线，$|\gamma_i|$ 为其周长，如图 3.2a所示。Taubin 指出[23]，若令 γ_i 向点 $\boldsymbol{q}$ 不断收缩逼近，这一积分将收敛至 $\boldsymbol{q}$ 处的平均曲率 $H(\boldsymbol{q})$ 与 $\boldsymbol{q}$ 处的单位法向量 $\boldsymbol{N_q}$ 的乘积，即

$$\lim_{|\gamma_i|\to 0} \frac{1}{|\gamma_i|} \int_{\boldsymbol{v}\in\gamma_i} (\boldsymbol{q}-\boldsymbol{v})\,\mathrm{d}l(\boldsymbol{v}) = H(\boldsymbol{q})\boldsymbol{N_q} \tag{3-5}$$

对式 3-5 的证明工作详见附录A。对比式 3-3、式 3-5 可以发现，式 3-5 给出了对局部 L-B 算子的取值，同时很好地捕获了局部欧氏空间中的平均曲率和法向量信息。在离散化背景下，流形 S 被视作一个三角网格，此时的 L-B 算子即通过近似式 3-5 获取。

如图 3.2b 所示，离散化后，环绕 $\boldsymbol{q}$ 的闭合曲线 γ_i 变成了 $\boldsymbol{q}$ 的 1-邻接顶点围成的环。若 $\boldsymbol{q}$ 位于流形的边缘，则称此处的 L-B 算子不存在，无需建立弯曲约束。记 $\boldsymbol{q}$ 在离散化下对应曲面 S 的第 i 个顶点的位置 $\boldsymbol{q}_i$，则对 L-B 算子在 $\boldsymbol{q}_i$ 处的取值 δ_i 的朴素离散化方案为

$$\delta_i = \frac{1}{d_i} \sum_{j\in N(i)} (\boldsymbol{q}_i - \boldsymbol{q}_j), \tag{3-6}$$

其中 d_i 表示 S 的第 i 个顶点的度（degree），$N(i)$ 为第 i 个顶点的 1-邻接顶点的下标集合，满足 $d_i = |N(i)|$。

但式 3-6在大多数三角网格上的表现均不尽人意，Meyer et al. 提出[26]，更好的方案是采用余切权重而非均匀权重：

$$\delta_i = \frac{1}{|\Omega_i|} \sum_{j\in N(i)} \frac{1}{2}(\cot\alpha_{ij} + \cot\beta_{ij})(\boldsymbol{q}_i - \boldsymbol{q}_j). \tag{3-7}$$

即所谓的余切公式（cotangent formula）。其中 $|\Omega_i|$ 为顶点 i 对应的 Voronoi 区域的面积，α_{ij}, β_{ij} 为边 ij 的两个对角，如图 3.3 所示。相对式 3-6，式 3-7 保证了其在平面上的取值为 0，具有更好的近似性，但在角度较大时，即 $\alpha_{ij} \to \pi$ 时，$\cot\alpha_{ij} \to \infty$，仍存在数值上的问题。随后 Floater 提出的平均值公式[27]（mean

value formula）解决了这一问题：

$$\delta_i = \sum_{j \in N(i)} \frac{\tan(\theta_{ij}^1/2) + \tan(\theta_{ij}^2/2)}{||\boldsymbol{q}_i - \boldsymbol{q}_j||}(\boldsymbol{q}_i - \boldsymbol{q}_j) \approx H_i \boldsymbol{N}_i. \tag{3-8}$$

其中 θ_{ij}^1 和 θ_{ij}^2 的定义如图 3.3 所示。式 3-8 相对式 3-7 和式 3-6 均表现出较强的稳健性，本毕业设计的算法实现即采用之计算局部离散 L-B 算子的取值。

实现上，本仿真器支持对三角形网格整体添加权重一定的弯曲约束。记 $N(i) = \{j_1, j_2, \ldots, j_{k-1}\}$，其中 $k = d_i + 1 = |V_c|$，且下标 $1, 2, \ldots, k-1$ 遵循以 $\boldsymbol{v}_i$ 为中心的逆时针排列。将列向量 $\boldsymbol{q} \in \mathbb{R}^{3n\times 1}$ 展开，得到 $\boldsymbol{q} = (\boldsymbol{q}_1, \ldots, \boldsymbol{q}_n)^T \in \mathbb{R}^{n\times 3}$，则约束的描述如下：

- $V_c = \{v_{j_0}, v_{j_1}, v_{j_2}, \ldots, v_{j_{k-1}}\}$，其中 $j_0 = i$。定义**下标集合** $I_c = \{j_0\} \cup N(i)$。
- $\boldsymbol{A}_c = \boldsymbol{\Delta}_S \boldsymbol{S}_c$，其中：
  - $\boldsymbol{S}_c$ 为 $k \times n$ 选择矩阵，满足 $\forall x \in [1, k], y \in [1, n]$，

$$\boldsymbol{S}_c(x, y) = \begin{cases} 1, & j_{x-1} \in I_c \text{ and } j_{x-1} = y, \\ 0, & \text{otherwise.} \end{cases}$$

    即使得 $\boldsymbol{S}_c\boldsymbol{q}$ 选出该约束覆盖的顶点位置。例如，若 $V_c = \{v_2, v_4, v_8, v_9\}$，即中心顶点为 v_2，其余顶点 v_4, v_8, v_9 逆时针环绕之，则 $\boldsymbol{S}_c$ 的第 1 行只有第 2 列有非零值 1，第 2 行只有第 4 列有非零值 1，以此类推。$\boldsymbol{S}_c(\boldsymbol{q}_1, \ldots, \boldsymbol{q}_n)^T = (\boldsymbol{q}_2, \boldsymbol{q}_4, \boldsymbol{q}_8, \boldsymbol{q}_9)^T$。
  - $\boldsymbol{\Delta}_S$ 为 $1 \times k$ 矩阵，用于计算 L-B 算子，其包含该约束的离散 L-B 算子在 v_j 处的计算权重 $w_j = -\frac{\tan(\theta_{ij}^1/2)+\tan(\theta_{ij}^2/2)}{||\boldsymbol{q}_i - \boldsymbol{q}_j||}$，满足 $\boldsymbol{\Delta}_S = \begin{bmatrix} w_0 & w_1 & \ldots & w_{k-1} \end{bmatrix}$。其中 w_0 根据式 3-8 定义为 $w_0 = -\sum_{j\in N(i)} w_j$。

  根据以上分析，$\boldsymbol{A}_c\boldsymbol{q}$ 计算了当前顶点位置 $\boldsymbol{q}$ 下，在约束 c 所涵盖的局部几何处的 L-B 算子的值。

- $\boldsymbol{A}'_c = \boldsymbol{R}\boldsymbol{\Delta}_S$，其中 $\boldsymbol{R} \in SO(3)$ 为任意三维旋转矩阵。如此定义的目的是，利用平均曲率在旋转变换下的不变性，使满足位置约束的顶点位置 $\boldsymbol{p}_c = (\boldsymbol{p}_i, \boldsymbol{p}_{j_1}, \ldots, \boldsymbol{p}_{j_{k-1}})^T$ 在此处的 $||\boldsymbol{A}'_c\boldsymbol{p}_c||$，即 L-B 算子得到的平均曲率不变。

注意到 L-B 算子的取值为平均曲率向量，记 $\boldsymbol{R}\boldsymbol{v}_m = \boldsymbol{R}\boldsymbol{N}_m H_m = \boldsymbol{A}'_c\boldsymbol{p}_c$，$\boldsymbol{v}_s = \boldsymbol{N}_s H_s = \boldsymbol{A}_c\boldsymbol{q}$，式 2-7 对应的原优化问题转化为

$$\min_{\boldsymbol{R}} ||\boldsymbol{v}_s - \boldsymbol{R}\boldsymbol{v}_m||^2, \tag{3-9}$$

约束条件为 $\boldsymbol{R} \in SO(3)$。

该优化问题的解实际上是显然的：对于已知的 $\boldsymbol{v}_s, \boldsymbol{v}_m$，只需让 $\boldsymbol{R}$ 将 $\boldsymbol{v}_m$ 转到与 $\boldsymbol{v}_s$ 同向，即可最小化它们之间的距离，此时

$$\boldsymbol{R}\boldsymbol{v}_m = \frac{\boldsymbol{v}_s}{||\boldsymbol{v}_s||}||\boldsymbol{v}_m||. \tag{3-10}$$

3.4.3 约束实现

弯曲约束的实现需要进行原始 L-B 算子的计算，并将之储存在约束中；PD 仿真过程中，再进行 local step 优化问题的求解。实现上，对于式 3-8 的计算，可同时判断顶点 v_i, v_{j_1}, v_{j_2}，即每个扇面三角形上的顶点位置 $\boldsymbol{q}_i, \boldsymbol{q}_{j_1}, \boldsymbol{q}_{j_2}$ 是否接近三点共线，若三点共线，则该曲面的三角化可能存在问题（如冗余结点等），这将导致 $\theta \to \pi$，$\tan(\theta/2) \to \infty$，引起数值问题，因此在控制台对用户给出警告。在计算 $\tan(\theta/2)$ 时，可采用公式

$$\tan(\theta/2) = \frac{1 - \cos\theta}{\sin\theta}, \tag{3-11}$$

并利用向量的点积和叉积计算 $\cos\theta$ 和 $\sin\theta$ 的值。

计算 L-B 算子的值时，尽管直接对各系数相乘相加能够少进行一定的减法

计算，即

$$\delta_i = \sum_{l=0}^{k-1} w_l \boldsymbol{q}_{j_l} \tag{3-12}$$

但实验显示，这样计算在顶点位置远离原点时，可能因浮点数误差导致结果上的偏移。为避免这一问题，算法实现仍采用了式 3-8，即

$$\delta_i = \sum_{l=1}^{k-1} -w_l(\boldsymbol{q}_i - \boldsymbol{q}_{j_l}) \tag{3-13}$$

进行计算。

在 local step 求解过程中，为避免除以一个接近于 0 的数，程序将在 $||\boldsymbol{v}_s||$ 很小时，不再计算式 3-10，而是计算当前局部几何的单位法向量 $\boldsymbol{N}_s$，并与原始平均曲率 H_m 相乘，得到 $\boldsymbol{A}_c'\boldsymbol{p}_c = \boldsymbol{N}_s H_m$。如此得到的结果与式 3-10 相同，因为

$$\boldsymbol{N}_s H_m = \frac{\boldsymbol{v}_s}{H_s} H_m = \frac{\boldsymbol{v}_s}{||\boldsymbol{v}_s||} ||\boldsymbol{v}_m||. \tag{3-14}$$

且这种方式可用于避免浮点数精度误差，但这种方法需要计算 $k-1$ 个三角形的面法向量并取平均，具有更高的计算量，因此一般情况下仍使用式 3-10 求解。

3.5 四面体张力约束

3.5.1 约束原理

弹性体中除了可归类为二维流形的布料、软管等，也有大量的三维流形，它们具有体积，因此纯粹使用表面三角形网格进行仿真具有很大的局限性。为仿真这类具体积弹性体，需要建立相应的体积元网格，并在相应的体积元上添加适当的约束。而最流行，也是最基础的体积元，是四面体体积元。

对于这些由四面体体积元的组成的四面体网格，它们在发生弹性形变后，需要有一个反向作用使该形变复原。描述这样的弹性形变及其复原，就是四面体张力约束需要达成的目标。

在超弹性模型的视角下，模型在每一时间步长均可看作微小形变，这使得

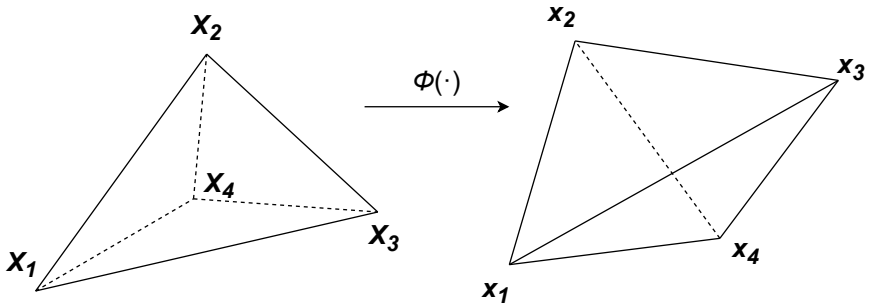


图 3.4 四面体的形变示意

每个四面体的形变均可用一个仿射变换表述[28]。记四面体的四顶点位置为 $\boldsymbol{X}_i \in \mathbb{R}^3, i \in \{1,2,3,4\}$，则变换的形变函数为

$$\phi(\boldsymbol{X}_i) = \boldsymbol{F}\boldsymbol{X}_i + \boldsymbol{b}, \tag{3-15}$$

其中 $\boldsymbol{F} \in \mathbb{R}^{3\times3}$，表示该变换的**形变梯度**（deformation gradient），$\boldsymbol{b} \in \mathbb{R}^3$，如图 3.4 所示。记 $\phi(\boldsymbol{X}_i) = \boldsymbol{x}_i$，由于 $\forall i \in \{1,2,3,4\}$，有

$$\boldsymbol{x}_i = \boldsymbol{F}\boldsymbol{X}_i + \boldsymbol{b}, \tag{3-16}$$

故

$$(\boldsymbol{x}_1 - \boldsymbol{x}_4, \boldsymbol{x}_2 - \boldsymbol{x}_4, \boldsymbol{x}_3 - \boldsymbol{x}_4) = \boldsymbol{F}(\boldsymbol{X}_1 - \boldsymbol{X}_4, \boldsymbol{X}_2 - \boldsymbol{X}_4, \boldsymbol{X}_3 - \boldsymbol{X}_4). \tag{3-17}$$

记上式右边为 $\boldsymbol{F}\boldsymbol{D}_m$，左边为 $\boldsymbol{D}_s$，则 $\boldsymbol{D}_m$ 和 $\boldsymbol{D}_s$ 分别表示形变前后的四面体整体形状。此时形变梯度

$$\boldsymbol{F} = \boldsymbol{D}_s\boldsymbol{D}_m^{-1}. \tag{3-18}$$

注意到，对于任一四面体的一次形变，形变梯度 $\boldsymbol{F}$ 完全描述了该形变导致的结果，因此，四面体张力约束应基于一定假设，找到 $\boldsymbol{F}$ 的“逆变换”，使该形变返回势能为零的状态。下面列出算法实现的四面体张力约束的描述：

- $V_c = \{v_i, v_j, v_k, v_l\}$，它们构成一个四面体元。

- $\boldsymbol{A}_c\boldsymbol{q} = \hat{D}_s(\boldsymbol{S}_c\boldsymbol{q})\boldsymbol{D}_m^{-1} = (\boldsymbol{q}_i - \boldsymbol{q}_l, \boldsymbol{q}_j - \boldsymbol{q}_l, \boldsymbol{q}_k - \boldsymbol{q}_l)\boldsymbol{D}_m^{-1}$。其中 $\hat{D}_s(\boldsymbol{S}_c\boldsymbol{q})$ 为一关于 $\boldsymbol{q}$ 的函数，作用是将 4 个顶点位置通过相应的选择矩阵 $\boldsymbol{S}_c$ 选出，并构造形如 $\boldsymbol{D}_s$ 的四面体整体形状。

- $\boldsymbol{A}_c'\boldsymbol{p}_c = \boldsymbol{T} = (\boldsymbol{p}_i - \boldsymbol{p}_l, \boldsymbol{p}_j - \boldsymbol{p}_l, \boldsymbol{p}_k - \boldsymbol{p}_l)\boldsymbol{D}_s^{-1} = \hat{D}_m(\boldsymbol{p}_c)\boldsymbol{D}_s^{-1} = (\hat{D}_s(\boldsymbol{p}_c)\boldsymbol{D}_m^{-1})^{-1}$。其中 $\hat{D}_m(\boldsymbol{p}_c)$ 为四面体形变前的合理形状——这里需要反向思考，注意 $\boldsymbol{q}$ 是形变后的位置，$\boldsymbol{p}_c$ 是形变前的位置。$\boldsymbol{T} \in \mathbb{R}^{3\times3}$ 的实际构造详见后文描述。

以上分析指出，形变后与形变前的能量之差 $||\boldsymbol{A}_c\boldsymbol{q} - \boldsymbol{A}_c'\boldsymbol{p}_c||^2 = ||\boldsymbol{F}_s^{-1} - \boldsymbol{T}||^2$ 具有有限元方法背景[29]。记 $\boldsymbol{A}_c\boldsymbol{q}$ 的奇异值分解（SVD）为

$$\boldsymbol{A}_c\boldsymbol{q} = \boldsymbol{F}_s = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T, \tag{3-19}$$

其中 $\boldsymbol{\Sigma} \in \mathbb{R}^{3\times3}$，对角元素记作 $\sigma_i, i \in \{1,2,3\}$。可见，该变换对四面体的体积的改变率的绝对值 $|\det(\boldsymbol{F}_s)| = \det(\boldsymbol{\Sigma}) = \sigma_1\sigma_2\sigma_3$。若假设 $\boldsymbol{\Sigma}$ 的对角元素均接近于 1，有

$$\boldsymbol{T} = (\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T)^{-1} = \boldsymbol{V}\boldsymbol{\Sigma}^{-1}\boldsymbol{U}^T \approx \boldsymbol{V}\boldsymbol{\Sigma}\boldsymbol{U}^T. \tag{3-20}$$

然而，在仿真过程中，由于时间步长的离散化、四面体的离散化等因素存在，这一假设不一定成立，此时 $\boldsymbol{\Sigma}$ 的对角元素可能不够接近 1，甚至 $\boldsymbol{F}_s$ 可能接近奇异。为使 PD 具有更高的稳健性，原始论文[5]采用了类似 SVD 求伪逆（广义逆）的方式。记

$$\boldsymbol{\Sigma}^* = \text{clamp}_{\sigma_{\min}\le\sigma_i\le\sigma_{\max}}(\boldsymbol{\Sigma}), \tag{3-21}$$

将 $\boldsymbol{\Sigma}$ 的对角元素限制在区间 $[\sigma_{\min}, \sigma_{\max}]$，其中 $\sigma_{\min} \le 1 \le \sigma_{\max}$，并令

$$\boldsymbol{T} = \boldsymbol{V}\boldsymbol{\Sigma}^*\boldsymbol{U}^T. \tag{3-22}$$

即可得到合理的保持体积的解。特别地，令 $\sigma_{\min} = \sigma_{\max} = 1$ 可最大程度保持体积，使物体难以被压缩或扩张。若三个奇异值 σ_i 对应的 $\sigma_{\min,i}$ 和 $\sigma_{\max,i}$ 均相同，

则弹性形变呈现各向同性。理论上，设置各方向的 $\sigma_{\min,i}$ 和 $\sigma_{\max,i}$ 不同，还可制造各向异性的弹性形变[30]。为使仿真合理，应当尽量使 $\sigma_{\min}$ 和 $\sigma_{\max}$ 取接近 1 的值。

此外，在仿真过程中，还需要防止四面体被翻转，这是因为奇异值分解可能导致 $\det(\boldsymbol{U}) < 0$ 或 $\det(\boldsymbol{V}) < 0$，从而 $\det(\boldsymbol{T}) < 0$，引起视觉上的错误。为此，程序在检测到 $\det(\boldsymbol{F}_s) < 0$ 时，将第三个奇异值 σ_3 在限制到区间 $[\sigma_{\min,3}, \sigma_{\max,3}]$ 后取相反数，即可保证 $\det(\boldsymbol{T}) > 0$ 且为一个接近 1 的值。

3.5.2 约束实现

四面体张力约束的实现与弯曲约束类似，首先需要进行 $\boldsymbol{D}_m^{-1}$ 的预计算；在 PD 仿真过程中，再进行 local step，利用式 3-22 求解 $\boldsymbol{T} = \boldsymbol{A}_c' \boldsymbol{p}_c$。在预计算时，还需求出原始四面体的体积 $V = \frac{1}{6}\det(\boldsymbol{D}_m)$，并让权重 ω_c 乘以该系数，以在网格四面体大小不一时获取更真实的结果。

在 CPU 上进行 local step 时，程序采用了 Eigen 库提供的 `Eigen::JacobiSVD` 进行奇异值分解；在 GPU 上的 SVD 则通过 Gao et al. 设计的高效 GPU SVD 算法[31]实现。通过 GPU 上的高效 SVD 和其他算法，四面体张力约束的局部并行求解十分迅速，没有成为本仿真器的性能瓶颈。对四面体张力约束的 local step 的实现源代码参见附录B。

在预计算 $\omega_c \boldsymbol{A}_c^T \boldsymbol{A}_c$ 时，记约束顶点集 $V_c = \{v_1, v_2, v_3, v_4\}$，可使用如下公式构造 $\boldsymbol{A}_c \in \mathbb{R}^{3\times n}$，即 $\forall j \in [1,3], i \in [1,n]$，

$$\boldsymbol{A}_c(j, v_i) = \begin{cases} -(\sum_{k=1}^{3} \boldsymbol{D}_m^{-1}(k,j)), & i = 4, \\ \boldsymbol{D}_m^{-1}(i,j), & i \in \{1,2,3\}, \\ 0, & \text{otherwise.} \end{cases} \tag{3-23}$$

其中 $\boldsymbol{D}_m^{-1}(i,j)$ 表示 $\boldsymbol{D}_m^{-1}$ 的第 i 行第 j 列（$i \in [1,3], j \in [1,3]$）的元素。可自行验证 $\boldsymbol{A}_c \boldsymbol{q} = (\boldsymbol{q}_i - \boldsymbol{q}_l, \boldsymbol{q}_j - \boldsymbol{q}_l, \boldsymbol{q}_k - \boldsymbol{q}_l)\boldsymbol{D}_m^{-1}$。构造完稀疏矩阵 $\boldsymbol{A}_c$ 后，即可计算

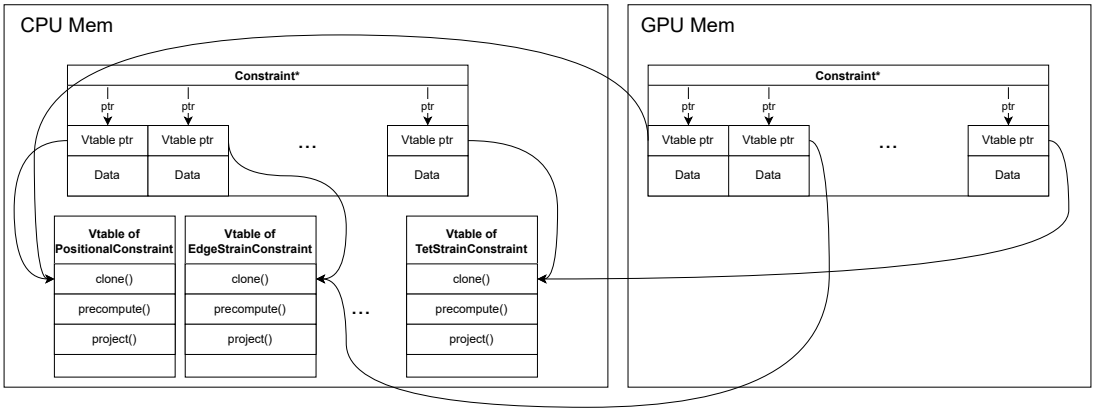


图 3.5 覆写虚函数表前的内存状况

$\omega_c \boldsymbol{A}_c^T \boldsymbol{A}_c$ 的值，并累加至全局线性系统的系数矩阵。在 local step 计算 $\omega_c \boldsymbol{A}_c^T \boldsymbol{A}_c' \boldsymbol{p}_c$ 时同理。

3.6 GPU 上的并行约束求解

上文已经介绍了各约束的计算方法，据此已经可以编写 CPU 程序进行串行约束求解，但编写 GPU 并行程序还需要开发者对并行框架和数据流的高度掌握。为了让各约束的局部投影过程在 GPU 上并行求解，程序需要将这些约束从它们创建时所在的 CPU 内存拷贝到 GPU 内存，并在 GPU 上实现多态。为此，在预计算阶段，若用户要求采用 GPU 计算 local step，则会进行一次内存拷贝，并在此时并行地覆写这些约束的虚函数指针，使它们正确地指向存在于 GPU 内存的虚函数表。在模拟阶段，对式 2-7 的计算即可在 GPU 上并行完成，并通过 `atomicAdd` 原子操作对线性方程组式 2-8 右边的常数项并行地累加（算法 2，第 8 行）。在用户重设模型约束时，这些原有的约束还需要被谨慎地从 GPU 内存中释放。

拷贝并覆写虚函数表的流程如图 3.5、图 3.6 所示。首先，程序需要将 CPU 上的多态 `Constraint` 指针数组深拷贝至 GPU 内存。但拷贝完成后，这些对象的虚函数表指针仍指向的是 CPU 上的虚函数表，在 GPU 上无法访问。因此，程序还需重定向这些 GPU 上的对象的虚函数表指针，即显式地将其首 8 字节（GPU 中指针的大小）的数据改写为 GPU 上的虚函数表地址。如此，便可在 GPU 上完成对虚函数表的动态绑定，正确地调用相应函数。

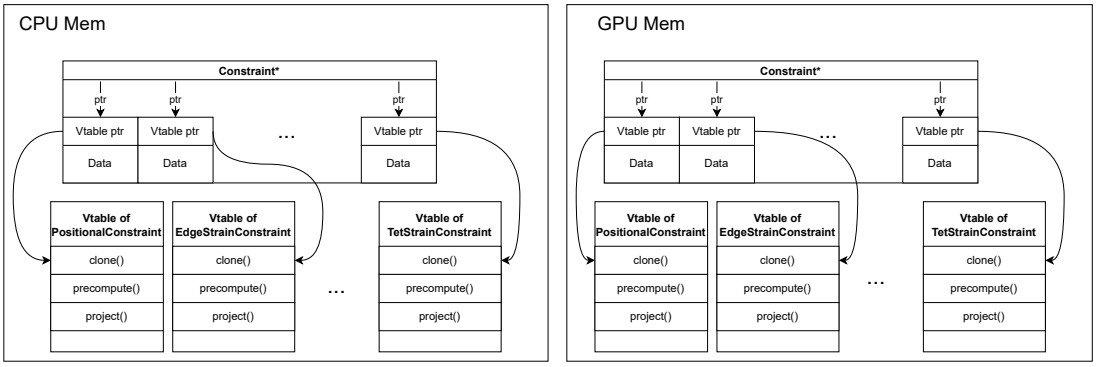


图 3.6 覆写虚函数表后的内存状况

当模型约束被重设时，程序在修改 CPU 上的 `Constraint` 指针数组后，原 GPU 上的 `Constraint` 将不再适用，需要先释放内存再重新拷贝内容。而在释放内存时，实验情况表明，尽管 CUDA GPU 对象的虚函数表正确，程序也无法正确地调用它们的虚析构函数。这一问题可能与 GPU 的堆内存回收机制有关，但限于 CUDA 运行时库的封闭性，进一步的分析十分困难。同时，CUDA 开发者论坛中的讨论尚未给出对此情况适用的解决方案，因此目前程序只能选择在 CPU 上释放内存。该过程描述如下：

1. 将 GPU 上的 `Constraint` 指针数组置入 CUDA 统一内存，即可在 CPU 上或 GPU 上同时访问这些对象。
2. 将这些对象的虚函数表覆写为 CPU 上的。
3. 在 CPU 上访问这些对象，CUDA 统一内存自动地将它们进行浅拷贝至 CPU，然后可以在 CPU 上正确地调用虚析构函数释放内存。

实验表明，在 GPU 上实现 local step 带来的性能提升是可观的，节 7 中将给出对比分析。

4 全局线性系统求解的实现

相对 local step，global step 的形式较为简单，仅为一个线性方程组。但实验表明，该线性方程组的求解很容易成为 PD 算法的瓶颈，因此给出合适的加速策

略十分必要。为实现高效求解，首先，需要对该线性系统进行数学建模，并给出相应的可行算法。

4.1 稀疏线性系统的建模

线性方程组式 2-8 是稀疏的，且对角元素非零，这一点可以从 $\boldsymbol{A}_c$ 和 $\boldsymbol{M}$ 的稀疏性和 $\boldsymbol{M}$ 的对角性直接得出。进一步可以发现，对于已实现的四种约束，位置约束、边张力约束和四面体张力约束的 $\boldsymbol{A}_c$ 都满足，当 v_i 与 v_j 之间没有边相连时，

$$(\boldsymbol{A}_c^T\boldsymbol{A}_c)_{3i,3j} = \boldsymbol{O}_3, \tag{4-1}$$

即 $\boldsymbol{A}_c^T\boldsymbol{A}_c$ 的第 $3i$ 行，第 $3j$ 列起对应的 3×3 块为零矩阵。进一步地，其否命题也成立。这一点是显然的——对于这三种约束，它们的约束顶点集中的顶点两两之间必然有边相连，从而使得 $\boldsymbol{A}_c^T\boldsymbol{A}_c$ 在相应的块上有值，这也可以通过计算各约束的 $\boldsymbol{A}_c^T\boldsymbol{A}_c$ 来证明。

而对于弯曲约束，尽管约束顶点集中的顶点两两之间未必有边相连，但注意到

$$\begin{aligned}\boldsymbol{A}_c^T\boldsymbol{A}_c &= (\boldsymbol{\Delta}_S\boldsymbol{S}_c)^T\boldsymbol{\Delta}_S\boldsymbol{S}_c \\ &= \boldsymbol{S}_c^T\boldsymbol{\Delta}_S^T\boldsymbol{\Delta}_S\boldsymbol{S}_c \\ &= \boldsymbol{S}_c^T\begin{bmatrix} w_0^2 & \dots & w_0w_{k-1} \\ \vdots & \ddots & \vdots \\ w_{k-1}w_0 & \dots & w_{k-1}^2 \end{bmatrix}\boldsymbol{S}_c,\end{aligned} \tag{4-2}$$

而 $\boldsymbol{S}_c$ 仅在决定 $\boldsymbol{A}_c$ 的某元素是否取 0 时起遮罩作用。由于中心顶点 v_i 与其他顶点 v_j 的连接边所贡献的 L-B 算子权重之积 w_0w_j 占主要成分，贡献了 $\boldsymbol{A}_c^T\boldsymbol{A}_c$ 的一次项，其余如 v_{j_1}, v_{j_2}，它们的权重之积 $w_{j_1}w_{j_2}$ 对 $\boldsymbol{A}_c^T\boldsymbol{A}_c$ 仅贡献二次项。因此，仍可近似认为弯曲约束也满足式 4-1。实验证明，这一近似是合理的，并且对于 A-Jacobi 算法是必要的。

基于以上的分析，存储该稀疏矩阵的空间复杂度已可以降低至 $O(V+\sum_i d_i) =$

$O(V+2E)=O(V+E)$。求解该稀疏线性系统的方法较多，而本毕业设计通过让具体求解器继承一个抽象父类 `LinearSystemSolver`，并实现其纯虚函数 `set_A()`（预计算）、`solve()`（求解）和 `clear()`（释放内存）的方法实现各求解器。

接下来给出求解该稀疏矩阵的 Cholesky 分解法和 A-Jacobi 法。

4.2 Cholesky 分解

线性代数中的 Cholesky 分解是 LU 分解在系数矩阵 $\boldsymbol{A}$ 为实对称矩阵时的特例。在经历 $O(n^3)$ 的分解过程（预计算）后，这两种分解法均将求解 m 次 $\boldsymbol{A}$ 不变，$\boldsymbol{b}$ 变化的 n 维线性系统的时间复杂度降低至 $O(mn^2)$，而非高斯消元法的 $O(mn^3)$，且相较求解 $\boldsymbol{A}^{-1}$ 的方法更稳健。实际应用中，Cholesky 分解的效率约为 LU 分解的两倍[32]。

在计算 $\boldsymbol{A}$ 的 Cholesky 分解时，首先求出 $\boldsymbol{A}=\boldsymbol{L}\boldsymbol{L}^T$ 对应的矩阵 $\boldsymbol{L}$。随后在每次求解 $\boldsymbol{A}\boldsymbol{x}_i=\boldsymbol{b}_i$ 时，先求解 $\boldsymbol{L}\boldsymbol{y}_i=\boldsymbol{b}_i$，再求解 $\boldsymbol{L}^T\boldsymbol{x}_i=\boldsymbol{y}_i$ 即可得到 $\boldsymbol{x}_i$。

原始 PD 论文中即采用 Cholesky 分解进行 global step，其效率稳定，且不需要像求解线性方程组的迭代法那样调整迭代次数或其他参数，因此在本仿真器中，默认情况下仍使用 Cholesky 分解求解 global step。

4.3 Jacobi 迭代求解与 A-Jacobi 算法

尽管如 Cholesky 分解这样的直接法的表现十分稳定，但也很容易成为整个求解器的性能瓶颈。为进一步提高效率，学术界也探索了对 global step 的并行性改造[15][7][33][8]，并提出了若干并行方案。在 PD 框架中，巨大但稀疏的系数矩阵 $\boldsymbol{A}$ 给出了这样的暗示——迭代法常常有比直接法更大的优化空间。由于直接求解线性方程组缺少可并行性，接下来将探讨求解线性方程组的迭代法，期望利用 GPU 高度并行的计算单元，在性能上取得优势。

对于维度为 n 的线性方程组 $\boldsymbol{A}\boldsymbol{x}=\boldsymbol{b}$，可用对角矩阵 $\boldsymbol{D}$ 和非对角矩阵 $\boldsymbol{B}$ 分

解 $\boldsymbol{A}$，得到 $\boldsymbol{A}=\boldsymbol{D}-\boldsymbol{B}$，并考虑如下的迭代形式：

$$\boldsymbol{x}^{(k+1)}=\boldsymbol{D}^{-1}(\boldsymbol{B}\boldsymbol{x}^{(k)}+\boldsymbol{b}). \tag{4-3}$$

易见，当该迭代过程收敛时，由于 $\boldsymbol{D}\boldsymbol{x}=\boldsymbol{B}\boldsymbol{x}+\boldsymbol{b}$，有 $\boldsymbol{x}^{(k+1)}=\boldsymbol{x}^{(k)}=\boldsymbol{x}=\boldsymbol{A}^{-1}\boldsymbol{b}$。若进一步分解 $\boldsymbol{D}^{-1}\boldsymbol{b}$，得到 $\boldsymbol{D}^{-1}\boldsymbol{A}\boldsymbol{x}=\boldsymbol{x}-\boldsymbol{D}^{-1}\boldsymbol{B}\boldsymbol{x}$，有

$$\boldsymbol{e}^{(k+1)}=\boldsymbol{x}^{(k+1)}-\boldsymbol{x}=\boldsymbol{D}^{-1}\boldsymbol{B}(\boldsymbol{x}^{(k)}-\boldsymbol{x})=\boldsymbol{D}^{-1}\boldsymbol{B}\boldsymbol{e}^{(k)}, \tag{4-4}$$

其中 $\boldsymbol{e}^{(k)}$ 为第 k 次迭代的误差向量。记 $\boldsymbol{D}^{-1}\boldsymbol{B}$ 的特征分解为 $\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{-1}$，其中 $\boldsymbol{\Lambda}$ 为特征值矩阵，有

$$\boldsymbol{e}^{(k)}=(\boldsymbol{D}^{-1}\boldsymbol{B})^{k}\boldsymbol{e}^{(0)}=\boldsymbol{Q}\boldsymbol{\Lambda}^{k}\boldsymbol{Q}^{-1}\boldsymbol{e}^{(0)}. \tag{4-5}$$

这意味着，若迭代法收敛，则其为线性收敛，且收敛率与 $\boldsymbol{D}^{-1}\boldsymbol{B}$ 的最大特征值有关，记为 $\boldsymbol{D}^{-1}\boldsymbol{B}$ 的**谱半径** $\rho(\boldsymbol{D}^{-1}\boldsymbol{B})$。进一步地，迭代法收敛当且仅当 $\rho(\boldsymbol{D}^{-1}\boldsymbol{B})<1$。

令 $\boldsymbol{B}=\boldsymbol{U}+\boldsymbol{L}$，其中 $\boldsymbol{U}$、$\boldsymbol{L}$ 分别为 $-\boldsymbol{A}$ 的严格上、下三角部分，就能得到 Jacobi 求解器的一般形式。

4.3.1 Jacobi 求解器

式 4-3 可整理如下：

$$\boldsymbol{x}^{(k+1)}=\boldsymbol{D}^{-1}(\boldsymbol{U}+\boldsymbol{L})\boldsymbol{x}^{(k)}+\boldsymbol{D}^{-1}\boldsymbol{b}. \tag{4-6}$$

这与矩阵分解形式

$$x_i^{(k+1)}=\frac{1}{d_i}\left(b_i-\sum_{j=1}^{i-1}l_{ij}x_j^{(k)}-\sum_{j=i+1}^{n}u_{ij}x_j^{(k)}\right) \tag{4-7}$$

等价，其中 d_i 为对角矩阵 $\boldsymbol{D}$ 的第 i 个元素，l_{ij} 和 u_{ij} 分别为 $\boldsymbol{L}$、$\boldsymbol{U}$ 的第 i 行，第 j 列的元素。注意到对任意 i，等式的右边一开始就是已知的，这意味着对迭代向量的每一元素，计算是可并行的。这使得 Jacobi 求解器特别适合在并行计算架构，如 GPU 上实现。然而，要使 Jacobi 法达到较小的求解误差，通常需要很大的迭代次数，因此实际的计算时间往往也较长。尽管 Jacobi 的单次迭代是迅速的，但在系统较复杂时，所需收敛的迭代次数也是较多的，这一缺陷已被学术界所关注，并提出了诸如 Chebyshev[6]、parallel descent[7]和 A-Jacobi[8]等优化方法，加速其收敛进程。

本仿真器实现了朴素的 Jacobi CPU/GPU 求解器，完全采用式 4-7 计算，并且直接使用二维数组存储系数矩阵 $\boldsymbol{A}$，不仅空间开销很大，且其效率在绝大多数场合都不如 Cholesky 分解；在 GPU 上，调用 CUDA 核函数（kernel function）和拷贝数据的开销，甚至可能超过计算的开销[8]，使得朴素 Jacobi 法的并行化对效率的提升幅度大大降低。可见一个谨慎的算法实现对算法效率的提高可能至关重要。

4.3.2 A-Jacobi GPU 算法

A-Jacobi 算法是 2022 年提出的一个高效的基于 GPU 的 Jacobi 迭代法[8]，通过在一次 GPU 内核并行计算时计算 l 次而非简单的 1 次 Jacobi 迭代，以提高计算的效率。记 $\boldsymbol{R}=\boldsymbol{D}^{-1}(\boldsymbol{U}+\boldsymbol{L})=\boldsymbol{D}^{-1}\boldsymbol{B}$，A-Jacobi 利用下式进行多次迭代：

$$
\begin{aligned}
\boldsymbol{x}^{(k)} &= \boldsymbol{R}\boldsymbol{x}^{(k-1)}+\boldsymbol{D}^{-1}\boldsymbol{b} \\
&= \boldsymbol{R}^2 x^{(k-2)}+\boldsymbol{R}\boldsymbol{D}^{-1}\boldsymbol{b}+\boldsymbol{D}^{-1}\boldsymbol{b} \\
&= (\boldsymbol{R}^0+\boldsymbol{R})\boldsymbol{D}^{-1}\boldsymbol{b}+\boldsymbol{R}^2\boldsymbol{x}^{(k-2)} \\
&= \dots \\
&= \sum_{j=0}^{l-1}\boldsymbol{R}^j\boldsymbol{D}^{-1}\boldsymbol{b}+\boldsymbol{R}^l\boldsymbol{x}^{(k-l)}
\end{aligned}
\tag{4-8}
$$

记 l 为 A-Jacobi 的阶数（order），可见，传统 Jacobi 法的 $l = 1$。上式中，$\boldsymbol{R}$ 完全由系数矩阵 $\boldsymbol{A}$ 决定，而 $\boldsymbol{A}$ 又是常数，故 $\boldsymbol{R}$ 可在 PD 框架中预计算。项 $\boldsymbol{\beta} = \sum_{j=0}^{l-1} \boldsymbol{R}^j \boldsymbol{D}^{-1}\boldsymbol{b}$ 对于固定的 $\boldsymbol{b}$ 进行的 Jacobi 迭代求解是常数，因此可在每次进行 global step 的迭代之前预计算；项 $\boldsymbol{R}^l\boldsymbol{x}^{(k-l)}$ 为迭代项，具有如下性质：

$$[\boldsymbol{R}\boldsymbol{x}^{(k-1)}]_i = D_{ii}^{-1}B_{ij}x_j^{(k-1)}, \tag{4-9}$$

$$[\boldsymbol{R}^2\boldsymbol{x}^{(k-2)}]_i = \overbrace{D_{ii}^{-1}D_{ss}^{-1}\underbrace{B_{is}B_{sj}}_{\text{A-product}}}^{\text{A-coefficient}} x_j^{(k-2)}. \tag{4-10}$$

式中采用爱因斯坦求和约定。可见：

- 当 $l = 1$ 时，注意到 $\boldsymbol{B}$ 为 $-\boldsymbol{A}$ 的非对角部分，故 B_{ij}（近似）有值当且仅当顶点 v_i、v_j 之间有边相连。考虑 $B_{ij}x_j^{(k-1)}$ 相当于固定顶点 v_i，遍历 v_i 的 1-邻接顶点集 $N(i)$ 得到顶点 $v_j, j \in N(i)$，此时求和 $D_{ii}^{-1}B_{ij}x_j^{(k-1)}$ 的几何意义为边权 B_{ij} 与顶点位置 $x_j^{(k-1)}$ 的相乘相加，然后经过 D_{ii}^{-1} 的放缩。
- 当 $l = 2$ 时，类似地，固定顶点 v_i，遍历其 2-邻接顶点集，求和 $D_{ii}^{-1}D_{ss}^{-1}B_{is}B_{sj}x_j^{(k-2)}$ 的几何意义为从 v_i 经过 v_s 到 v_j 的边权之积 $B_{is}B_{sj}$ 与顶点位置 $x_j^{(k-2)}$ 的相乘相加，然后经过 $D_{ii}^{-1}D_{ss}^{-1}$ 的放缩。

可以类推该规律对较大的 l 也成立，即 $[\boldsymbol{R}^l\boldsymbol{x}^{(k-l)}]_i$ 相当于遍历 v_i 的 l-邻接顶点集，并进行相乘相加和放缩。

本仿真器在实现上采用如下的数据结构保存 l-邻接顶点集及其对应权重的信息：

代码 4.1: 计算 $\boldsymbol{R}^l\boldsymbol{x}$ 的数据结构

```
Scalar** l_ring_neighbors[A_JACOBI_MAX_ORDER];
VertexIndex** l_ring_neighbor_indices[A_JACOBI_MAX_ORDER];
int* l_ring_neighbor_sizes[A_JACOBI_MAX_ORDER];
Scalar* diagonals;
```

其中 `A_JACOBI_MAX_ORDER` 表示一个编译时常量，为系统提供的最大 A-Jacobi 阶数。`l_ring_neighbors[l][i][j]` 保存的是在计算 $\boldsymbol{R}^l\boldsymbol{x}$ 的第 i 个分量时，所需使

用的第 j 个非零 A-product（定义见式 4-10）与路径中的放缩项 $D_{ss}^{-1}\ldots$（不包含 D_{ii}^{-1}）的乘积，而 `l_ring_neighbor_indices[l][i][j]` 为第 j 个非零 A-product 对应的目的顶点的下标，`l_ring_neighbor_sizes[l][i]` 为顶点 v_i 的 l-邻接顶点集的大小。`diagonals[i]` 则保存 D_{ii}^{-1} 的值。

以上结构均可在 CPU 上预计算，并拷贝至 GPU 中进行 global step 的求解。

4.3.3 Chebyshev 加速

Checyshev 加速策略可为 Jacobi 迭代法提供更高的收敛速率[6]，它也适用于 A-Jacobi[8]，详细的推导过程可见原始论文。在 Chebyshev 加速中，存在三个超参数 S, γ, ρ，其中 S 常设为 10 可满足绝大多数应用需求，γ 为亚松弛（under relaxation）因子，ρ 为估计的谱半径 $\rho(\boldsymbol{R}^l)$。在第 $k+1$ 次迭代求解线性方程组的末尾，$\boldsymbol{x}^{(k+1)}$ 的更新遵循如下公式：

$$\boldsymbol{x}^{(k+1)} = \omega_{k+1}(\gamma(\boldsymbol{x}^{(k+1)} - \boldsymbol{x}^{(k)}) + \boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k-1)}) + \boldsymbol{x}^{(k-1)}. \tag{4-11}$$

其中 ω_{k+1} 在每次迭代下更新：

$$\omega_{k+1} = \begin{cases} 1, & k < S, \\ 2/(2-\rho^2), & k = S, \\ 4/(4-\rho^2\omega_k), & k > S. \end{cases} \tag{4-12}$$

实验证明，对于部分场景，Chebyshev 加速的提升是明显的，但仍存在部分场景的 Chebyshev 加速导致了求解过程发散，但这些场景下即使不采用 (A-)Jacobi 方法，求解过程也存在发散的问题。注意到原始 Jacobi 法对应的 Chebyshev 参数是 $\omega = 1$，$\gamma = 1$。

根据以上分析，A-Jacobi 求解过程可由算法 3 描述，其中 N_l 表示代码 4.1 对应的 A-Jacobi 阶数为 l 时的数据结构。算法 3 的第 7 行和第 8 行完全在 GPU 上并行计算，而其他行则在 CPU 上运行。值得注意的是，算法的具体实现在求解

算法 3: 本仿真器实现的 A-Jacobi 求解器

1 $\boldsymbol{x}_{\text{prev}} = \boldsymbol{x} = \boldsymbol{x}_{\text{next}} = \mathbf{0}$;
2 $\boldsymbol{\beta} = \texttt{precompute\_beta}(\boldsymbol{b}, N_0, N_1, \ldots, N_{l-1})$;
3 $k = 0$;
4 $\omega = 1$;
5 **while** $k <$ max #SolverIteration **do**
6 $\quad \omega = \texttt{update\_omega}(k, S, \rho, \omega)$;
7 $\quad \boldsymbol{x}_{\text{next}} = \texttt{a\_jacobi}(\boldsymbol{x}_{\text{prev}}, \boldsymbol{x}, \boldsymbol{\beta}, N_l)$;
8 $\quad \boldsymbol{x}_{\text{next}} = \omega(\gamma(\boldsymbol{x}_{\text{next}} - \boldsymbol{x}) + \boldsymbol{x} - \boldsymbol{x}_{\text{prev}}) + \boldsymbol{x}_{\text{prev}}$;
9 $\quad \boldsymbol{x}, \boldsymbol{x}_{\text{prev}} = \boldsymbol{x}_{\text{next}}, \boldsymbol{x}$;
10 $\quad k = k + 1$;
11 **end**
12 $\texttt{check\_convergence}(\boldsymbol{x}_{\text{next}}, \boldsymbol{x})$;
13 **return** $\boldsymbol{x}_{\text{next}}$;

$\boldsymbol{x}_{\text{next}}$ 后，进行了 triple buffer 更名，即在下次迭代时仅向 `a_jacobi()` 函数（算法 3，第 7 行）传入次序不同的 $\boldsymbol{x}$ 参数，而不会做额外的内存拷贝（算法 3，第 9 行），以提高算法的效率。有关 A-Jacobi 的性能对比和发散问题，节 7 将给出更多说明。

5 离散碰撞检测的实现

进行多个物体的物理模拟时，模型难免产生交互，此时碰撞检测的稳健性和高效性对于模拟复杂场景而言十分重要。至今，对复杂几何体之间的高效碰撞检测和处理算法，仍是学术界讨论的热点之一。限于规模，本仿真器对碰撞检测的实现比较简单，仅实现了弹性体与简单凸几何体（平面、球、立方体和环面）的碰撞检测和处理，无法处理更复杂的碰撞和模型的自相交。但对于一个个人开发的仿真器而言，这样的碰撞检测系统已经能产生较丰富的仿真结果。下面将介绍仿真器对简单凸几何体的生成与具体碰撞相关算法的实现。

5.1 简单凸几何体的生成

与之前对约束和线性系统求解器的实现相同,程序采用继承抽象父类 `Primitive` 的方式实现各具体几何体 `Floor`、`Sphere`、`Block` 和 `Torus`。程序化生成几何体

的命名空间 meshgen 则实现了若干用于几何体生成的静态函数。而各具体几何体类通过调用 meshgen 中的各函数，实现了它们的虚函数 generate_model()，如 generate_plane()、generate_sphere() 和 generate_torus() 等，用于可视化模型的生成。这一设计采用了访问者模式（visitor pattern），将数据与代码进行了封装与解耦。

5.2 碰撞检测与处理算法

在提出 PD 算法框架的原始论文[5]中，使用了运行时添加碰撞约束的方式来处理可能的碰撞，如此实现将会大幅降低求解器的效率，因为求解过程在 GPU 上计算，动态改变 GPU 上的约束存储将需要许多代价。本程序处理碰撞的方式与原始论文中基于约束的方法不同，是一种原创的，适用于 PD 求解框架的离散碰撞检测方法，性能更佳。

参见算法 2 的第 2 行和第 12 行，算法在仿真前后各进行了一次碰撞检测与处理，它们的意义有细微的不同。第一次的 ResolveCollision() 调用紧接着 $\boldsymbol{s}_t$ 的求解，注意到 $\boldsymbol{s}_t$ 的物理意义为不考虑内力、碰撞等其他因素，各顶点在当前状态下继续移动一时间步长后的位置。此时程序检测顶点可能与碰撞体产生的相交，并将这些相交顶点推出碰撞体，更新 $\boldsymbol{s}_t$ 的值，这样 $\boldsymbol{s}_t$ 将变得更加合理，在后续的 L-G 迭代中呈现出更好的收敛数值过程。第二次的 ResolveCollision() 调用则在 L-G 迭代结束后，解决仍可能产生的相交情况，确保结果的正确性。ResolveCollision() 的内部实现为遍历各顶点和各碰撞体，逐一询问它们是否有相交，若有则将顶点位置投影至最近的碰撞体表面无相交处。这一实现调用了各碰撞体对顶点的碰撞检测和处理的函数 collision_handle()。

以上介绍的简易碰撞检测和处理系统已可以仿真较丰富的交互情况。最后，需要强调的是，碰撞检测与仿真算法仍在很大程度上具有“正交性”，这是因为大多数的碰撞检测算法都位于物理仿真流程的“后处理”部分，即在仿真算法完全结束后，将可能相交的顶点移至安全位置，以确保碰撞处理的正确性。可见，这一流程与实际仿真流程的求解的相关性不高。因此，程序的碰撞检测实现虽

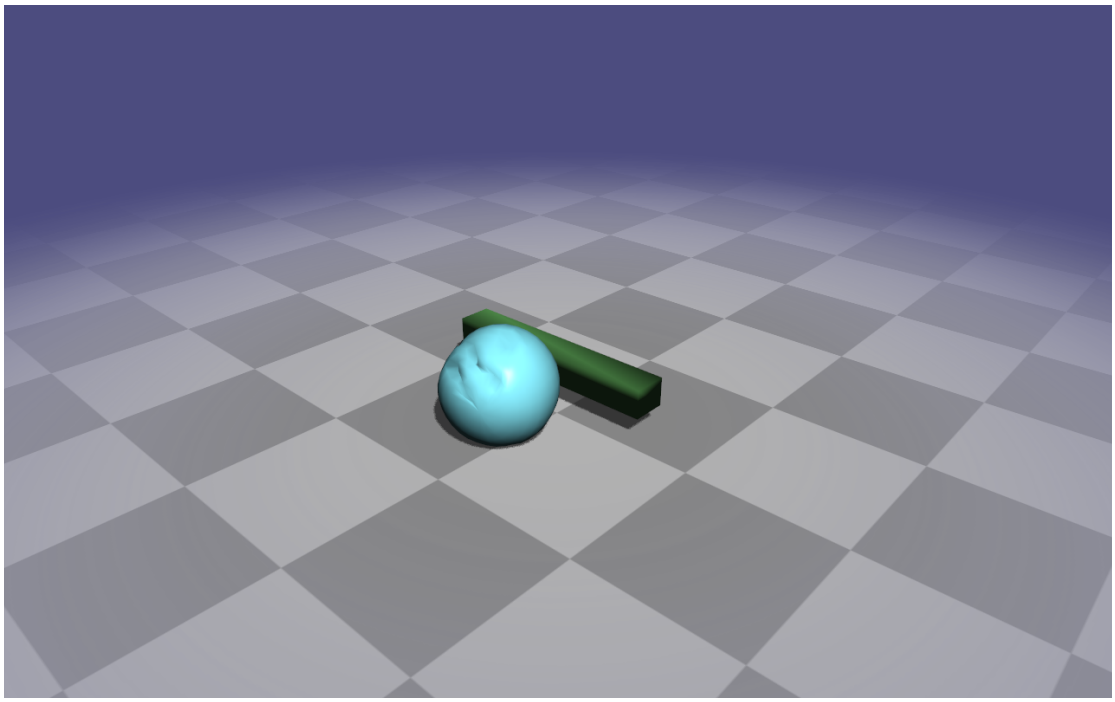


图 5.1 平面上的弹性球与横梁相撞的仿真

仍具相当大的改进空间，但已不属于仿真算法的核心内容。本毕业设计的后续将预期通过引入空间散列算法、连续碰撞检测等改进该碰撞检测方法，以打造更稳健的仿真器。

除碰撞类型的支持外，本仿真器仍存在一个重要的改进空间，即碰撞接触摩擦的实现。目前，仿真器的所有碰撞都是光滑的，这意味着碰撞不会损失动能，物体在平面上的滑行不会休止，如球体的滚动等物理现象基本上无法实现。这一点在图 5.1 描绘的弹性球撞击刚性横梁的场景中尤为明显，该仿真场景中，球的移动呈现出滑动为主而非滚动，且碰撞后球的动能不会损失。对实时摩擦碰撞的支持在工业界仿真器中的表现也不甚理想，因此一直是学术界探索的前沿话题，近年来 Ly et al. 的工作[34]在这一方面有不错的成果，实现了高效的布料之间的摩擦碰撞求解，或可成为本仿真器进一步的改进方向。

6 模型管理与 GUI 的实现

本毕业设计的模型管理与 GUI 为用户操作仿真器提供了便利，其实现高度依赖开源库 Dear ImGui 和 libigl 提供的 API。由于编写此类代码与仿真算法的关系不大，这里仅作简要说明。

本仿真器将模型分为两种：弹性体与碰撞体，其中弹性体可建立约束用于仿真，碰撞体不可直接参与仿真，仅在算法中参与碰撞处理。模型的增、删、改功能通过 `ObjManager` 类和若干渲染选项的切换实现。

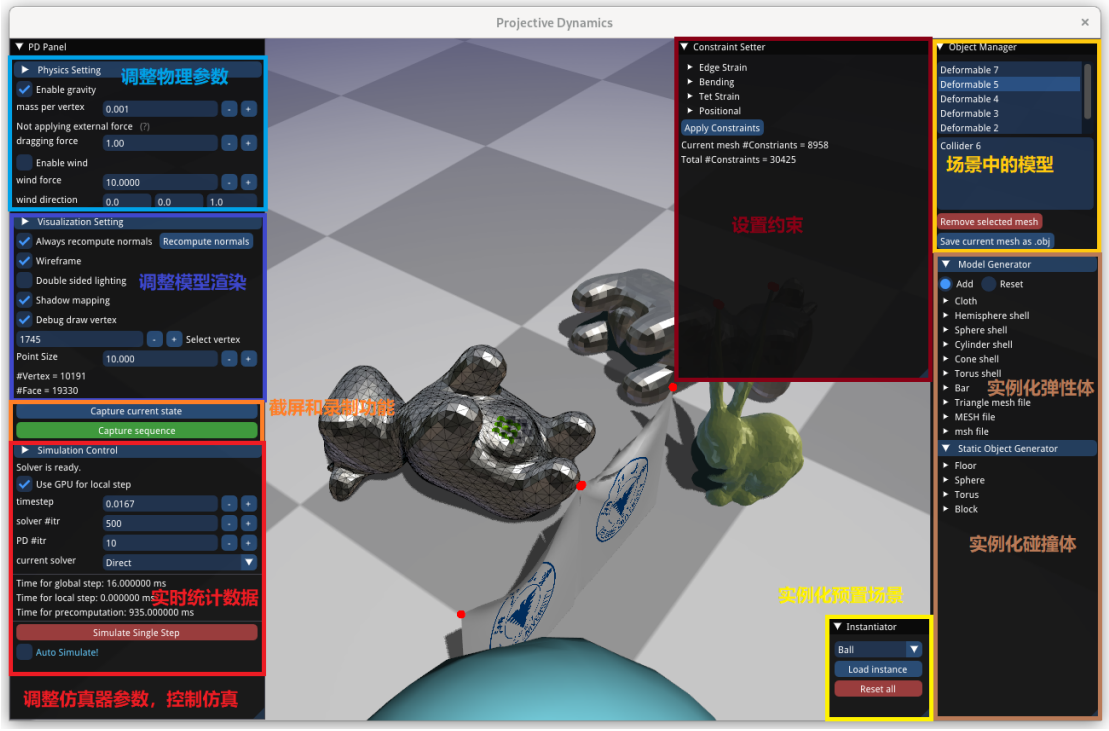
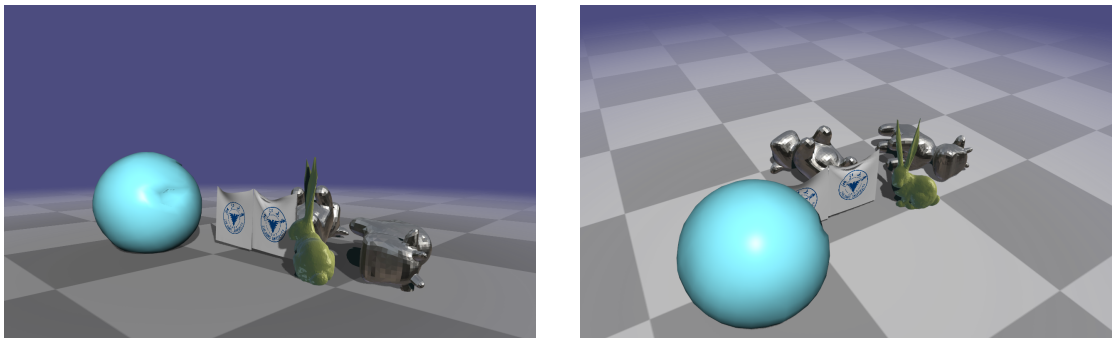


图 6.1 仿真器 GUI 展示及说明



(a) 视角 1　　(b) 视角 2

图 6.2 沙盘场景的仿真

仿真器的 GUI 实现如图 6.1 所示。这是一个用户自创建的沙盘场景，GUI 的各部分功能在图中已进行说明。当用户点击视口中的模型时，本仿真器会自动进行射线追踪，选中距离用户点击位置最近的模型，方便进一步的处理。图 6.2 展示了更多对该仿真场景的观察视角。该场景共包括 6 个弹性体模型，1 个碰撞体模型，总约束数目为 8958，使用 Cholesky 直接法和 GPU 上的约束求解，仿真的 FPS（frames per second，每秒帧数）达到了 200 以上。

7 实验与对比

本节将展示使用本仿真器进行的若干仿真实例，所有统计数据在计时中均包含渲染（有时使用轻便的 MatCap、shadow mapping 技术），所有仿真实例默认均添加了恒定向下的重力。实验环境统一为：

- 操作系统：Arch Linux x86_64
- CPU: 12th Gen Intel(R) Core(TM) i7-12700
- GPU: NVIDIA Corporation GA104 [GeForce RTX 3060 Ti Lite Hash Rate]
- 编译器：gcc (g++) 12.2.1、nvcc 11.8

表 7.1 不同仿真场景的统计数据汇总表。注：一个模型的总元素（element）数指的是组成其的三角形或四面体元的数目。Armadillo 的第一个总约束数为其被悬挂时，加入位置约束时的总约束数；第二个总约束数为其取消位置约束时的总约束数。

仿真场景	总顶点数	总元素数	总约束数
2 个 20 × 20 布料（图 7.1）	882	1600	2845
风场中的 2 个 20 × 20 布料（图 7.2）	882	1600	2845
140 × 140 布料（图 7.14）	19881	39200	59082
5 个半球壳（图 7.4）	1700	3200	6400
2 个圆柱形软管（图 7.3）	680	1280	2340
自由下垂的软棒（图 7.5）	208	540	556
软性桥梁（图 7.7）	189	400	418
弹性球（图 5.1）	530	1539	1539
沙盘场景（图 6.2）	10191	29214	30425
Bunny（图 7.9）	2531	8121	8125
Dragon（图 7.11）	41262	167882	167882
Armadillo（图 7.12）	15666	49393	51362、49393

表 7.2 不同仿真场景的统计数据汇总表（续）

仿真场景	预计算时间 (ms)	最小 FPS	最大 FPS
2 个 20 × 20 布料（图 7.1）	39	200	>200
风场中的 2 个 20 × 20 布料（图 7.2）	40	200	>200
140 × 140 布料（图 7.14）	595	52	59
5 个半球壳（图 7.4）	63	91	100
2 个圆柱形软管（图 7.3）	32	>200	>200
自由下垂的软棒（图 7.5）	11	>200	>200
软性桥梁（图 7.7）	12	>200	>200
弹性球（图 5.1）	37	>200	>200
沙盘场景（图 6.2）	935	62	66
Bunny（图 7.9）	215	83	90
Dragon（图 7.11）	56091	8.1	10.2
Armadillo（图 7.12）	6704	45	47

各仿真场景的统计数据汇总如表 7.1、表 7.2 所示。除本节外，本毕业设计的中期报告中也展示了部分实验结果可供参考。

7.1 布料仿真

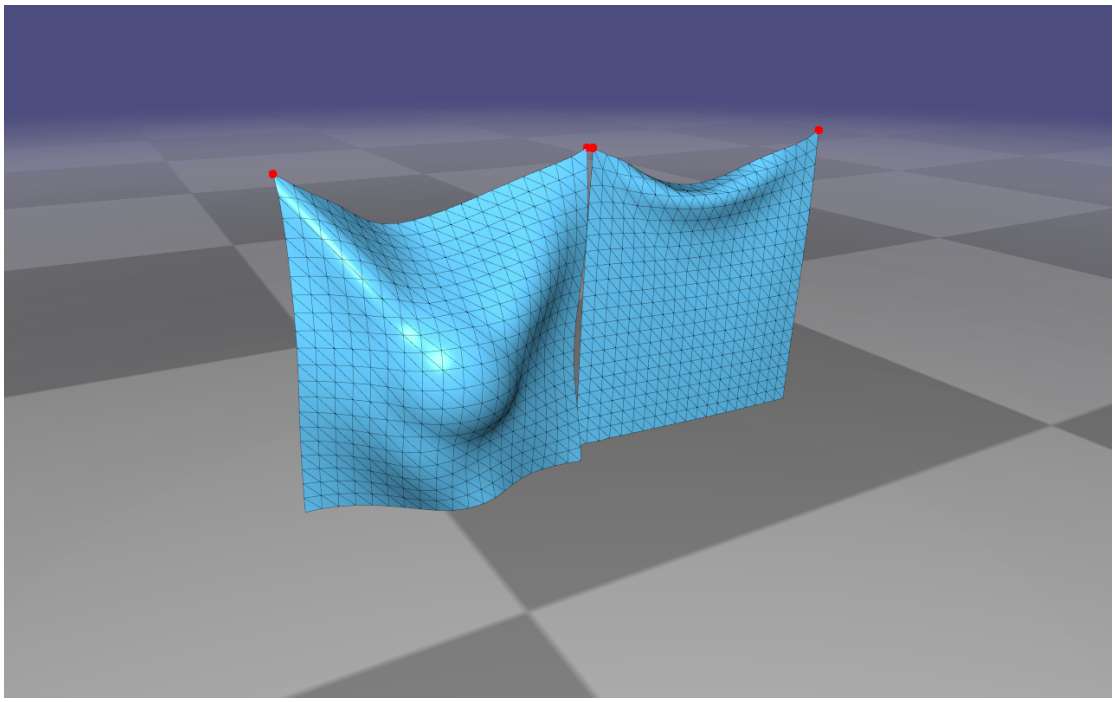


图 7.1 基础布料仿真

布料是基础的二维弹性体，本仿真器能够稳健地仿真延展性、刚性不同的各类布料。图 7.1 展示了仿真器对两种具有相同几何体，而约束不同的布料的仿真截图，其中左侧布料建立的弯曲约束权重更大，右侧布料建立的权重更小，两侧布料的边张力约束和位置约束权重大小一致。仿真过程中，左侧布料有一球体以一定速度沿布料法线方向移动。可见，仿真器正确地展示了布料与球碰撞的结果。在实验仿真进行时，动态改变球体的位置对仿真 FPS 的影响可忽略不计。

在风场中，程序对布料的仿真过程仍保持高度的稳健性。图 7.2 展示了当用户动态控制风场的大小、方向时，布料的仿真情况。在实验中，即使风力的大小呈现急剧增大，PD 算法对布料的仿真仍保持着稳定的收敛过程，布料的视觉表现也呈现出正确性。风场的改变对仿真 FPS 的影响也可忽略不计。

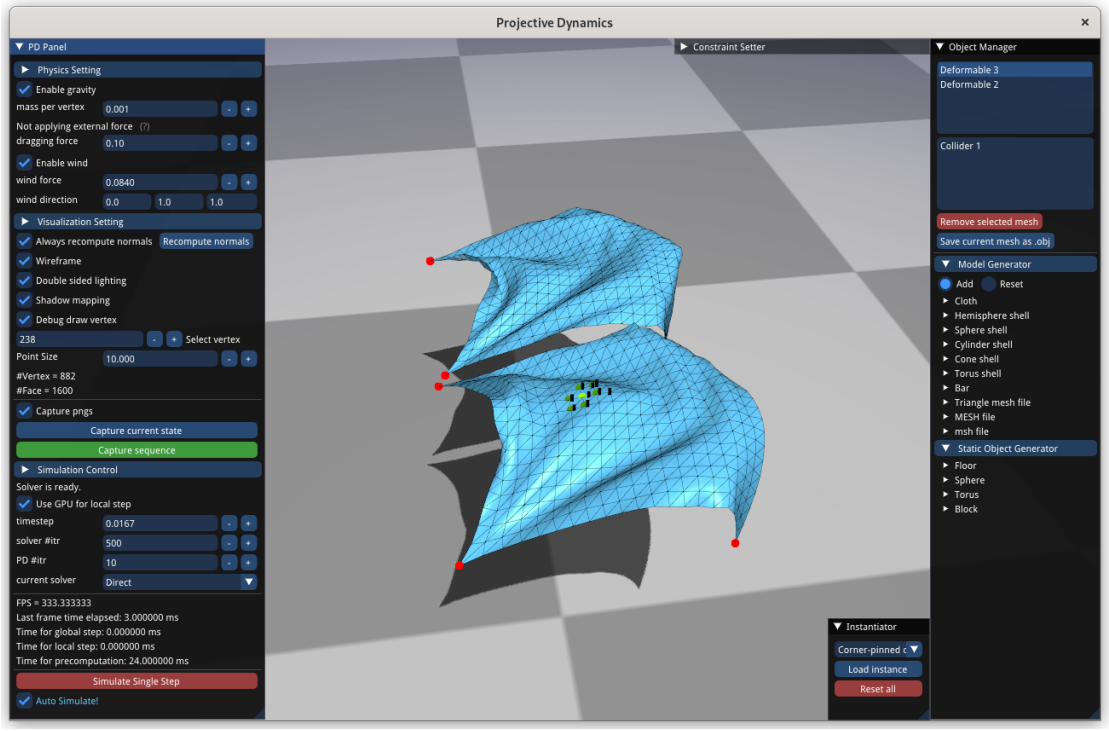
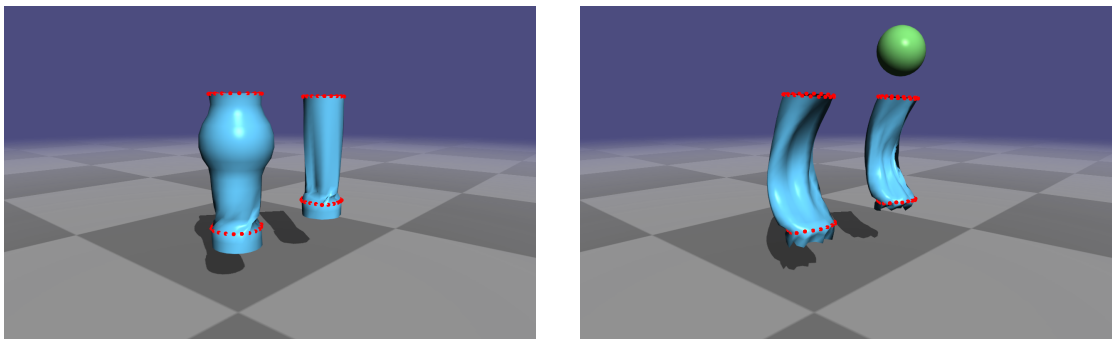
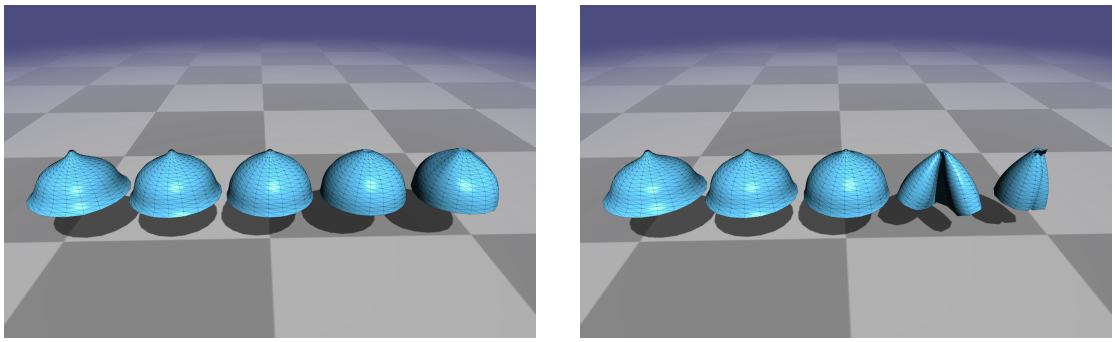


图 7.2 风场中的稳健布料仿真



(a) 球穿越时软管的场景　　(b) 施加风场时的场景

图 7.3 软管的仿真



(a) 静止时的场景 **(b) 用户施加外力后的场景**

图 7.4 不同刚度半球壳的仿真

7.2 软壳仿真

“软壳”仍属于二维流形曲面，仿真过程与布料并无二致。如图 7.3 所示，利用三种局部约束，本仿真器高效且稳健地仿真了圆柱形软管。图中的左侧软管相对右侧具有更大的弯曲约束权重，因此保持了其初始时表面上的弯曲程度，并在风场中产生了自然的表面形变；而右侧软管在风场中的形变则产生了自相交。可见，仿真过程正确地处理了软管与球体的交互和它们风场中的表现。

如图 7.4 所示，对初始的半球形曲面设置不同大小的弯曲约束，各模型能够对相同外力呈现不同的反应。图中从左到右，模型的弯曲约束大小依次减小。可见，对于最左侧的模型，由于高度的弯曲约束，使得模型收敛到了与原始半球形曲面不同的一个几何模式，视觉上和物理上的表现均更“坚固”；而对于最右侧的模型，由于重力的存在和较小的弯曲约束，初始状态下就有坍塌的趋势（若不设置弯曲约束，将直接坍塌）。当外力作用时，仿真过程无法收敛至局部能量极小值点，导致模型迅速坍塌，呈现图 7.4b 的状态。由于程序的碰撞相关算法无法处理自碰撞，故模型出现了自相交的情况。

7.3 具体积弹性体仿真

对各类复杂具体积弹性体的仿真，是对算法的正确性、高效性和稳健性的考验。图 7.5 是固定一端的均匀弹性软棒，即材料力学和有限元分析领域的“悬臂梁”，其仿真结果具有可供参考的解析解。因此程序可通过仿真，确认四面体网

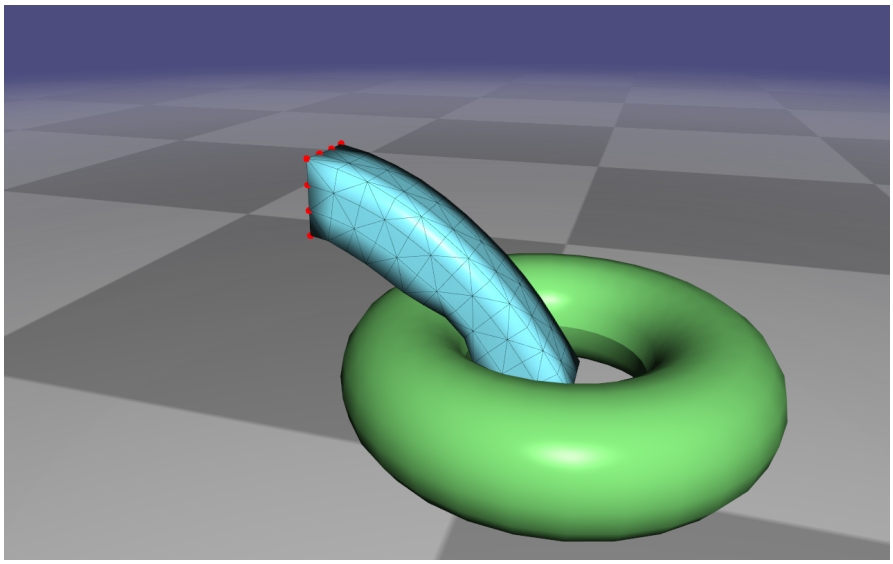
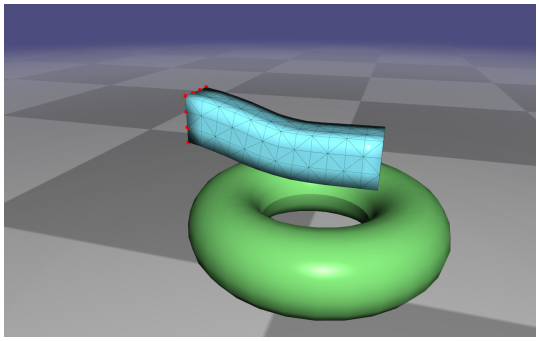
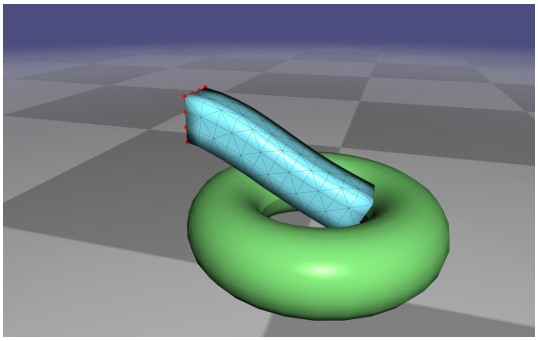


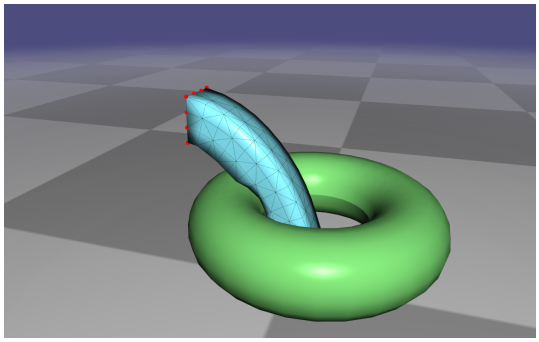
图 7.5 自由下垂的软棒（悬臂梁）的仿真



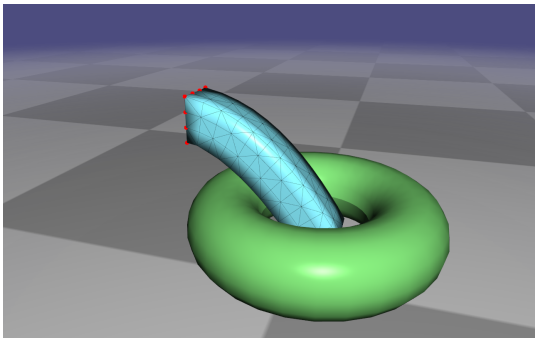
(a) 第 30 帧



(b) 第 60 帧



(c) 第 90 帧



(d) 第 120 帧

图 7.6 对软棒（悬臂梁）仿真的逐 30 帧演示

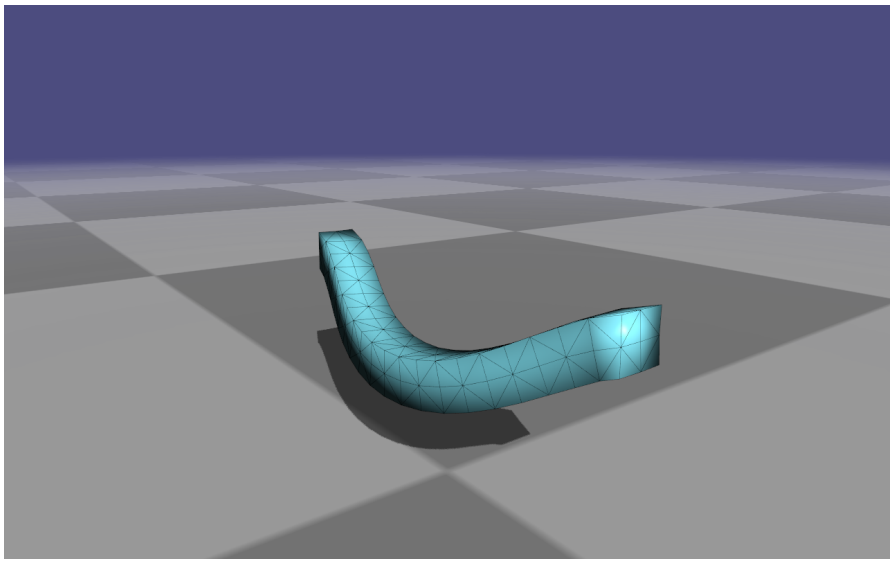


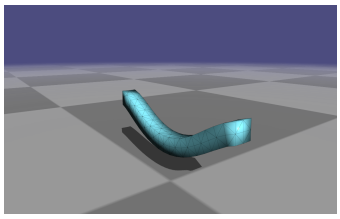
图 7.7 软性桥梁的仿真

格建立和计算过程的正确性。软棒模型的四面体网格采用程序实现的简单几何体生成算法完成。如图 7.6 所示，该场景中软棒在重力下的自然下垂行为，及其与环面的碰撞得到了正确的仿真结果。同时，对仿真中间数据和收敛结果的数值过程对比，也与解析解近似相符。

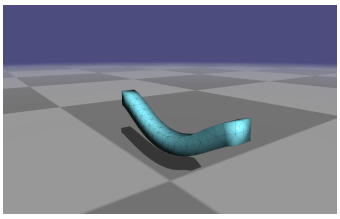
同理，用户也可以固定悬臂梁的两端，仿真软性桥梁，如图 7.7 所示。通过调整四面体张力约束权重的大小，能够得到下垂程度不同的桥体。在实验中，通过对该桥梁模型施加不同大小、方向的外力，桥梁随之产生了真实的摇晃行为。如图 7.8 所示，用户在某次仿真序列中的第 40 帧、第 70 帧和第 150 帧（以及其他帧）附近对桥梁施加了如图 7.8d、图 7.8g、图 7.8o 所示的外力，经过仿真后的表现符合预期。

图 7.9 展示了对固定了耳朵上端的 Stanford Bunny 模型的弹性体仿真。在实验中，用户可通过动态改变地面的高度，产生 Bunny 弹跳的实时动画，实际效果饶有趣味。

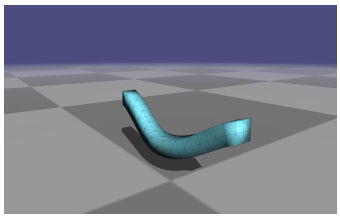
对于高分辨率模型 Dragon，程序采用了 TetWild[20]生成了原始 obj 模型的四面体网格，导出为 msh 格式，然后在仿真器中导入该格式的模型。图 7.10 展示了在仿真器中对其四面体网格的可视化。图 7.11 展示了 Dragon 模型的仿真情况，整个场景的总约束数目为 167882，仿真的平均 FPS 在 8.5 左右。



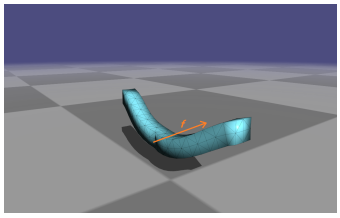
(a) 第 10 帧



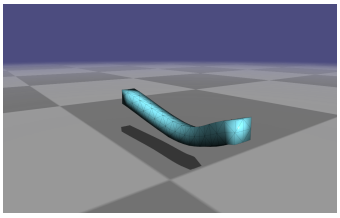
(b) 第 20 帧



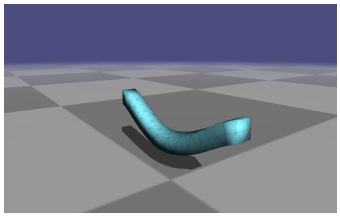
(c) 第 30 帧



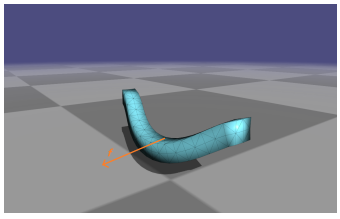
(d) 第 40 帧



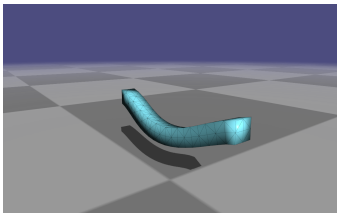
(e) 第 50 帧



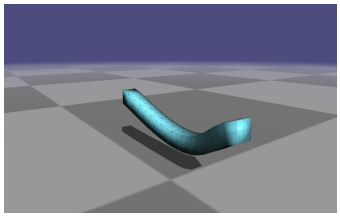
(f) 第 60 帧



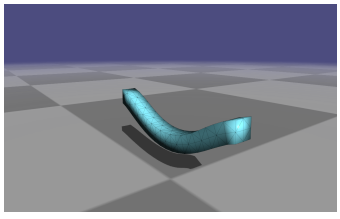
(g) 第 70 帧



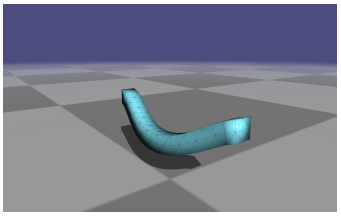
(h) 第 80 帧



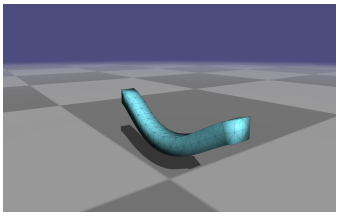
(i) 第 90 帧



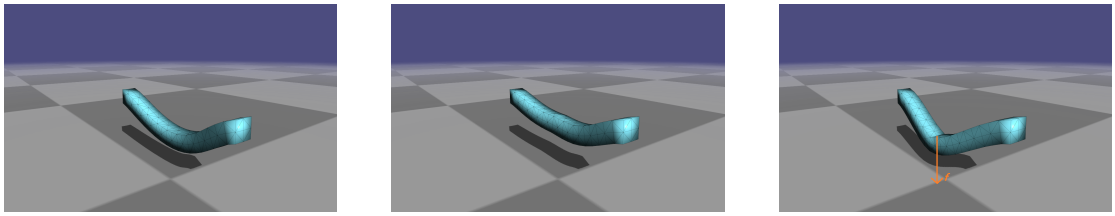
(j) 第 100 帧



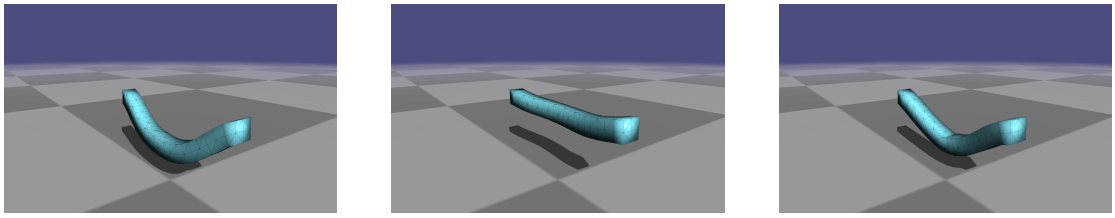
(k) 第 110 帧



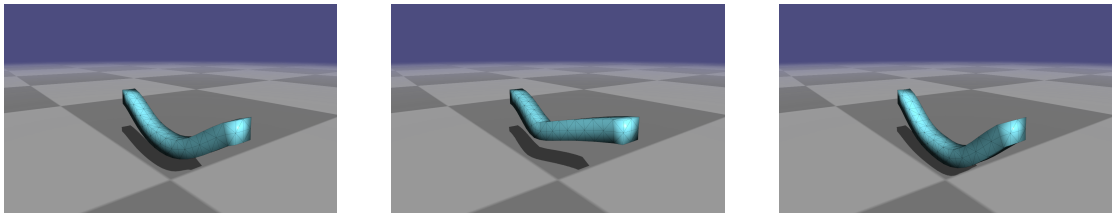
(l) 第 120 帧



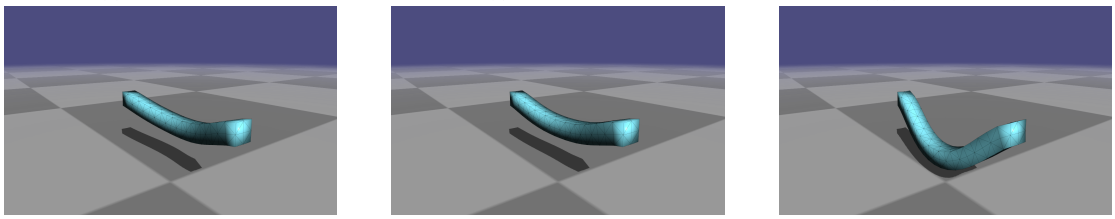
(m) 第 130 帧　(n) 第 140 帧　(o) 第 150 帧



(p) 第 160 帧　(q) 第 170 帧　(r) 第 180 帧



(s) 第 190 帧　(t) 第 200 帧　(u) 第 210 帧



(v) 第 220 帧　(w) 第 230 帧　(x) 第 240 帧

图 7.8 对桥梁仿真的逐 10 帧演示。限于论文形式，此处只能展示逐帧图片。作者将在毕业设计答辩现场的幻灯片中展示本节所有实验对应的动图。

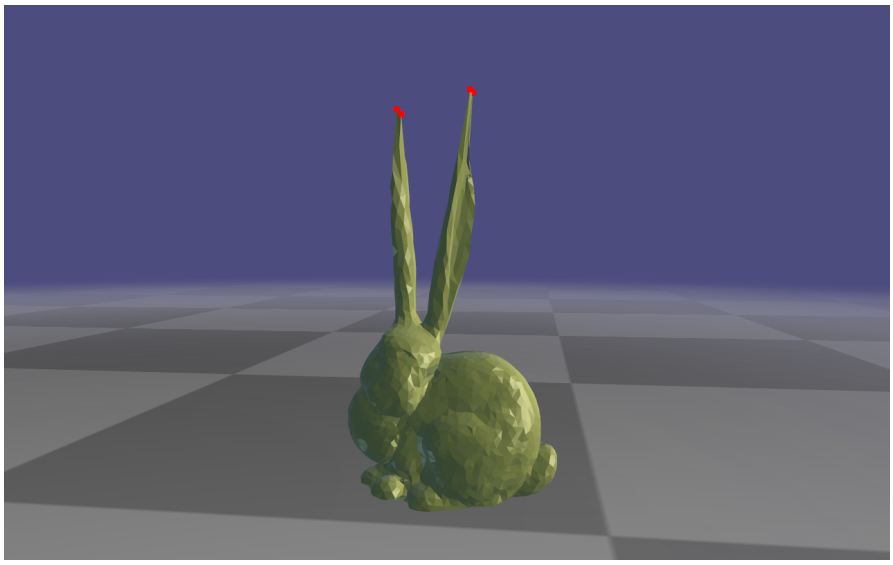
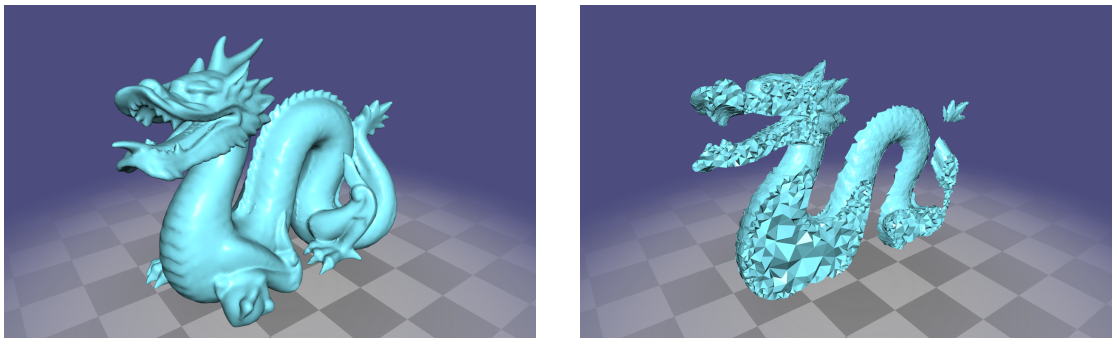


图 7.9 固定耳朵的 Stanford Bunny 的仿真



(a) 模型表面渲染 **(b) 模型内部的若干四面体渲染**

图 7.10 Dragon 模型的四面体网格可视化

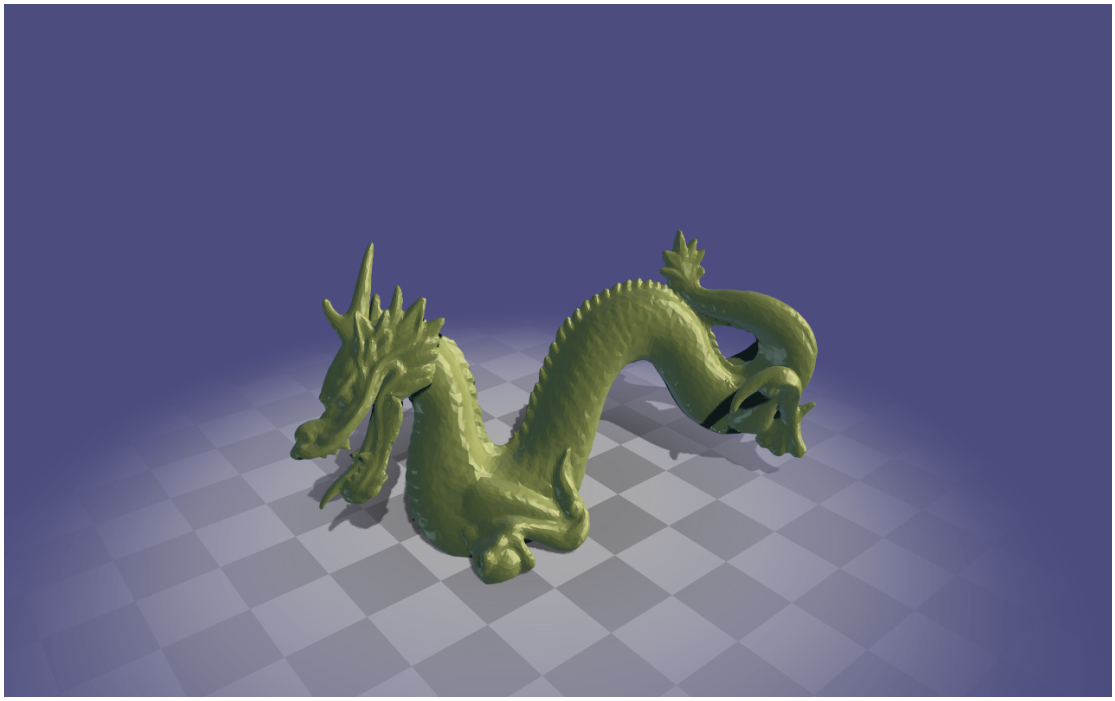
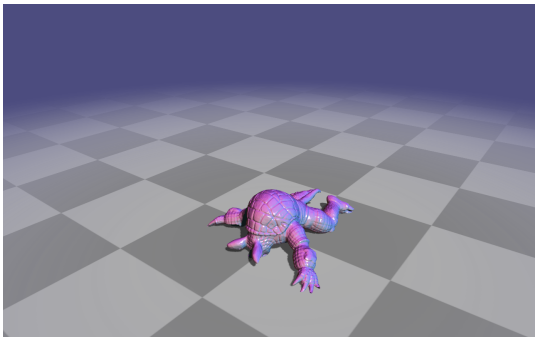


图 7.11 高分辨率模型 Dragon 的仿真

(a) 悬挂时的场景



(b) 落下后的场景

图 7.12 Armadillo 模型的仿真

本仿真器对于 Armadillo 弹性模型的仿真如图 7.12 所示。该模型的四面体网格采用 TetGen[19]生成。在悬挂 Armadillo 模型时，程序通过硬编码的方式，在其背部的若干个顶点处建立位置约束；实验中，Armadillo 在悬挂时表现出手臂随重力的真实感摇晃。用户随后通过取消位置约束的方式释放 Armadillo 模型，使之自然落下并在地面上呈现真实的轻微反弹。整个场景的总约束数目为 49393，仿真的平均 FPS 在 45 左右。

7.4 不同算法的性能对比

表 7.3 CPU 串行求解 local step 所需时间表

约束数目	local step 计算时间 (ms)
8K	20
16K	51
40K	120
78K	297

通过对比局部约束求解的 CPU 和 GPU 算法，可发现两者的性能存在十分显著的差距。使用四面体张力约束模型进行测试，约束数目与 CPU local step 所需时间的关系如表 7.3 所示。可见，当约束数目较大时，串行求解方法很容易成为性能瓶颈。而经过测试，在约束数目超过 200K 时，GPU 并行求解 local step 所需时间仍为一个十分接近于 0 的数，效率惊人。

表 7.4 不同 global step 算法对场景的仿真 FPS 数据表

仿真场景	Cholesky	朴素 Jacobi	1 阶 A-Jacobi
8 个 20 × 20 布料	90	2	100
100 × 100 布料	75	内存超限	112
圆柱形软管（图 7.3）	>200	12	200
弹性球（图 5.1）	>200	发散	38
Bunny（图 7.9）	83	发散	发散

表 7.5 不同 global step 算法对场景的仿真 FPS 数据表（续）

仿真场景	2 阶 A-Jacobi	3 阶 A-Jacobi
8 个 20 × 20 布料	90	71
100 × 100 布料	89	77
圆柱形软管（图 7.3）	200	111
弹性球（图 5.1）	27	14
Bunny（图 7.9）	发散	发散

而对 global step 求解算法的对比，可以发现，A-Jacobi 算法的性能并未表现出彩。在部分场景中，比较 Cholesky 直接法、朴素 Jacobi 和不同阶数的 A-Jacobi 的算法性能如表 7.4、表 7.5 所示。可见，在布料仿真上，A-Jacobi 相对 Cholesky 有更高的效率，这是因为当模型的边张力约束占约束的主要成分时，此时系数矩阵呈现显著的对角占优，迭代谱半径可估计得较小，故 Chebyshev 加速的收敛速率可观。在具有较高的弯曲约束成分的软壳仿真中，A-Jacobi 不能乐观地向下估计迭代矩阵 $\boldsymbol{R}^l$ 的谱半径，故加速能力有所下降。而当仿真四面体网格时，若模型的四面体元很小，基于 Jacobi 的迭代法常常呈现发散表现。研究[6]指出，Chebyshev 加速此时必须采用一个更小的亚松弛因子 γ，但这也会导致较慢的收敛速率。如对弹性球的仿真，用户可将 γ 设为一个较小的数 0.7，以避免发散，但也使得加速效率并不可观。而对 Bunny 的仿真中，实验显示，无论如何调整参数，也无法避免迭代的发散行为，只能采用 Cholesky 法作为 global step 进行仿真。

同时，令人疑惑的是，仿真器实现的高阶 A-Jacobi 算法的表现不如低阶，这与原始论文[8]的图表展现的结果不一致。在经过初步分析对比后，可能的原因指向了 CUDA 调用核函数时，分配的 grid、thread 的数目与物理设备（GPU）匹配程度不高，导致程序的实际并行计算效率受到影响。另外，原始论文对 A-Jacobi 的算法实现比较模糊，省略了许多细节，可能也将导致本仿真器的实现在效率相对不可观。本毕业设计后续将进一步尝试优化 A-Jacobi 的表现，但 Cholesky 分解的稳定和不需调参的性质，使其在这些算法对比中具有独特优势，也因此成为本仿真器的默认算法。而由于在布料仿真中，总是呈现对角占优的系数矩阵，收敛表现可得到保证，故此时采用 A-Jacobi 更具优势。

7.5 与 PhysX (NvCloth) 和 Bullet 的对比

通过将本仿真器与工业界流行的 PhysX (NvCloth) 和 Bullet 进行对比，可更好地揭示其特点、优势与不足之处，并为用户群体提供参考。对比实验的环境与上述实验相同，PhysX 版本为 5.1.3，NvCloth 版本为 1.1.6，Bullet 版本为 3.25，均为 2022 至 2023 年发布的稳定版本。

7.5.1 对比指标

鉴于各仿真器在算法设计上的差异性，且本仿真器使用的算法框架并非 PBD，直接将本仿真器与其他同类产品进行比较在某种意义上是不妥的。但对于使用本实时仿真器的用户，如游戏开发者、建模设计师等而言，他们并不会关注算法的调优，而是重点关注算法的两个方面：效率与视觉表现。因此，本文集中在这两个方面阐述对比的结果，供用户和进一步开发参考。

7.5.2 布料仿真对比

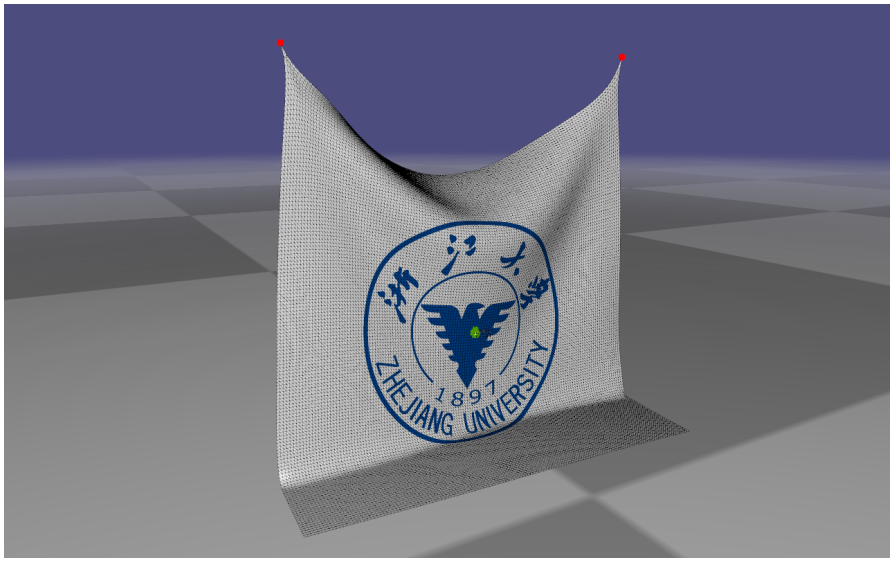


图 7.13 本仿真器的高分辨率布料仿真（显示网格）

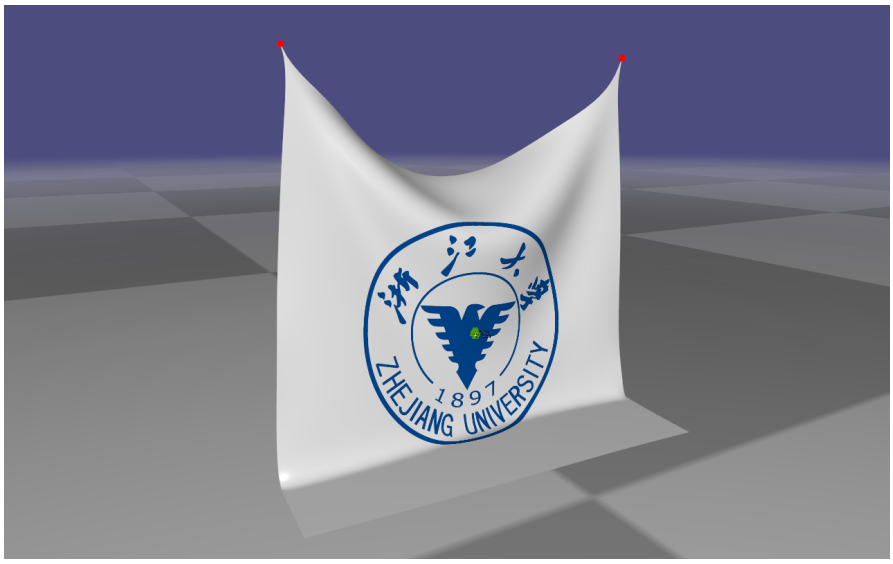


图 7.14 本仿真器的高分辨率布料仿真

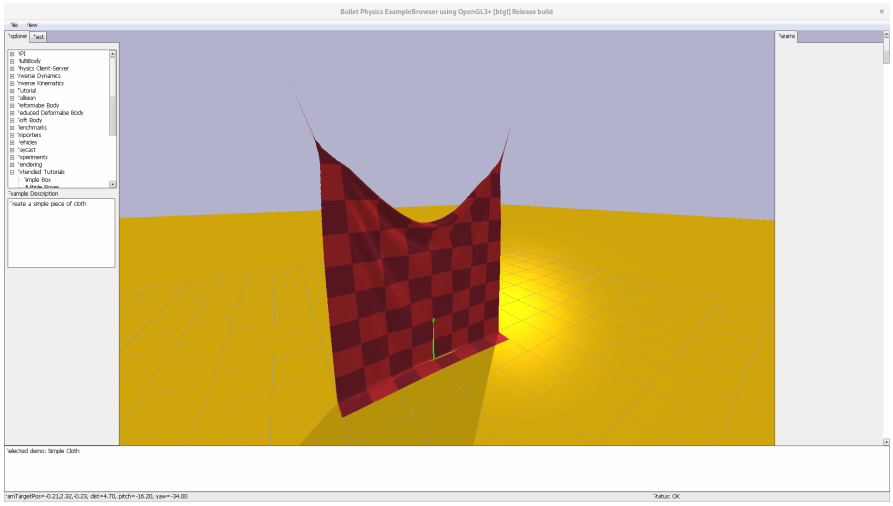


图 7.15 Bullet 的高分辨率布料仿真，基于 Simple Cloth example

表 7.6 本仿真器与 PhysX (NvCloth) 和 Bullet 的仿真平均 FPS 数据表

仿真场景	本仿真器	PhysX (NvCloth)	Bullet
140 × 140 布料	54 (A-Jacobi)	48	41
2 个 Armadillo	21 (Cholesky)	44	17

为对比布料仿真的效果和效率，实验过程在本仿真器、NvCloth 和 Bullet 上分别创建了长、宽均为 140，即 140×140 的布料，并进行了相同悬挂方式的仿真，如图 7.14、图 7.15 所示。由于均基于质点-弹簧模型，三者的视觉和物理正确性

图 7.16 本仿真器的 Armadillo 仿真

均毋庸置疑，NvCloth 的仿真结果因官方不提供渲染器，且其坐标输出与其他结果的相似性不再列出。仿真 FPS 的对比如表 7.6 所示。在采用了高效的 A-Jacobi 算法后，本仿真器在效率上略微领先。

7.5.3 Armadillo 弹性体仿真对比

实验同时对比了具体积物体的仿真——在本仿真器、PhysX 和 Bullet 上均加载了相同的 Armadillo obj 模型，让 PhysX 用内部实现的算法生成四面体网格，而用 TetGen[19]为本仿真器和 Bullet 生成四面体网格。在此前提下，本仿真器和 Bullet 使用了相同的四面体网格配置，而与 PhysX 不相同。如图 7.16、图 7.17、图 7.18 所示，三种仿真器呈现出不太一致的结果。在进行性能比较时，采用的是在场景中同时仿真 2 个相同 Armadillo 的实验，结果如表 7.6 所示。

实验结果显示，PhysX 的仿真在调节了各项参数后，其结果仍表现出较强的刚性，仅在地面上反弹时表现出一定柔软度，如图 7.17 所示。在调整了线性刚度（linear stiffness）和质量后，Bullet 的仿真结果呈现仍十分不理想，各四面体出现了严重的自相交和随机滑行的状态，如图 7.18 所示，这可能与四面体模型仿真功能在 Bullet 属于新功能，成熟度不高有关。进一步的实验过程中，通过调研 PhysX 文档，并审阅 PhysX 运行时结果，可以发现 PhysX 仿真弹性体采用了

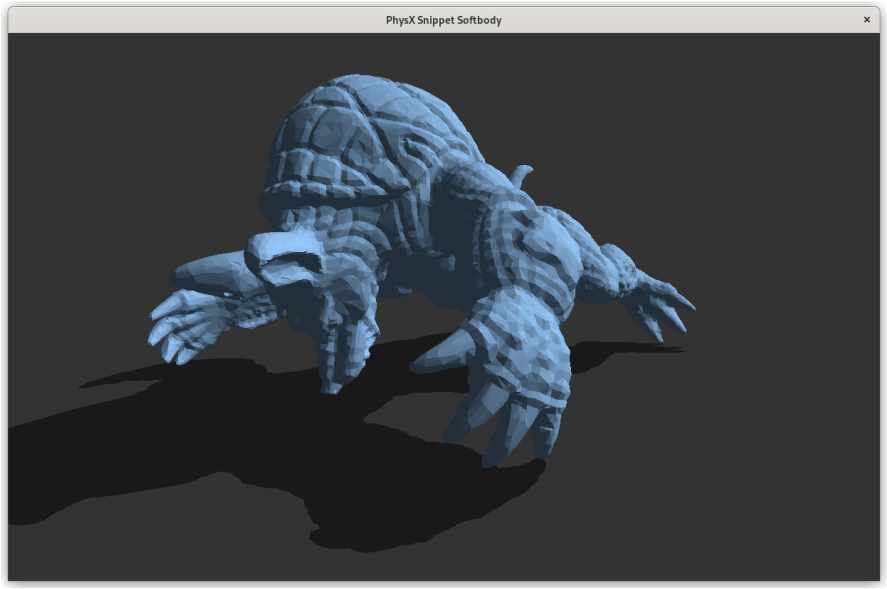


图 7.17 PhysX 的 Armadillo 仿真，基于 Softbody Snippet

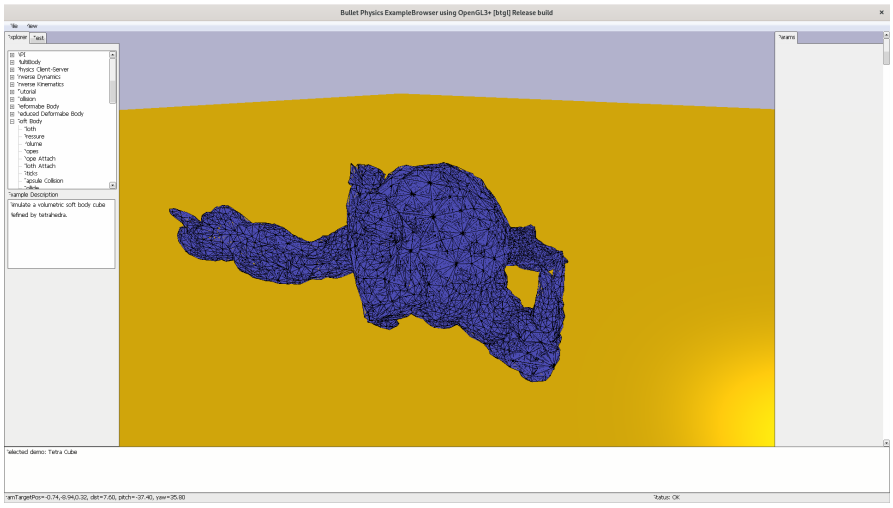


图 7.18 Bullet 的 Armadillo 仿真，基于 Tetra Cube example

基于体素（voxel）的四面体网格构建形式，对单个 Armadillo 模型仅构建了 355 个四面体，远少于 TetGen 算法构建的 49393 个四面体，故其效率高于本仿真器是必然的。但使用这一低分辨率四面体模型也导致了 PhysX 缺乏仿真时对模型细节的表现。例如，图 7.16 中的 Armadillo 表现出鼻子和手指因触碰到地板而产生方向的改变，而使用 PhysX 的基于体素的四面体网格很难体现这一点。

8 结论与展望

本毕业设计基于 PD 算法框架和 CUDA 并行程序设计完成了一个基于 GPU 的高效的弹性体仿真器。该仿真器同时在与工业界的 PhysX、Bullet 的对比中不占下风，这使得其具有较强的应用价值，或可整合入国产工业引擎 RaysEngine 的物理仿真模块，应用于实际的生产实践中。

但不可否认的是，本仿真器仍具许多不足之处和改进空间。首先，目前的碰撞检测和处理功能相对简单，仅支持弹性体与简单几何碰撞体之间的无摩擦碰撞处理，为在更多的场景得到应用，更丰富的碰撞功能尚待实现。其次，A-Jacobi 算法的效率或仍存在问题，具有较高的改进空间，需要作者进一步学习 CUDA 并行程序设计在底层方面的知识，以分析其计算过程的开销。最后，尽管 4 种约束已经可以仿真绝大多数超弹性模型，仍可为仿真器设计更多类型的约束，以适应实际应用中的需要。

本毕业设计让作者从对物理仿真一无所知到把握了 PBD 和 PD 框架的实现，并顺带学习了微分几何、有限元分析和 CUDA 编程的相关知识，是一次不小的成长。通过完成毕业设计时大量的文献查阅，作者对近年弹性体物理仿真的最新成果也有了一定的把握，更深刻地理解到了该领域的高难度和独特的魅力。

希望本毕业设计能够成为作者在物理仿真领域的“敲门砖”。“长风破浪会有时，直挂云帆济沧海”。

参考文献

[1] NORRIE D H, VRIES G D. The finite element method[J]. Osborne McGraw-Hill, 1973.

[2] BARAFF D. Large Steps in Cloth Simulation[J]. Proc. SIGGRAPH '98, 1998: 43-54.

[3] MÜLLER M, HEIDELBERGER B, TESCHNER M, et al. Meshless Deformations Based on Shape Matching[J]. ACM Transactions on Graphics, 2005, 24(3): 471-478. DOI: 10.1145/1073204.1073216.

[4] MÜLLER M, HEIDELBERGER B, HENNIX M, et al. Position Based Dynamics[J]. Journal of Visual Communication and Image Representation, 2007, 18(2): 109-118. DOI: 10.1016/j.jvcir.2007.01.005.

[5] BOUAZIZ S, MARTIN S, LIU T, et al. Projective Dynamics: Fusing Constraint Projections for Fast Simulation[J]. ACM Transactions on Graphics, 2014, 33(4): 1-11. DOI: 10.1145/2601097.2601116.

[6] WANG H. A Chebyshev Semi-Iterative Approach for Accelerating Projective and Position-Based Dynamics[J]. ACM Transactions on Graphics, 2015, 34(6): 1-9. DOI: 10.1145/2816795.2818063.

[7] WANG H, YANG Y. Descent Methods for Elastic Body Simulation on the GPU[J]. ACM Transactions on Graphics (TOG), 2016, 35(6): 1-10.

[8] LAN L, MA G, YANG Y, et al. Penetration-Free Projective Dynamics on the GPU[J]. ACM Transactions on Graphics, 2022, 41(4): 69:1-69:16. DOI: 10.1145/3528223.3530069.

[9] MACKLIN M, MÜLLER M, CHENTANEZ N. XPBD: Position-Based Simulation of Compliant Constrained Dynamics[C]//Proceedings of the 9th International Conference on Motion in Games. Burlingame California: ACM, 2016: 49-54. DOI: 10.1145/2994258.2994272.

[10] BRANDT C, EISEMANN E, HILDEBRANDT K. Hyper-Reduced Projective Dynamics[J]. ACM Transactions on Graphics, 2018, 37(4): 1-13. DOI: 10.1145/3197517.3201387.

[11] LIU T, BARGTEIL A W, O'BRIEN J F, et al. Fast Simulation of Mass-Spring Systems[J]. ACM Transactions on Graphics, 2013, 32(6): 1-7. DOI: 10.1145/2508363.2508406.

[12] LIU T, BOUAZIZ S, KAVAN L. Quasi-Newton Methods for Real-Time Simulation of Hyperelastic Materials[J]. ACM Transactions on Graphics, 2017, 36(3): 1-16. DOI: 10.1145/2990496.

[13] WANG H, O'BRIEN J, RAMAMOORTHI R. Multi-Resolution Isotropic Strain Limiting[J]. ACM Transactions on Graphics, 2010, 29(6): 156:1-156:10. DOI: 10.1145/1882261.1866182.

[14] NICKOLLS J R, BUCK I, GARLAND M, et al. Scalable Parallel Programming with CUDA [C]//Hot Chips 20 Symposium. 2008. DOI: 10.1145/1365490.1365500.

[15] FRATARCANGELI M, TIBALDO V, PELLACINI F. Vivace: A Practical Gauss-Seidel Method for Stable Soft Body Dynamics[J]. ACM Transactions on Graphics (TOG), 2016, 35(6): 1-9.

[16] OGDEN R W. Non-Linear Elastic Deformation[J]. Engineering Analysis with Boundary Elements, 1984, 1(2): 119.

[17] GUENNEBAUD G, JACOB B, et al. Eigen v3[Z]. http://eigen.tuxfamily.org. 2010.

[18] JACOBSON A, PANOZZO D. Libigl: Prototyping Geometry Processing Research in C++ [C]//SIGGRAPH Asia 2017 Courses. 2017. DOI: 10.1145/3134472.3134497.

[19] SI H. TetGen, a Delaunay-Based Quality Tetrahedral Mesh Generator[J]. ACM Transactions on Mathematical Software, 2015, 41(2): 1-36. DOI: 10.1145/2629697.

[20] HU Y, ZHOU Q, GAO X, et al. Tetrahedral Meshing in the Wild[J]. ACM Transactions on Graphics, 2018, 37(4): 1-14. DOI: 10.1145/3197517.3201353.

[21] BERGOU M, WARDETZKY M, HARMON D, et al. A Quadratic Bending Model for Inextensible Surfaces[J].

[22] CARNO D. Differential Geometry of Curves and Surfaces[M]. Differential Geometry of Curves and Surfaces, 1976. DOI: 10.1007/b137116.

[23] TAUBIN G. A Signal Processing Approach to Fair Surface Design[C]//Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques - SIGGRAPH ’95. Not Known: ACM Press, 1995: 351-358. DOI: 10.1145/218380.218473.
[24] SORKINE O. Laplacian Mesh Processing[J]. Proc of Eurographics, 2005.
[25] CAISSARD T, COEURJOLLY D, LACHAUD J O, et al. Laplace‐Beltrami Operator on Digital Surfaces[J]. Journal of Mathematical Imaging and Vision, 2019, 61(3): 359-379. DOI: 10.1007/s10851-018-0839-4.
[26] MEYER M, DESBRUN M, SCHRÖDER P, et al. Discrete Differential-Geometry Operators for Triangulated 2-Manifolds[G]//FARIN G, HEGE H C, HOFFMAN D, et al. Visualization and Mathematics III. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003: 35-57. DOI: 10.1007/978-3-662-05105-4_2.
[27] FLOATER M S. Mean Value Coordinates[J]. Computer Aided Geometric Design, 2003, 20(1): 19-27. DOI: 10.1016/S0167-8396(03)00002-5.
[28] SIFAKIS E, BARBIC J. FEM Simulation of 3D Deformable Solids: A Practitioner’s Guide to Theory, Discretization and Model Reduction[C]//SIGGRAPH ’12: ACM SIGGRAPH 2012 Courses. New York, NY, USA: Association for Computing Machinery, 2012: 1-50. DOI: 10.1145/2343483.2343501.
[29] CHAO I, PINKALL U, SANAN P, et al. A Simple Geometric Model for Elastic Deformations [J]. ACM Transactions on Graphics, 2010, 29(4): 1-6. DOI: 10.1145/1778765.1778775.
[30] HERNANDEZ F, CIRIO G, PEREZ A G, et al. Anisotropic Strain Limiting[J]. 2013.
[31] GAO M, WANG X, WU K, et al. GPU Optimization of Material Point Methods[J]. ACM Transactions on Graphics, 2018, 37(6): 254:1-254:12. DOI: 10.1145/3272127.3275044.
[32] BURDEN R, FAIRES J D. Numerical Analysis, 9th Edition[J]. 2010.
[33] FRATARCANGELI M, WANG H, YANG Y. Parallel Iterative Solvers for Real-Time Elastic Deformations[G]//SIGGRAPH Asia 2018 Courses. 2018: 1-45.
[34] LY M, JOUVE J, BOISSIEUX L, et al. Projective Dynamics with Dry Frictional Contact[J]. ACM Transactions on Graphics, 2020, 39(4). DOI: 10.1145/3386569.3392396.

附录

A 对式 3-5 的证明

提出式 3-5 的论文[23]及其后继成果均未给出对该式的正确性的严格证明，且这些论文的参考文献也未提供该式的相关信息，但作者仍在简单类圆闭曲线上证明了这一结论。虽然作者未系统受过微分几何训练，无法对任意闭曲线进行证明，但可以认为，将结论推广到任意闭曲线上应是合理的。

证明 设嵌入于 $\mathbb{R}^3$ 的二维流形 S 在点 $p \in S$ 附近的 ε-邻域二阶可微。不失一般性，不妨设 p 附近有显式函数表达 $z = f(x,y)$，且 $\boldsymbol{q} = c(p)$ 的坐标为 $(0,0,0)$，经过其的切平面为 $z=0$，法向量为 $\boldsymbol{N} = (0,0,1)$（当 p 不满足该条件时，总可以用对 S 进行等距变换使 p 移动到坐标系的原点，并满足以上条件）。记 $\boldsymbol{u} = (u_1, u_2, 0) = (u_1, u_2)$ 为 $\boldsymbol{q}$ 所在切平面上的任一单位向量，由微分几何中的第二基本形式的推导过程可知，

$$\kappa_{\boldsymbol{u}} = \boldsymbol{u}^T \boldsymbol{H}|_{\boldsymbol{q}} \boldsymbol{u} \tag{A-1}$$

为 $\boldsymbol{q}$ 处的在 $\boldsymbol{u}$ 方向上的曲率，其中

$$\boldsymbol{H}|_{\boldsymbol{q}} = \begin{bmatrix} f_{xx}(\boldsymbol{q}) & f_{xy}(\boldsymbol{q}) \\ f_{xy}(\boldsymbol{q}) & f_{yy}(\boldsymbol{q}) \end{bmatrix} \tag{A-2}$$

为 f 在 $\boldsymbol{q}$ 处的 Hessian。根据微分几何相应概念的定义，$\kappa_{\boldsymbol{u}}$ 的最大值 κ_1、最小值 κ_2 称为 S 在 $\boldsymbol{q}$ 处的**主曲率**，其平均值 $H(\boldsymbol{q}) = (\kappa_1 + \kappa_2)/2$ 则称为**平均曲率**。记 Hessian 的最大、最小特征值分别为 λ_1, λ_2，若使 $\kappa_{\boldsymbol{u}}$ 取最大值，由线性代数结论，有

$$\kappa_1 = \boldsymbol{u}^T \boldsymbol{H}|_{\boldsymbol{q}} \boldsymbol{u} \le \boldsymbol{u}^T \lambda_1 \boldsymbol{u} = \lambda_1 \boldsymbol{u}^T \boldsymbol{u} = \lambda_1. \tag{A-3}$$

且对最小特征值有类似结论，故 $H(\boldsymbol{q}) = (\lambda_1 + \lambda_2)/2$。此时，取到最值的 $\boldsymbol{u}$ 所在方向即为主曲率的方向，且两个主曲率方向互相正交。进而，可以旋转 S 使得

两个主曲率的方向恰好为 x 轴、y 轴的方向，而不影响以上结论。此时，$\boldsymbol{H}|_{\boldsymbol{q}}$ 成为对角矩阵，即

$$f_{xy}(\boldsymbol{q}) = 0. \tag{A-4}$$

且 $f_{xx}(\boldsymbol{q})$ 和 $f_{yy}(\boldsymbol{q})$ 成为两个主曲率 κ_1, κ_2，不妨记 $f_{xx}(\boldsymbol{q}) = \kappa_1, f_{yy}(\boldsymbol{q}) = \kappa_2$。

由于经过 $\boldsymbol{q}$ 的切平面为 $z = 0$，有 $f_x(\boldsymbol{q}) = f_y(\boldsymbol{q}) = 0$。由 $\boldsymbol{q}$ 处的二阶泰勒展开，有

$$\begin{aligned} f(x,y) &\approx f(\boldsymbol{q}) + f_x(\boldsymbol{q})x + f_y(\boldsymbol{q})y + \frac{1}{2}f_{xx}(\boldsymbol{q})x^2 + \frac{1}{2}f_{yy}(\boldsymbol{q})y^2 + f_{xy}(\boldsymbol{q})xy \\ &= \frac{1}{2}f_{xx}(\boldsymbol{q})x^2 + \frac{1}{2}f_{yy}(\boldsymbol{q})y^2 \\ &= \frac{1}{2}(\kappa_1 x^2 + \kappa_2 y^2). \end{aligned} \tag{A-5}$$

其中 $(x,y) \in S$ 为 $\boldsymbol{q}$ 附近任意一点。取充分小的 $\varepsilon > 0$，考虑环绕 $\boldsymbol{q}$ 的，半径为 ε 的圆周 $\gamma_\varepsilon \subset S$，满足参数方程

$$\begin{cases} x = \varepsilon\cos\theta, \\ y = \varepsilon\sin\theta, \\ z = \frac{\varepsilon^2}{2}(\kappa_1\cos^2\theta + \kappa_2\sin^2\theta). \end{cases} \tag{A-6}$$

那么，$|\gamma_\varepsilon| \approx 2\pi\varepsilon^2$。考虑积分

$$\begin{aligned} \lim_{|\gamma_\varepsilon|\to 0} \frac{1}{\varepsilon^2|\gamma_\varepsilon|}\int_{\boldsymbol{v}\in\gamma_\varepsilon} \boldsymbol{v}\,\mathrm{d}l(\boldsymbol{v}) &= \lim_{\varepsilon\to 0}\frac{1}{2\pi\varepsilon^3}\int_0^{2\pi}\boldsymbol{v}\varepsilon\,\mathrm{d}\theta \\ &= \lim_{\varepsilon\to 0}\frac{1}{2\pi\varepsilon^3}\int_0^{2\pi}\left(\varepsilon\cos\theta\hat{\boldsymbol{i}} + \varepsilon\sin\theta\hat{\boldsymbol{j}} + \frac{\varepsilon^2}{2}(\kappa_1\cos^2\theta + \kappa_2\sin^2\theta)\hat{\boldsymbol{k}}\right)\varepsilon\,\mathrm{d}\theta \\ &= \lim_{\varepsilon\to 0}\frac{1}{2\pi\varepsilon^3}\left(0 + 0 + \frac{\varepsilon^3\pi}{2}(\kappa_1+\kappa_2)\boldsymbol{N}\right) \\ &= \frac{1}{4}(\kappa_1+\kappa_2)\boldsymbol{N} \\ &= \frac{1}{2}H\boldsymbol{N}. \end{aligned} \tag{A-7}$$

注意到上述计算给出的结果与式 3-5仅有系数上的差异。当 $\boldsymbol{q}$ 为原点时，$(\boldsymbol{q}-\boldsymbol{v})=-\boldsymbol{v}$，故上式在简单类圆闭曲线 γ_ε 上验证了式 3-5 的结论。若 γ_ε 有其他不同的形状，$H\boldsymbol{N}$ 可能具有其他系数，因此 Taubin 和 Sorkine 均提出[23][24]，在离散化场合可使用不同的参数 w_{ij} 估计平均曲率向量，即计算 $\sum_{j\in N(i)} w_{ij}(\boldsymbol{q}_i-\boldsymbol{q}_j)$，这导致了余切公式（式 3-7）与平均值公式（式 3-8）的提出。

B 对四面体张力约束局部求解的源代码

```
__host__ __device__ void TetStrainConstraint::project_c_AcTAchpc(
    SimScalar* __restrict__ b, const SimScalar* __restrict__ q)
    const
{
    SimVector3 cur_pos[4];
    for (int i = 0; i < 4; i++)
    {
        const VertexIndexType v = vertices[i];

        SimScalar x = (SimScalar)q[3 * v];
        SimScalar y = (SimScalar)q[3 * v + 1];
        SimScalar z = (SimScalar)q[3 * v + 2];

        cur_pos[i] = { x, y, z };
    }

    // v[3] as pivot
    SimMatrix3 D_s;
    for (int i = 0; i < 3; i++)
    {
        D_s.col(i) = cur_pos[i] - cur_pos[3];
    }

#ifdef __CUDA_ARCH__
    const SimMatrix3 F = multiply3x3(D_s, D_m_inv); // deformation
    gradient
#else
    const SimMatrix3 F = D_s * D_m_inv;
#endif

    const bool tet_inverted = determinant3(F) < 0;
#ifdef __CUDA_ARCH__
    // GPU side SVD
    SimMatrix3 U;
    SimMatrix3 V;
    SimVector3 sigma;
    gpu_svd3(F, U, sigma, V);
#else
    // CPU side SVD
    Eigen::JacobiSVD<SimMatrix3> svd(F, Eigen::ComputeFullU | Eigen
    ::ComputeFullV);
    const SimMatrix3& U = svd.matrixU();
    SimVector3 sigma = svd.singularValues();
    const SimMatrix3& V = svd.matrixV();
#endif

    for (int i = 0; i < 3; i++)
```

```
    {
        sigma(i) = std::clamp(sigma(i), (SimScalar)min_strain_xyz(i
    ), (SimScalar)max_strain_xyz(i));
    }
    if (tet_inverted)
    {
        sigma(2) = -sigma(2);
    }

#ifdef __CUDA_ARCH__
    const SimMatrix3 Achpc = multiply3x3(V, multiply_diagx3(sigma,
    U.transpose())); // equivalent to (U * sigma * V^T)^{-1}
#else
    const SimMatrix3 Achpc = V * sigma.asDiagonal() * U.transpose()
    ;
#endif

    // apply A_c^T
    for (int i = 0; i < 4; i++)
    {
        const VertexIndexType v = vertices[i];

        const SimRowVector3 neg_D_m_inv = -(D_m_inv.row(0) +
    D_m_inv.row(1) + D_m_inv.row(2));

        SimRowVector3 sum_of_products;
        sum_of_products.setZero();
        for (int j = 0; j < 3; j++)
        {
            // v3 as pivot vertex
            if (i == 3)
            {
                sum_of_products += Achpc.row(j) * neg_D_m_inv(j);
            }
            else
            {
                sum_of_products += Achpc.row(j) * D_m_inv.coeff(i,
    j);
            }
        }

        for (int j = 0; j < 3; j++)
        {
        #ifdef __CUDA_ARCH__
            atomicAdd(&b[3 * v + j], sum_of_products[j] * wc);
        #else
            b[3 * v + j] += sum_of_products[j] * wc;
        #endif
        }
```

```
    }
}
```