Soft Body Simulation

Dong She
University of Copenhagen
qbp758@alumni.ku.dk

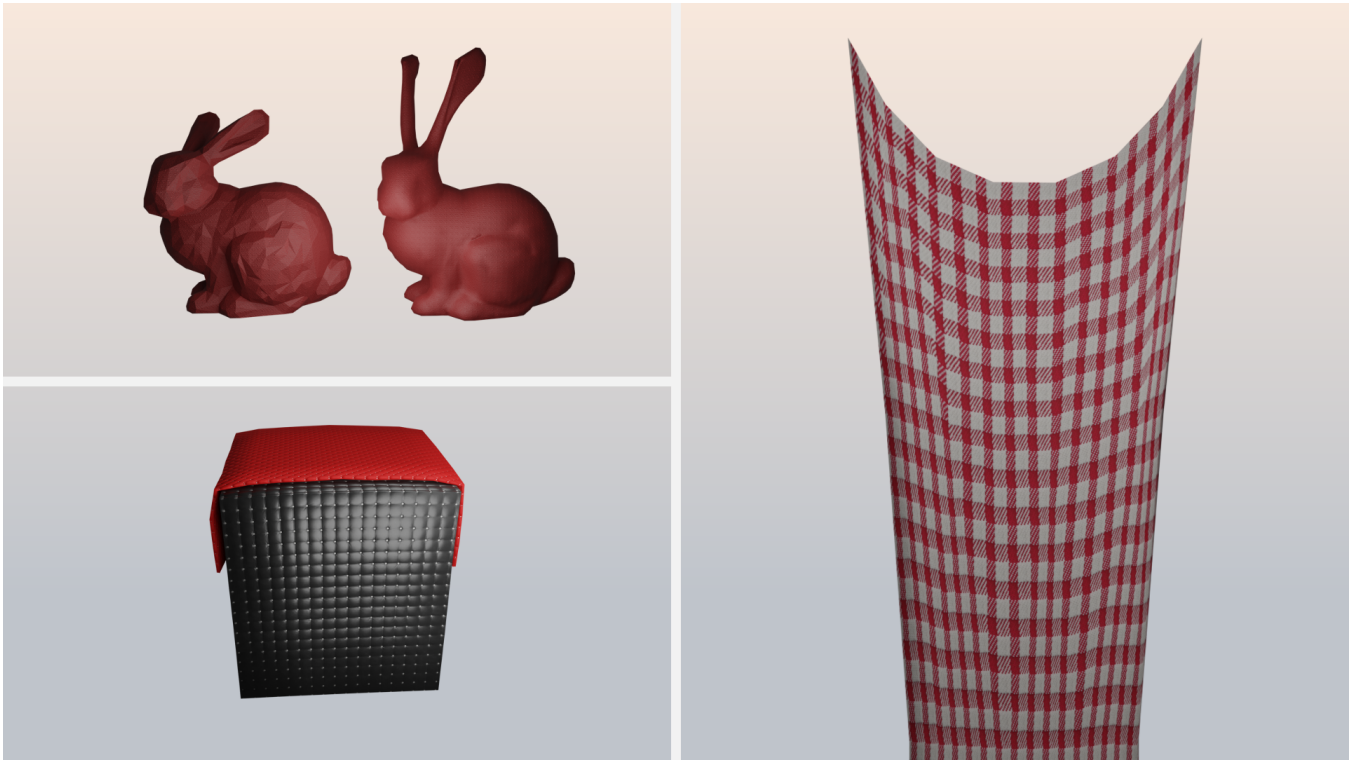


Figure 1: Using the RAINBOW software for soft body simulations: on the left, the top image showcases bunny ears stretched long, while the bottom shows a soft sheet resting atop a black box. On the right, a cloth is hung from its upper corners.

1 INTRODUCTION

Soft-body simulation is integral to computer graphics, influencing various fields such as gaming, robotics, and animation. Given the complexity and the need for realism in these areas, the efficiency and accuracy of soft-body simulation have become an important research topic.

This project investigates the core concepts of soft-body simulation [Andrews et al. 2022], focusing on improving the performance and stability of existing soft-body simulators. This initiative will involve the analysis of the RAINBOW [Erleben 2022] soft-body simulator, followed by the design and implementation of improved solutions.

I systematically explore and propose enhancements to the soft body simulation within the RAINBOW software. In the next of this report, I will comprehensively explain the project in detail and present the results of the experiments. Section 2 initiates the analysis of the existing simulation framework, delving into its fundamental theories and operational demonstrations. Following this, Section 3 outlines potential improvements aimed at bolstering the system's efficiency and accuracy, including refined collision detection and fully implicit time stepping. The subsequent sections, 4 and 5, are dedicated to evaluating the experimental outcomes of these proposed improvements and concluding with a reflective overview of the project as well as prospects for future advancements.

2 ANALYSIS OF SOFT BODY SIMULATION OF THE RAINBOW

2.1 Baseline of Soft Body Simulator of the RAINBOW

The RAINBOW software is used to build a simluator by python, it can run simulations of rigid bodies and soft bodies. In this section, it will introduce the essential features and functionalities of the soft body simulator of the RAINBOW software.

Consitutive Models. The RAINBOW software provides three different consitutive models for the soft body simulator, which are Neo-Hookean, St. Venant-Kirchhoff and Corotational Linear Elastic. These three models enable us to simulate the soft body with different material properties within the sofy body simulation.

Domain Discretization. To represent the soft bodies in the simulation, the RAINBOW software provides a lots of geometry helper functions to create the tetrahedra mesh, which is the most common discretization of the domain for soft body simulation.

Boundary Conditions. The simluator includes the helper function to create the boundary conditions. For example, it is easy to set up the Dirichlet boundary conditions in the simulation.

Time Integration. For the numerical solution of the soft body dynamics, the simulator implements the semi-implicit Euler time integration method.

Collision Detection. A fundamental feature of the soft body simulator is the collision detection system. RAINBOW's simluators are based on the the bounding volume hierarchy. The bounding volume hierarchy is a tree structure, which is used to speed up the collision detection.

Visualization. The RAINBOW software includes a visiualization system for view the soft body motion with the 'pythreejs' package. Additionally, in the simulation scene, it aledry setting up the crema, and lighting ans others of the rendering pass.

2.2 Soft Body Simulation Demostrations

In this section, it will present some simulations using the soft body simulator of the RAINBOW software. These simulations aim to illustrate the dynamic behaviors in different scenes. Below, we explain each demonstration in detail:

Demonstration 1. The demonstration showcases a simulation of a soft beam, which is hung on the wall. Specifically, the soft beam is fixed on the left side of the wall. As the simulation progresses, the right part of the soft beam will fall down because of gravity. The final state of the soft beam is shown in Figure 2.

Demonstration 2. In this demonstration, it shows a soft beam, which is fixed on the left and right sides along the horizontal axis. Following this, a vertical traction force is applied from the bottom boundary of the soft beam, causing the beam to bend. The final state of the soft beam is shown in Figure 4.

Demonstration 3. In this scene, we simulate a soft flat surface that is also fixed on both the left and right sides. On top of this soft plat surface, we place a cube. As the simulation progresses, the soft flat surface gradually bends downward. This demonstration showcases the deformation behavior of the soft surface and the impact of the cube's weight on it. The initial state and final state of the deformed surface are shown in in Figure 3.

2.3 The Theory of the Collision Detection System

This section discusses the theoretical background of the collision detection system implemented in the RAINBOW software. The methodology primarily employs the surface-based and local method integrated with Signed Distance Function(SDF) as reference in [Erleben 2018; Macklin et al. 2020] to detect the collision and generate the corresponding contact points between the two objects. Utilize the Frank-Wolfe to identify the deepest contact point in the RAINBOW software, noted for its efficiency in achieving local optimization as reference in [Macklin et al. 2020]. Furthermore, to speed up the collision detection, the chunked bounding volume hierarchy (BVH) [Schmidtke and Erleben 2018] is used to reduce the number of collision detections. The following sections will explain the details of the collision detection system.

2.3.1 Basic Concepts. Before delving into the collision detection system, it is essential to introduce some fundamental concepts. The first one is the signed distance function (SDF), which is a function that returns the shortest distance from a point to the surface of an object, mathematically represented as:

$$s(\mathbf{x}) = \begin{cases} d(\mathbf{x}) & \text{if } \mathbf{x} \in \Omega \\ -d(\mathbf{x}) & \text{if } \mathbf{x} \in \Omega^c \end{cases}$$

where $d(\mathbf{x})$ is the distance from the point $\mathbf{x}$ to the surface of the object, Ω is the object, and Ω^c is the complement of Ω. The sign indicates that the value is negative inside the object, positive outside the object, and zero on the surface of the object.

The second concept is the barycentric coordinates. In a 2D plane, barycentric coordinates are a method to determine the position of a point relative to a triangle. The representation of a point $\mathbf{P}$ in a triangle $\mathbf{A}, \mathbf{B}, \mathbf{C}$ is given by:

$$\mathbf{P} = \alpha_1 \mathbf{A} + \alpha_2 \mathbf{B} + \alpha_3 \mathbf{C}$$

s

where $\alpha_1, \alpha_2, \alpha_3$ are the barycentric coordinates of the point **Ps**. The barycentric coordinates are defined as:

$$\alpha_1 = A_{\text{BPC}}$$
$$\alpha_2 = A_{\text{APC}}$$
$$\alpha_3 = A_{\text{APB}}$$

here, A_{BPC} is the area of the triangle **BPC**, A_{APC} is the area of the triangle **APC**, and A_{APB} is the area of the triangle **APB**. The barycentric coordinates are illustrated in Figure 5. and subject to $\alpha_1 + \alpha_2 + \alpha_3 = 1$.

Similarly, extend this concept to 3D, the barycentric coordinates of a point $\mathbf{P}$ in a tetrahedron $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}$ is defined as:

$$\mathbf{P} = \alpha_1 \mathbf{A} + \alpha_2 \mathbf{B} + \alpha_3 \mathbf{C} + \alpha_4 \mathbf{D}$$

where $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ are the barycentric coordinates of the point $\mathbf{x}$. The barycentric coordinates are defined as:

$$\alpha_1 = V_{\text{BPCD}}$$
$$\alpha_2 = V_{\text{APCD}}$$
$$\alpha_3 = V_{\text{APBD}}$$
$$\alpha_4 = V_{\text{APBC}}$$

here, V_{BPCD} is the volume of the tetrahedron **BPCD**, V_{APCD} is the volume of the tetrahedron **APCD**, V_{APBD} is the volume of the tetrahedron **APBD**, and V_{APBC} is the volume of the tetrahedron **APBC**. The barycentric coordinates are illustrated in Figure 6. and subject to $\alpha_1 + \alpha_2 + \alpha_3 + \alpha_4 = 1$.

The BVH[Wikipedia contributors 2023b], which is a k-tree, where each node represents a bounding box of the object, and each leaf node represents an individual object, plays a pivotal role in collision detection. An example of the BVH is shown in Figure7. There are various different bounding volume types[Wikipedia contributors 2023a], including the axis-aligned bounding box (AABB),

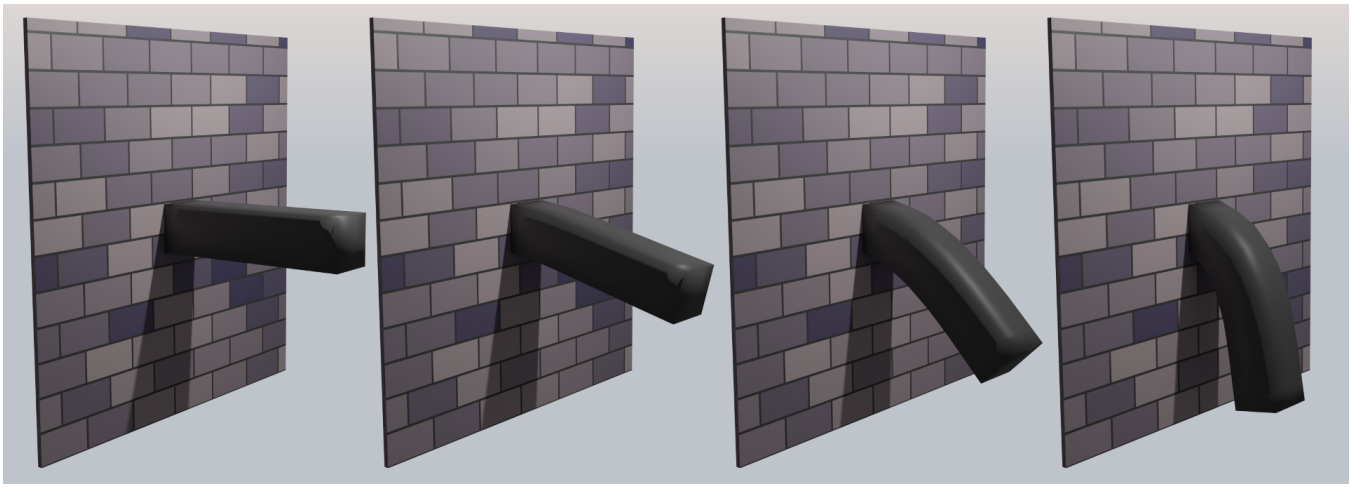
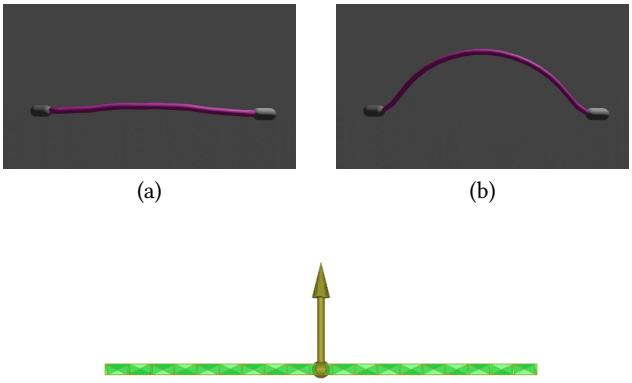


Figure 2: In this figure, a soft beam (illustrated in a coffee-brown color) is fixed to a wall on its left side. From 0 to 1 second of the simulation, the soft beam is gradually affected by the force of gravity, falling downward. The four pictures from left to right illustrate the transition from its initial upright position to its final, deformed state, demonstrating the soft beam's response to gravitational forces.

Figure 3: In this simulation, a soft sheet (depicted in dark) is firmly fixed on both sides, with a cube situated on it. As the simulation progresses, the surface gradually sags under the cube's weight, illustrating the deformation behavior. The sequence of images, arranged from left to right, and from top to bottom, shows the progression of the simulation.

discrete-oriented polytope(k-DOP), and others. In the context of the RAINBOW software, the k-DOP is used to construct the BVH. The k-DOP is a convex polytope, which is defined as the intersection of k half-spaces as:

$$\mathrm{DOP}_k = \{\mathbf{x} \in \mathbb{R}^3 | \mathbf{a}_i^T \mathbf{x} \leq b_i, i = 1, 2, \cdots, k\}$$



(a) (b)

(c)

Figure 4: In this simulation, a soft beam is depicted experiencing forces applied at its center, and it is fixed on two ends. Initially presented as a straight line in Figure (a), the beam begins to exhibit a noticeable bend at its center in Figure (b). In Figure (c), the traction force(yellow arrow) exerted on the beam is clearly illustrated.

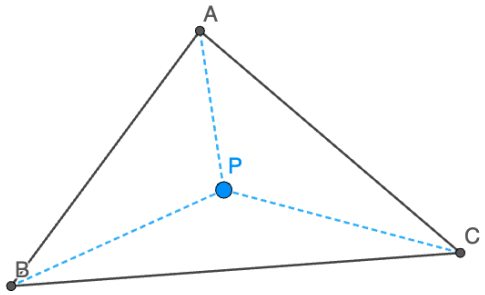


Figure 5: The illustration displays a triangle formed by the vertices A, B, and C, with point P representing the barycentric point within the triangle.

Moreover, the RAINBOW software employs an advanced BVH data structure known as Chunked BVH[Schmidtke and Erleben 2018], In the Chunked BVH, a chunk is a set of subtrees of the BVH,

$$\text{Chunk} = \{T_1, T_2, \cdots, T_n\}$$

here, T_i is a subtree of the BVH.

2.3.2 Surface-based Collision Detection. In RAINBOW software, the Chunked BVH is used to speed up collision detection. Given M objects in space, and considering that each subtree encompasses N elements, the calculation of the number of leaves in a balanced binary subtree and the value of the chunks C can be computed as:

$$L = \frac{N+1}{2}$$

$$C = \lceil \frac{M}{L} \rceil$$

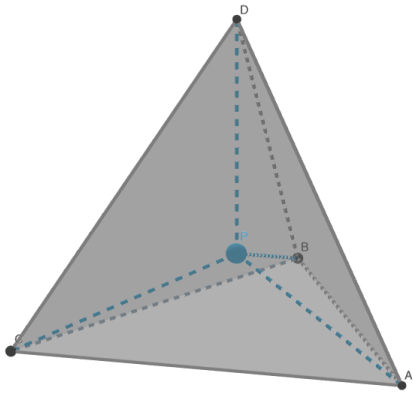


Figure 6: The illustration displays a tetrahedron formed by the vertices A, B, C, and D, with point P representing the barycentric point within the tetrahedron.

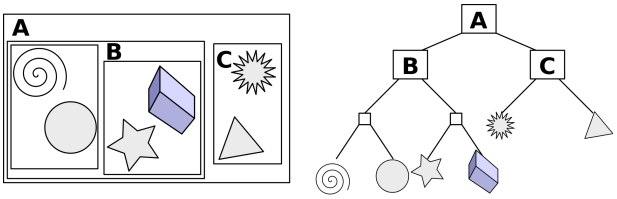


Figure 7: Left: different geometry shapes in the space. Right: the BVH of the geometry shapes.[Wikipedia contributors 2023b]

The construction of the chunked BVH tree adopts a top-down in-order traversal scheme, as detailed in Algorithm 1. Additionally, the BVH tree undergoes updates at each step to account for deformations in the soft body.

The computation of the contact point, a pivotal component of the collision detection system, follows the identification of intersections. RAINBOW software undertakes precise collision detection between Signed Distance Fields (SDFs) and triangle meshes, employing the Frank-Wolfe algorithm to pinpoint the deepest contact point. This section elucidates the steps involved in computing the contact point.

Convert to the Local Coordinate System. In a 3D space, it is critical to map one of the intersecting triangles onto the material space of the tetrahedron corresponding with the other triangle. Operating in a localized material space not only simplifies calculations, making transformations and computations more straightforward but also enhances numerical stability. This is particularly beneficial when world coordinates have large magnitudes, potentially leading to numerical errors.

This transformation process from the world space to the material space encompasses several steps. Let's denote the vertices of the tetrahedron in the world space as $\mathbf{V_{0w}}$, $\mathbf{V_{1w}}$, $\mathbf{V_{2w}}$ and $\mathbf{V_{3w}}$, and in the material space as $\mathbf{V_{0m}}$, $\mathbf{V_{1m}}$, $\mathbf{V_{2m}}$ and $\mathbf{V_{3m}}$. The vertices of the triangle mesh are represented as **A**, **B**, and **C**.

Initially, we aim to compute the barycentric coordinates of the triangle mesh vertices within the tetrahedron in the world space.

Algorithm 1 Narrow Phase of Collision Detection

```
1: B_s are the total objects in the space.
2: Create all combinations C_s of the B_s.
3: C is the number of the chunks
4: Res is the result of the collision detection
5: for each pair (A, B) ∈ C_s do
6:     BVH_A ← the BVH tree of A
7:     BVH_B ← the BVH tree of B
8:     for each subtree_A ∈ BVH_A.chunks do
9:         for each subtree_B ∈ BVH_B.chunks do
10:            Queue Q
11:            Node_A ← the root of subtree_A
12:            Node_B ← the root of subtree_B
13:            push(Q, Node_A)
14:            push(Q, Node_B)
15:            while Q is not empty do
16:                pop(Q, Node_A)
17:                pop(Q, Node_B)
18:                if Node_A and Node_B are not colliding then
19:                    continue
20:                end if
21:                if Node_A and Node_B are leaf nodes then
22:                    Append the intersections information to Res
23:                end if
24:                if Node_A is not a leaf node and Node_B is not
   a leaf node then
25:                    for each child_A in Node_A.children do
26:                        for each child_B in Node_B.children do
27:                            push(Q, child_A)
28:                            push(Q, child_B)
29:                        end for
30:                    end for
31:                    continue
32:                end if
33:                if Node_A is not a leaf node then
34:                    for each child_A in Node_A.children do
35:                        push(Q, child_A)
36:                        push(Q, Node_B)
37:                    end for
38:                    continue
39:                end if
40:                if Node_B is not a leaf node then
41:                    for each child_B in Node_B.children do
42:                        push(Q, Node_A)
43:                        push(Q, child_B)
44:                    end for
45:                    continue
46:                end if
47:            end while
48:        end for
49:    end for
50: end for
51: Return Res
```

his can be achieved using the subsequent equations, which involve calculating individual coordinates $\mathbf{A_x}$, $\mathbf{A_y}$, and $\mathbf{A_z}$ by utilizing a weighted sum of the vertices' coordinates and an additional equation to normalize the weights:

$$w_1V_{0wx} + w_2V_{1wx} + w_3V_{2wx} + w_4V_{3wx} = A_x$$
$$w_2V_{0wy} + w_2V_{1wy} + w_3V_{2wy} + w_4V_{3wy} = A_y$$
$$w_3V_{0wz} + w_2V_{1wz} + w_3V_{2wz} + w_4V_{3wz} = A_z$$
$$w_1 + w_2 + w_3 + w_4 = 1$$

Thus, the relationship can be modeled as follows by the matrix style:

$$\begin{bmatrix} A_x \\ A_y \\ A_z \\ 1 \end{bmatrix} = \begin{bmatrix} V_{0wx} & V_{1wx} & V_{2wx} & V_{3wx} \\ V_{0wy} & V_{1wy} & V_{2wy} & V_{3wy} \\ V_{0wz} & V_{1wz} & V_{2wz} & V_{3wz} \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{A} \\ 1 \end{bmatrix} = \underbrace{\left[\begin{array}{c|c|c|c} \mathbf{V_{0w}} & \mathbf{V_{1w}} & \mathbf{V_{2w}} & \mathbf{V_{3w}} \\ 1 & 1 & 1 & 1 \end{array}\right]}_{4*4 matrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix}$$

Therefore, we can compute the w_1, w_2, w_3, and w_4 by solving the linear equation. However, we can use an easy way to figure out the barycentric coordinates. The idea is that we can use the $w_4 = 1 - w_1 - w_2 - w_3$ to rewrite the linear equation. Then, we get:

$$\begin{aligned} \mathbf{A} &= w_1\mathbf{V_{0w}} + w2\mathbf{V_{1w}} + w_3\mathbf{V_{2w}} + (1 - w_1 - w_2 - w_3)\mathbf{V_{3w}} \\ &= w_1(\mathbf{V_{0w}} - \mathbf{V_{3w}}) + w_2(\mathbf{V_{1w}} - \mathbf{V_{3w}}) + w_3(\mathbf{V_{2w}} - \mathbf{V_{3w}}) + \mathbf{V_{3w}} \end{aligned}$$

Hence,

$$\begin{aligned} \mathbf{A} - \mathbf{V_{3w}} &= w_1(\mathbf{V_{0w}} - \mathbf{V_{3w}}) + w_2(\mathbf{V_{1w}} - \mathbf{V_{3w}}) + w_3(\mathbf{V_{2w}} - \mathbf{V_{3w}}) \\ &= \underbrace{\left[\begin{array}{c|c|c} \mathbf{V_{0w}} - \mathbf{V_{3w}} & \mathbf{V_{1w}} - \mathbf{V_{3w}} & \mathbf{V_{2w}} - \mathbf{V_{3w}} \end{array}\right]}_{3*3 matrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} \end{aligned}$$

Now, we can use a matrix $\mathbf{B}$, $\mathbf{b_1}$, $\mathbf{b_2}$, and $\mathbf{b_3}$ instead of the $3 * 3$ matrix:

$$\mathbf{b_1} = \mathbf{V_{0w}} - \mathbf{V_{3w}}$$
$$\mathbf{b_2} = \mathbf{V_{1w}} - \mathbf{V_{3w}}$$
$$\mathbf{b_3} = \mathbf{V_{2w}} - \mathbf{V_{3w}}$$

Hence,

$$\mathbf{B} = [\mathbf{b_1} \quad \mathbf{b_2} \quad \mathbf{b_3}]$$

Assume the $\mathbf{w} = [w_1w_2w_3]^T$, the barycentric coordinates of the triangle mesh vertice $\mathbf{A}$ in the tetrahedron can be computed by the following equation:

$$\mathbf{B}w = \mathbf{A} - \mathbf{V_{3w}}$$

Consequently, the $\mathbf{w}$ can be computed by :

$$\mathbf{w} = \mathbf{B}^{-1}(\mathbf{A} - \mathbf{V_{3w}})$$

Last, we can compute the w_1 by the following equation:

$$w_4 = 1 - w_1 - w_2 - w_3$$

The same method is applied to compute the barycentric coordinates for the other vertices $\mathbf{B}$ and $\mathbf{C}$, denoted as $\mathbf{u}$ and $\mathbf{v}$, respectively.

Following this, we transition to convert the barycentric coordinates from the world space to the material space of the tetrahedron. Assume that the local space matrix is $\mathbf{M}$, which is defined as:

$$\mathbf{M} = [\mathbf{V_{0m}} \quad \mathbf{V_{1m}} \quad \mathbf{V_{2m}} \quad \mathbf{V_{3m}}]$$

The barycentric coordinates in the tetrahedron space can be computed by the following equation:

$$\mathbf{A'} = \mathbf{Mw}$$
$$\mathbf{B'} = \mathbf{Mu}$$
$$\mathbf{C'} = \mathbf{Mv}$$

where $\mathbf{A'}$, $\mathbf{B'}$, $\mathbf{C'}$ are the barycentric coordinates in the tetrahedron space.

Compute the Contact Closest Point by Local Optimization. Upon converting to the local coordinate system, the closest point can be computed by solving the following optimization problem over the barycentric coordinates:

$$\arg\min_{w,u,v} s(w\mathbf{A}, u\mathbf{B}, v\mathbf{C})$$

In RAINBOW software, we use the Frank-Wolfe algorithm to facilitate the identification of the deepest contact point. During each iteration, the minimum SDF gradient norm of the three vertices is the next point to update, denoted as s_i. The update rule is given by:

$$\mathbf{x_{i+1}} = \mathbf{x_i} + \alpha(\mathbf{s_i} - \mathbf{x_i})$$

here $\alpha = \frac{2}{i+2}$, i is the iteration number, $\mathbf{x_i} = u_i, v_i, w_i$,

2.4 The Theory of Proximal Operator Solver

The proximal method is a higher level of abstraction optimization algorithm. This section presents the utility of the proximal algorithm in computing contact force within the RAINBOW software. Initially, we will introduce the definition of the proximal algorithm. Subsequently, a detailed contact force model of the soft body will be present, followed by an exploration of determining the contact force utilizing the proximal algorithm.

2.4.1 Definition of the Proximal Algorithm. The definition of the proximal algorithm is presented herein and refers to the foundational framework established by [Parikh and Boyd 2014].

Suppose a closed, proper, and convex function $f : \mathbf{R}^n \to \mathbf{R} \cup \{+\infty\}$, the proximal operator $\mathbf{prox}_f : \mathbf{R}^n \to \mathbf{R}^n$ of f is formulated as

$$\text{prox}_f(z) = \arg\min_x \left(f(x) + \frac{1}{2}\|x - z\|_2^2 \right) \tag{1}$$

where $\| \cdot \|_2$ denotes the second Euclidean norm, and given that the function minimized on the right-hand side is strongly convex and not everywhere infinite, it guarantees a unique minimizer for every $z \in \mathbf{R}^n$.

In frequent instances, the proximal operator is combined with a scaled function λf, where $\lambda > 0$. As a result,

$$\text{prox}_{\lambda f}(z) = \arg\min_x \left(f(x) + \frac{1}{2\lambda}\|x - z\|_2^2 \right) \tag{2}$$

Consider f defined as the indicator function:

$$I_C(x) = \begin{cases} 0 & x \in C \\ +\infty & x \notin C \end{cases}$$

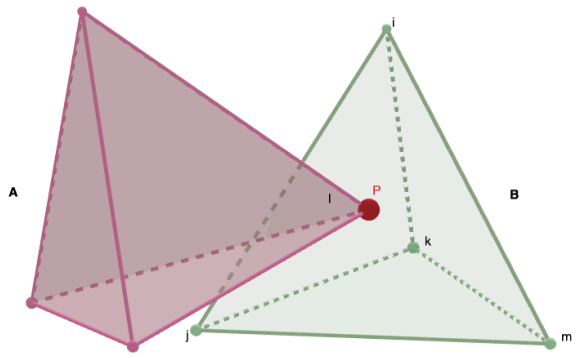


Figure 8: This illustration depicts a collision between two entities, body A and body B, highlighting the contact point P in red. The nodes of the bodies are indexed as i, j, k, m, and l. The velocity of the contact point P is calculated utilizing barycentric coordinates: $\mathbf{v}_{p,B} = w_i\mathbf{v}_{i,B} + w_j\mathbf{v}_{j,B} + w_k\mathbf{v}_{k,B} + w_m\mathbf{v}_{m,B}$

where C is a closed nonempty convex set, Under this consideration, the proximal operator of f simplifies to Euclidean projection onto C, expressed as follows:

$$\begin{aligned} \text{prox}_C(z) &= \arg\min_{x \in C} \|x - z\|_2 \\ &= \arg\min_{x \in C} \|z - x\|_2 \end{aligned} \tag{3}$$

2.4.2 Contact Force Model with the Proximal Operator.

The Normal Impulse λ_n : In the context of simulating collisions between two bodies, it is imperative to impose a non-interpenetration contact constraint to prevent physical impossibilities and ensure realism in the simulation.

Let us define the ϕ as the gap function and λ_n as the magnitude of an impulse along the normal direction to the contact plane, ensuring the absence of interpenetration between two bodies. The relative velocity along the normal direction of the contact plane can be mathematically defined as $\mathbf{v}_n = \dot{\phi}$.

The non-interpenetration contact constraint can be defined by two conditions:

(1) The relative velocity between the two colliding bodies is null, expressed as $\mathbf{v}_n = 0$. In such a scenario, an impulse is essential to prevent overlap necessitating $\lambda_n > 0$.
(2) The impulse is null, denoted as $\mathbf{v}_n > 0$, indicative of a non-collision state, thereby result in $\lambda_n = 0$

Combine the both of two aforementioned conditions can be mathematical representations:

$$\mathbf{v}_n \geq 0, \quad \lambda_n \geq 0, \quad \text{and} \quad \mathbf{v}_n\lambda_n = 0 \tag{4}$$

We suppose that the Δv is the relative contact point velocity, then

$$\mathbf{v}_n = \mathbf{n} \cdot \Delta v$$

Considering a collision between two bodies, namely A and B, at point p as depicted in Figure 8, where i, j, k, m, l denote indices of the bodies, we explain the non-interpenetration contact constraint:

The relative velocity $\mathbf{v_n}$ along the normal direction can be defined as:

$$\mathbf{v}_n = \mathbf{n} \cdot (\mathbf{v}_{pB} - \mathbf{v}_{lA})$$

where $\mathbf{v}_{p,B}$ is the velocity of the contact point p of the body B, the $\mathbf{v}_{l,A}$ is the velocity of the node l of body A.

Employing the barycentric coordinate, the $\mathbf{v}_{pB}$ can be represented as:

$$\mathbf{v}_{p,B} = w_i\mathbf{v}_{i,B} + w_j\mathbf{v}_{j,B} + w_k\mathbf{v}_{k,B} + w_m\mathbf{v}_{m,B}$$

As a result in:

$$\mathbf{v}_n = \mathbf{n} \cdot (w_i\mathbf{v}_{i,B} + w_j\mathbf{v}_{j,B} + w_k\mathbf{v}_{k,B} + w_m\mathbf{v}_{m,B} - \mathbf{v}_{l,A})$$

Matrix representation of this expression introduces the contact Jacobian, denoted as $\mathbf{J}$, leading to:

$$\mathbf{v}_n = \underbrace{\begin{bmatrix} -n^T & w_in^T & w_jn^T & w_kn^T & w_mn^T \end{bmatrix}}_{\mathbf{J}} \underbrace{\begin{bmatrix} \mathbf{v}_{l,A} \\ \mathbf{v}_{i,B} \\ \mathbf{v}_{j,B} \\ \mathbf{v}_{k,B} \\ \mathbf{v}_{m,B} \end{bmatrix}}_{\mathbf{u}} = \mathbf{Ju}$$

here, the matrix $\mathbf{J}$ and the velocity vector $\mathbf{u}$ are defined as:

We can use the proximal operator with the equation 4. Then we obtain:

$$\lambda(n) = \text{prox}(n)(\lambda_n - r_n\mathbf{v}_n) \quad \text{for} \quad r_n > 0 \tag{5}$$

here r_n is a scalar variable is called `r-factor`, which will influence the convergence.

We extend the one contact point scene to multiple contact points scenario, and implementing the Augmented Lagrangian method, the contact constraints employing proximal operators:

$$\lambda(ni) = \text{prox}(ni)(\lambda_{ni} - r_{ni}\mathbf{v}_{ni}) \quad \text{for} \quad \forall i \tag{6}$$

where sub-index i references the contact point index.

According to the Newton impact law, we assume that the $\mathbf{v}^-$ and $\mathbf{v}^+$ denote the pre- and post-impact contact velocities. We have:

$$\mathbf{v}_{ni} = \mathbf{v}^+_{ni} + \epsilon_{ni}\mathbf{v}^-_{ni} \tag{7}$$

We proceed to integrate the equation the equation 7 for $\mathbf{v}_{ni}$ into equation 6, resulting in the subsequent formulation:

$$\lambda(ni) = \text{prox}(ni)(\lambda_{ni} - r_{ni}(\mathbf{v}^+_{ni} + \epsilon_{ni}\mathbf{v}^-_{ni})) \quad \text{for} \quad \forall i \tag{8}$$

Employing the Newton-Euler equations and kinematic maps, the relative velocity is computed as follows:

$$\begin{aligned} \mathbf{M}\dot{\mathbf{u}} &= \mathbf{h} + \mathbf{J}^T\lambda, \\ \dot{\mathbf{q}} &= \mathbf{u} \end{aligned} \tag{9}$$

where $\dot{u} = \frac{\mathbf{u}^{t+\Delta t} - \mathbf{u}^t}{\Delta t}$, and $\mathbf{u}$ epresents the generalized velocity vector, $\mathbf{q}$ is generalized position vector, $\mathbf{M}$ is the mass matrix, $\mathbf{J}$ is the contact Jacobian, and $\mathbf{h}$ contains external and gyroscopic force terms.

Utilizing the time discretization to obtain:

$$\mathbf{u}^{t+\Delta t} = \mathbf{u}^t + \Delta t\mathbf{M}^{-1}\mathbf{h} + \mathbf{M}^{-1}\mathbf{J}^T\lambda \tag{10}$$

Applying the semi-implicit time-stepping method in RAINBOW software, so the $\mathbf{v}^- = \mathbf{Ju}^t$, and $\mathbf{v}^+ = \mathbf{Ju}^{t+\Delta t}$. We have:

$$\begin{aligned} \mathbf{v}^+ &= \mathbf{Ju}^{t+\Delta t} \\ &= \mathbf{J}(\mathbf{u}^t + \Delta t\mathbf{M}^{-1}\mathbf{h} + \mathbf{M}^{-1}\mathbf{J}^T\lambda) \\ &= \mathbf{JM}^{-1}\mathbf{J}^T\lambda + \mathbf{Ju}^t + \Delta t\mathbf{JM}^{-1}\mathbf{h} \end{aligned} \tag{11}$$

With $\mathbf{R}$ and $\mathbf{E}$ representing the diagonal matrices containing `r-Factors` and ϵ coefficients respectively. , we define:

$$\begin{aligned} \mathbf{z} &= \lambda - \mathbf{R}(\mathbf{v}^+ + \mathbf{E}\mathbf{v}^-) \\ &= \lambda - \mathbf{R}(\mathbf{JM}^{-1}\mathbf{J}^T\lambda + \mathbf{Ju}^t + \Delta t\mathbf{JM}^{-1}\mathbf{h} + \mathbf{EJu}^t) \end{aligned} \tag{12}$$

With the established definitions, we have:

$$\mathbf{z}_{ni} = \lambda_{ni} - \mathbf{R}_{ni}(\mathbf{J}_{ni}\mathbf{M}^{-1}_{ni}\mathbf{J}^T_{ni}\lambda_{ni} + \mathbf{J}_{ni}\mathbf{u}^t_{ni} + \Delta t\mathbf{J}_{ni}\mathbf{M}^{-1}_{ni}\mathbf{h}_{ni} + \mathbf{E}_{ni}\mathbf{J}_{ni}\mathbf{u}^t_{ni})$$

Then, we integrate the $\mathbf{z}_{ni}$ into the equation 8 to obtain:

$$\lambda(n_i) = \text{prox}(n_i)(\mathbf{z}_{ni}) \quad \text{for} \quad \forall i \tag{13}$$

Before we introduce the iterative proximal method to solve this equation, we introduce how to compute the friction force λ_f.

The Friction Force λ_f***:*** Based on Coulomb friction law, two conditions are shown as follows:

(1) **Slip:** When bodies slide relative to each other, the friction force is opposite to the tangential relative velocity, and its magnitude is determined by $\mu\lambda_n$, where μ denotes the coefficient of friction.
(2) **Stick:** In the absence of relative motion between the bodies, the friction force can assume any direction provided that the inequality $|\lambda_f|_2 \leq \mu\lambda_n$ is satisfied.

Mathematically, these scenarios can be expressed as follows:

$$\begin{aligned} \mu \cdot \lambda_n - \sqrt{\lambda_f \cdot \lambda_f} &\geq 0 \\ \|\mathbf{v}_f(\mu\lambda_n - \sqrt{\lambda_f \cdot \lambda_f}) &= 0 \\ \|\mathbf{v}_f\|\|\lambda_f\| &= -\mathbf{v}_f \cdot \lambda_f \end{aligned} \tag{14}$$

Here, the term $|\mathbf{v}_f|$ functions as a selector between the slip and stick conditions. When $|\mathbf{v}_f| = 0$, the final two conditions are inherently met, with the first condition emphasizing that the frictional force should lie within the friction cone. Conversely, if $|\mathbf{v}_f| > 0$, the last two conditions become significant.

These stick-slip conditions can be generalized, particularly when considering energy dissipation. By positing that the frictional force dissipates energy at its maximum rate, the stick-slip conditions can be deduced for an isotropic circular friction cone based on the principle of maximal dissipation. For more complex forces, the friction cone definition (illustrated in Figure 9) can be described as follows:

$$\mathcal{F}(\mu\lambda_n) \equiv \left\{ \gamma \in \mathbf{R}^2 \;\middle|\; \|\gamma\|_2 \leq \mu\lambda_n \right\} \tag{15}$$

Subsequently, maximum energy dissipation can be converted to a minimization problem:

$$\lambda_f = \underset{\gamma \in \mathcal{F}(\mu\lambda_n)}{\arg\min}\ \mathbf{v}_f \cdot \gamma \tag{16}$$

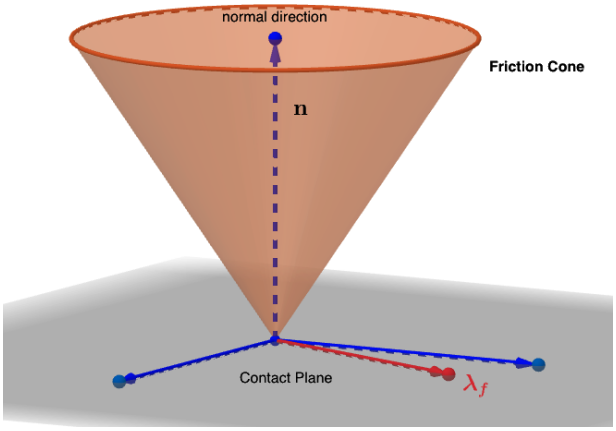


Figure 9: This figure depicts the friction cone on the contact plane, with the orange cone representing the frictional aspect, the vector n signifying a normal perpendicular to the contact plane, and the red line illustrating the friction force.

According to section 1.5 of [Andrews et al. 2022], when $\mathcal{F}(\mu\lambda_n)$ forms a strictly convex set, we can establish the variational inequality (VI) for the first-order optimality condition of the above minimization problem:

$$\forall\gamma \in \mathcal{F}(\mu\lambda_n) \quad \text{and} \quad (\gamma - \lambda_f)\cdot \mathbf{v}_f \geq 0 \tag{17}$$

Now a solution to this variational inequality is equivalent to the fixed point of the proximal operator:

$$\lambda_f = \text{prox}_{\mathcal{F}(\mu\lambda_n)}(\lambda_f - r_f\mathbf{v}_f) \quad \text{for} \quad r_f > 0 \tag{18}$$

Now, We extend the one contact point scene to multiple contact point scenarios similarly as previously,

$$\lambda_{fi} = \text{prox}_{\mathcal{F}(\mu\lambda_{ni})}(\lambda_{fi} - r_{fi}\mathbf{v}_{fi}) \quad \text{for} \quad r_{fi} > 0 \tag{19}$$

where sub-index i references the contact point index.

Next, we define the $\mathbf{z}_{fi}$:

$$\mathbf{z}_{fi} = \lambda_{fi} - \mathbf{R}_{fi}(\mathbf{J}_{fi}\mathbf{M}_{fi}^{-1}\mathbf{J}_{fi}^T\lambda_{fi} + \mathbf{J}_{fi}\mathbf{u}_{fi}^t + \Delta t\mathbf{J}_{fi}\mathbf{M}_{fi}^{-1}\mathbf{h}_{fi} + \mathbf{E}_{fi}\mathbf{J}_{fi}\mathbf{u}_{fi}^t)$$

Then, we integrate the $\mathbf{z}_{fi}$ into the equation 19:

$$\lambda(f_i) = \text{prox}(\mathcal{F}_i)(\mathbf{z}_{fi}) \quad \text{for} \quad \forall i \tag{20}$$

Combine the proximal operator on the normal impulse we have:

$$\begin{aligned} \forall i \quad \lambda_{ni} &= \text{prox}_{ni}(\mathbf{z}_{ni}) \\ \forall i \quad \lambda_{fi} &= \text{prox}_{\mathcal{F}_i}(\mathbf{z}_{fi}) \end{aligned} \tag{21}$$

The solutions of these are given by:

$$\begin{aligned} \forall i \quad \lambda_{ni}^{k+1} &= \max(0, \mathbf{z}_{ni}) \\ \forall i \quad \lambda_{fi}^{k+1} &= \begin{cases} \mathbf{z}_{fi} & ; \mathbf{z}_{fi\in\mathcal{F}_i} \\ \arg\min_{\gamma\in\mathcal{F}_i} \left\|\mathbf{z}_{fi} - \gamma\right\|^2 & ; \text{ otherwise} \end{cases} \end{aligned} \tag{22}$$

2.4.3 Iterative Solver. Iterative methods, especially the Jacobian and Gauss-Seidel schemes, are pivotal in computational mechanics for addressing numerical problems with distinct approaches. This section particularly underscores the Gauss-Seidel scheme. The idea of the Gauss-Seidel scheme is to always use the most updated λ-values in any computation. Let the $\mathbf{z}$ represent the most updated value at every computational iteration. The normal and frictional solutions for the i_{th} contact in subsequent iterations (denoted by superscript $k+1$) can be computed by:

$$\begin{aligned} \lambda_{ni}^{k+1} &= \text{prox}_{ni}(\mathbf{z}_{ni}) \\ \lambda_{fi}^{k+1} &= \text{prox}_{\mathcal{F}_i^{k+1}}(\mathbf{z}_{fi}) \end{aligned} \tag{23}$$

Further, the equation for $\mathbf{z}$ can be reformulated by leveraging factorization techniques:

$$\begin{aligned} \mathbf{z} &= \lambda - \mathbf{R}(\underbrace{\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^T\lambda}_{\mathbf{w}} + \underbrace{\mathbf{J}\mathbf{u}^t + \Delta t\mathbf{J}\mathbf{M}^{-1}\mathbf{h} + \mathbf{E}\mathbf{J}\mathbf{u}^t}_{\mathbf{b}}) \\ &= \lambda - \mathbf{R}(\mathbf{J}\mathbf{w} + \mathbf{b}) \end{aligned} \tag{24}$$

Suppose B as the index set for all bodies, and defining $I \equiv ni, fi$, fi as the index set for the i^{th} contact point, the most updated value of $\mathbf{z}_I$ can be determined prior to computing the contact forces by:

$$\mathbf{z}_I = \lambda_I^K - \mathbf{R}_i(\mathbf{J}_{IB}\mathbf{w} + \mathbf{b}_I) \tag{25}$$

Subsequent to updating the contact forces, $\mathbf{w}$ can be updated as follows:

$$\mathbf{w} = \left(\mathbf{M}^{-1}\mathbf{J}^T\right)_{BI}\left(\lambda_I^{k+1} - \lambda_I^k\right) \tag{26}$$

The iterations may continue until the residual error either converges or surpasses a predetermined threshold for maximum iterations. The entirety of the algorithmic approach can be referred to in Algorithm 2.

Algorithm 2 The PROX Gauss-Seidel Solver

Require: K: indices of all contacts, B indices of all bodies, **J**, **M**, **b**, **R**, λ^0, v

$(k, \lambda^k, \epsilon^k) \leftarrow (0, \lambda^0, \infty)$
while not converged **do**
  $\mathbf{w} \leftarrow \mathbf{M}^-1\mathbf{J}^T\lambda^k$
  for $i \in K$ **do**
    I ≡ indices of block ni, fi
    $\mathbf{z}_I \leftarrow \lambda_I^k - \mathbf{R}_i(\mathbf{J}_{IB}\mathbf{w} + \mathbf{b}_I)$
    $\lambda_{ni}^{k+1} \leftarrow \text{prox}_{ni}(\mathbf{z}_{ni})$
    $\lambda_{fi}^{k+1} \leftarrow \text{prox}_{\mathcal{F}_i(\lambda_{ni}^{k+1})}(\mathbf{z}_{fi})$
    $\mathbf{w} \leftarrow \mathbf{w} + \left(\mathbf{M}^{-1}\mathbf{J}^T\right)_{B,I}\left(\lambda_I^{k+1} - \lambda_I^k\right)$
  end for
  $\epsilon^{k+1} = \|\lambda^{k+1} - \lambda^k\|_\infty$
  if $\epsilon^{k+1} > \epsilon^k$ **then**
    $\mathbf{R} \leftarrow v\mathbf{R}$
  else
    $(\lambda^k, \epsilon^k, k) \leftarrow (\lambda^{k+1}, \epsilon_{k+1}, k+1)$
  end if
end while

3 IMPROVEMENTS FOR SOFT BODY SIMULATION OF RAINBOW

3.1 More Efficient Collision Detection System

Our initial observations from the soft body demonstrations in the RAINBOW software indicated a significant time consumption in the simulation processes. For instance, the third soft body demonstration alone took about 10 hours. Recognizing the need for a more expedited simulation, we delved deeper into the software's performance metrics.

3.1.1 Analysis of the Execution Time. To accurately identify performance bottlenecks, we deployed the `CProfile` tool, profiling the execution time during the soft body simulation. This was subsequently visualized using `SnakeViz` to obtain a holistic view of the simulation's runtime. As can be found from Figure 10, the collision detection phase emerged as the predominant time consumer. A more detailed investigation into this phase pinpointed the computing contact points algorithm (specifically, Frank-Wolfe) as the main time sink. This algorithm calculates contact points for overlaps, which are derived from the narrow phase. Additionally, the processes of contact point reduction and updating the BVH tree within the narrow phase also account for significant time consumption in collision detection.

Deep Dive into Collision Detection: On delving deeper into the collision detection segment of the RAINBOW software, we observed that the contact point computation, implemented by a nested loop, consumed the maximum time. The pseudocode for this calculation is as follows:

```
    for body_overlap in body_overlaps:
        for overlap in body_overlap.overlaps:
            contact_points_computing(overlap)
```

This structure (nested loops), while functional, can be flattened. By aggregating all overlaps into a singular list, we can simplify the operation to a single loop:

```
    # Construct the whole overlaps list
    whole_overlaps = []
    for body_overlap in body_overlaps:
        for overlap in body_overlap.overlaps:
            whole_overlaps.append(overlap)

    # Contact points computing phase
    for overlap in whole_overlaps:
            contact_points_computing(overlap)
```

With the current structure, the contact points computation phase is readily parallelizable on a GPU. Transitioning this phase to the GPU is expected to yield a substantial speedup based on the underlying logic. Consequently, this presents a promising approach for enhancing the performance of the collision detection system in RAINBOW.

Contact Points Reduction: Post contact point computation, the system proceeds to the reduction phase, aimed at eliminating redundant contact points. The extant algorithm, as straightforward, is brute force in nature:

```
    # Initialize an empty list called: reduced_list
    reduced_list = []

    # Loop all contact points
    for cp1 in contact_points:
        for cp2 in reduced_list:
            if cp1 != cp2:
                reduced_list.append(cp1)
```

This strategy, with its worst-case time complexity of $O(n^2)$ can be computationally heavy. To overcome this, we introduce a single-loop mechanism coupled with Python's `set` data type. Within the single loop, we can leverage Python's `set` data type to store each contact point. By checking if a new contact point is already present in the `set`, we can determine its uniqueness. If the contact point is not found in the `set`, it's deduced as unique and consequently added to the result list. The associated pseudo code is provided below for clarity:

```
    # Initialize an empty list called: reduced_list
    reduced_list = []
    unique_ids = set()

    # Loop all contact points
    for cp in contact_points:
        # Create a unique id of a contact point
        uid = uid_contact_point(cp)
        if uid not in unique_ids:
            reduced_list.append(cp)
            unique_ids.add(uid)
```

This improved strategy exhibits a worst-case-time complexity of $\Theta(n)$ greatly enhancing efficiency given the $O(1)$ query time for sets in Python.

Refinement of the Narrow Phase: Within RAINBOW, the BVH facilitates overlap calculations for scene objects. As a step forward in optimization, we propose switching to spatial hashing [Teschner et al. 2003] for this phase. This approach is more GPU-friendly compared to the BVH, potentially further augmenting performance.

Following the above thorough review of the collision detection of RAINBOW, we have identified three key points for performance optimization: one is parallelizing the contact points computation on the GPU, the other is changing the time complexity of contact points reduction from $O(n^2)$ to $\Theta(n)$, and the last one is incorporating spatial hashing to enhance the narrow phase of the collision detection process. In the subsequent sections, we will delve into the detailed implementation and present the experimental results.

3.1.2 GPU-based Contact Point Computation.

Implementation: To easily parallelization the program on GPU, we employed the Numba Cuda [Numba 2022] to write the CUDA code running on the GPU, This decision was primarily influenced by Numba CUDA's seamless integration with Python.

Our GPU-based computation approach can be delineated as following steps:

- **Data Preparation:** Initially, we aggregate all overlaps into a singular list on the host(CPU). Concurrently, we assemble data pertaining to the scene's objects into another list.
- **Data Transferred to GPU:** Once data is prepared, this data is transferred from the host to the device.
- **Kernel Execution:** Post data transferred to the device, we invoke the contact points computation kernel on the host.
- **Result Transferred to CPU:** Upon kernel function execution complete, computed results are transferred back from the device to the host.

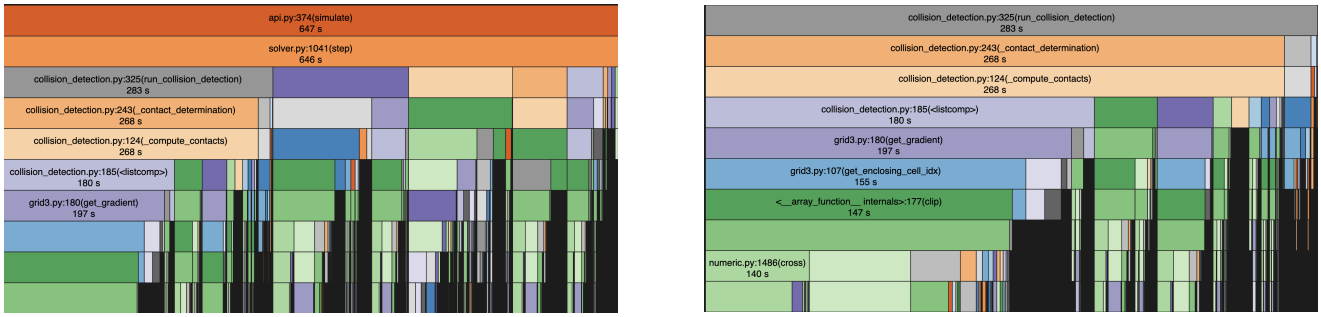


Figure 10: Visualization of the simulation's runtime using SnakeViz, the left figure shows the running time of the whole simulation process, and the right figure shows the running time of the `collision_detection` function. Highlight the collision detection phase as the primary time consumer.

To facilitate kernel function execution, adaptations were necessary for RAINBOW's functions to cater to the GPU's parallel environment version. This entailed crafting device-specific auxiliary functions that would assist the kernel in its contact point computation on GPU. For instance, basic tensor operations were re-implemented as device functions, encompassing tasks such as tensor addition and subtraction, among others.

After the establishment of these foundational device functions, our main task was the kernel function. In our approach, a one-dimensional grid was leveraged to process the overlaps list, which was the optimal option due to the inherent nature of this task. Overlap data was accessed using the thread's unique index, while data corresponding to specific objects was fetched based on their unique object ID within the scene. After that, the Frank-Wolfe algorithm was deployed to compute the contact points.

Experiment Result: We conducted two experiments to assess the performance of contact point computations when shifted from the CPU to the GPU. Consequently, we observed a dramatic acceleration, achieving up to a tenfold speed-up in computation times compared to the traditional CPU-based approach. The details of the two experiments are shown as follows:

- *Frame-wise Running Time Evaluation:* This experiment primarily tracked the execution time for computing contact points on each frame. Intriguingly, our initial findings showcased an increased runtime at the start point of the execution for the GPU version. This overhead can be attributed to the data transfer latency from the CPU to the GPU. Nevertheless, post this initialization phase, the GPU's runtime plot stabilized near the zero line, illustrating an almost real-time computation. On the other hand, the CPU's runtime was invariably greater than the GPU version after the initialization point, a detailed depiction of which is provided in Figure 11.
- *Scalability with Tetrahedrons:* Our second experiment measured the computation performance as the increasing number of tetrahedrons. The CPU's runtime exhibited a direct correlation, i.e., as the tetrahedron count escalated, so did its runtime. Conversely, the GPU-based computation, even under a burgeoning tetrahedron count, maintained its near-zero runtime.

From both of the two above experiments, a compelling observation: the GPU-accelerated approach for contact point computation bolsters the performance manifold. This pivot to a GPU-based approach undeniably amplifies the computational efficiency of simulations in RAINBOW.

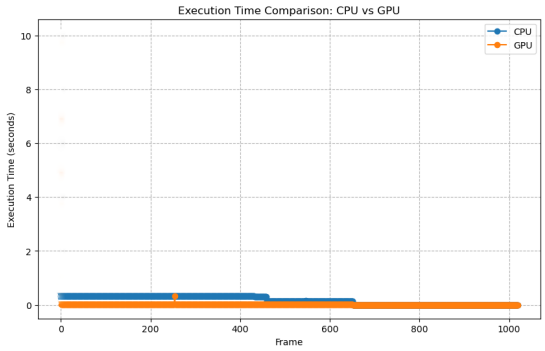


Figure 11: Comparative runtime analysis of contact point computations per frame between the traditional CPU-based approach and the optimized GPU-driven method. An initial overhead is evident for GPU processing due to data transfer, but subsequent frames highlight the GPU's superior, real-time performance.

3.1.3 *Efficiency Improvements in Contact Points Reduction.*

Implementation. The refinement of the contact points reduction is straightforward. From our earlier mentioned in the provided pseudo code, we predominantly employed the `set` data structure in Python. This structure facilitates the efficient recording of unique contact point identifiers. Consequently, with the singular loop, we managed to eliminate redundant contact points from the list. Theoretically, the time complexity for reducing contact points has been improved from $O(n^2)$ to $\Theta(n)$,

Experiment Result. In alignment with our GPU-based contact point computation tests, we conduct two experimental evaluations: a *Frame-wise Running Time Assessment* during the simulation process and a *Scalability Analysis with Tetrahedrons*. These experiments

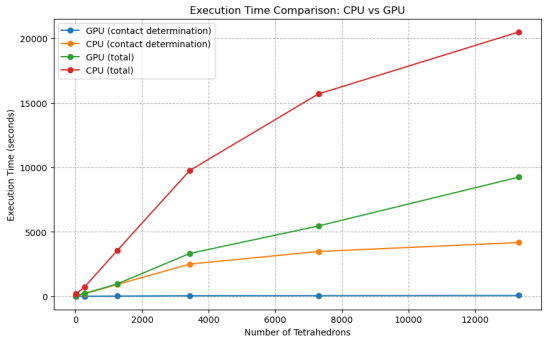


Figure 12: Runtime performance against an increasing number of tetrahedrons. While CPU processing times escalate sharply with more tetrahedrons, specifically the number of tetrahedrons over 10000, the GPU-based approach consistently maintains a near-zero computation duration, highlighting its scalability and efficiency.

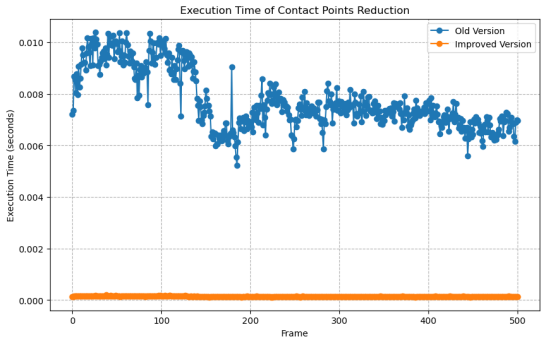


Figure 13: Performance evaluation of the optimized contact point reduction method. The graph contrasts the stable execution time of the new method with the fluctuating times of the previous approach, especially as the number of frames increases. The blue line represents the running time of the previous approach, while the orange line illustrates the running time of the revised method.

show the tangible speed enhancements of our revised contact points reduction method when juxtaposed against the previous method.

- *Frame-wise Running Time Evaluation:* Our first experiment centered on measuring the efficiency of contact point computations across consecutive frames. Evidently, the revised approach consistently delivered almost instantaneous results, maintaining a run-time nearing 0 seconds. In contrast, the performance of the erstwhile method fluctuate as the frame count increased, as visualized in Figure 13.
- *Scalability with Tetrahedrons:* Our second experiment was architected to evaluate the computational scalability in the face of an escalating tetrahedron count. Mirroring observations from our GPU-based experiments, the new methodology continued to register nearly 0 seconds of computation durations, irrespective of the tetrahedron volume. Conversely, the previous method of compact point reduction exhibited a direct correlation between extended run times and tetrahedron increments.

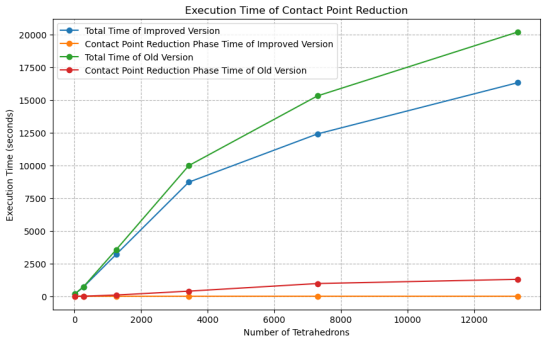


Figure 14: Comparison of computation performance with an increasing number of tetrahedrons. The orange line signifies the running time of the previous method, while the blue line showcases the efficiency of the revised approach.

3.1.4 *Spatial Hash.*

Implementation. We adapted the spatial hash method as detailed in the reference paper [Teschner et al. 2003]. The spatial hashing approach can be summarized into two key phases:

- *Triangles Insertion to Hash Table:* During this phase, we iterate over all surface triangles of all bodies and compute the Axis-Aligned Bounding Box (AABB) for each of them. Traversal the all cells in each AABB, compute their hash values and subsequently insert these values into the hash table, which is shown in Figure 16 in 2D for convenience understanding, however, extending this to 3D, you can visualize the division of the 3D space into lots of cubes, as depicted in Figure 15
- *Identifying Overlapping Pairs:* This involves traversing all collision candidates within the hash table and checking the AABBs of the triangle pairs. If any AABB overlaps, the corresponding triangle pairs are recorded for use in the computation of contact points during the subsequent phase.

The complete procedure can be found detailed in Algorithm 3. In RAINBOW implementation, we use the vectorized operation to figure the AABBs in once, hence, we remove the loops by vectorization.

Turning our attention to the hash function, we've opted for the perfect hash method as discussed in [Lefebvre and Hoppe 2006]. The formulation for the hash value computation is given by:

$$h(p) = h_0(p) + \Phi\Big[h_1(P)\Big] \bmod \mathrm{m}$$

where, m represents the size of the hash table.
The map h_0 is defined as:

$$h_0(p) = \mathbf{M_0}p \bmod m$$

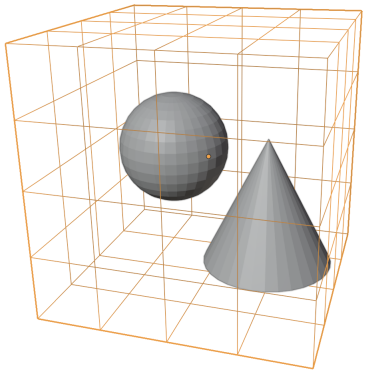


Figure 15: In this figure, the 3D space is divided into cubes, demonstrating that the sphere and the cone are encompassed by these cubes.

with $\mathbf{M_0}$ being a transformation matrix.
The map h_1 is defined as:

$$h_1(p) = \mathbf{M_1}p \bmod r$$

here, $\mathbf{M_1}$ is also a transformation matrix, and r is the offset table Φ size.

Owing to our simulations being situated in a 3D context, both the M_0 and M_1 are two 3*3 matrices. In our particular implementation, we strategically utilize two identity matrices for these transformations.

Algorithm 3 Spatial Hashing for Collision Detection

Require: The List of bodies in the scene
Initialize: Create an empty hash table **H**, the candidate overlap results **C**, and the true overlap results **R**
for each body in the scene **do**
  for each surface triangle of body **do**
    Create the AABB of the triangle
    for each cell in the AABB **do**
      Insert the triangle information to **H**
      if collisions found in hash table **then**
        Append the triangle pair to **C**
      end if
    end for
  end for
end for
for each candidate c in **C** **do**
  if overlap occurs between the AABBs of the triangle pair **then**
    Append the triangle pair to the **R**
  end if
end for

Experiment Result. We not only conducted two experiments to measure the efficacy of the spatial hashing method as in previous experiments but also conducted an experiment to find the best cell size in our simulation. Here are the detailed insights from these experiments:

- *Frame-wise Running Time Evaluation:* In this experiment, we tracked the execution time of the narrow phase across successive frames with 1250 tetrahedrons. The findings were clear: the running time for the spatial hashing method is consistently less than that of the BVH technique, a trend graphically demonstrated in Figure 17.
- *Scalability with Tetrahedrons:* Our second experiment was specifically devised to evaluate the performance efficiency of the spatial hashing method and the BVH (Bounding Volume Hierarchy) as the tetrahedron count in the scene increases. Preliminary findings indicated that when the scene comprised a relatively small number of tetrahedrons, the spatial hashing algorithm required a more prolonged execution time compared to the BVH technique. Conversely, as the tetrahedron population increased (the turning point was over 7500 tetrahedrons in my experiment), the spatial hashing method showcased superior efficiency, surpassing the BVH in terms of computational speed. This relationship between the tetrahedron count and algorithmic efficacy is comprehensively delineated in Figure 17.
- *Spatial Hashing Execution Time with Varied Cell Sizes:* The third experiment aimed to explore the relationship between spatial hashing execution time and varying cell sizes, specifically considering cell size as a multiple of the average edge length. Two test scenarios were employed, distinguished by their tetrahedron counts: one with 13,310 tetrahedrons, and the other with 1,250. As depicted in Figure 18, the X-axis represents the ratio of cell size to average edge length, while the orange and blue lines reflect the results from the tests with 13,310 and 1,250 tetrahedrons respectively. A clear pattern is observed, indicating that a smaller cell size is associated with a longer spatial hashing execution time. Remarkably, an optimal execution time was identified at a ratio of 2.2, meaning that the most efficient spatial hashing occurred when the cell size was 2.2 times greater than the average edge length.

Self-collision. In the context of soft body dynamics, the inherent deformable nature of soft bodies poses the complex challenge of self-collisions. To ensure computational efficiency while addressing this challenge, we incorporated a streamlined self-collision detection method within our spatial hashing framework.

Within this framework, when a collision is detected on the hash table, we check the colliding triangle pairs within the same hash slot. If these triangle pairs belong to the same body, it indicates a potential self-collision scenario. However, before confirming the self-collision, we determine whether the two triangles share a common vertex. Triangles sharing a vertex are inherently connected and do not constitute a true self-collision. If they do not share a vertex, we then proceed to check if the triangles actually intersect or merely reside within the same spatial cell.

For efficient and accurate intersection verification, we adopted the Moller-Trumbore intersection algorithm [Wikipedia contributors 2023c]. If a true intersection is confirmed, we subsequently

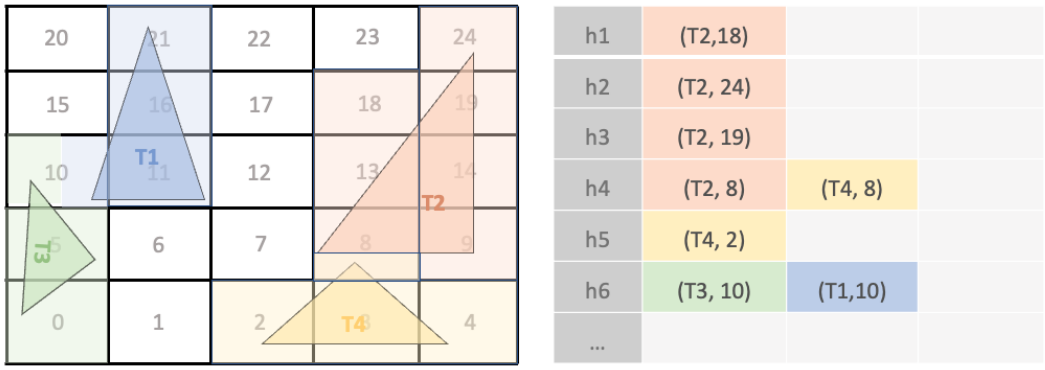


Figure 16: This figure illustrates the Tetrahedron Hashing phase, the second phase of the spatial hashing method, represented in a 2D perspective. The 2D surface is segmented into 25 cells. Triangle 2 (labeled as T2 in the figure) contains 7 cells, specifically numbers 8, 9, 13, 14, 18, 19, and 24. We iterate through all its cells, compute their hash values, and subsequently cross-reference them with the hash table displayed on the right side of the figure, for example, both (T3, 10) and (T1, 10) cells are collisions at the hash value $h6$ on the hash table.

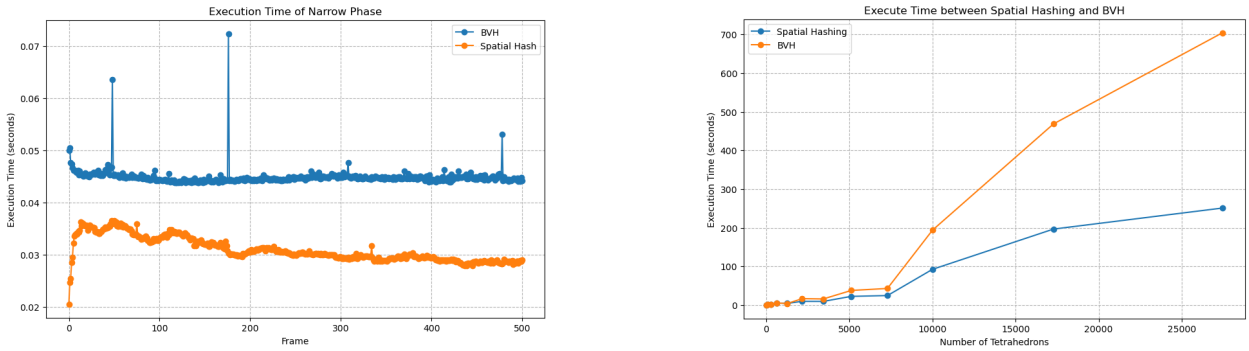


Figure 17: (Left) Evaluation of Frame-wise Running Time, showcasing the consistent execution time advantage of the spatial hashing method across successive frames with 1250 tetrahedrons. (Right) Scalability in Relation to Tetrahedron Count, illustrating a performance comparison between spatial hashing and BVH methods; spatial hashing exhibits huge improved efficiency with a growing number of tetrahedrons in the scene beyond the turning point. The turning point, in this experiment, is just over 7500 tetrahedrons

report this triangle pair for the computation phase of the contact points.

3.2 More Efficient Proximal Operator Solver

Achieving optimal performance in numerical computations often requires the integration of parallel processing techniques. In our quest to optimize the proximal algorithm, we delved into various parallelization strategies. One of the approaches we worked with is the Jacobi-scheme proximal. Its structure and operations are inherently parallel, making it a suitable candidate for efficient computation on the RAINBOW software.

Besides, we ventured to design a hybrid approach that combined the strengths of both the Jacobi and Gauss-Seidel schemes. While the Jacobi scheme offers the advantage of natural parallelism, the Gauss-Seidel scheme, on the other hand, provides a better convergence rate for problem-solving.

Lastly, this sequential nature of Gauss-Seidel makes its parallelization non-trivial and presents an intriguing challenge. In the subsequent sections, we discuss our strategies to overcome this challenge by the graph-coloring method.

3.2.1 Parallel Jacobi-Scheme Algorithm. Parallelizing the Jacobi-Scheme algorithm is straightforward due to its inherent support for concurrent operations. We can execute all iterations updating both λ_n and λ_f in parallel, and the parallel Jacobi-Scheme algorithm is shown in Algorithm 4.

For our implementation, we employed Numba for parallel computing, specifically using its non-Python mode. In this mode, Numba compiles the decorated function to run without involving the Python interpreter, thereby achieving optimal performance. This approach aligns with the guidelines provided in Numba's official documentation [Numba-Development-Team 2022].

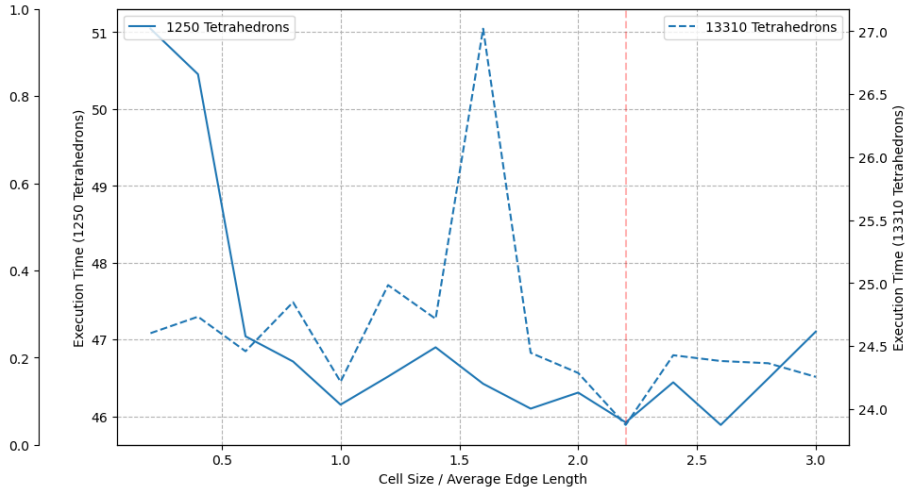


Figure 18: In this figure, the X axis represents the ratio of cell size to average edge length. Two distinct scenarios are plotted: the orange line correlates to a test with 13,310 tetrahedrons, while the blue line corresponds to a scenario with 1,250 tetrahedrons. Overall, a diminutive cell size is discerned to correlate with a longer spatial hashing execution time. The optimum execution time for spatial hashing is observed at a ratio of 2.2, indicating that the cell size is 2.2 times larger than the average edge length.

Algorithm 4 The Parallel PROX Jacobi-Scheme Solver

Require: K: indices of all contacts, B indices of all bodies, $\mathbf{J}$, $\mathbf{M}$, $\mathbf{b}$, $\mathbf{R}$, $\lambda^{\mathbf{0}}$, v

$(k, \lambda^k, \epsilon^k) \leftarrow (0, \lambda^0, \infty)$
while not converged **do**
  $\mathbf{w} \leftarrow \mathbf{M}^- 1\mathbf{J}^T\lambda^k$
  $\mathbf{z} \leftarrow \lambda^k - \mathbf{R}(\mathbf{J}\mathbf{w} + \mathbf{b})$
  for $i \in K$ in parallel computing **do**
    $\mathbf{I} \equiv$ indices of block ni, fi
    $\lambda_{ni}^{k+1} \leftarrow \text{prox}_{ni}(\mathbf{z}_{ni})$
    $\lambda_{fi}^{k+1} \leftarrow \text{prox}_{\mathcal{F}_i(\lambda_{ni}^k)}(\mathbf{z}_{fi})$
  end for
  $\epsilon^{k+1} = \|\lambda^{k+1} - \lambda^k\|_\infty$
  if $\epsilon^{k+1} > \epsilon^k$ **then**
    $\mathbf{R} \leftarrow v\mathbf{R}$
  else
    $(\lambda^k, \epsilon^k, k) \leftarrow (\lambda^{k+1}, \epsilon_{k+1}, k+1)$
  end if
end while

3.2.2 Parallel Hybrid Algorithm. In our implementation based on the Jacobi-scheme parallel approach, we focused on concurrently computing the values for λ_n and λ_f. Interestingly, while we compute these values in parallel, we still ensure that λ_f is updated using the most recent value of λ_n. This method can be represented mathematically as:

$$\lambda_n^{k+1} \leftarrow \text{prox}_n(\mathbf{z}_n)$$
$$\lambda_f^{k+1} \leftarrow \text{prox}_{\mathcal{F}_i(\lambda_n^{k+1})}(\mathbf{z}_f)$$

One of the key benefits of our approach is that, for the sake of easier parallel computation, we've eliminated the need to consistently update $\mathbf{z}$ and $\mathbf{w}$ with the latest λ_n in each iteration. This differentiates our method significantly from the Gauss-Seidel scheme approach. A comprehensive outline of our technique is depicted in the referred Algorithm 5.

Algorithm 5 The Parallel PROX Hybrid Solver

Require: K: indices of all contacts, B indices of all bodies, $\mathbf{J}$, $\mathbf{M}$, $\mathbf{b}$, $\mathbf{R}$, $\lambda^{\mathbf{0}}$, v

$(k, \lambda^k, \epsilon^k) \leftarrow (0, \lambda^0, \infty)$
while not converged **do**
  $\mathbf{w} \leftarrow \mathbf{M}^- 1\mathbf{J}^T\lambda^k$
  $\mathbf{z} \leftarrow \lambda^k - \mathbf{R}(\mathbf{J}\mathbf{w} + \mathbf{b})$
  for $i \in K$ in parallel computing **do**
    $\mathbf{I} \equiv$ indices of block ni, fi
    $\lambda_{ni}^{k+1} \leftarrow \text{prox}_{ni}(\mathbf{z}_{ni})$
    $\lambda_{fi}^{k+1} \leftarrow \text{prox}_{\mathcal{F}_i(\lambda_{ni}^{k+1})}(\mathbf{z}_{fi})$
  end for
  $\epsilon^{k+1} = \|\lambda^{k+1} - \lambda^k\|_\infty$
  if $\epsilon^{k+1} > \epsilon^k$ **then**
    $\mathbf{R} \leftarrow v\mathbf{R}$
  else
    $(\lambda^k, \epsilon^k, k) \leftarrow (\lambda^{k+1}, \epsilon_{k+1}, k+1)$
  end if
end while

3.2.3 Parallel Gauss-Seidel Scheme Algorithm. In the case of the Gauss-Seidel method, non-parallelism inherent challenges arise when aiming for straightforward parallelization. To overcome these challenges, we employ a strategy that involves decomposing the computational domain into smaller, independent subdomains. To achieve this, we rely on the Contact Jacobian matrix to construct what we call a "contact graph."

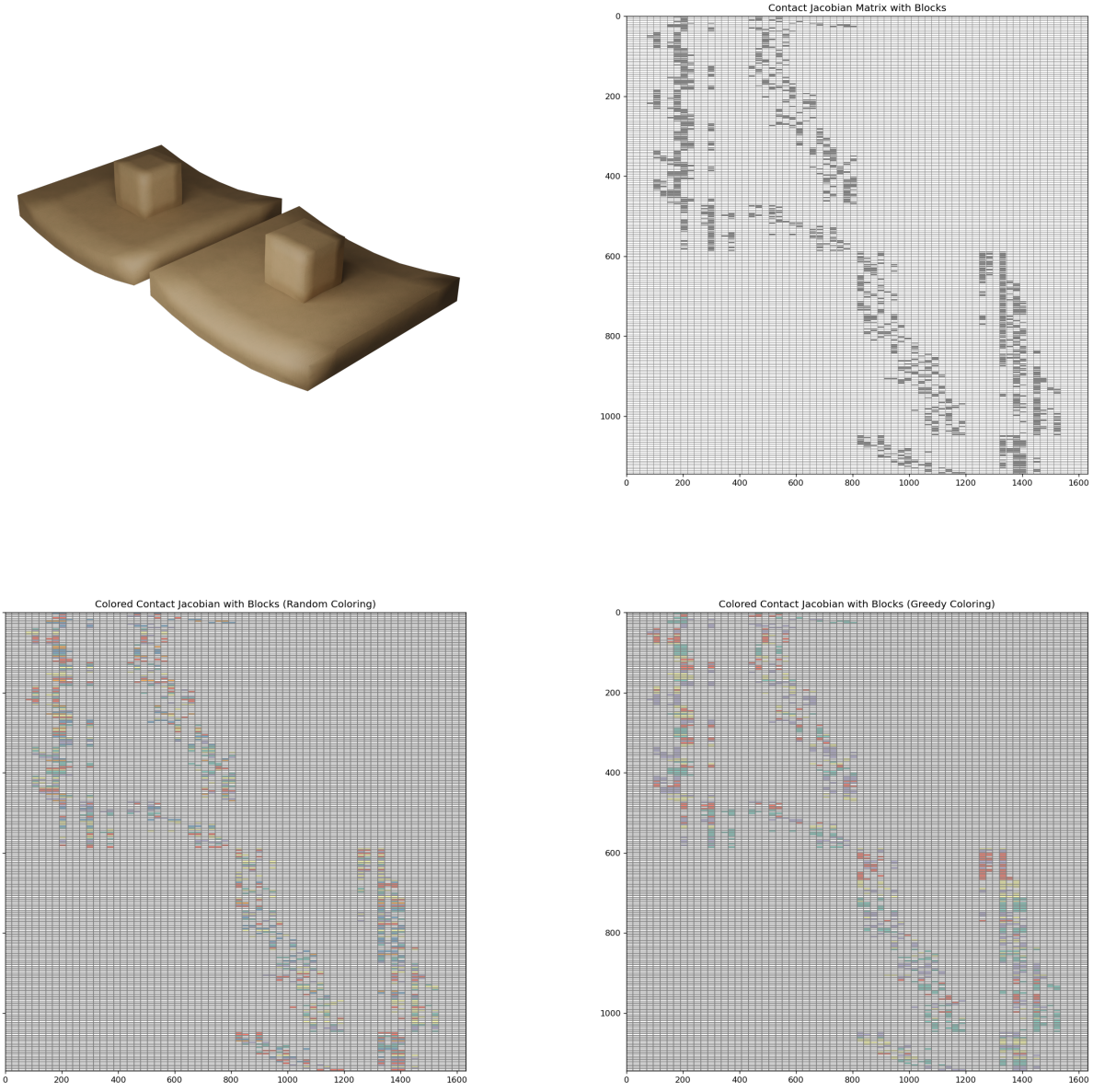


Figure 19: In the depicted scene, two soft cubes rest atop two soft boxes, generating 286 contact points. While the random graph coloring method grouped these into 125 distinct categories, the greedy graph coloring method was more efficient, categorizing them into just 95 groups. Progressing from top to bottom and left to right: the first image displays the simulation outcome of the scene; the second shows the original contact Jacobi matrix; the third reveals the outcome using the random graph coloring method; and the fourth illustrates the results from the greedy graph coloring approach.

Constructing the Contact Graph: The process of creating the contact graph is based on [Lazarevych et al. 2010], the idea is to treat each contact point as an individual vertex within the graph. The method begins by scanning the entire Contact Jacobian matrix. Whenever two contact points share a common mass point, we establish an edge between them in the graph. The constructing contact graph algorithm is shown in the Algorithm 6.

Graph Coloring: With the contact graph in place, the next step is to partition it into distinct, independent subgraphs. Several

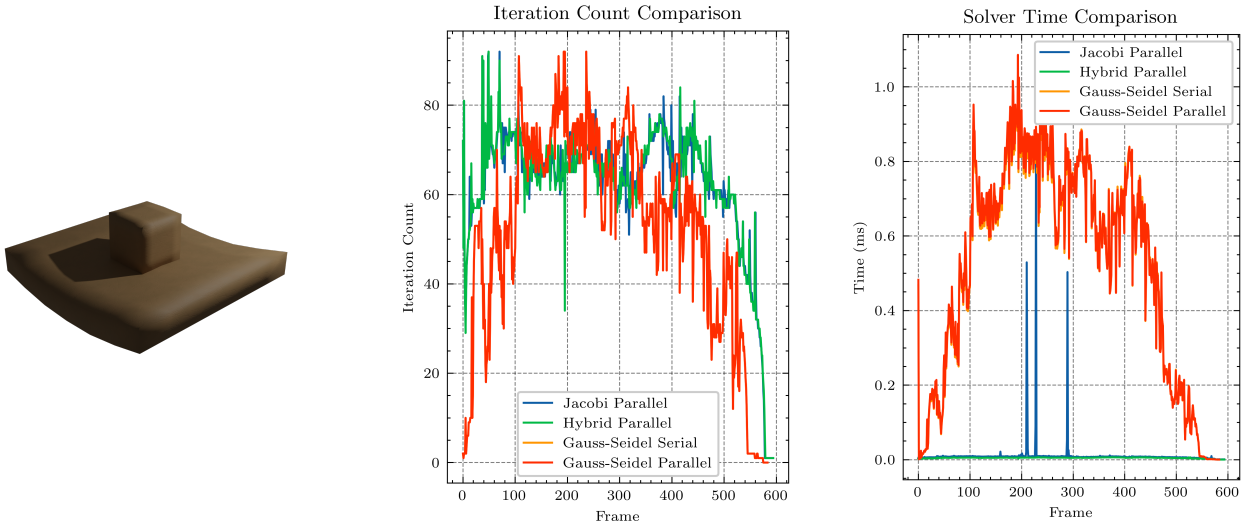


Figure 20: In a basic scene, a soft cube is placed on top of a soft box, leading to 198 contact points. Moving from left to right: the first image depicts the simulated scene; the next presents a comparison of iteration counts across frames; while the last highlights the solver time results. The reason analysis of time spikes in the parallel Jacobi algorithm is shown in Section 3.2.4.

Algorithm 6 Constructing the Contact Graph

Require: **J**: the contact Jacobi matrix
Contact points count K = rows of **J**
Initialize a empty graph **G**
for $k \in range(K)$ **do**
    G.add_vertex(k)
end for
for *column* ∈ columns of **J** **do**
    contacts ← Find every non-zero contact point in this *column*
    for $c_1 \in contacs$ **do**
        for $c_2 \in contacs$ **do**
            if $c_1 != c_2$ **then**
                G.add_edge(c1, c2)
            end if
        end for
    end for
end for
Return **G**

approaches can be utilized for this purpose, but we opt for a graph coloring strategy. Our implementation features two methods of this approach: random graph coloring and greedy graph coloring algorithms.

- Random Graph Coloring: This technique has proven effective in soft body simulation [Fratarcangeli et al. 2016]; however, our experimentation within RAINBOW has revealed certain limitations, including instability and inconsistency in its performance. As such, we do not show the details of this algorithm in this report.
- Greedy Graph Coloring: In contrast, the greedy graph coloring algorithm has emerged as the more suitable and effective choice for RAINBOW. It offers greater stability and consistently outperforms random graph coloring. In our evaluations, for example, random graph coloring produced 125 distinct groups for a scenario with 286 contact points, while the greedy graph coloring approach yielded a more efficient division into just 95 groups, which is shown in Figure 19.

Parallel Computing of Contact Forces: Once the various coloring groups are established, we can systematically traverse these groups in sequence. Within each group, the computation of contact forces for individual contact points can occur in parallel. We propose this parallel Gauss-Seidel algorithm which is shown in Algorithm 7

3.2.4 Experiment Result. We initiated our evaluation with a simplistic scenario: a cube placed atop a box, yielding a total of 198 contact points, which is shown in Figure 20. Several key observations were found from our experiment as follows.

When comparing the number of iterations per frame, the Gauss-Seidel scheme consistently required fewer iterations compared to the Jacobi scheme. For solver time, the hybrid parallel algorithm was considerably superior in terms of computational efficiency. Its solver time for calculating contact forces was nearly negligible for every frame. However, it's worth noting that in this simplistic scenario, the Gauss-Seidel parallel did not exhibit a significant advantage over its serial counterpart. This observation can be attributed to the overhead introduced by constructing the contact graph and performing graph coloring, both of which added to the total solver time in parallel Gauss-Seidel scheme algorithm. Besides, time spikes when using the parallel Jacobi algorithm could stem from occasional resource contention where other processes demand computational resources, leading to a temporary slowdown. Another possibility is memory bottlenecks, where the system might momentarily tap into slower disk-based memory. Lastly, inherent thread synchronization

Algorithm 7 The Parallel PROX Gauss-Seidel Solver

Require: B indices of all bodies, **J**, **M**, **b**, **R**, $\lambda^{\mathbf{0}}$, v
1: $(k, \lambda^k, \epsilon^k) \leftarrow (0, \lambda^0, \infty)$
2: Constructing the Contact graph G by J
3: Creating color groups C by graph coloring algorithm
4: **while** not converged **do**
5: $\mathbf{w} \leftarrow \mathbf{M}^{-}1\mathbf{J}^T\lambda^k$
6: **for** $c \in \mathbf{C}$ **do**
7: Initialize an empty list Δw_s
8: **for** $i \in c$ **do** in parallel computing
9: $\mathbf{I} \equiv$ indices of block ni, fi
10: $\mathbf{z}_I \leftarrow \lambda_I^k - \mathbf{R}_i(\mathbf{J}_{IB}\mathbf{w} + \mathbf{b}_I)$
11: $\lambda_{ni}^{k+1} \leftarrow \text{prox}_{ni}(\mathbf{z}_{ni})$
12: $\lambda_{fi}^{k+1} \leftarrow \text{prox}_{\mathcal{F}_i(\lambda_{ni}^{k+1})}(\mathbf{z}_{fi})$
13: Append $\left(\mathbf{M}^{-1}\mathbf{J}^T\right)_{B,I}\left(\lambda_I^{k+1} - \lambda_I^k\right)$ to Δw_s
14: **end for**
15: $\mathbf{w} \leftarrow \mathbf{w} + \frac{sum(\Delta w_s)}{len(\Delta w_s)}$
16: **end for**
17: $\epsilon^{k+1} = \|\lambda^{k+1} - \lambda^k\|_\infty$
18: **if** $\epsilon^{k+1} > \epsilon^k$ **then**
19: $\mathbf{R} \leftarrow v\mathbf{R}$
20: **else**
21: $(\lambda^k, \epsilon^k, k) \leftarrow (\lambda^{k+1}, \epsilon_{k+1}, k+1)$
22: **end if**
23: **end while**

within the algorithm can introduce brief delays, especially if threads await shared resources.

As a result, we hypothesize that as the complexity of the scenario grows and the number of contact points increases, the Gauss-Seidel parallel algorithm would surpass the performance of the Gauss-Seidel serial. Conversely, for scenarios with minimal complexity and fewer contact points, the serial variant of the Gauss-Seidel algorithm is adequately efficient. In light of these results, while the Gauss-Seidel parallel shows potential for complex environments, its utility in simpler contexts may be limited due to the overheads associated with graph operations.

3.3 Fully Implicit Time Stepping

In the RAINBOW software, a semi-implicit time-stepping approach is employed. More specifically, the velocity is updated explicitly, and the position of the vertices is implicitly evolved. This method is mathematically depicted by the following equations:

$$\begin{aligned} \mathbf{v}^{t+1} &= \mathbf{v}^{\mathbf{t}} + \frac{\mathbf{F}^{\mathbf{t}}}{m}\Delta t \\ \mathbf{x}^{t+1} &= \mathbf{v}^{\mathbf{t}} + \mathbf{v}^{\mathbf{t+1}}\Delta t \end{aligned} \tag{27}$$

This method, while being more efficient and easy to implement compared to the fully implicit approach, can sometimes lead to instability in the results. Let's delve into the Courant-Friedrichs-Lewy (CFL) condition to understand this better.

Analyzing the CFL Condition. For clarity, consider an object divided into numerous triangles in 2D. Focusing on a single vertex, P_0 , of one such triangle, which is shown Figure 21, imagine a traction force pushing P_0 from top to bottom. The objective is to ensure that this vertex does not exceed a threshold, such as crossing the bottom edge. This constraint can be represented mathematically as:

$$\left|\mathbf{X}^{\mathbf{t+1}} - X^t\right| < h$$

From the second equation of the semi-implicit for position update:

$$\left|\mathbf{v}^{\mathbf{t+1}}\Delta t\right| < h$$

Because of the time is positive:

$$\left|\mathbf{v}^{\mathbf{t+1}}\right|\Delta t < h$$

From the first equation of the semi-implicit method to instead of the $\mathbf{v}^{\mathbf{t+1}}$, we have:

$$\left|\mathbf{v}^{\mathbf{t}} + \frac{\mathbf{F}^{\mathbf{t}}}{m}\Delta t\right|\Delta t < h$$

We know that **F** is proportional to Young's Modulus E. So we can let the left and right sides time a constant c and use the E instead of **F**, we have:

$$\Delta t\left|\mathbf{V}^{\mathbf{t}} + \frac{\Delta t}{m}E\right| < hc$$

Expanding further:

$$\Delta t\mathbf{V}_{\mathbf{max}} + \frac{\Delta t^2}{m}E < hc$$

From this, the time step constraint becomes:

$$\Delta t < c\sqrt{\frac{hm}{E}}$$

This final expression outlines the CFL condition for our semi-implicit method. As per this condition, Therefore, as Young's Modulus E increases, indicating stiffer materials, there's a corresponding need to decrease the time step Δt to ensure stability.

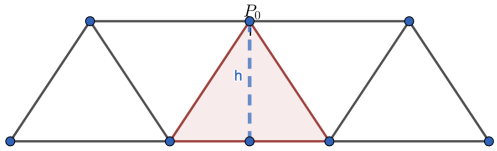


Figure 21: Illustration of a 2D object divided into triangles, and the blue triangle highlights the vertex P_0, positioned h units away from the bottom edge. To prevent the triangle from collapsing when a force is applied to P_0 in the top-to-bottom direction, it is necessary to ensure that $\|X_{t+1} - X_t\| < h$.

For enhanced stability in soft body dynamics simulations, especially when desiring larger time steps, the implicit time integration approach is often favored, and the implicit formulation can be describe as:

$$\begin{aligned} \mathbf{v}^{t+1} &= \mathbf{v}^{\mathbf{t}} + \mathbf{M}^{-1}\mathbf{f}^{\mathbf{t+1}}\Delta t \\ \mathbf{x}^{t+1} &= \mathbf{v}^{\mathbf{t}} + \mathbf{v}^{\mathbf{t+1}}\Delta t \end{aligned} \tag{28}$$

here the term $\mathbf{f}^{\mathbf{t+1}}$ is dependent on the state $\mathbf{x}^{t+1}$, mathematically,

$$\mathbf{f}^{\mathbf{t+1}} = \mathbf{f}(\mathbf{x}^{t+1}, t + \Delta t)$$

The distinction between the implicit and semi-implicit 27 formulations is primarily the evaluation of forces at $t+1$ in the implicit method.

Analysis of implicit formulation, we can find that we do know the system state at $t+1$ (means we do know $\mathbf{x}^{t+1}$ and $\mathbf{v}^{t+1}$), therefore we can not directly compute the $\mathbf{f}^{t+1}$. Generally, the problem at hand is non-linear. While Newton's method is typically used to find a solution, we can opt for a simplified approach using a single iteration of Newton's method [Baraff and Witkin 1998]. In this method, the system is linearized around the current state, and the linearized problem is solved. Furthermore, efficient solution techniques like the Conjugate Gradient method, a Krylov subspace method, can be employed. The steps to linearize the system are detailed below:

Firstly, the original non-linear system is constructed in matrix form as:

$$\begin{bmatrix} \mathbf{M}\mathbf{v}^{t+1} - \mathbf{M}\mathbf{v}^{t} - \mathbf{F}^{t+1}\Delta t \\ \mathbf{x}^{t+1} - \mathbf{v}^{t+1} - \mathbf{v}^{t+1}\Delta t \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix} \tag{29}$$

where

$$\mathbf{f}^{t+1} = \mathbf{f}_d^{t+1} + \mathbf{f}_e^{t+1} + \mathbf{f}_c^{t+1} + \mathbf{f}_{ext} \tag{30}$$

Here, $\mathbf{f}_d$ represents the damping force, $\mathbf{f}_e$ the elastic force, $\mathbf{f}c$ the contact force, and $\mathbf{f}ext$ the external force. Inserting the force equation into Equation 29, we obtain:

$$\begin{bmatrix} \mathbf{M}\mathbf{v}^{t+1} - \mathbf{M}\mathbf{v}^{t} - \mathbf{f}_d^{t+1}\Delta t - \mathbf{f}_e^{t+1}\Delta t - \mathbf{f}_c^{t+1}\Delta t - \mathbf{f}_{ext}\Delta t \\ \mathbf{x}^{t+1} - \mathbf{v}^{t+1} - \mathbf{v}^{t+1}\Delta t \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix} \tag{31}$$

Assume k is the k^{th} iteration step, then the update equation as follows:

$$\begin{aligned} \mathbf{x}^{k+1} &\leftarrow \mathbf{x}^{k} + \Delta\mathbf{x} \\ \mathbf{v}^{k+1} &\leftarrow \mathbf{v}^{k} + \Delta\mathbf{v} \end{aligned} \tag{32}$$

To approximate the terms in Equation 31, the first-order Taylor series expansion is used. By taking the differential, we obtain a set of relations for the changes in $\mathbf{v}$, $\mathbf{f}_d$, $\mathbf{f}_e$, and $\mathbf{f}_c$ with respect to their respective matrices.

$$\begin{aligned} \mathbf{M}\mathbf{v}^{k+1} &\approx \mathbf{M}\mathbf{v}^{k} + \mathbf{M}\Delta\mathbf{v} \\ \mathbf{f}_d^{k+1} &\approx \mathbf{f}_d^{k} + \mathbf{D}^{k}\Delta\mathbf{v} \\ \mathbf{f}_e^{k+1} &\approx \mathbf{f}_e^{k} + \mathbf{K}^{k}\Delta\mathbf{x} \\ \mathbf{f}_c^{k+1} &\approx \mathbf{f}_c^{k} - \mathbf{H}_v^{k}\Delta\mathbf{v} - \mathbf{H}_x^{k}\Delta\mathbf{x} \end{aligned} \tag{33}$$

here the $\mathbf{D}$ and $\mathbf{K}$ are the damping matrix and the stiffness matrix. Rewrite the Equation 31, we have:

$$\begin{bmatrix} (\mathbf{M} - \Delta t\mathbf{B}^{k} + \Delta t\mathbf{H}_v^{K})\Delta\mathbf{v} - \Delta t\Delta\mathbf{x}(\mathbf{H}_x^{k} - \mathbf{K}^{k}) \\ -\Delta t\Delta\mathbf{v} + \Delta\mathbf{x} \end{bmatrix} = \begin{bmatrix} \mathbf{M}(\mathbf{v}^{t} - \mathbf{v}^{k}) + \Delta t\mathbf{f} \\ \mathbf{x}^{t} - \mathbf{x}k + \Delta\mathbf{v}^{k} \end{bmatrix} \tag{34}$$

where $\mathbf{f} = \mathbf{f}_d^{k} + \mathbf{f}_e^{k} + \mathbf{f}_c^{k} + \mathbf{f}_{ext}$.

After linearization and simplification using the relation $\Delta\mathbf{x} = \Delta t\Delta\mathbf{v}$, the system can be expressed as:

$$\begin{bmatrix} (\mathbf{M} - \Delta t\mathbf{B}^{k} + \mathbf{H}_v^{K}\Delta t + \mathbf{H}_x^{k}\Delta t^{2} - \mathbf{K}^{k}\Delta t^{2})\Delta\mathbf{v} \\ -\Delta t\Delta\mathbf{v} + \Delta\mathbf{x} \end{bmatrix} = \begin{bmatrix} \mathbf{M}(\mathbf{v}^{t} - \mathbf{v}^{k}) + \Delta t\mathbf{f} \\ \mathbf{x}^{t} - \mathbf{x}k + \Delta\mathbf{v}^{k} \end{bmatrix} \tag{35}$$

Now, Equation 35 is a linear system. However, computing the stiffness matrix $\mathbf{K}$ is not easy, in practical scenarios, as mentioned in [Sifakis and Barbic 2012], it's often more effective to compute force differentials, such as $\Delta\mathbf{f}_e$, directly without explicitly constructing the $\mathbf{K}$ matrix.

To compute the elastic force differential for tetrahedra, $\Delta\mathbf{f}_e$, we consider the deformation gradient, $\mathbf{F}$, which relates the deformed space matrix, $\mathbf{D}_s$, to the material space matrix, $\mathbf{D}_m$. The relationship is established as:

$$\mathbf{D}_s = \begin{bmatrix} \mathbf{x}_1 - \mathbf{x}_0 \mid \mathbf{x}_2 - \mathbf{x}_0 \mid \mathbf{x}_3 - \mathbf{x}_0 \end{bmatrix} \tag{36}$$

$$\mathbf{D}_m = \begin{bmatrix} \overline{\mathbf{x}}_1 - \overline{\mathbf{x}}_0 \mid \overline{\mathbf{x}}_2 - \overline{\mathbf{x}}_0 \mid \overline{\mathbf{x}}_3 - \overline{\mathbf{x}}_0 \end{bmatrix} \tag{37}$$

We know the deformation gradient $\mathbf{F}$:

$$\mathbf{F} = \mathbf{D}_s\mathbf{D}_m^{-1} \tag{38}$$

For a tetrahedron, the elastic force differential $\Delta\mathbf{f}_e$ is defined as:

$$\Delta\mathbf{f}_e = \begin{bmatrix} \Delta\mathbf{f}_{e0} & \Delta\mathbf{f}_{e1} & \Delta\mathbf{f}_{e2} & \Delta\mathbf{f}_{e3} \end{bmatrix} \tag{39}$$

Given the constraints, we can directly compute the values of $\Delta\mathbf{f}_{e0}$, $\Delta\mathbf{f}_{e1}$ and $\Delta\mathbf{f}_{e2}$ first, then the fourth element $\Delta\mathbf{f}_{e3} = -(\Delta\mathbf{f}_{e0} + \Delta\mathbf{f}_{e1} + \Delta\mathbf{f}_{e2})$,

We introduce a matrix $\Delta\mathbf{H}_e$ represent the first three elements:

$$\Delta\mathbf{H}_e = \begin{bmatrix} \Delta\mathbf{f}_{e0} & \Delta\mathbf{f}_{e1} & \Delta\mathbf{f}_{e2} \end{bmatrix} \tag{40}$$

$$\Delta\mathbf{H}_e = -V\Delta\mathbf{P}(\mathbf{F}, \Delta\mathbf{F})\mathbf{D}_m^{-T} \tag{41}$$

where the V is the undeformed volume of the tetrahedron equals $\frac{1}{6}|det\mathbf{D}_m|$, $\mathbf{P}$ is the Piola stress.

Starting with the differential of the deformation gradient, we have:

$$\Delta\mathbf{F} = (\Delta\mathbf{D}_s)\mathbf{D}_m^{-1} \tag{42}$$

The only remaining task is to present a clear formula for $\Delta\mathbf{P}(\mathbf{F}, \Delta\mathbf{F})$, the formulation of this equation will vary based on the specific constitutive model, such as the St. Venant-Kirchhoff or the Neohookean material models. The procedure to compute the force differential can now be outlined in Algorithm 8.

Algorithm 8 Compute the elastic force differential

Require: $\mathcal{M}$: is the tetrahedral mesh
1: $\Delta\mathbf{f} \leftarrow \mathbf{0}$
2: **for** $\mathbf{T}_e \in \mathcal{M}$ **do**
3: Compute $\mathbf{D}_s$
4: Compute $\Delta\mathbf{D}_s$
5: $\mathbf{F} \leftarrow \mathbf{D}_s\mathcal{M}[e]$
6: $\Delta\mathbf{F} \leftarrow (\Delta\mathbf{D}_s)\mathcal{M}[e]$
7: $\Delta P \leftarrow \Delta\mathbf{P}(\mathbf{F}, \Delta\mathbf{F})$
8: $\Delta\mathbf{H}_e \leftarrow -V(\Delta\mathbf{P})\mathcal{M}[e]^{-T}$
9: $\Delta\mathbf{f}[e] \leftarrow \begin{bmatrix} \Delta\mathbf{H}_e[0] & \Delta\mathbf{H}_e[1] & \Delta\mathbf{H}_e[2] & -(\Delta\mathbf{H}_e[0] + \Delta\mathbf{H}_e[1] + \Delta\mathbf{H}_e[2]) \end{bmatrix}$
10: **end for**
11: return $\Delta\mathbf{f}$

4 CONCLUSION

In this project, we gained significant insights into soft body simulation, particularly its collision detection system. For spatial discretization of soft body simulation, the Finite Elements Method (FEM) with tetrahedrons is generally employed to discretize the mesh. In terms of temporal discretization, we utilized semi-implicit integration in the RAINBOW software. Our analysis of its Courant–Friedrichs–Lewy (CFL) condition in Section 3.3 indicated instability with larger time steps, prompting us to explore the implicit integration method for soft body simulation. Instead of directly computing the stiffness matrix $\mathbf{K}$, we learned to determine the elastic force differential $\Delta\mathbf{f}_e$.An efficient numerical method for computing the stiffness matrix $\mathbf{K}$ will be our focus in the upcoming block 2 project.

Regarding the collision detection system, our journey included understanding the constraint-based method in RAINBOW software and enhancing its performance. We delved into its narrow phase, contact point generation, and contact point reduction mechanisms. The narrow phase employs chunked-BVH to expedite overlap detection of surface triangle pairs with SDF. Our implementation of the Spatial Hashing method aimed to accelerate the narrow phase further, detailed in Section 3.1.4. During the contact generation phase, we familiarized ourselves with the Frank-Wolfe algorithm. Its compatibility with GPU-based computations led us to implement it on GPUs using the Numba package, as described in Section 3.1.2. In the contact point reduction phase, we transitioned from an $O(n^2)$ algorithm to a more efficient $\Theta(n)$ one, with the specifics in Section 3.1.3.

Additionally, we tackled the computation of contact force using the Proximal optimization algorithm. Beginning with its foundational mathematical theory in Section 2.4, we subsequently explored the proximal Gauss-Seidel and Jacobi scheme algorithms for determining normal and friction forces. We enhanced these algorithms by adopting parallel versions: parallel proximal Gauss-Seidel, parallel proximal Jacobi, and the parallel proximal hybrid scheme, with a comprehensive overview in Section 3.2.

Moreover, recognizing that intricate soft body simulations can be time-consuming, we integrated the Universal Scene Description (USD) functionality. This allows for saving the entire simulation process into a USD file. By executing complex simulations on remote machines and then downloading the USD files to local systems, we can conveniently visualize the simulations using tools like Blender or usdview. An example of such a complex simulation was presented earlier in this report.

Looking ahead, our future endeavors will center on a deeper exploration of soft body simulation in the upcoming block 2 project and our thesis. In the block 2 project, we'll investigate soft body simulations using differentiable physical simulations and implicit integration methods. Our thesis work will likely build upon this project and the block 2 initiative, delving deeper into aspects like hybrid collision detection systems (blending constraint-based and penalty-based approaches), the application of deep learning in simulations as highlighted in [Du 2023], and more intricate friction models as mentioned in [Andrews et al. 2022]. Our ultimate goal is to create a simulator that's both precise and efficient.

REFERENCES

Sheldon Andrews, Kenny Erleben, and Zachary Ferguson. 2022. Contact and Friction Simulation for Computer Graphics. In *ACM SIGGRAPH 2022 Courses* (Hybrid Event, Vancouver, Canada) *(SIGGRAPH '22)*. Association for Computing Machinery, New York, NY, USA, Article 2, 124 pages.

David Baraff and Andrew Witkin. 1998. *Large Steps in Cloth Simulation* (1 ed.). Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3596711.3596792

Tao Du. 2023. Deep Learning for Physics Simulation. In *ACM SIGGRAPH 2023 Courses* (Los Angeles, California) *(SIGGRAPH '23)*. Association for Computing Machinery, New York, NY, USA, Article 6, 25 pages. https://doi.org/10.1145/3587423.3595518

Kenny Erleben. 2018. Methodology for Assessing Mesh-Based Contact Point Methods. *ACM Trans. Graph.* 37, 3, Article 39 (jul 2018), 30 pages. https://doi.org/10.1145/3096239

Kenny Erleben. 2022. RAINBOW. https://github.com/diku-dk/RAINBOW. GitHub repository.

Marco Fratarcangeli, Valentina Tibaldo, and Fabio Pellacini. 2016. Vivace: A Practical Gauss-Seidel Method for Stable Soft Body Dynamics. *ACM Trans. Graph.* 35, 6, Article 214 (dec 2016), 9 pages. https://doi.org/10.1145/2980179.2982437

Olexiy Lazarevych, Gabor Szekely, and Matthias Harders. 2010. Decomposing the Contact Linear Complementarity Problem into Separate Contact Regions. *Computer Vision Laboratory, ETH Zurich* (2010).

Sylvain Lefebvre and Hugues Hoppe. 2006. Perfect Spatial Hashing. *ACM Trans. Graph.* 25, 3 (jul 2006), 579–588. https://doi.org/10.1145/1141911.1141926

Miles Macklin, Kenny Erleben, Matthias Müller, Nuttapong Chentanez, Stefan Jeschke, and Zach Corse. 2020. Local Optimization for Robust Signed Distance Field Collision. *Proc. ACM Comput. Graph. Interact. Tech.* 3, 1, Article 8 (may 2020), 17 pages. https://doi.org/10.1145/3384538

Numba. 2022. Numba: A Just-In-Time Compiler for Numerical Functions in Python. https://numba.pydata.org/numba-doc/latest/cuda/index.html Accessed: 2023.

Numba-Development-Team. 2022. Numba: A Just-In-Time Compiler for Numerically-focused Python. https://numba.pydata.org/

Neal Parikh and Stephen Boyd. 2014. Proximal Algorithms. *Found. Trends Optim.* 1, 3 (Jan 2014), 127–239.

Robert Schmidtke and Kenny Erleben. 2018. Chunked Bounding Volume Hierarchies for fast digital prototyping using volumetric meshes. *I E E E Transactions on Visualization and Computer Graphics* 24, 12 (2018), 3044–3057. https://doi.org/10.1109/TVCG.2017.2784441

Eftychios Sifakis and Jernej Barbic. 2012. FEM Simulation of 3D Deformable Solids: A Practitioner's Guide to Theory, Discretization and Model Reduction. In *ACM SIGGRAPH 2012 Courses* (Los Angeles, California) *(SIGGRAPH '12)*. Association for Computing Machinery, New York, NY, USA, Article 20, 50 pages. https://doi.org/10.1145/2343483.2343501

Matthias Teschner, Bruno Heidelberger, Matthias Müller, Danat Pomerantes, and Markus H. Gross. 2003. Optimized Spatial Hashing for Collision Detection of Deformable Objects. In *International Symposium on Vision, Modeling, and Visualization*. https://api.semanticscholar.org/CorpusID:12035329

Wikipedia contributors. 2023a. Bounding volume — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Bounding_volume&oldid=1149445876 [Online; accessed 14-September-2023].

Wikipedia contributors. 2023b. Bounding volume hierarchy — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Bounding_volume_hierarchy&oldid=1147718410 [Online; accessed 14-September-2023].

Wikipedia contributors. 2023c. Möller–Trumbore intersection algorithm — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=M%C3%B6ller%E2%80%93Trumbore_intersection_algorithm&oldid=1155554391 [Online; accessed 17-October-2023].