ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ І. І. МЕЧНИКОВА
(повне найменування закладу вищої освіти)

Факультет математики, фізики та інформаційних технологій
(повне найменування факультету)

Кафедра комп'ютерних систем та технологій
(повна назва кафедри)

Кваліфікаційна робота

на здобуття ступеня вищої освіти «бакалавр»
(рівень вищої освіти)

«Розробка імітаційної моделі для інтегрованого моделювання взаємодії фізичних тіл»

(тема кваліфікаційної роботи українською мовою)

«Development of a Simulation Model for Integrated Modeling of the Interaction of Physical Bodies»

(тема кваліфікаційної роботи англійською мовою)

Виконав: здобувач денної форми навчання
спеціальності 122 – Комп'ютерні науки
(шифр і назва напряму підготовки, спеціальності)

Освітня програма «Комп'ютерні науки»
(назва освітньої програми)

Ляшенко Святослав Геннадійович
(прізвище, ім'я, по-батькові)

Керівник д.т.н., професор Приходько С. Б.
(науковий ступінь, вчене звання, прізвище та ініціали, підпис)

Рецензент ст. викл. Шаріпова І. В.
(науковий ступінь, вчене звання, прізвище та ініціали)

Рекомендовано до захисту:
Протокол засідання кафедри
комп'ютерних систем та технологій
№ ____ від ___ . _____ . 20__ р.

Завідувач кафедри
_____ Юрій ГУНЧЕНКО
(підпис) (ім'я, прізвище)

Захищено на засіданні ЕК № 5
протокол № __ від ___ . ____ . 20__ р.

Оцінка _____ / _____ /
(за національною шкалою, шкалою ECTS, бали)

Голова ЕК
_____ Микола МАЛАКСІАНО
(підпис) (ім'я, прізвище)

Одеса – 2025

АНОТАЦІЯ

Кваліфікаційна робота присвячена розробці імітаційної моделі для інтегрованого моделювання взаємодії фізичних тіл у реальному часі. Актуальність теми зумовлена потребами комп'ютерної графіки, ігрової індустрії та віртуальної реальності в реалістичних симуляціях. Традиційні методи, як-от FEM, є точними, але не придатними для реального часу, тоді як XPBD і PBF забезпечують оптимальний баланс продуктивності та стабільності.

Мета роботи – створення імітаційної моделі для моделювання твердих тіл, м'яких тіл і рідин. Теоретичний розділ включає огляд методів моделювання та обґрунтування вибору XPBD і PBF. Розглянуто алгоритми XPBD для твердих тіл (з позиційними та кутовими обмеженнями) і м'яких тіл (з відстаневими та об'ємними корекціями), а також PBF для рідин (з корекцією щільності). Інтегрована імітаційна модель об'єднує ці алгоритми для взаємодії тіл.

Тестування проведено на чотирьох сценах: тканина з торусом (реалістична деформація при жорсткості 100), шарніри (механічні рухи), ланцюг з кроликом (провисання) і рідина (стабілізація при в'язкості 0.02). Результати підтвердили стабільність моделі, хоча виявлено штучну податливість XPBD при низькій кількості ітерацій і відсутність повної інтеграції рідин з іншими типами тіл.

Робота має практичне значення для ігрових рушіїв і віртуальної реальності.

Обсяг роботи становить 100 сторінку, вона включає 34 рисунки, 1 таблицю, 21 джерело.

ABSTRACT

The qualification work is devoted to the development of a simulation model for integrated computer modeling of the interaction of physical bodies in real time. The relevance of the topic is due to the needs of computer graphics, the gaming industry and virtual reality in realistic simulations. Traditional methods, such as FEM, are accurate, but not suitable for real time, while XPBD and PBF provide an optimal balance of performance and stability.

The purpose of the work is to create a model for modeling solids, soft bodies and fluids. The theoretical section includes an overview of modeling methods and justification for the choice of XPBD and PBF. The XPBD algorithms for solids (with positional and angular constraints) and soft bodies (with distance and volume corrections), as well as PBF for fluids (with density correction) are considered. The integrated simulation model combines these algorithms for the interaction of bodies.

Testing was conducted on four scenes: fabric with a torus (realistic deformation at stiffness 100), hinges (mechanical movements), chain with a rabbit (sag) and liquid (stabilization at viscosity 0.02). The results confirmed the stability of the model, although artificial compliance of XPBD was detected at a low number of iterations and the lack of full integration of liquids with other types of bodies.

The work has practical significance for game engines and virtual reality.

The work is completed on 100 pages, it includes 34 figures, 1 table, 21 sources.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ І ТЕРМІНІВ

AABB — Axis-Aligned Bounding Box (Осьово-вирівняна обмежувальна коробка)

FEM —Finite Element Method (Метод скінченних елементів)

PBD — Position-Based Dynamics (Динаміка на основі позицій)

PBF — Position-Based Fluids (Моделювання рідин на основі позицій)

SPH — Smoothed Particle Hydrodynamics (Згладжена гідродинаміка частинок)

XPBD — Extended Position-Based Dynamics (Розширена позиційна динаміка)

ВСТУП

Актуальність теми. Моделювання фізичних взаємодій за допомогою ЕОМ відіграє ключову роль у багатьох сферах науки і техніки, зокрема в комп'ютерній графіці, анімації, ігровій індустрії та інженерному моделюванні.

Використання комп'ютерних симуляцій фізичних процесів дозволяє відтворювати реальні процеси, аналізувати їхній перебіг та прогнозувати наслідки взаємодій між об'єктами різної природи, включаючи тверді тіла, м'які матеріали, гази та рідини.

На сьогодні існують два основні підходи до моделювання фізичних процесів: так звані *офлайн (offline) симуляції* та *симуляції в реальному часі (real-time).* Офлайн-методи зазвичай використовуються у кінематографі для деталізованих сцен та інженерних розрахунках, де головним критерієм є точність, а час обчислень не є критичною змінною. Натомість, у інтерактивних системах, таких як комп'ютерні ігри, віртуальна та доповнена реальність, а також системи управління робототехнікою, необхідно забезпечити високу швидкість обчислень (інтерактивність) при достатньому рівні фізичної достовірності.

У цьому контексті особливого значення набуває використання методів моделювання, орієнтованих на реальний час, серед яких одним із ефективних підходів є XPBD (eXtended Position-Based Dynamics), запропонований Мюллером та ін. [1, 2]. XPBD дозволяє стабільно та ефективно моделювати широкий спектр фізичних явищ, зокрема взаємодію твердих та м'яких тіл, їхні деформації, навіть взаємодію частинок у рідинах та газах за допомогою такого розширення як PBF (Position Based Fluids) [3]. Завдяки системі обмежень цей метод може забезпечувати як жорсткі зв'язки між точками тіла, так і часткову деформацію, яка дозволяє моделювати пружні властивості матеріалів та речовин.

Мета роботи. Розробка імітаційної моделі на основі методів XPBD та PBF для інтегрованого моделювання взаємодії різних типів фізичних тіл (тверді тіла, м’які тіла, тканина, частинкові системи тощо) в реальному часі.

Завдання роботи. Для досягнення поставленої мети необхідно вирішити наступні задачі:

- Дослідити сучасні підходи до комп’ютерного моделювання фізичних процесів, зокрема метод XPBD та його розширення PBF, для симуляції різних типів фізичних тіл.
- Розробити архітектуру імітаційної моделі інтегрованого моделювання, що підтримує взаємодію твердих тіл, м’яких тіл і рідин.
- Реалізувати алгоритми XPBD для симуляції твердих і м’яких тіл із урахуванням обчислювальної ефективності та стабільності в реальному часі.
- Реалізувати алгоритм PBF для моделювання рідин із забезпеченням коректної обробки щільності та зіткнень.
- Провести тестування розробленої моделі на різних сценаріях взаємодії фізичних тіл та оцінити її функціональність.

Об’єкт роботи – процес розробки імітаційної моделі для інтегрованого моделювання взаємодії фізичних тіл у реальному часі, що охоплює тверді тіла, м’які тіла та рідини.

Розроблена імітаційна модель спрямована на забезпечення реалістичної симуляції фізичних взаємодій у реальному часі, що робить її цінним інструментом для ігрових рушіїв та систем віртуальної реальності. Результати роботи можуть бути використані для створення інтерактивних симуляцій, ігрових рушіїв та інших програмних систем, що потребують реалістичної фізики в реальному часі.

1 ТЕОРЕТИЧНІ ОСНОВИ ІМІТАЦІЙНОГО МОДЕЛЮВАННЯ ВЗАЄМОДІЇ ФІЗИЧНИХ ТІЛ

1.1 Загальний опис моделювання фізичних процесів

Комп’ютерне моделювання фізичних процесів є ключовою складовою сучасних комп’ютерних симуляцій, що дозволяє відтворювати поведінку об’єктів у віртуальному середовищі з урахуванням законів фізики. Воно широко застосовується в комп’ютерних іграх, анімації, віртуальній реальності та інженерних розрахунках [4]. Основна мета такого моделювання – створення реалістичних сцен, де об’єкти рухаються, стикаються та деформуються подібно до реального світу.

У моделюванні фізичних процесів розрізняють кілька типів тіл:

- **Тверді тіла**: об’єкти, що не деформуються під дією сил, наприклад, куби, сфери або складні форми в іграх.
- **М’які тіла**: деформівні об’єкти, такі як тканина, гума чи плоть, що змінюють форму під впливом сил.
- **Рідини**: рідкі або газоподібні середовища, що течуть і взаємодіють з іншими об’єктами, для симуляції яких часто використовують методи на основі частинок, як-от PBF або Smoothed Particle Hydrodynamics (SPH) та його вдосконалення [5].

Взаємодія між цими тілами є важливою, наприклад, деформація тканини під впливом твердого об’єкта або рух рідини навколо перешкод (рис. 1.1).

Основні концепції моделювання фізичних процесів включають:

- **Частинки**: базові одиниці з масою, позицією, швидкістю та іншими властивостями.
- **Обмеження**: правила, що обмежують рух частинок, наприклад, підтримка відстаней або форм.
- **Сили**: зовнішні впливи, як-от гравітація, тертя чи взаємодія з користувачем.

- **Методи інтегрування**: числові техніки для оновлення позицій і швидкостей частинок, такі як метод Ейлера або інтегрування Верле [6].

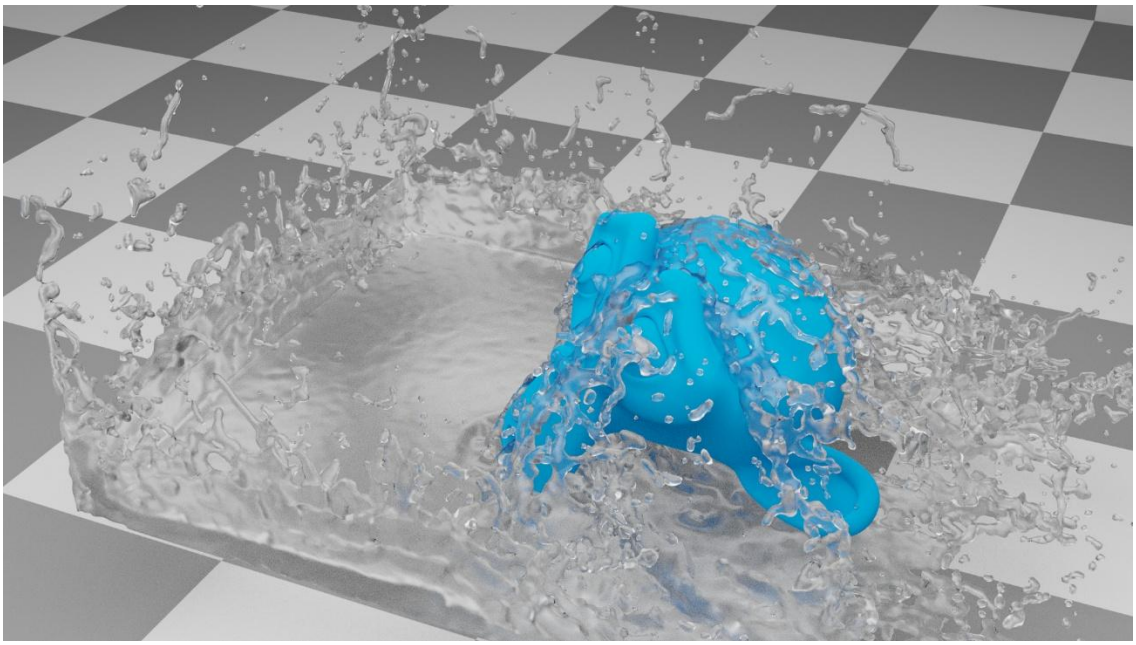

Рисунок 1.1 – Вода, що зтикається з головою мавпи

1.2 Огляд існуючих підходів до моделювання фізичних процесів

У комп’ютерній графіці існує кілька основних підходів до моделювання фізичних процесів, кожен з яких має свої сильні та слабкі сторони [7]:

- **Методи на основі сил**: базуються на другому законі Ньютона ($F = ma$), де сили (внутрішні та зовнішні) обчислюються для визначення прискорень, які потім інтегруються для оновлення швидкостей і позицій. Приклади включають:
  - **метод скінченних елементів (FEM)**: числовий метод для розв’язання диференціальних рівнянь, що використовується для деформівних об’єктів. Він точний, але обчислювально складний і не підходить для реального часу [10].
  - **масо-пружинні системи**: об’єкти моделюються як набір точкових мас, з’єднаних пружинами. Цей метод простий, але має

проблеми з моделюванням об’ємних ефектів, таких як збереження об’єму.

– **методи скінченних різниць і скінченних об’ємів:** використовуються для розв’язання диференціальних рівнянь, але повільні для реального часу.

– **метод граничних елементів:** фокусується на поверхневих взаємодіях, зменшуючи обчислювальну складність, але менш універсальний.

- **Імпульсна динаміка:** використовується переважно для твердих тіл, де імпульси застосовуються для обробки зіткнень і обмежень. Цей метод ефективний для простих сценаріїв, але менш гнучкий для деформованих об’єктів.
- **Кінематичні методи (constraint-based):** використовують множники Лагранжа та глобальні обмеження, розв’язуючи системи лінійних рівнянь. Точні для моделювання механізмів, але обчислювально ємні.
- **Динаміка на основі позицій (PBD/XPBD)**: геометричний метод, що коригує позиції об’єктів для виконання обмежень, уникаючи обчислення сил. PBD використовує симплектичний метод Ейлера та ітеративний розв’язувач Гауса-Зейделя, забезпечуючи швидкість і стабільність для реального часу, але з нижчою точністю порівняно з силовими методами [1]. XPBD розширює PBD, вводячи параметр *податливості* α, який дозволяє точно налаштувати жорсткість обмежень незалежно від часового кроку Δt.
- **Інші геометричні методи**: наприклад, *метод відповідності форм* (від англ. *shape matching*) використовується для моделювання еластичних і пластичних деформацій шляхом зіставлення деформованої конфігурації з початковою формою.

Нижче наведено таблицю із коротким порівнянням переваг та недоліків вищенаведених методів (табл. 1).

Таблиця 1. Порівняння методів моделювання фізичних процесів

Метод	**Переваги**	**Недоліки**
Методи на основі сил	Висока точність, наукова достовірність	Висока обчислювальна складність, нестабільність в реальному часі
Імпульсна динаміка	Ефективність для твердих тіл	Обмежена гнучкість для деформацій
Кінематичні методи	Точність для механізмів	Висока обчислювальна складність
PBD/XPBD	Швидкість, стабільність, керованість	Менша точність, залежність від ітерацій

1.3 Обґрунтування використання XPBD

XPBD обрано для цієї роботи завдяки оптимальному балансу між продуктивністю, стабільністю та універсальністю, що ідеально підходить для інтерактивних симуляцій [8].

Переваги XPBD:

- *Незалежність від часового кроку та кількості ітерацій*: на відміну від PBD, де жорсткість обмежень залежить від кроку часу Δt і кількості ітерацій, XPBD забезпечує послідовну поведінку завдяки податливості (*compliance*), що полегшує налаштування параметрів.
- *Покращена точність*: XPBD точно моделює еластичні та диссипативні поведінки, наближаючись до аналітичних розв’язків. Наприклад, у тесті простої пружини XPBD відтворює правильний період коливань незалежно від кількості ітерацій, тоді як PBD цього не робить (див. рис. 2 у [2]).
- *Оцінка сил обмежень*: XPBD надає оцінки сил через множники Лагранжа, що робить його застосовним до ширшого кола застосувань.
- *Простота та стабільність*: XPBD зберігає простоту PBD, додаючи мінімальні обчислювальні витрати.

- *Універсальність*: XPBD підтримує широкий спектр обмежень (відстань, згинання, об’єм) і може бути розширений до складних моделей, таких як FEM для балок і тканин [9].

Недоліки XPBD:

- *Фізична достовірність*: через спрощення фізичних моделей, точність XPBD нижча, ніж у офлайн-методах, як наприклад FEM.
- *Налаштування*: потребує налаштування параметру податливості для оптимальних результатів.
- *Штучна податливість*: при недостатній кількості ітерацій XPBD може проявляти штучну податливість обмежень, що впливає на точність.

Загалом, XPBD забезпечує стабільну симуляцію в реальному часі, що робить його ефективним для моделювання взаємодії фізичних тіл у даній роботі.

1.4 Моделювання твердих тіл за допомогою XPBD

Тверді тіла — це об’єкти, що не деформуються під дією зовнішніх сил, наприклад, куби, сфери чи складні форми, такі як деталі механізмів [10]. В XPBD вони представляються як єдині сутності з позиційними та обертальними станами, включаючи позицію $\mathbf{x}$, швидкість $\mathbf{v}$, орієнтацію $\mathbf{q}$ (кватерніони), кутову швидкість $\boldsymbol{\omega}$ і тензор інерції $\mathbf{I}$ [3]. Параметр податливості α дозволяє контролювати жорсткість обмежень, моделюючи як наближено жорсткі, так і злегка деформовані об’єкти. Цей підхід забезпечує точне моделювання зіткнень, контактів і сполучень, що є важливим для створення реалістичних симуляцій у відеоіграх чи анімаціях [5].

1.4.1 Передбачення стану

Для моделювання руху твердих тіл XPBD застосовує явну інтеграцію, оновлюючи позиції та орієнтації на основі зовнішніх сил $\mathbf{f}_{\text{ext}}$ і моментів τ_{ext}.

Наприклад, лінійна швидкість оновлюється за формулою:

$$\mathbf{v} \leftarrow \mathbf{v} + h\mathbf{f}_{\text{ext}}/m, \tag{1.1}$$

а позиція — як:

$$\mathbf{x} \leftarrow \mathbf{x} + h\mathbf{v}. \tag{1.2}$$

Для обертань кутова швидкість коригується з урахуванням моментів:

$$\boldsymbol{\omega} \leftarrow \boldsymbol{\omega} + h\mathbf{I}^{-1}\left(\tau_{\text{ext}} - \left(\boldsymbol{\omega} \times (\mathbf{I}\boldsymbol{\omega})\right)\right), \tag{1.3}$$

а орієнтація оновлюється через кватерніон:

$$\mathbf{q} \leftarrow \mathbf{q} + h\frac{1}{2}[\omega_x, \omega_y, \omega_z, 0]\mathbf{q}, \tag{1.4}$$

з подальшою нормалізацією $\mathbf{q}$**:**

$$\mathbf{q} \leftarrow \mathbf{q}/|\mathbf{q}|, \tag{1.5}$$

щоб зберегти його одиничну довжину.

1.4.2 Позиційні обмеження

Наступним етапом є розв’язування обмежень за допомогою *нелінійного розв’язувача Гауса-Зейделя*. Для позиційних обмежень, таких як збереження відстані між точками на тілах, обчислюються узагальнені обернені маси:

$$\begin{aligned} w_1 &= \frac{1}{m_1} + (\mathbf{r}_1 \times \mathbf{n})^T \mathbf{I}_1^{-1} (\mathbf{r}_1 \times \mathbf{n}), \\ w_2 &= \frac{1}{m_2} + (\mathbf{r_2} \times \mathbf{n})^T \mathbf{I}_2^{-1} (\mathbf{r_2} \times \mathbf{n}), \end{aligned} \tag{1.6}$$

де $\mathbf{r}_1$, $\mathbf{r_2}$ — вектори від центрів мас до точок обмеження, а $\mathbf{n}$ — нормаль $\Delta\mathbf{x}$. Множник Лагранжа оновлюється так:

$$\Delta\lambda = \frac{-c - \alpha\lambda}{w_1 + w_2 + \alpha}, \quad \lambda \leftarrow \lambda + \Delta\lambda, \tag{1.7}$$

де c — модуль (довжина) $\Delta\mathbf{x}$, а $\alpha = \alpha/h^2$ — параметр податливості, що контролює жорсткість. Потім застосовується імпульс $\mathbf{p} = \Delta\lambda\mathbf{n}$ для корекції позицій і орієнтацій:

$$\mathbf{x}_1 \leftarrow \mathbf{x}_1 + \mathbf{p}/m_1, \quad \mathbf{x}_2 \leftarrow \mathbf{x}_2 - \mathbf{p}/m_2, \tag{1.8}$$

$$\begin{aligned} \mathbf{q}_1 &\leftarrow \mathbf{q}_1 + \frac{1}{2}[\mathbf{I}_1^{-1}(\mathbf{r}_1 \times \mathbf{p}), 0]\mathbf{q}_1, \\ \mathbf{q}_2 &\leftarrow \mathbf{q}_2 - \frac{1}{2}[\mathbf{I}_2^{-1}(\mathbf{r}_2 \times \mathbf{p}), 0]\mathbf{q}_2. \end{aligned} \tag{1.9}$$

Негайне оновлення тіл після обробки кожного обмеження запобігає проблемі перевищення (від англ. *overshooting*) та є однією з причин стійкості (X)PBD. Це призводить до нелінійного проектованого розв'язку Гауса-Зейделя. Після цього, сили, що діють уздовж обмежень можуть бути отримані як:

$$\mathbf{f} = \lambda\mathbf{n}/h^2. \tag{1.10}$$

1.4.3 Кутові обмеження

Для кутових обмежень, таких як вирівнювання орієнтації, використовуються аналогічні формули, але з фокусом на обертання.

Наприклад, узагальнені обернені маси для обертань:

$$w_1 = \mathbf{n}^T\mathbf{I}_1^{-1}\mathbf{n}, \quad w_2 = \mathbf{n}^T\mathbf{I}_2^{-1}\mathbf{n}, \tag{1.11}$$

а корекція множника Лагранжа виконується за схожою формулою:

$$\Delta\lambda = \frac{-\theta - \alpha\lambda}{w_1 + w_2 + \alpha}, \quad \lambda \leftarrow \lambda + \Delta\lambda, \tag{1.12}$$

але на цей раз, корекція впливає тільки на орієнтацію:

$$\begin{aligned} \mathbf{q}_1 &\leftarrow \mathbf{q}_1 + \frac{1}{2}[\mathbf{I}_1^{-1}\mathbf{p}, 0]\mathbf{q}_1, \\ \mathbf{q}_2 &\leftarrow \mathbf{q}_2 - \frac{1}{2}[\mathbf{I}_2^{-1}\mathbf{p}, 0]\mathbf{q}_2. \end{aligned} \tag{1.13}$$

Аналогічно до сил, що діють вздовж обмеження, ми можемо отримати крутний момент за формулою:

$$\boldsymbol{\tau} = \lambda \mathbf{n}/h^2. \tag{1.14}$$

1.4.4 З'єднання (Joints)

З'єднання у XPBD обмежують відносний рух твердих тіл, обробляючи обертальні та поступальні ступені свободи через позиційні корекції.

Обертальні ступені свободи

Для вирівнювання орієнтацій двох тіл обчислюють кутову корекцію:

$$\mathbf{q} = \mathbf{q}_1 \mathbf{q}_2^{-1}, \quad \Delta\mathbf{q}_{fixed} = 2(q_x, q_y, q_z). \tag{1.15}$$

Для шарнірного з'єднання вирівнюють осі a_1 та a_2:

$$\Delta\mathbf{q}_{hinge} = \mathbf{a}_1 \times \mathbf{a}_2. \tag{1.16}$$

Щоб досягти цільового кута α, обертають $\mathbf{b}_1$ навколо $\mathbf{a}_1$, отримуючи $\mathbf{b}_{target}$, і застосовують:

$$\Delta\mathbf{q}_{target} = \mathbf{b}_{target} \times \mathbf{b_2}. \tag{1.17}$$

Жорсткість контролюється параметром податливості α. З обмеженням на цільовий кут ми можемо змоделювати мотор оновленням $\alpha \leftarrow \alpha + \ hv$, де v — цільова швидкість.

Обмеження кутів з'єднань забезпечуються через:

$$\phi = \arcsin\big((\mathbf{n}_1 \times \mathbf{n}_2) \cdot \mathbf{n}\big), \tag{1.18}$$

з коригуванням ϕ до діапазону $[\alpha, \beta]$ для м'яких ($\alpha > 0$) або жорстких ($\alpha = 0$) обмежень.

Поступальні ступені свободи

Позиційні обмеження фіксують точки кріплення:

$$\Delta\mathbf{r} = \mathbf{r_2} - \mathbf{r_1}, \quad \Delta\mathbf{x} = \Delta\mathbf{r}. \tag{1.19}$$

Також можна розширити це представалення, ввівши максимальну відстань $d_{\max}$:

$$\Delta \mathbf{x} = \frac{\Delta \mathbf{r}}{|\Delta \mathbf{r}|}(|\Delta \mathbf{r}| - d_{\max}). \tag{1.20}$$

Для призматичних з'єднань обмеження застосовуються вздовж осей, накопичуючи корекції, якщо зміщення виходить за межі $a_{\min}$ або $a_{\max}$. Цільовий зсув d_{target} задається з податливістю $\alpha = 1/f(|\Delta \mathbf{r}| - d_{\text{target}})$.

1.4.5 Обробка контактів і тертя

Пари потенційних зіткнень виявляються один раз на крок Δt з використанням розширених обмежувальних прямокутників. На кожному підкроці обчислюються нормалі контактів і позиції $\mathbf{r}_1$, $\mathbf{r}_2$. Глибина проникнення:

$$d = (\mathbf{p}_1 - \mathbf{p}_2) \cdot \mathbf{n}, \quad \mathbf{p}_i = \mathbf{x}_i + \mathbf{q}_i \mathbf{r}_i. \tag{1.21}$$

Якщо $d > 0$, застосовують $\Delta \mathbf{x} = d\mathbf{n}$ з $\alpha = 0$.

Статичне тертя забезпечує $\Delta \mathbf{p}_t = 0$, де:

$$\Delta \mathbf{p} = (\mathbf{p}_1 - \overline{\mathbf{p}}_1) - (\mathbf{p}_2 - \overline{\mathbf{p}}_2), \quad \Delta \mathbf{p}_t = \Delta \mathbf{p} - (\Delta \mathbf{p} \cdot \mathbf{n})\mathbf{n}, \tag{1.22}$$

за умови $\lambda_t < \mu_s \lambda_n$, де μ_s — коефіцієнт статичного тертя.

1.4.6 Корегування швидкостей

Після розв'язання позицій оновлюють швидкості для обробки динамічного тертя та відбиття.

Відносні швидкості в точці контакту отримуються так:

$$\begin{gathered} \mathbf{v} = (\mathbf{v}_1 + \boldsymbol{\omega}_1 \times \mathbf{r}_1) - (\mathbf{v}_2 + \boldsymbol{\omega}_2 \times \mathbf{r}_2), \\ v_n = \mathbf{n} \cdot \mathbf{v}, \quad \mathbf{v}_t = \mathbf{v} - \mathbf{n} v_n. \end{gathered} \tag{1.23}$$

Динамічне тертя застосовується через:

$$\Delta \mathbf{v} = -\frac{\mathbf{v}_t}{|\mathbf{v}_t|} \min(h\mu_d |f_n|, |\mathbf{v}_t|), \tag{1.24}$$

де $f_n = \lambda_n / h^2$.

Демпфування з'єднань реалізується через:

$$\begin{aligned} \Delta \mathbf{v} &= (\mathbf{v}_2 - \mathbf{v}_1)\min(\mu_{\text{lin}} h, 1), \\ \Delta \boldsymbol{\omega} &= (\boldsymbol{\omega}_2 - \boldsymbol{\omega}_1)\min(\mu_{\text{ang}} h, 1). \end{aligned} \tag{1.25}$$

Відбиття коригує нормальну швидкість до $-e\overline{v_n}$, де e — коефцієнт реституції:

$$\Delta \mathbf{v} = \mathbf{n}(-v_n + \min(-e\overline{v_n}, 0)), \tag{1.26}$$

з порогом для уникнення тремтіння (від англ. *jittering*).

1.4.7 Візуалізація

Так як тверді тіла в XPBD моделюються суцільно через центр мас, це дозволяє рендерити їх як тривимірні моделі або геометричні примітиви (куби, сфери). У [11] показана візуалізація (рендеринг) об'єктів, таких як автомобіль із шинами чи ланцюг із 100 ланок.

Позиція та орієнтація об'єкта визначаються через: $p_i = x + qr_i,$ де p_i — позиція точки на тілі, r_i — локальний вектор відносно центру мас. Рендеринг також може включати відображення сил $f = \lambda n/h^2$ для аналізу контактів.

1.4.8 Переваги та обмеження

Переваги:

- Стабільність при $\alpha = 0$ для жорстких обмежень.
- Ефективність завдяки підкрокам і одній ітерації Гаусса-Зейделя.
- Підтримка складних систем із великими співвідношеннями мас.

Обмеження:

- Підкроки збільшують обчислювальну складність.
- Складні геометрії потребують додаткових алгоритмів (наприклад, [12, 13]).

- Менша ефективність для високочастотних зіткнень порівняно з імпульсними методами.

1.5 Моделювання м’яких тіл за допомогою XPBD

М’які тіла, такі як тканина, консольні балки чи надувні об’єкти, моделюються в XPBD як набори частинок, з’єднаних обмеженнями, що дозволяють деформації. Кожна частинка має позицію $\mathbf{x}$, швидкість $\mathbf{v}$ і масу m, а система описується матрицею мас $\mathbf{M}$. Параметр податливості α та демпфування β контролюють жорсткість і дисипацію енергії. XPBD використовує неявну дискретизацію часу, що забезпечує стабільність і незалежність від розміру часового кроку чи кількості ітерацій [2].

1.5.1 Неявна інтеграція

Основою методу є рівняння руху з неявною дискретизацією:

$$\mathbf{M}\left(\frac{\mathbf{x}^{n+1}-2\mathbf{x}^{n}+\mathbf{x}^{n-1}}{\Delta t^{2}}\right)=-\nabla U^{T}(\mathbf{x}^{n+1}), \tag{1.27}$$

де $\mathbf{M}$ — матриця мас; $\mathbf{x}^{n+1}$, $\mathbf{x}^{n}$, $\mathbf{x}^{n-1}$ — позиції частинок у послідовні моменти часу; ∇U^{T} — градієнт енергетичного потенціалу. Потенціал енергії визначається через функції обмежень $\mathbf{C}(\mathbf{x})=[C_1(\mathbf{x}),\ C_2(\mathbf{x}),\ \dots,\ C_m(\mathbf{x})]^{T}$:

$$U(\mathbf{x})=\frac{1}{2}\mathbf{C}(\mathbf{x})^{T}\boldsymbol{\alpha}^{-1}\mathbf{C}(\mathbf{x}), \tag{1.28}$$

де $\boldsymbol{\alpha}$ — матриця податливості (обернена до жорсткості).

1.5.2 Обмеження для деформацій

Обмеження $C(x)$ моделюють деформації, такі як розтяг, зсув чи збереження об’єму. Еластична сила обчислюється як:

$$\mathbf{f}_{\text{elastic}}=-\nabla\mathbf{C}^{T}\boldsymbol{\alpha}^{-1}\mathbf{C}, \tag{1.29}$$

а множник Лагранжа для еластичних обмежень як:

$$\boldsymbol{\lambda}_{\text{elastic}} = -\boldsymbol{\alpha}^{-1}\mathbf{C}(\mathbf{x}), \tag{1.30}$$

де $\boldsymbol{\alpha} = \frac{\boldsymbol{\alpha}}{\Delta t^2}$.

Тоді дискретні рівняння руху отримають вигляд:

$$\begin{aligned} \mathbf{M}(\mathbf{x}^{n+1} - \mathbf{x}) - \nabla\mathbf{C}(\mathbf{x}^{n+1})^T\boldsymbol{\lambda}^{n+1} = \mathbf{0}, \\ \mathbf{C}(\mathbf{x}^{n+1}) + \boldsymbol{\alpha}\boldsymbol{\lambda}^{n+1} = \mathbf{0}, \end{aligned} \tag{1.31}$$

де $\mathbf{x}$ — передбачена (інерціальна) позиція. Ці рівняння розв'язуються ітеративно за допомогою методу Ньютона, що забезпечує точність і стабільність.

Для окремого обмеження j корекція множника Лагранжа отримується формулою:

$$\Delta\lambda_j = \frac{-C_j(\mathbf{x}_i) - \alpha_j\lambda_{ij}}{\nabla C_j \mathbf{M}^{-1}\nabla C_j^T + \alpha_j}. \tag{1.32}$$

Позиції оновлюються наступним чином:

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{M}^{-1}\nabla C_j^T \Delta\lambda_j. \tag{1.33}$$

1.5.3 Демпфування

Для моделювання демпфування використовується потенціал розсіювання Релея:

$$D(\mathbf{x}, \mathbf{v}) = \frac{1}{2}\mathbf{C}(\mathbf{x})^T\boldsymbol{\beta}\mathbf{C}(\mathbf{x}) = \frac{1}{2}\mathbf{v}^T\nabla\mathbf{C}^T\boldsymbol{\beta}\nabla\mathbf{C}\mathbf{v}, \tag{1.34}$$

де $\boldsymbol{\beta}$ — матриця коефіцієнтів демпфування.

Згідно з лагранжевою динамікою, сила демпфування отримуються з від'ємного градієнту D відносно швидкості:

$$\mathbf{f}_{damp} = -\nabla_{\boldsymbol{v}} D^T = -\nabla\mathbf{C}^T\boldsymbol{\beta}\nabla\mathbf{C}\mathbf{v}. \tag{1.35}$$

Загальний множник Лагранжа включає еластичні та демпфуючі компоненти:

$$\boldsymbol{\lambda} = \boldsymbol{\lambda}_{\text{elastic}} + \boldsymbol{\lambda}_{\text{damp}} = -\boldsymbol{\alpha}^{-1}\mathbf{C}(\mathbf{x}) - \boldsymbol{\beta}\nabla\mathbf{C}\mathbf{v}, \qquad \boldsymbol{\beta} = \Delta t^2\boldsymbol{\beta}.$$

Для окремого обмеження j оновлення множника Лагранжа виконується так:

$$\Delta\lambda_j = \frac{-C_j(\mathbf{x}_i) - \alpha_j\lambda_{ij} - \gamma_j\nabla C_j(\mathbf{x}_i - \mathbf{x}^n)}{(1+\gamma_\mathrm{j})\nabla C_j\mathbf{M}^{-1}\nabla C_j^\mathrm{T} + \alpha_j}, \qquad (1.36)$$

де $\gamma_j = \frac{\widetilde{\alpha}_j\beta_j}{\Delta t}$. Цей підхід дозволяє моделювати як еластичні, так і диссипативні властивості м'яких тіл, забезпечуючи високу точність і стабільність симуляції.

1.5.4 Візуалізація

М'які тіла в XPBD моделюються як набори частинок, з'єднаних обмеженнями, і візуалізуються через деформовані сітки або частинкові системи, що відображають об'єкти, такі як тканина чи балки.

1.5.5 Переваги та обмеження

Переваги:

- Гнучкість для моделювання еластичних деформацій.
- Стабільність завдяки неявній інтеграції.
- Придатність для реального часу з помірною кількістю частинок.

Обмеження:

- Висока обчислювальна складність при великій кількості частинок.
- Потреба в точному налаштуванні параметрів α та β.
- Нестабільність при екстремальних деформаціях за неоптимальних параметрів.

1.6 Моделювання рідин за допомогою PBF

Моделювання рідин у методі Position Based Fluids (PBF), який базується на динаміці на основі позицій (PBD), зосереджується на забезпеченні нестисливості рідини шляхом підтримки сталої густини, що є ключовою властивістю рідин, таких як вода. PBF використовує позиційні обмеження для ітеративної корекції позицій частинок, щоб досягти густини, близької до густини стану спокою ρ_0, що робить його ефективним для симуляцій у реальному часі, наприклад, у відеоіграх чи інтерактивних додатках [3]. Метод дозволяє створювати візуально реалістичні ефекти, такі як хвилі, бризки чи вихори, зберігаючи стабільність навіть при великих часових кроках [14].

1.6.1 Основи PBF

PBF базується на ідеї використання частинок для представлення рідини, де кожна частинка i має позицію $\boldsymbol{p}_i$, швидкість $\mathbf{v}_i$ та масу m_i [15]. Густина рідини в позиції частинки i обчислюється за допомогою функції згладжування W (smoothing kernel) з радіусом h:

$$\rho_i = \sum_j m_j W(\mathbf{p}_i - \mathbf{p}_j, h), \tag{1.37}$$

де j — індекс сусідніх частинок, m_j — їхня маса, а $\mathbf{p}_j$ — позиція [5]. Функція W є радіально симетричною і має скінченну область підтримки, що обмежується радіусом h, що зменшує обчислювальну складність.

Для забезпечення нестисливості рідини вводиться обмеження густини:

$$C_i(\mathbf{p}_i) = \frac{\rho_i}{\rho_0} - 1 \leq 0, \tag{1.38}$$

де ρ_0 —густина рідини в стані спокою. Це обмеження гарантує, що густина ρ_i не перевищує ρ_0, що відповідає фізичній властивості нестисливості.

1.6.2 Ітеративне розв'язання обмежень

Для виконання обмеження густини позиції частинок коригуються ітеративно за допомогою нелінійного розв'язувача, подібного до методу Гауса-Зейделя, який використовується в (X)PBD. Корекція позиції $\Delta \mathbf{p}_i$ для частинки i обчислюється як:

$$\Delta \mathbf{p}_i = \frac{1}{\rho_0} \sum_j (\lambda_i + \lambda_j) \nabla W(\mathbf{p}_i - \mathbf{p}_j, h), \tag{1.39}$$

де λ_i та λ_j — множники Лагранжа, які відповідають обмеженням густини для частинок i та j, а ∇W — градієнт функції згладжування [3]. Множник Лагранжа λ_i визначається через:

$$\lambda_i = -\frac{C_i(\mathbf{p}_i)}{\sum_j |\nabla \mathbf{p}_i W(\mathbf{p}_i - \mathbf{p}_j, h)|^2 + \varepsilon}, \tag{1.40}$$

де ε — мала константа для числової стабільності. Цей підхід дозволяє ітеративно зменшувати помилку густини, досягаючи нестисливості рідини за кілька ітерацій (зазвичай 2–4 для реального часу) [3].

1.6.3 Штучний тиск і поверхневий натяг

Однією з проблем у SPH-базованих методах, таких як PBF, є скупчення частинок (*clustering*) через нестабільність при негативних тисках, що може призводити до нерівномірного розподілу частинок (див. рис. 2 у [3]). Для вирішення цієї проблеми PBF вводить термін штучного тиску s_{corr}, який моделює поверхневий натяг і запобігає скупченню:

$$s_{\text{corr}} = -k \left(\frac{W(\mathbf{p}_i - \mathbf{p}_j, h)}{W(\Delta \mathbf{q}, h)} \right)^n, \tag{1.41}$$

де $k \approx 0.1$ — мала константа, $n \approx 4$ — показник степеня, а $\Delta \mathbf{q}$ — фіксована відстань у межах радіуса згладжування ($|\Delta \mathbf{q}| \approx 0.1h \dots 0.3h$) [3]. Оновлення позицій з урахуванням цього терміну виглядає так:

$$\Delta \mathbf{p}_i = \frac{1}{\rho_0} \sum_j (\lambda_i + \lambda_j + s_{\text{corr}}) \nabla W(\mathbf{p}_i - \mathbf{p}_j, h). \tag{1.42}$$

Цей термін є суто відштовхувальним і створює ефект, подібний до поверхневого натягу, що сприяє формуванню гладких поверхонь рідини. Однак, цей ефект є нефізичним артефактом антикластеризаційного доданку та вимагає компромісу між помилками кластеризації та силою поверхневого натягу [3].

1.6.4 Вихрова конфінація

Для збереження деталей руху рідини, таких як вихори, PBF застосовує вихорову конфінацію (*vorticity confinement*) як постпроцес до швидкостей частинок. Цей метод, запропонований у [16], компенсує числову дисипацію, яка виникає в позиційних методах, і додає енергію в систему для створення більш реалістичних візуальних ефектів. Вихрова конфінація обчислюється через локальний вихор ω_i:

$$\omega_i = \sum_j \frac{m_j}{\rho_j} (\boldsymbol{v}_j - \boldsymbol{v}_i) \times \nabla W(\boldsymbol{p}_i - \boldsymbol{p}_j, h), \tag{1.43}$$

а корекція швидкості додається як:

$$\Delta \mathbf{v}_i = \varepsilon_v (\mathbf{N}_i \times \omega_i) \Delta t, \tag{1.44}$$

де $\mathbf{N}_i = \nabla \rho_i / |\nabla \rho_i|$ — нормалізований градієнт густини, ε_v — коефіцієнт конфінації, а Δt — часовий крок. Це дозволяє створювати більш виразні бризки та завихрення, як показано на рис. 5 у [3].

1.6.5 Алгоритм PBF

Загальний алгоритм PBF для моделювання рідини складається з таких етапів:

1) **Пошук сусідів**: для кожної частинки i визначаються сусідні частинки j у радіусі h за допомогою ґраткової структури даних для зменшення

обчислювальної складності з $O(n^2)$ до $O(nm)$, де m — середня кількість сусідів.

2) **Обчислення густини**: розрахунок ρ_i.

3) **Ітеративне розв'язання обмежень**: обчислення множників Лагранжа λ_i та корекція позицій $\Delta \mathbf{p}_i$.

4) **Застосування штучного тиску**: Додавання s_{corr} для стабілізації розподілу частинок і моделювання поверхневого натягу.

5) **Вихорова конфінація**: Корекція швидкостей для збереження вихорів.

6) **Оновлення позицій і швидкостей**: Застосування корекцій до $\mathbf{p}_i$ та $\mathbf{v}_i$ з використанням схеми інтеграції, наприклад, Leap-Frog [5].

Цей алгоритм дозволяє створювати стабільні симуляції рідин у реальному часі з великими часовими кроками ($\Delta t \approx 0.016$ с), що значно перевищують обмеження традиційних SPH-методів, таких як PCISPH ($\Delta t \approx 0.0016$ с) [3].

1.6.6 Візуалізація

Зазвичай, формування поверхні рідини в реальному часі виконується за допомогою методу розбризкування еліпсоїдів (*ellipsoid splatting*) на основі графічного процесора (GPU). Анізотропія частинок спочатку обчислюється за допомогою методу [17], а поверхня реконструюється за допомогою методу, заснованого на фільтрації екранного простору, представленого в [18]. Також, для рендерингу симульованих рідин у реальному часі застосовуються такі методи, як рендеринг в екранному просторі [19].

1.6.7 Переваги та обмеження

Переваги:

- Забезпечує стабільність симуляції при великих часових кроках завдяки ітеративній корекції нестисливості.
- Ефективно моделює зіткнення з твердими тілами через позиційні обмеження.
- Дозволяє реальне моделювання складних потоків, таких як бризки та вихори.

Обмеження:

- Вимагає значної кількості ітерацій для точного підтримання густини, що підвищує обчислювальну складність.
- Чутливий до вибору параметрів ядра та радіусу згладжування.
- Обмежений у точному відтворенні тонких струменів без додаткових оптимізацій.

1.7 Імітаційна модель інтегрованого моделювання фізичних тіл

Виходячи з вищеописаних підходів для моделювання різних типів тіл та того факту, що (X)PBD та інші суміжні з ним методи (наприклад, PBF) моделюють їх, використовуючи уніфіковану концпецію *частинки*, можна розробити систему, що дозволяє обробляти тверді тіла, м’які тіла, рідини, гранулярні матеріали та тканину в єдиному фреймворку [8]. Усі об’єкти представляються як набори частинок, з’єднаних обмеженнями, що обробляються ітеративно. Для уніфікації можна використати абстрактні класи *Constraint* (для обмежень) та *Body* (для тіл), від яких походять спеціалізовані реалізації, що спрощує цикл симуляції, усуває зайві перевірки та зменшує код.

1.7.1 Моделювання різних типів тіл

- **Тверді тіла:**

– *представлення*: набір частинок із обмеженнями відповідності форми (*shape-matching constraints*) для забезпечення жорсткості. Фазовий ідентифікатор запобігає самоколізіям.

– *обробка*: передбачення позицій частинок, застосування зовнішніх сил, розв’язання обмежень відповідності форми та контактних обмежень із SDF (Signed Distance Field) для колізій.

– *взаємодія*: двостороння взаємодія через контактні обмеження, наприклад, із рідинами при обчисленні густини.

- **М’які тіла:**

– *представлення*: частинки, з’єднані обмеженнями на розтяг, зсув та збереження об’єму. Податливість (α) та демпфування (β) контролюють деформацію.

– *обробка*: передбачення позицій, застосування еластичних та демпфуючих сил, ітеративне розв’язання обмежень за методом Ньютона.

– *взаємодія*: контактні обмеження для взаємодії з твердими тілами чи рідинами, наприклад, тканина, що обтікає сферу.

- **Рідини:**

– *представлення*: частинки з обмеженнями густини; функція згладжування (W) обчислює густину та градієнти.

– *обробка*: пошук сусідів, обчислення густини, розв’язання обмежень густини, застосування штучного тиску та вихрової конфінації.

– *взаємодія*: включення частинок твердих тіл у розрахунок густини для плавучості чи тиску.

- **Тканина:**

– *представлення*: сітка частинок із обмеженнями на розтяг, зсув та самоколізії між частинками.

– *обробка*: передбачення позицій, розв’язання обмежень розтягу/зсуву, обробка самоколізій.

– *взаємодія*: контактні обмеження для обтікання твердих тіл чи взаємодії з рідиною.

- **Гранулярні матеріали:**

– *представлення*: частинки з високим тертям, оброблювані позиційними корекціями для статичного тертя.

– *обробка*: розв’язання контактних обмежень із тертям на основі тангенціального зміщення.

– *взаємодія*: контактні обмеження для накопичення на поверхнях чи реакції на рідину.

1.7.2 Цикл симуляції

На рис. 1.2 наведено спрощений псевдокод циклу симуляції, що використовує ідею об’єднання різних типів тіл та обмежень під абстрактними класами для зменшення перевірок та зайвиї операцій.

Algorithm 1: General XPBD

Input: $bodies$, Δt
Output: Updated positions $\mathbf{x}$ and velocities $\mathbf{v}$
while *simulating* **do**
  CollectCollisionPairs();
  $h \leftarrow \Delta t / numSubsteps$;
  for $numSubsteps$ **do**
    Predict($bodies$);
    for $numPosIters$ **do**
      SolvePositions($bodies$);
    Update($bodies$);
    SolveVelocities($bodies$) ;

Рисунок 1.2 – Псевдокод головного циклу системи моделювання

Головний цикл умовно можно поділити на дві складових: пошук потенційних зіткнень тіл (функція *CollectCollisionPairs* на рис. 1.2) та оновлення їх станів в залежності від взаємодій, які відбулися.

Оновлення стану відбувається в 4 етапи: передбачення (рис. 1.3), розв'язання обмежень та знаходження корекцій позицій Δx_i (найнижчий вкладений цикл на рис. 1.2), оновлення (рис. 1.4) та корекція швидкостей (фукнція *SolveVelocities* на рис. 1.2). Корекція швидкостей відбувається для твердих тіл задля врахування таких ефектів, як тертя та відбиття (див. пункт 1.4.6).

Algorithm 2: Predict

Input: *bodies*, Δt
Output: Predicted positions $\mathbf{x}$ and velocities $\mathbf{v}$, ω
for *body in bodies* **do**
  if *body.type* = "*rigid*" **then**
    $\mathbf{x}_{prev} \leftarrow \mathbf{x}_{COM}$;
    $\mathbf{v}_{COM} \leftarrow \mathbf{v}_{COM} + h\mathbf{f}_{ext}/m$;
    $\mathbf{x}_{COM} \leftarrow \mathbf{x}_{COM} + h\mathbf{v}$;
    $\mathbf{q}_{prev} \leftarrow \mathbf{q}$;
    $\omega \leftarrow \omega + h\mathbf{I}^{-1}(\tau_{ext} - (\omega \times (\mathbf{I}\omega)))$;
    $\mathbf{q} \leftarrow \mathbf{q} + h\frac{1}{2}[\omega_x, \omega_y, \omega_z, 0]\mathbf{q}$;
    $\mathbf{q} \leftarrow \mathbf{q}/|\mathbf{q}|$;
  else
    for *n particles* **do**
      $\mathbf{x}_{prev} \leftarrow \mathbf{x}$;
      $\mathbf{v} \leftarrow \mathbf{v} + h\mathbf{f}_{ext}/m$;
      $\mathbf{x} \leftarrow \mathbf{x} + h\mathbf{v}$;

Рисунок 1.3 – Крок передбачення позицій та швидкостей

Algorithm 3: Update

Input: *bodies*, Δt
Output: Updated velocities $\mathbf{v}$, ω
for *body in bodies* **do**
  if *body.type* = "*rigid*" **then**
    $\mathbf{v} \leftarrow (X - X_{prev})/h$;
    $\Delta\mathbf{q} \leftarrow \mathbf{q}\mathbf{q}_{prev}^{-1}$;
    $\omega \leftarrow 2[\Delta\mathbf{q}_x, \Delta\mathbf{q}_y, \Delta\mathbf{q}_z]/h$;
    $\omega \leftarrow (\Delta q_w \geq 0)\ ?\ \omega : -\omega$;
  else
    for *n particles* **do**
      $\mathbf{v} \leftarrow (\mathbf{x} - \mathbf{x}_{prev})/h$;

Рисунок 1.4 – Крок оновлення швидкостей

Таким чином, здійснюється можливість інтегрувати обробки різних типів тіл, що використовують різні типи обмежень, в один цикл.

2 РОЗРОБКА ІМІТАЦІЙНОЇ МОДЕЛІ ДЛЯ МОДЕЛЮВАННЯ ВЗАЄМОДІЙ ФІЗИЧНИХ ТІЛ

Цей розділ присвячено розробці (реалізації) описаної імітаційної моделі для інтегрованого моделювання взаємодій фізичних тіл у реальному часі на основі методів розширеної позиційної динаміки (XPBD) та позиційних рідин (PBF). Модель орієнтована на створення єдиної системи, що об’єднує різні типи тіл у модульному дизайні. Вона забезпечує гнучкість, стабільність і підтримку інтерактивних взаємодій, що робить її придатною для застосувань у комп’ютерній графіці, віртуальній реальності та ігровій індустрії. У цьому розділі описано архітектуру моделі, реалізацію відповідних алгоритмів, а також перспективи інтеграції в цілісну систему.

Розробка базується на уніфікованій системі частинок, де кожен об’єкт представлено як набір частинок із відповідними фізичними властивостями, такими як позиція, швидкість і маса. Використання поліморфних обмежень і централізованого розв’язувача XPBD дозволяє адаптувати модель до різних типів тіл (твердих, м’яких, тканин) та їхніх взаємодій. Особливу увагу приділено модульності, що полегшує розширення системи та оптимізацію продуктивності. Результати роботи включають архітектурні схеми, псевдокод і попередні оцінки ефективності, які будуть деталізовані в наступних підрозділах.

2.1 Опис архітектури імітаційної моделі

Архітектура імітаційної моделі для моделювання фізичних взаємодій, представлена на рис. 2.1, базується на модульному підході, який забезпечує чіткий розподіл функцій між компонентами та дозволяє легко розширювати систему в майбутньому. Вона складається з двох основних етапів: **попередньої обробки даних** (*Data Preprocessing*) та **ініціалізації з симуляцією** (*Initialization & Simulation*). Ці етапи пов’язані між собою і

формують цілісний процес — від підготовки вхідних даних до візуалізації результатів у реальному часі. Діаграма чітко ілюструє, як система переходить від сирих даних до готової сцени, а потім запускає симуляцію, спираючись на стандартні інструменти та бібліотеки, такі як TetGen, SDL і OpenGL, що широко застосовуються в проєктах імітаційного моделювання і графічних застосунках.

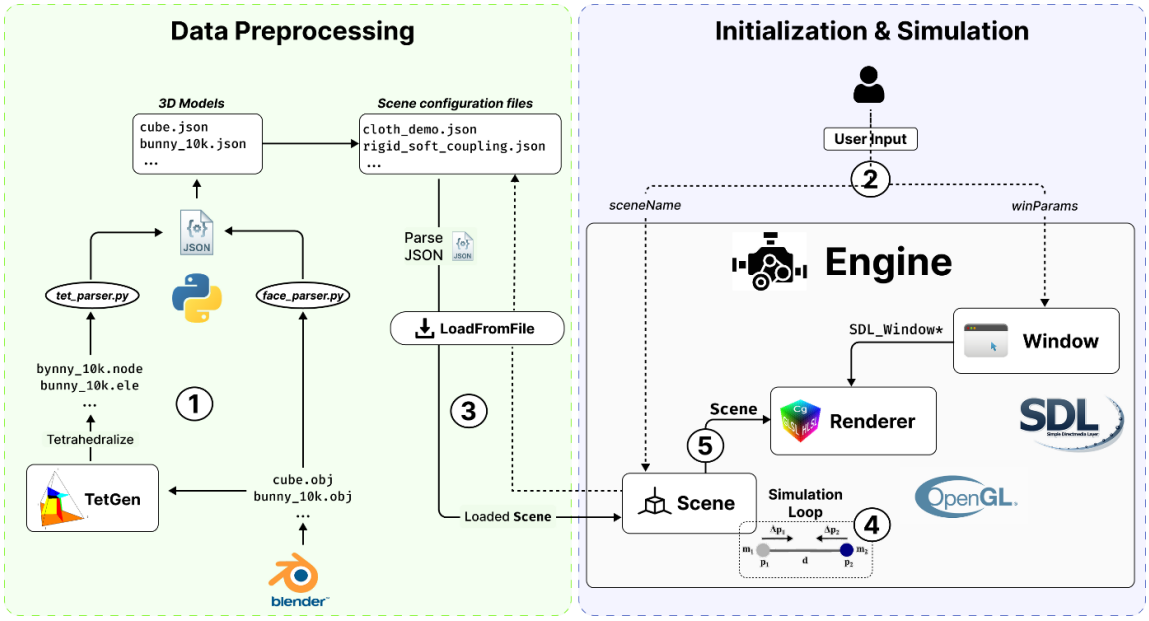

Рисунок 2.1 – Діаграма загальної архітектури проєкту

2.1.1 Попередня обробка даних (Data Preprocessing)

Етап попередньої обробки даних розпочинається з генерування 3D-моделей за допомогою середи 3D-моделювання Blender. Після цього, отримавши файли .obj, в залежності від того, який тип сітки (mesh) ми хочем отримати – поверхневу (face) чи тетраедальну (tetrahedral) – за допомогою реалізованих Python-скриптів генеруються всі необхідні дані про моделі у форматі JSON. У випадку, коли потрібно згенерувати тетраедальну сітку, файли .obj чи .ply передаються до спеціальної програми TetGen, розроблену інститутом прикладного аналізу та стохастики імені Вейєрштрасса [15], що генерує на основі поверхнової сітки та методу тріангуляції Делоне внутрішні

тетраедри [16]. Після обробки файлів відповідним скриптом – tet_parser.py чи face_parser.py – отримуються дані про моделі в форматі .json (наприклад, “cube.json” чи “bunny_10k.json” на рис. 2.1). Ці файли містять геометричні дані об’єктів, зокрема координати вершин, ребра та грані, які необхідні для створення сіток. При налаштуванні конфігурації всієї сцени через той самий формат .json, шляхи до цих файлів вказуються як атрибути відповідних тіл, як показано на прикладі рис. 2.2.

Рисунок 2.2 – Зазначення шляху до сітки у конфігураційному файлі сцени

Конфігураційні файли сцени, наприклад, "cloth_demo.json" чи "rigid_soft_coupling.json", створюються завчасно або генеруються спеціальним скриптом SceneGenerator.py. Вони визначають фізичні властивості об’єктів, параметри їх взаємодії та загальні налаштування симуляції. Обробка цих даних покладається на конструктор класу Scene, який відіграє центральну роль на цьому етапі. Конструктор Scene приймає на вхід назву файлу сцени та передає його методу LoadFromFile(), який об’єднує геометричну інформацію з конфігураційними параметрами і готує їх для подальшого використання,

зберігаючи в об’єкті Scene. Цей об’єкт включає геометрію об’єктів, їхні фізичні параметри та конфігурацію сцени, створюючи повноцінний набір даних, готовий для передачі на етап ініціалізації та симуляції. Така організація попередньої обробки даних дозволяє моделі бути модульною і гнучкою, адже кожен крок — від завантаження файлів до генерації сіток — виконується незалежно, але в рамках єдиного процесу підготовки.

2.1.2 Ініціалізація та симуляція (Initialization & Simulation)

Переходячи до етапу ініціалізації та симуляції, ми бачимо, як модель починає працювати з користувацьким вводом і запускає основні процеси моделювання. Цей етап розпочинається з отримання двох ключових параметрів від користувача: sceneName та winParams. Параметр sceneName визначає, яку сцену потрібно завантажити, наприклад, "cloth_demo" для симуляції тканини чи "rigid_soft_coupling" для моделювання взаємодії твердих і м’яких тіл. Параметр winParams задає налаштування вікна для візуалізації, такі як роздільна здатність чи режими відображення, що впливають на те, як користувач бачитиме результати симуляції. Ці дані передаються до центрального компонента цього етапу — Engine, який координує всю подальшу роботу системи.

Модель створює об’єкт Scene через конструктор із використанням sceneName для вибору відповідної конфігурації сцени. Також налаштовується вікно для відображення, спираючись на winParams і бібліотеку SDL (Simple DirectMedia Layer). SDL є популярним інструментом завдяки своїй кросплатформенній підтримці для створення вікон, обробки подій і управління введенням. У цьому випадку SDL створює вікно, яке стає основою для візуалізації сцени, дозволяючи моделі працювати однаково добре на різних операційних системах. Для рендерингу використовується бібліотека OpenGL, потужний графічний API, який забезпечує апаратне прискорення для промальовування 2D- і 3D-об’єктів. Renderer отримує дані зі Scene і, за

потреби, застосовує шейдери, написані на GLSL (OpenGL Shading Language), для створення складних візуальних ефектів, таких як освітлення чи текстури, що часто зустрічається в системах моделювання.

Після ініціалізації вікна та рендерингу Engine запускає *Simulation Loop* — головний цикл симуляції, який є ядром динамічної роботи моделі. У цьому циклі модель ітеративно оновлює стан сцени на основі фізичних законів і заданих правил. Наприклад, для кожного об’єкта в сцені обчислюються нові позиції, швидкості та взаємодії з іншими об’єктами, що може включати прості пружинні сили чи складніші зіткнення. Діаграма показує приклад із двома частинками, p_1 і p_2, із масами m_1 і m_2, відстанню d між ними та змінами Δp_1 і Δp_2, що ілюструє базову фізичну взаємодію методу XPBD. Кожна ітерація циклу завершується передачею оновлених даних до *Renderer*, який використовує OpenGL для промальовування сцени у вікні, забезпечуючи користувачу можливість бачити результати в реальному часі.

2.1.3 Головний цикл симуляції

Головний цикл симуляції, зображений на рис. 2.3, є основою для моделювання. Він включає кілька ключових етапів, які виконуються послідовно та повторюються, доки вікно програми залишається відкритим.

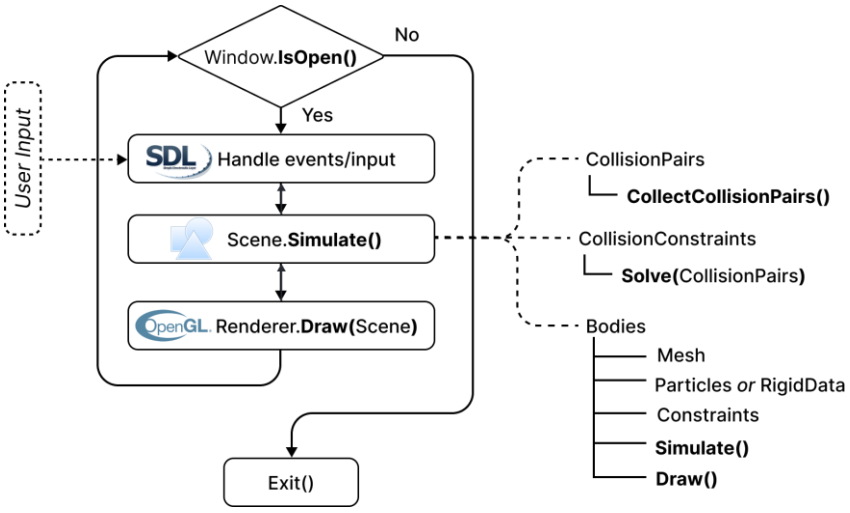

Рисунок 2.3 – Головний алгоритм моделі (системи)

Цикл починається з перевірки стану вікна за допомогою методу *Window.**IsOpen()***. Ця перевірка є умовою для продовження виконання програми: якщо вікно відкрите, модель переходить до наступних етапів; якщо ні, програма завершує роботу, що зображено викликом *Exit()*. Цей механізм є стандартним для графічних додатків, що використовують бібліотеки на зразок SDL, і забезпечує коректне завершення програми при закритті вікна користувачем.

Наступним етапом є обробка подій та введення користувача через *SDL Handle events*, де обробляються дії користувача, такі як натискання клавіш, через функції на кшталт *SDL_PollEvent()*. На цьому кроці система зчитує та обробляє всі події, що надійшли від користувача, такі як натискання клавіш, рухи миші або команди для зміни параметрів симуляції. Обробка подій дозволяє моделі бути інтерактивною, реагуючи на дії користувача в реальному часі. Наприклад, користувач може змінити положення об'єкта або запустити нову симуляцію.

Після обробки подій модель переходить до етапу симуляції через виклик *Scene.**Simulate()***. Цей метод є основним для оновлення стану сцени: він обчислює нові позиції, швидкості та інші параметри об'єктів на основі фізичних законів і заданих обмежень. Симуляція може включати інтегрування рівнянь руху, перевірку зіткнень або застосування обмежень, залежно від типу моделювання. Цей етап є найбільш обчислювально складним, але водночас і ключовим для забезпечення реалістичної поведінки об'єктів.

Нарешті, після оновлення сцени, система виконує рендеринг через *Renderer.**Draw(**Scene**)***. *Renderer* використовує OpenGL для візуалізації оновленої сцени на екрані, забезпечуючи користувачу можливість бачити результати симуляції. Після рендерингу цикл повертається до перевірки стану вікна, і процес повторюється.

Важливо також зазначити, що етап *Scene.**Simulate()*** включає внутрішні компоненти, які деталізують процес оновлення сцени. Зокрема, це об’єкти

Bodies і *Constraints*, які відповідають за фізичну поведінку та взаємодію тіл. Ці елементи є ключовими для розуміння того, як модель інтегрує різні типи об’єктів, таких як м’які тіла чи рідини.

2.2 Опис ключових модулів для моделювання

Імітаційна модель базується на наборі ключових модулів, які забезпечують ефективне моделювання фізичних об’єктів та їхніх взаємодій. Кожен модуль відповідає за конкретну частину процесу, від представлення тіл до обробки обмежень, інтегруючись із сучасними методами, такими як XPBD і PBF. Нижче наведено детальний опис двох основних модулів, що лежать в основі функціональності моделі та її алгоритмів.

2.2.1 Модуль тіл (Body)

Модуль *Body* є основою для моделювання фізичних об’єктів імітаційної моделі, забезпечуючи гнучку ієрархію класів для представлення різних типів тіл, таких як жорсткі тіла, м’які тіла, тканини та рідини. Він інтегрується з методами XPBD для моделювання деформівних і жорстких об’єктів та PBF для рідин, що дозволяє реалізовувати широкий спектр фізичних взаємодій.

Діаграма ієрархії класу Body (рис. 2.4) зображує базовий клас Body із властивостями Mesh, Particles or RigidParams, Constraints та методами Simulate() і Draw(). Від нього успадковуються підкласи RigidBody, SoftBody, Cloth і Fluid, кожен із власними специфічними властивостями та обмеженнями.

Як можна зрозуміти з цього рисунку, клас Body є абстрактним базовим класом, який визначає спільний інтерфейс для всіх фізичних об’єктів. Його основна роль — забезпечити уніфікований доступ до геометричних і фізичних даних, а також методів для симуляції та рендерингу.

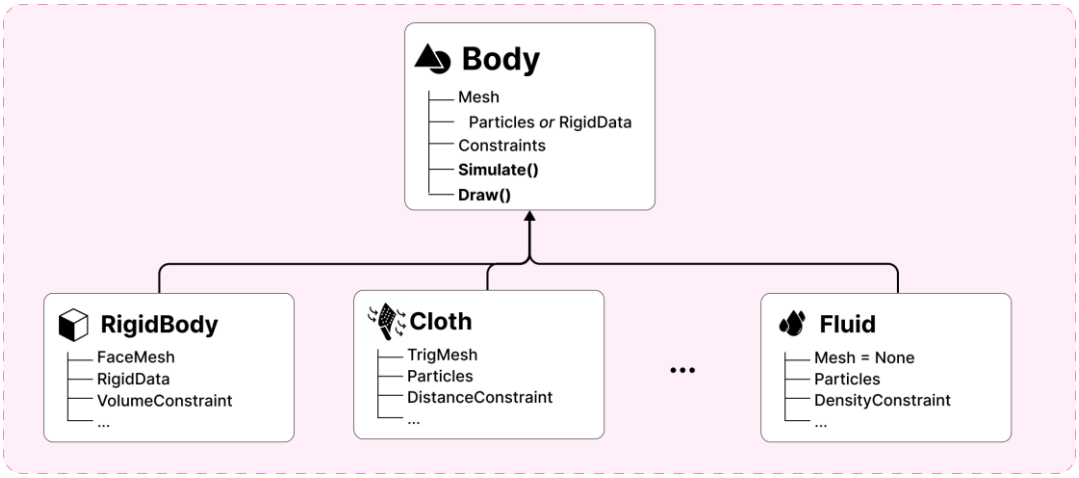

Рисунок 2.4 – Ієрархія успадкування класа Body

При цьому, кожен підклас спеціалізується на певному типі фізичної поведінки:

- **RigidBody**

Жорстке тіло (RigidBody) представляє об’єкти, які не деформуються під дією сил, наприклад, куби чи сфери. Його ключові властивості включають:

– *FaceMesh* — сітка граней, що визначає поверхню об’єкта, зазвичай трикутна або полігональна.

– *RigidData* — параметри, такі як маса, момент інерції та коефіцієнти тертя, які визначають динамічну поведінку.

– *VolumeConstraint* — обмеження для збереження об’єму та форми, що застосовується через XPBD для стабільної симуляції зіткнень і рухів.

– *DampingCoefficient* — коефіцієнт демпфування, який зменшує коливання об’єкта.

– *RestitutionCoefficient* — коефіцієнт реституції, що визначає еластичність при зіткненнях. Симуляція жорстких тіл базується на інтеграції рівнянь Ньютона-Ейлера, де XPBD використовується для розв’язання обмежень.

- **SoftBody**

М’яке тіло (SoftBody) моделює деформовані об’єкти, такі як желе чи м’які тканини. Воно включає:

– *TrigMesh* — трикутна сітка для представлення поверхні об’єкта.

– *Particles* — набір частинок, що формують внутрішню структуру, зазвичай у вигляді тетраедричної сітки.

– *DistanceConstraint* — обмеження відстані між частинками для підтримки еластичності.

– *VolumeConstraint* — обмеження для збереження об’єму, що є критично важливим для запобігання колапсу під дією сил.

– *ElasticityCoefficient* — коефіцієнт пружності, що регулює ступінь деформації.

– *Viscosity* — параметр в’язкості, який впливає на внутрішнє тертя. Симуляція м’яких тіл використовує XPBD для ітеративного розв’язання обмежень.

- **Cloth**

Тканина (Cloth) моделює гнучкі поверхні, такі як одяг чи прапори. Вона включає:

– *TrigMesh* — трикутна сітка для представлення поверхні тканини.

– *Particles* — частинки, що формують структуру тканини.

– *DistanceConstraint* — обмеження для підтримки відстаней між частинками, що забезпечує натягування та згинання.

– *BendingCoefficient* — коефіцієнт згинання, що регулює жорсткість тканини.

– *WindCoefficient* — параметр впливу вітру, який додає динамічність рухам. Симуляція тканини зосереджена на поверхневих ефектах, використовуючи XPBD для обробки обмежень.

- **Fluid**

Рідина (Fluid) моделює рідкі середовища, такі як вода чи олія. Вона включає:

– *Mesh = None* — відсутність традиційної сітки, оскільки рідини представлені частинками.

– *Particles* — набір частинок для моделювання потоку.

– *DensityConstraint* — обмеження щільності, реалізоване через PBF для підтримки об’єму рідини та її поверхневої напруги.

– *Viscosity* — коефіцієнт в’язкості, що визначає опір течії.

– *SurfaceTension* — параметр поверхневого натягу, який стабілізує форму рідини. Симуляція рідин базується на PBF, що дозволяє ефективно моделювати їхню поведінку в реальному часі.

2.2.2 Модуль обмежень (Constraint)

Модуль *Constraint* є ключовим компонентом середи моделювання фізичних взаємодій, який забезпечує дотримання фізичних правил шляхом застосування обмежень до тіл. Обмеження регулюють взаємодію між частинками або об’єктами, забезпечуючи стабільність і реалістичність симуляції. У контексті методів XPBD і PBF обмеження використовуються для розв’язання задач, таких як збереження відстаней, об’ємів, щільності чи обробка зіткнень.

Діаграма ієрархії класу Constraint (рис. 2.5) зображує базовий клас Constraint із методом Solve(). Від нього успадковуються підкласи *DistanceConstraint*, *VolumeConstraint*, *DensityConstraint*, а також додаткові типи, такі як *ContactConstraint* (з варіантами для різних комбінацій тіл), які розширюють функціональність системи.

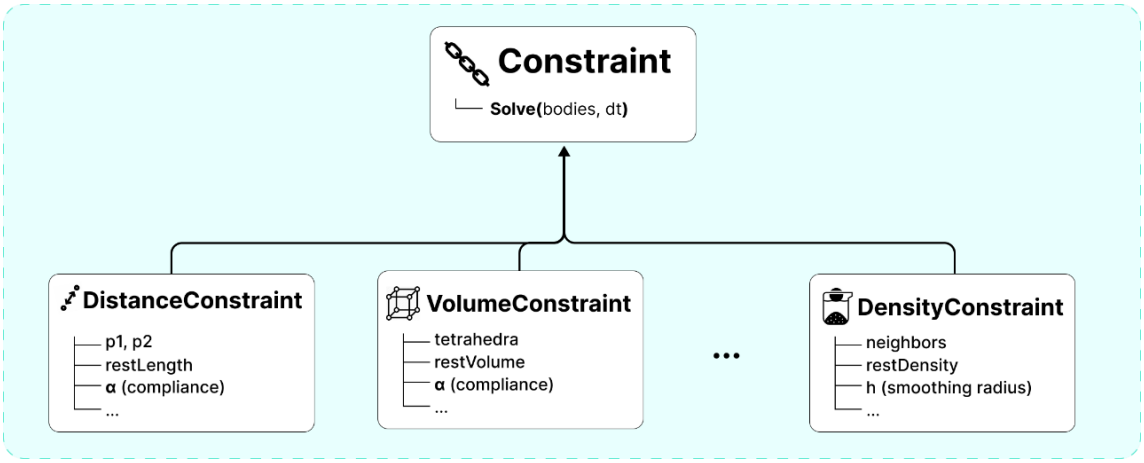

Рисунок 2.5 – Ієрархія успадкування класа Constraint

Клас Constraint є абстрактним базовим класом, який визначає спільний інтерфейс для всіх обмежень у системі. Його основна роль — забезпечити метод Solve(), який коригує позиції частинок або тіл для дотримання фізичних правил. Кожен підклас спеціалізується на певному типі фізичних обмежень, таких як відстань, об’єм, щільність чи зіткнення, адаптуючи метод Solve() до конкретних потреб:

- **DistanceConstraint**

Забезпечує збереження відстані між двома частинками, що є основою для моделювання пружних зв’язків у м’яких тілах чи тканинах.

– *p1*, *p2* — позиції двох частинок, між якими підтримується відстань.

– *restLength* — початкова відстань між частинками.

– *stiffness* — коефіцієнт жорсткості, що визначає силу корекції.

– *compliance* — параметр податливості, що регулює гнучкість обмеження в XPBD.

Псевдокод розв’язання зображено на рис. 2.6.

```
def solve(p1, p2, restLength, stiff, comp, dt):
    delta = p2.position - p1.position
    currentLength = delta.length()
    error = currentLength - restLength
    alpha = comp / (dt * dt)
    correction = error * stiff / (p1.mass + p2.mass + alpha)
    delta.normalize()
    p1.position += delta * correction * p1.mass
    p2.position -= delta * correction * p2.mass
```

Рисунок 2.6 – Псевдокод розв’язання обмеження відстані

- **VolumeConstraint**

Забезпечує збереження об’єму тетраедричних елементів у м’яких тілах, запобігаючи їхньому колапсу. Параметри:

– *particles* — список частинок, що формують тетраедр.

– *restVolume* — початковий об’єм тетраедра.

– *stiffness* — коефіцієнт жорсткості для корекції.

– *compliance* — параметр податливості для XPBD.

Псевдокод розв’язання зображено на рис. 2.7.

```
def compute_current_volume(particles):
    p0, p1, p2, p3 = particles
    v1 = p1.position - p0.position
    v2 = p2.position - p0.position
    v3 = p3.position - p0.position
    return abs(v3.dot(v1.cross(v2))) / 6.0

def solve(particles, restVolume, stiff, comp, dt):
    currentVolume = compute_current_volume(particles)
    error = currentVolume - restVolume
    alpha = comp / (dt * dt)
    correction = error * stiff / (sum(p.mass for p in particles) + alpha)
    for particle in self.particles:
        particle.position += correction * particle.mass
```

Рисунок 2.7 – Псевдокод розв’язання обмеження об’єму

- **DensityConstraint**

Забезпечує нестисливість рідин у PBF шляхом підтримки постійної щільності. Параметри:

– *particles* — список частинок рідини.

– *restDensity* — бажана щільність рідини.

– *h* — радіус згладжування для пошуку сусідів.

– *stiffness* — коефіцієнт жорсткості для корекції щільності.

Псевдокод розв’язання зображено на рис. 2.8.

```
def solve(particles, restDensity, h, stiff, dt):
    for i, particle in enumerate(particles):
        density = 0.0
        for j in get_neighbors(i, h):
            r = particle.position - particles[j].position
            density += poly6_kernel(r.length(), h)
        error = density - restDensity
        correction = error * stiff / (dt * dt)
        grad = compute_gradient(i, particles, h)
        particle.position += correction * grad
```

Рисунок 2.8 – Псевдокод розв’язання обмеження густини

- **BendingConstraint**

Забезпечує згинання поверхонь у тканинах, підтримуючи їхню гнучкість і реалістичність. Параметри:

– *particles* — частинки, що формують трикутники тканини.

– *restAngle* — початковий кут між сусідніми трикутниками.

– *stiffness* — коефіцієнт жорсткості згинання.

– *compliance* — параметр податливості для XPBD.

Псевдокод розв’язання зображено на рис. 2.9.

```
def solve(particles, restAngle, stiff, comp, dt):
    p1, p2, p3, p4 = particles
    n1 = (p2 - p1).cross(p3 - p1).normalize()
    n2 = (p2 - p1).cross(p4 - p1).normalize()
    currentAngle = acos(n1.dot(n2))
    error = currentAngle - restAngle
    alpha = comp / (dt * dt)
    correction = error * stiff / (sum(p.mass for p in particles) + alpha)
    for particle in particles:
        particle.position += correction * compute_gradient(particle, n1, n2)
```

Рисунок 2.9 – Псевдокод розв’язання обмеження згинання

- **ContactConstraint (Rigid-Rigid)**

Обробляє зіткнення між двома жорсткими тілами, запобігаючи їхньому проникненню. Параметри:

– *rb1*, *rb2* — жорсткі тіла, що зіткнулися.

– *contactPoint* — точка зіткнення.

– *normal* — нормаль до поверхні зіткнення.

– *restitution* — коефіцієнт реституції, що визначає еластичність зіткнення.

– *friction* — коефіцієнт тертя для обробки ковзання.

Псевдокод розв’язання зображено на рис. 2.10.

```
def solve(rb1, rb2, contactPoint, normal, restitution, friction, dt):
    relativeVelocity = rb2.velocity - rb1.velocity
    velocityAlongNormal = relativeVelocity.dot(normal)
    if velocityAlongNormal > 0:
        return
    impulse = -(1 + restitution) * velocityAlongNormal
    impulse /= (1 / rb1.mass + 1 / rb2.mass)
    rb1.velocity -= (impulse / rb1.mass) * normal
    rb2.velocity += (impulse / rb2.mass) * normal
    rb1.angularVelocity -= rb1.inverseInertia * (contactPoint - rb1.position).cross(normal * impulse)
    rb2.angularVelocity += rb2.inverseInertia * (contactPoint - rb2.position).cross(normal * impulse)
```

Рисунок 2.10 – Псевдокод розв’язання обмеження контакту тверде тіло – тверде тіло

- **ContactConstraint (Rigid-Soft)**

Обробляє зіткнення між жорстким і м’яким тілом, забезпечуючи коректну взаємодію. Параметри:

– *rb* — жорстке тіло.
– *sb* — м’яке тіло (з частинками).
– *contactPoints* — список точок зіткнення.
– *normals* — нормалі до поверхні зіткнення.
– *restitution* — коефіцієнт реституції для еластичності.

Псевдокод розв’язання зображено на рис. 2.11.

```
def solve(rb, sb, contactPoints, normals, restitution, dt):
    for i, contactPoint in enumerate(.contactPoints):
        normal = .normals[i]
        particle = .sb.get_particle_at(contactPoint)
        relativeVelocity = particle.velocity - .rb.velocity
        velocityAlongNormal = relativeVelocity.dot(normal)
        if velocityAlongNormal > 0:
            continue
        impulse = -(1 + .restitution) * velocityAlongNormal
        impulse /= (1 / particle.mass + 1 / .rb.mass)
        particle.velocity -= (impulse / particle.mass) * normal
        .rb.velocity += (impulse / .rb.mass) * normal
```

Рисунок 2.11 – Псевдокод розв’язання обмеження контакту тверде тіло – м’яке тіло

2.3 Реалізація алгоритмів XPBD та PBF

Цей підрозділ детально описує реалізацію алгоритмів Extended Position-Based Dynamics (XPBD) і Position-Based Fluids (PBF) у системі симуляції фізичних взаємодій, що розроблюэться. XPBD застосовується для моделювання твердих і м’яких тіл, тоді як PBF використовується для симуляції рідин. Розглянемо нижче алгоритми для кожного типу тіл, методи обробки зіткнень і способи поєднання різних типів тіл у єдиній симуляції.

2.3.1 Алгоритм моделювання твердих тіл

Моделювання твердих тіл базується на методі XPBD, який забезпечує стабільну симуляцію жорстких об’єктів із використанням фізичних одиниць і

параметра податливості (compliance), що є оберненим до жорсткості. Як зазначається в [11], XPBD дозволяє точно обробляти контактні обмеження та з'єднання, використовуючи субкроки (substeps) для підвищення точності та збереження енергії. Основний цикл симуляції включає оновлення позицій, орієнтацій і швидкостей, із застосуванням ітеративного розв'язання обмежень.

Алгоритм для твердих тіл передбачає виконання кількох субкроків у кожному кадрі, де кожен субкрок оновлює стан об'єктів і розв'язує обмеження. Зіткнення виявляються один раз на кадр за допомогою axis-aligned bounding boxes (AABB). Для кожного субкроку розміром $h = \Delta t / \mathrm{numSubsteps}$ (зазвичай $\Delta t = 1/60$ с) виконуються наступні дії: оновлення позицій і швидкостей, оновлення орієнтацій, розв'язання позиційних обмежень і оновлення швидкостей на основі змін позицій (рис. 2.12).

```
def simulate_rigid_bodies(bodies, dt, numSubsteps):
    h = dt / numSubsteps
    collision_pairs = collect_collision_pairs(bodies, dt)

    for substep in range(numSubsteps):
        for body in bodies:
            body.x_prev = body.x
            body.v += h * body.f_ext / body.m
            body.x += h * body.v

            body.q_prev = body.q
            body.omega += h * body.invI * (body.tau_ext - cross(body.omega, body.I * body.omega))
            body.q += (h / 2) * quaternion(body.omega, 0) * body.q
            normalize(body.q)

        solve_positions(bodies, collision_pairs)

        for body in bodies:
            body.v = (body.x - body.x_prev) / h
            delta_q = body.q * inverse(body.q_prev)
            body.omega = 2 * vector_part(delta_q) / h
            if delta_q.w < 0:
                body.omega = -body.omega

        solve_velocities(bodies)
```

Рисунок 2.12 – Фрагмент коду, що описує моделювання твердих тіл

Цей псевдокод відображає основний цикл симуляції твердих тіл, де *solve_positions()* і *solve_velocities()* включають розв'язання контактних і з'єднувальних обмежень.

2.3.2 Алгоритм моделювання м’яких тіл

М’які тіла, такі як тканини чи еластичні матеріали, моделюються з використанням XPBD, який дозволяє ітеративно розв’язувати обмеження, такі як відстань, згинання та збереження об’єму. Згідно з [2], XPBD усуває залежність жорсткості від розміру часового кроку та кількості ітерацій, використовуючи параметр податливості (α) для точного контролю деформацій. Основний цикл симуляції для м’яких тіл включає передбачення позицій частинок, ітеративне розв’язання обмежень і оновлення швидкостей (рис. 2.13).

```
def simulate_soft_body(particles, constraints, dt, num_iterations):
    for i in range(len(particles)):
        particles[i].v += dt * particles[i].f_ext / particles[i].m
        particles[i].x_pred = particles[i].x + dt * particles[i].v

    for iter in range(num_iterations):
        for constraint in constraints:
            i, j, d, alpha = constraint
            delta = particles[j].x_pred - particles[i].x_pred
            current_length = length(delta)
            if current_length == 0:
                continue
            C = current_length - d
            grad_i = -delta / current_length
            grad_j = delta / current_length
            w = (1/particles[i].m) * dot(grad_i, grad_i) + (1/particles[j].m) * dot(grad_j, grad_j)
            alpha_tilde = alpha / (dt * dt)
            delta_lambda = (-C - alpha_tilde * constraint.lambda) / (w + alpha_tilde)
            constraint.lambda += delta_lambda
            delta_x_i = (1/particles[i].m) * grad_i * delta_lambda
            delta_x_j = (1/particles[j].m) * grad_j * delta_lambda
            particles[i].x_pred += delta_x_i
            particles[j].x_pred += delta_x_j

    for i in range(len(particles)):
        particles[i].v = (particles[i].x_pred - particles[i].x) / dt
        particles[i].x = particles[i].x_pred
```

Рисунок 2.13 – Фрагмент коду, що описує моделювання м’яких тіл

Для м’яких тіл модель використовує частинкову модель, де кожна частинка має масу, позицію та швидкість. Обмеження, такі як відстань, забезпечують еластичність, тоді як об’ємні обмеження запобігають колапсу. У кожній ітерації розв’язувач обчислює корекції позицій на основі похибки обмеження та параметра податливості, що дозволяє стабільно моделювати деформації.

2.3.3 Алгоритм моделювання рідин

Симуляція рідин у моделі базується на методі Position-Based Fluids (PBF), який моделює нестискувані рідини через частинковий підхід із обмеженнями щільності. Як зазначається в [3], PBF забезпечує реалістичну поведінку рідин у реальному часі, використовуючи ітеративне розв'язання обмежень для підтримки постійної щільності. Основний цикл симуляції включає передбачення позицій частинок, пошук сусідів, обчислення корекцій позицій і оновлення швидкостей (рис. 2.14).

```
def simulate_fluid(particles, dt, solver_iterations):
    for i in range(len(particles)):
        particles[i].v += dt * f_ext(particles[i].x)
        particles[i].x_star = particles[i].x + dt * particles[i].v

    for i in range(len(particles)):
        particles[i].neighbors = find_neighbors(particles[i].x_star)

    for iter in range(solver_iterations):
        for i in range(len(particles)):
            lambda_i = calculate_lambda(i, particles[i].neighbors)
            particles[i].lambda = lambda_i
        for i in range(len(particles)):
            delta_p_i = calculate_delta_p(i, particles[i].neighbors, particles[i].lambda)
            collision_response(i, delta_p_i)
            particles[i].x_star += delta_p_i

    for i in range(len(particles)):
        particles[i].v = (particles[i].x_star - particles[i].x) / dt
        apply_vorticity_and_viscosity(i)
        particles[i].x = particles[i].x_star
```

Рисунок 2.14 – Фрагмент коду, що описує моделювання рідин

Алгоритм PBF передбачає обчислення множника Лагранжа λ_i для кожної частинки на основі похибки щільності, а потім застосування корекцій позицій Δp_i для забезпечення нестискуваності. Додаткові ефекти, такі як вихрове стиснення та XSPH-в'язкість, додають реалістичності поведінці рідини [20].

2.3.4 Загальна обробка зіткнень

Обробка зіткнень у системі здійснюється через двофазний підхід, який включає широку та вузьку фази виявлення зіткнень. У широкій фазі використовуються просторові структури, такі як AABB, для швидкого визначення потенційних пар зіткнень, як описано в [11]. У вузькій фазі проводяться точні тести для визначення точок контакту, нормалей і глибини проникнення.

Для XPBD зіткнення обробляються через створення контактних обмежень, які розв'язуються в рамках основного циклу симуляції. Наприклад, для твердих тіл контактні обмеження коригують позиції та орієнтації, щоб запобігти проникненню. Для м'яких тіл можуть застосовуватися аналогічні обмеження або методи штрафів. У PBF зіткнення обробляються в рамках корекції позицій частинок, де частинки відштовхуються від твердих меж або інших об'єктів, як зазначено в [3].

2.3.5 Поєднання різних типів тіл

Поєднання твердих, м'яких тіл і рідин у єдиній симуляції є завданням, яке вимагає уніфікованого підходу до розв'язання обмежень. У даній системі це досягається через інтеграцію різних типів тіл у спільний цикл симуляції, де обмеження, що зв'язують їхні ступені свободи, розв'язуються одночасно. Наприклад, для взаємодії між твердими і м'якими тілами використовуються обмеження прикріплення або контактні обмеження, які забезпечують коректну передачу сил, як описано в [2].

Для взаємодії рідин із твердими тілами PBF застосовує граничні умови, де частинки рідини відштовхуються від поверхонь твердих тіл. Для рідин і м'яких тіл можуть використовуватися частинкові взаємодії або спеціальні обмеження, що враховують щільність і деформацію, проте в даній роботі це не використовується.

3 РЕЗУЛЬТАТИ МОДЕЛЮВАННЯ

Цей розділ присвячений аналізу результатів тестування та моделювання сцен, описаних у попередніх розділах, а саме демонстрацій поєднання твердих та м’яких тіл та рідини на основі PBF. Результати базуються на алгоритмах бібліотеки PositionBasedDynamics, яка є відкритим програмним забезпеченням і розроблена Computer Animation Group з RWTH Aachen University [21]. У розділі представлено оцінку поведінки об’єктів у симуляціях, вплив ключових параметрів на динаміку та стабільність, а також порівняння отриманих результатів із очікуваною поведінкою фізичних систем. Дані отримані на основі проведених експериментів із різними налаштуваннями, що дозволяє зробити висновки про ефективність застосованих методів та можливі напрями подальшого дослідження.

3.1 Тканина та тверде тіло

Сцена колізії тканини з твердим тілом (торусом) демонструє інтерактивну симуляцію взаємодії м’якої тканини з нерухомим об’єктом, реалізовану за допомогою методу XPBD. Ця сцена ілюструє поведінку тканини, представленої у вигляді сітки частинок, при її контакті з геометрією торуса, що виступає статичним об’єктом. Симуляція відображає реалістичне падіння тканини під дією гравітації та деформацію при зіткненні з поверхнею торуса.

Сцена починається з плоскої квадратної тканини, розміщеної над торусом, із заданими розмірами 10×10 одиниць (ширина $\times$ довжина) та роздільною здатністю 50×50 частинок. Спочатку тканина розташована на висоті 4 одиниці над основною площиною, з кутами, закріпленими під кутом $90°$ для імітації натягу (рис. 3.1).

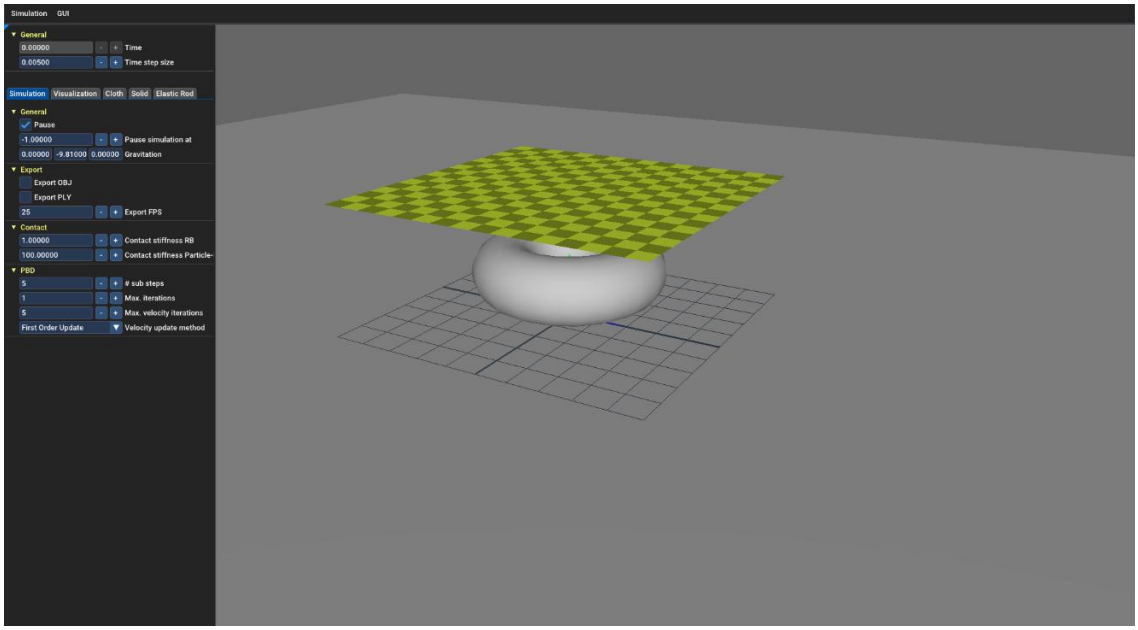

Рисунок 3.1 – Початкове розташування тканини над торусом

Під дією гравітації, що спрямована вниз із прискоренням 9.81 м/с², тканина поступово опускається, вступаючи в контакт із торусом, розташованим у позиції $(0.0, 1.5, 0.0)$ із радіусом 1.0 метри. Цей процес призводить до реалістичної деформації тканини, яка обволікає поверхню торуса, утворюючи природні складки та драпірування, що відображає фізичну поведінку текстильних матеріалів (рис. 3.2).

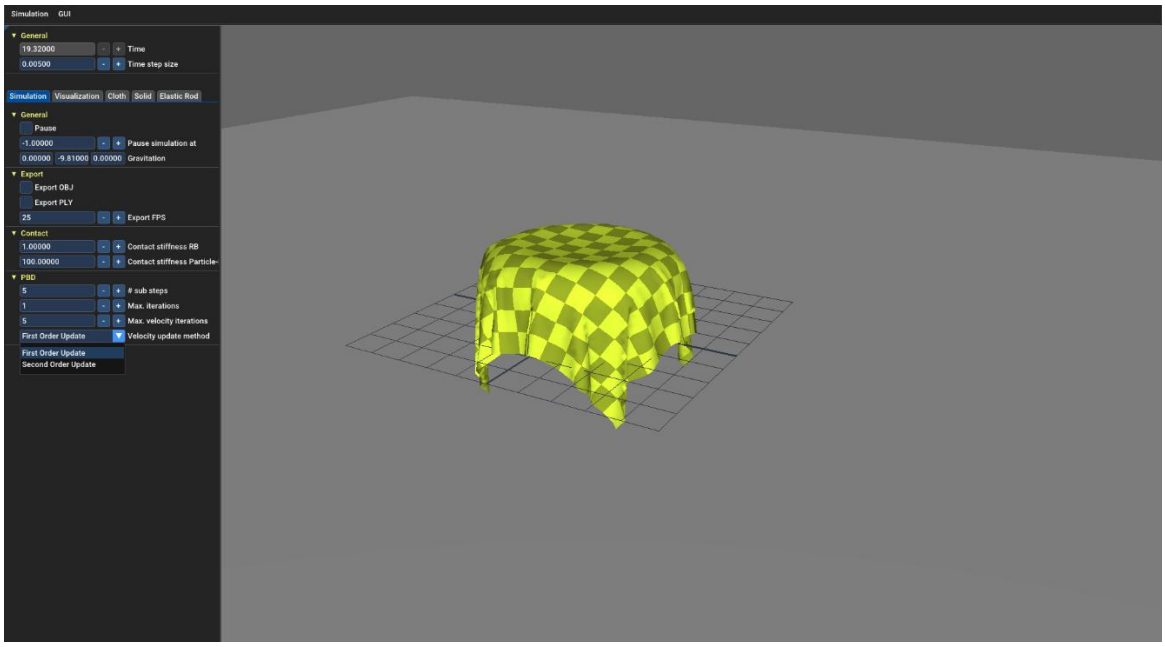

Рисунок 3.2 – Деформація тканини при контакті з торусом

Основними параметрами, що визначають поведінку тканини, є її гнучкість і сила взаємодії з об’єктами сцени. Сітка тканини складається з 2500 частинок із радіусом 0.075 метри, що забезпечує деталізоване моделювання зморшок і складок (рис. 3.3).

Жорсткість тканини регулюється коефіцієнтом відстані, встановленим на рівні 1.0, що імітує еластичність типового текстилю, тоді як метод ізометричного згинання (*isometric bending*) із жорсткістю 0.01 додає реалістичність формуванню вигинів (рис. 3.3). Контакт із торусом характеризується коефіцієнтом тертя 0.1, який дозволяє тканині плавно ковзати по поверхні, а жорсткість контакту, встановлена на 100, забезпечує стабільну взаємодію без проникаючих ефектів. Якщо ж встановити останній параметр замалим, це призведе до нереалістичного проникнення всередину торуса (рис. 3.5) з подальшим падінням на поверхню симуляції.

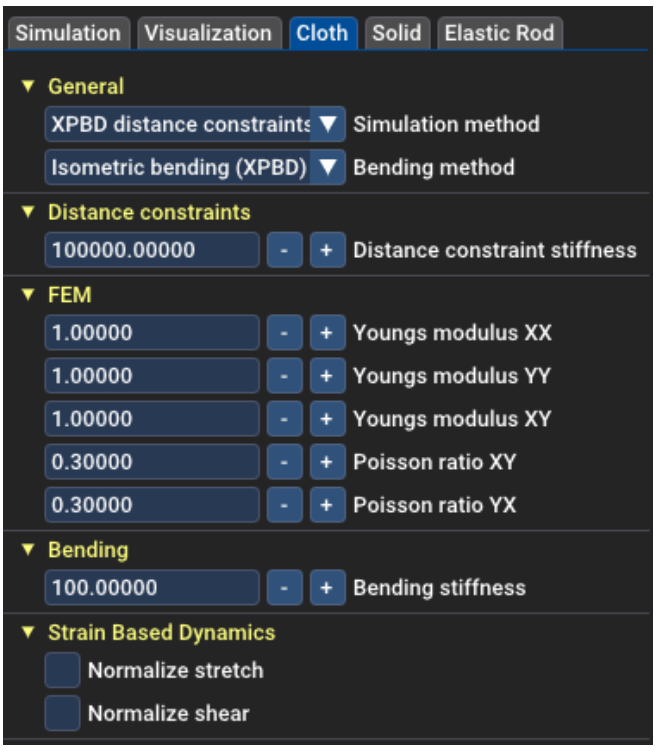

Рисунок 3.3 – Меню налаштування параметрів тканини

Часовий крок 0.005 секунди гарантує плавність анімації, тоді як 5 ітерацій на крок часу підтримують числову стабільність симуляції (рис. 3.4).

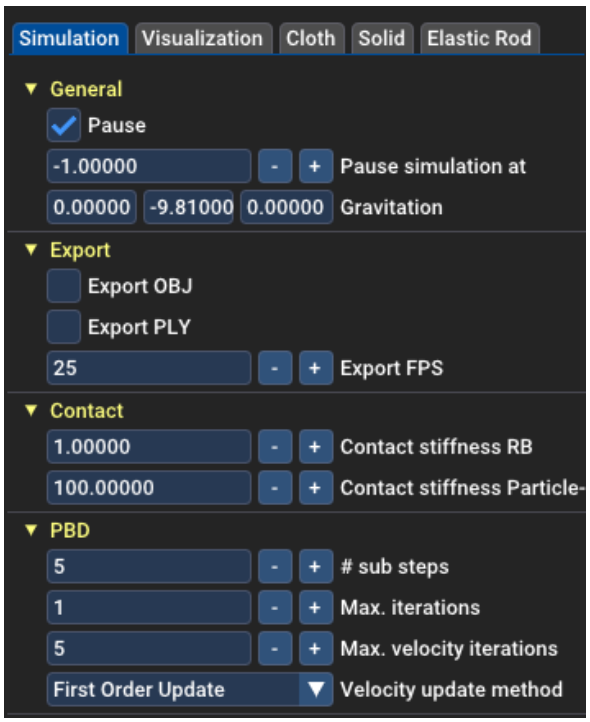

Рисунок 3.4 – Меню налаштування параметрів симуляції

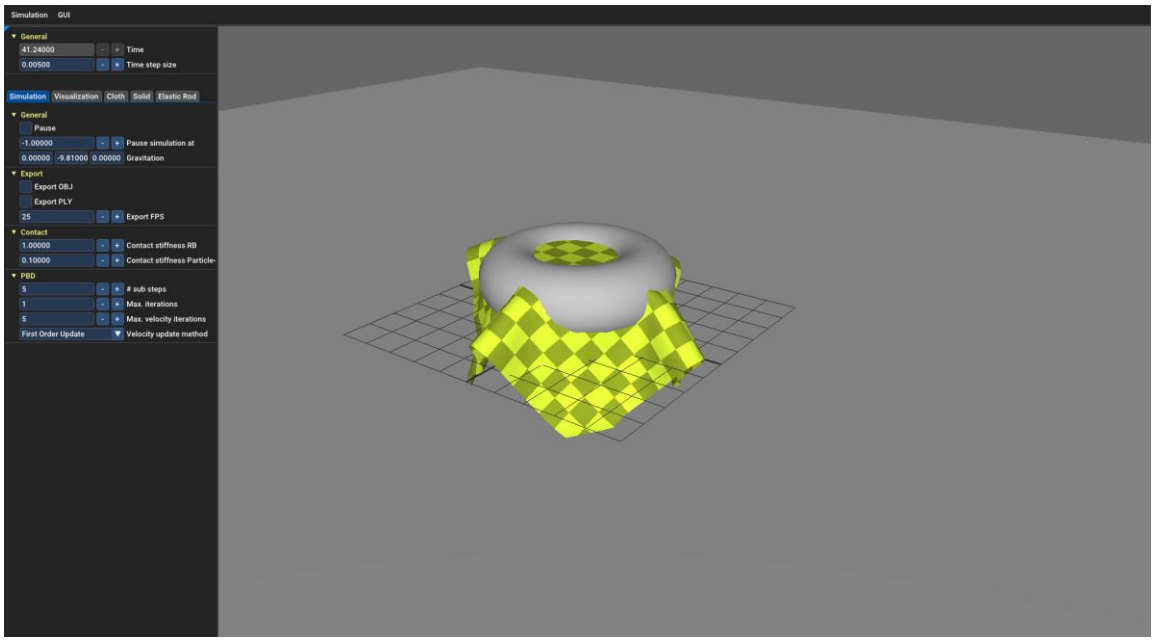

Рисунок 3.5 – Проникнення тканини через торус при низькій жорсткості контактів з твердими тілами

Додаткові візуальні елементи, такі як дротові каркаси (*wireframe*) та контактні точки, дозволяють аналізувати динаміку взаємодії, роблячи сцену корисною для вивчення поведінки м’яких тіл. (рис. 3.6).

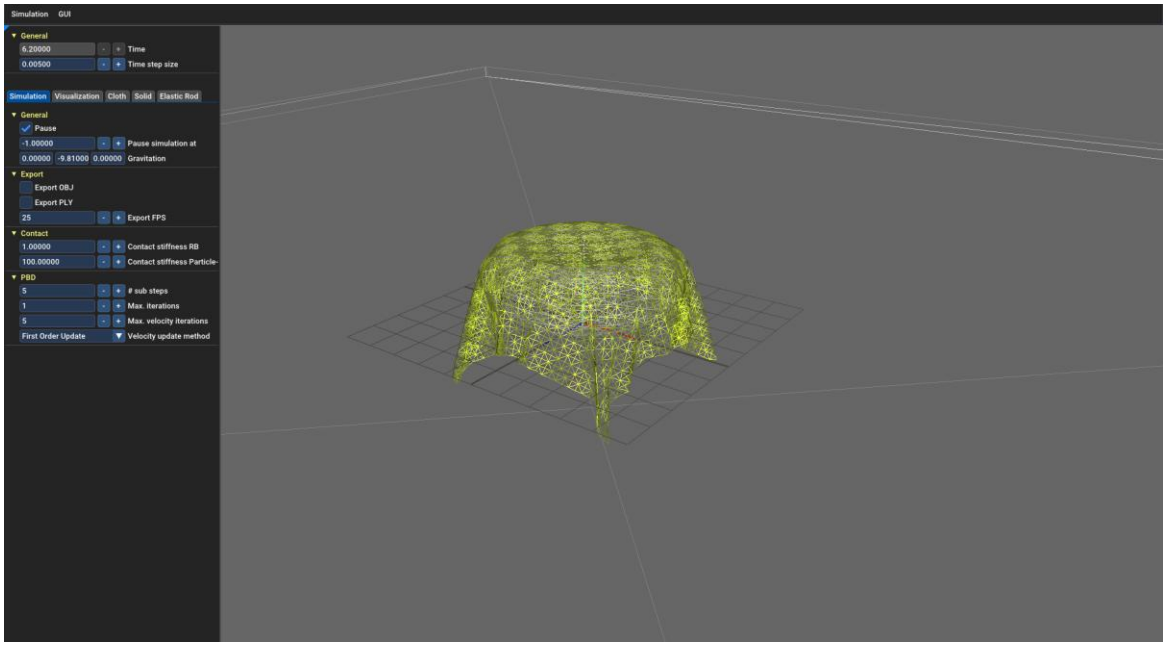

Рисунок 3.6 – Візуалізація контактних точок і каркасу тканини

Сцена колізії тканини з торусом успішно демонструє інтегровану модель симуляції м’яких та твердих тіл із використанням XPBD. Налаштування параметрів дозволяють адаптувати поведінку тканини до різних сценаріїв, що робить цю модель придатною для застосувань у комп’ютерній графіці та віртуальній реальності.

3.2 Шарніри

Дана сцена представляє інтерактивну симуляцію поведінки жорстких тіл, з’єднаних різними типами шарнірів, під впливом гравітації та анімаційних рухів. У початковому стані сцена включає 11 пар кубів, де кожен динамічний стрижень із розмірами $0.4 \times 0.4 \times 2.0$ метрів з’єднаний із статичним стрижнем розміром $0.5 \times 0.5 \times 0.5$ метрів за допомогою кульових, шарнірних, універсальних та моторизованих шарнірів (рис 3.7).

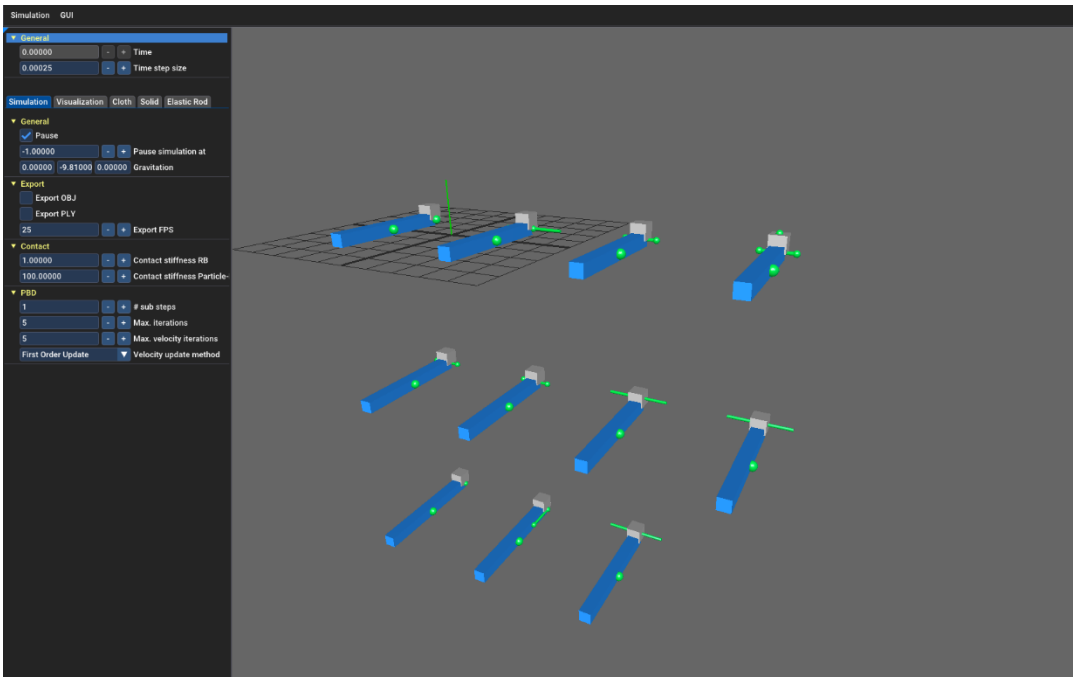

Рисунок 3.7 – Початкове розташування стрижнів із шарнірами

Ці об’єкти розташовані вертикально по осі Y від 6.5 до -5.5 одиниць із кроком 5.5 одиниць і розподілені по осі X від 0.0 до 48.0 одиниць.

Під дією гравітації, спрямованій вниз із прискоренням 9.81 м/с², динамічні стрижні падають, а моторизовані шарніри, такі як TargetAngleMotor і TargetPositionMotor, додають періодичні обертальні та лінійні рухи, що імітують механічні системи (рис. 3.8).

Основними параметрами, що визначають поведінку тіл і шарнірів, є їхня маса, інерція та типи з’єднань. Кожен динамічний стрижень має масу 1.0 із інерційним тензором, обчисленим на основі розмірів, що забезпечує реалістичне обертання під час падіння. Моторизовані шарніри створюють анімацію: TargetAngleMotor обертає стрижні з частотою 0.25 Гц в межах $\pm\pi/4$, тоді як TargetPositionMotor забезпечує зміщення з частотою 2.0 Гц в межах ± 1.5 метрів.

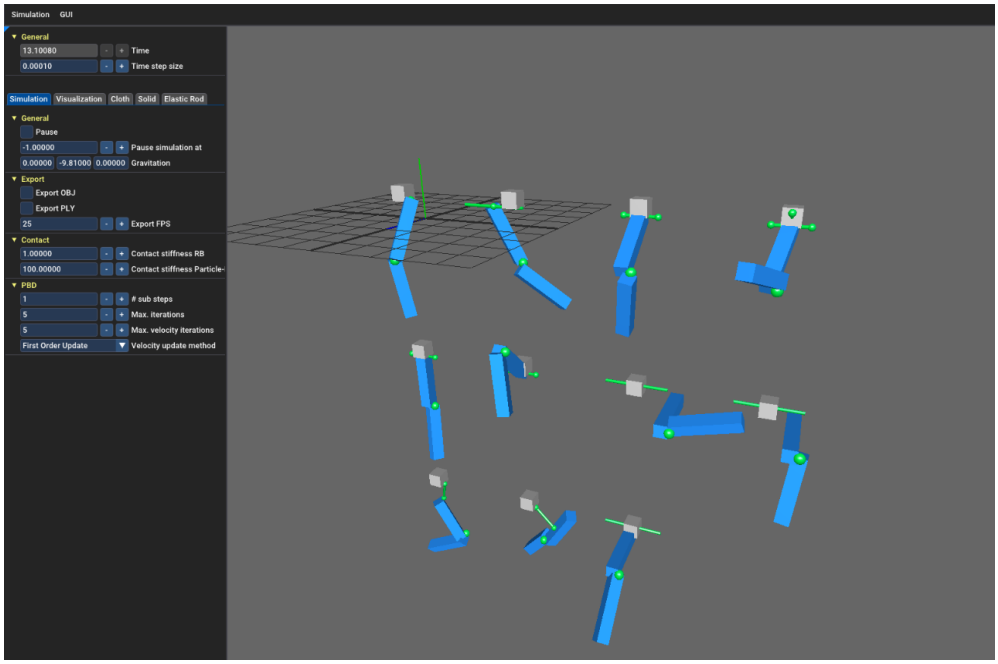

Рисунок 3.8 – Динамічний рух стрижнів під дією моторизованих шарнірів

Часовий крок 0.001 секунди гарантує плавність рухів, а 5 ітерацій на крок часу стабілізують симуляцію, уникаючи артефактів. Статичні стрижні, з масою 0.0, слугують як точки кріплення, дозволяючи шарнірам визначати свободу рухів.

Пунтки налаштування симуляції залишаються такими ж самими, як і в попередньому випадку моделювання взаємодії тканини та твердого тіла. Єдине, що не було згадано в попередньому підрозділі, так це пункт меню «Solid», який дозволяє змінювати параметри, що стосуються моделювання твердих тіл, такі як методи симуляції, жорсткість обмеження об’єму та інші (рис. 3.9).

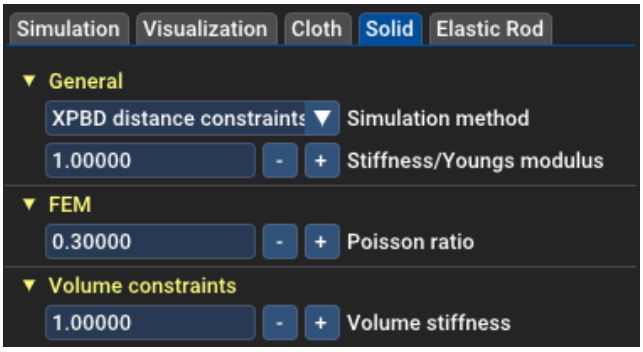

Рисунок 3.9 – Меню налаштування параметрів твердих тіл

Як і в усіх інших випадках, є можливість зобразити дротові каркаси та контактні точки для більш наочної візуалізації (рис. 3.10).

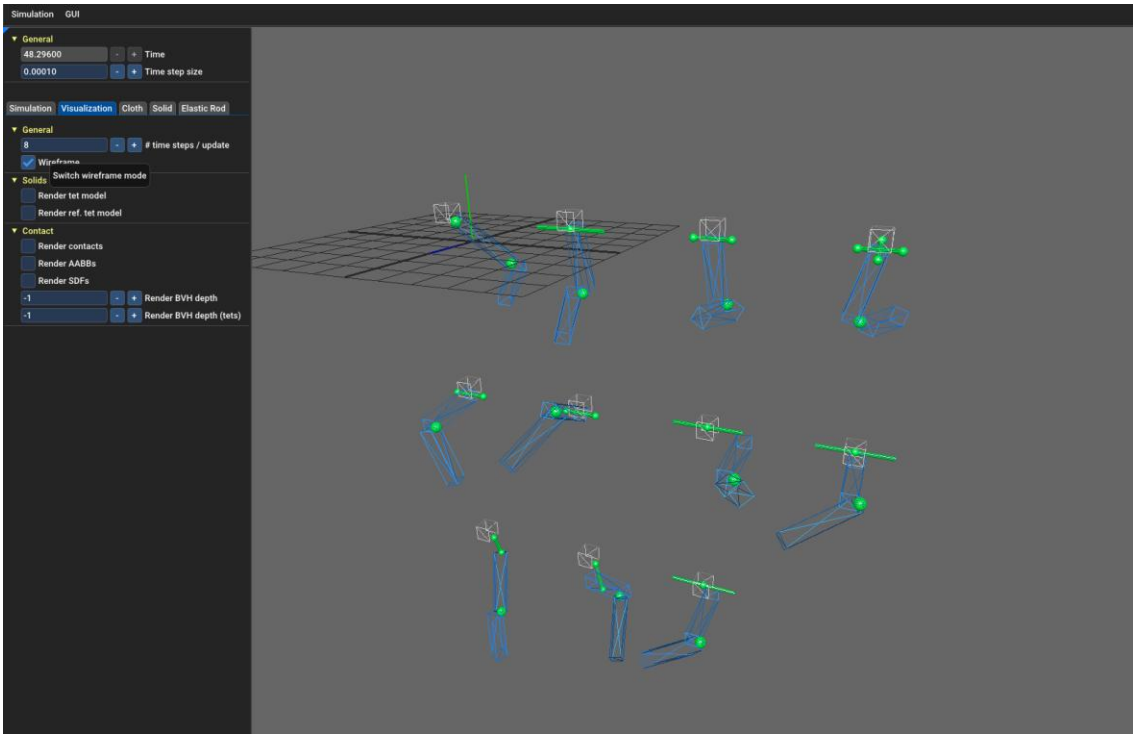

Рисунок 3.10 – Дротові каркаси та точки з’єднання шарнірів

Візуалізація сцени підкреслює динаміку завдяки чіткому рендерингу стрижнів у блакитному кольорі з виділенням осей і точок з’єднання, що полегшує аналіз взаємодії. Дротові каркаси та контактні точки відображаються для детального вивчення поведінки суглобів. Сцена ефективно демонструє різноманітність механічних рухів, від простого падіння до складних анімацій, що робить її придатною для вивчення динаміки жорстких тіл.

3.3 Ланцюг та кролик

Наступна сцена демонструє симуляцію гнучкого ланцюга, складеного з жорстких тіл, з’єднаних кульовими шарнірами, під дією гравітації. У

початковому стані ланцюг, утворений 10 сегментами розміром $1.0 \times 0.1 \times 0.1$ метрів, розміщується горизонтально уздовж осі X із кроком 1.0 одиниці, починаючи з позиції $(0.0, 0.0, 0.0)$, де перший сегмент є статичним, а решта — динамічними з масою, обчисленою на основі густини 1.0 (рис. 3.11).

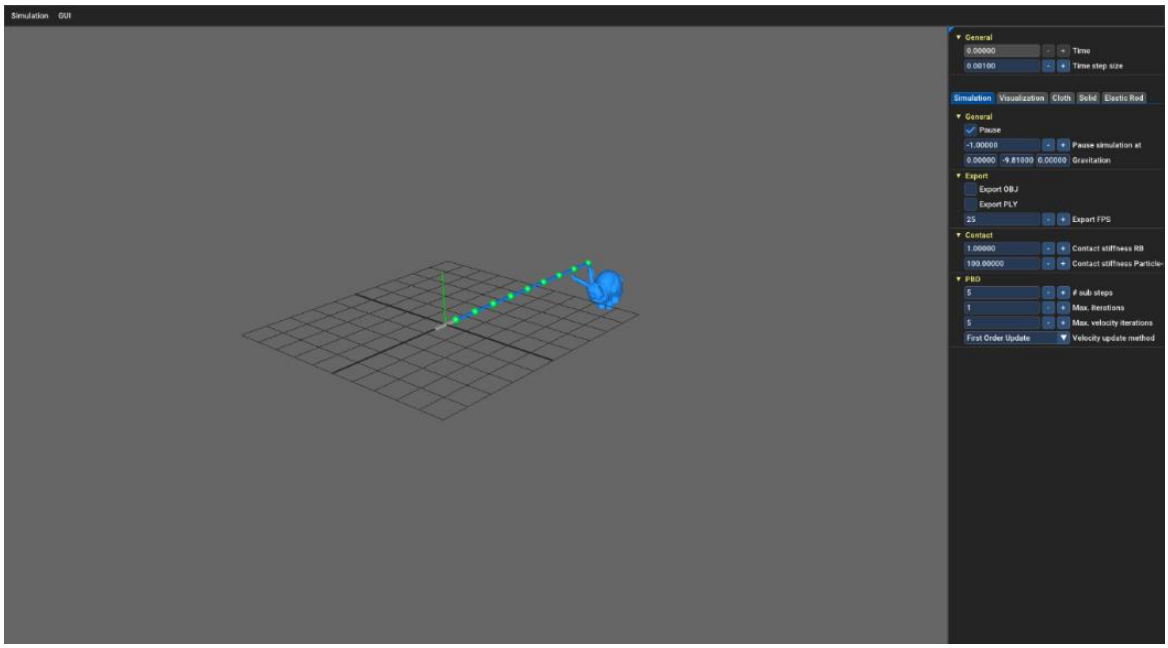

Рисунок 3.11 – Початкове розташування ланцюга з кроликом

Під впливом гравітації, спрямованій вниз із прискоренням 9.81 м/с², ланцюг опускається, утворюючи природну дугу, тоді як кінцевий сегмент, представлений 3D-моделлю кролика (масштаб $2.0 \times 2.0 \times 2.0$), з'єднаний із дев'ятим сегментом через кульовий шарнір і повторює рух, додаючи візуальної складності. Через деякий (доволі довгий) час ланцюг стабілізується, імітуючи поведінку гнучкого об'єкта (рис. 3.12).

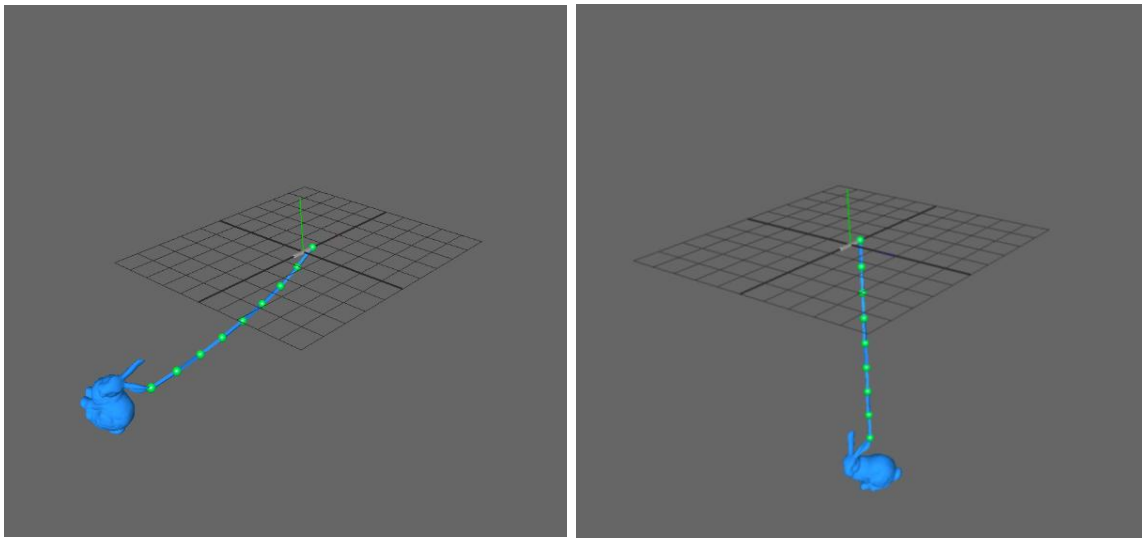

Рисунок 3.12 – Крайнє положення та положення рівноваги ланцюга

Основними параметрами, що визначають динаміку ланцюга, є його сегменти та взаємодія з гравітацією. Кожен динамічний сегмент має масу, пропорційну густині 1.0, що забезпечує реалістичне провисання, тоді як статичний сегмент слугує як точка кріплення. Кролик, завантажений із файлу `bunny_10k.obj` і розташований із початковою позицією $(0.411 + 9.0, -1.776, 0.356)$ із кутом обертання $\pi/6$ навколо осі Z, додає унікальний візуальний елемент, що сприяє природному падінню та стабілізації.

3.4 Рідина на основі PBF

Рідина на основі методу PBF (Position-Based Fluids Demo) в даному фреймворку симулюється автономно, зосереджуючись виключно на внутрішній динаміці частинок рідини під впливом гравітації, без взаємодії з іншими тілами.

Сцена розпочинається з компактного блоку, утвореного 4500 частинками з радіусом 0.025 одиниці, розміщеним у контейнері розміром $7.5 \times 4.0 \times 1.5$ метри (рис. 3.13), і еволюціонує через етапи розбризкування (рис. 3.14) та падіння під дією гравітації $(0.0, -9.81, 0.0)$, завершуючись стабілізацією на дні (рис. 3.15).

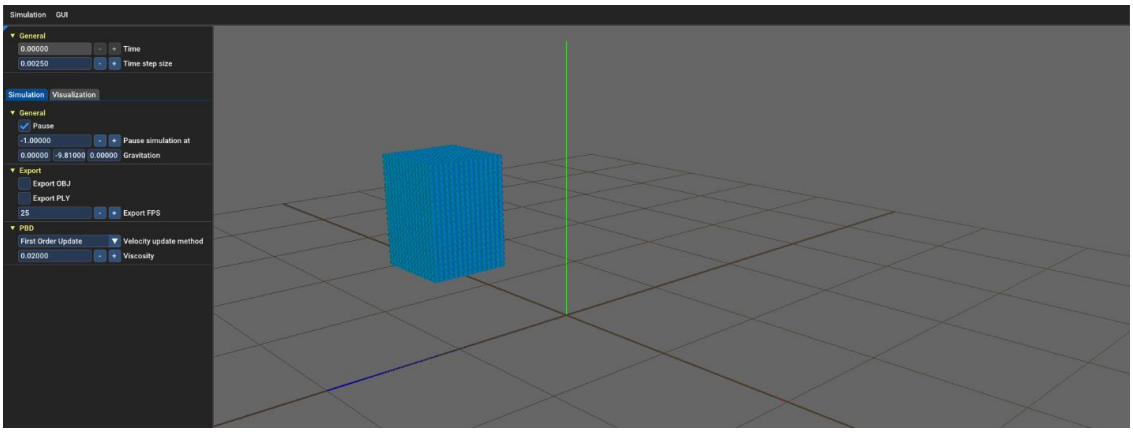

Рисунок 3.13 – Початковий блок рідини в контейнері

Цей процес, який включає обчислення густини, проекцію обмежень і корекцію швидкостей, імітує природну поведінку рідини, де в’язкість сприяє згладжуванню рухів, а межі контейнера лише задають геометричні рамки.

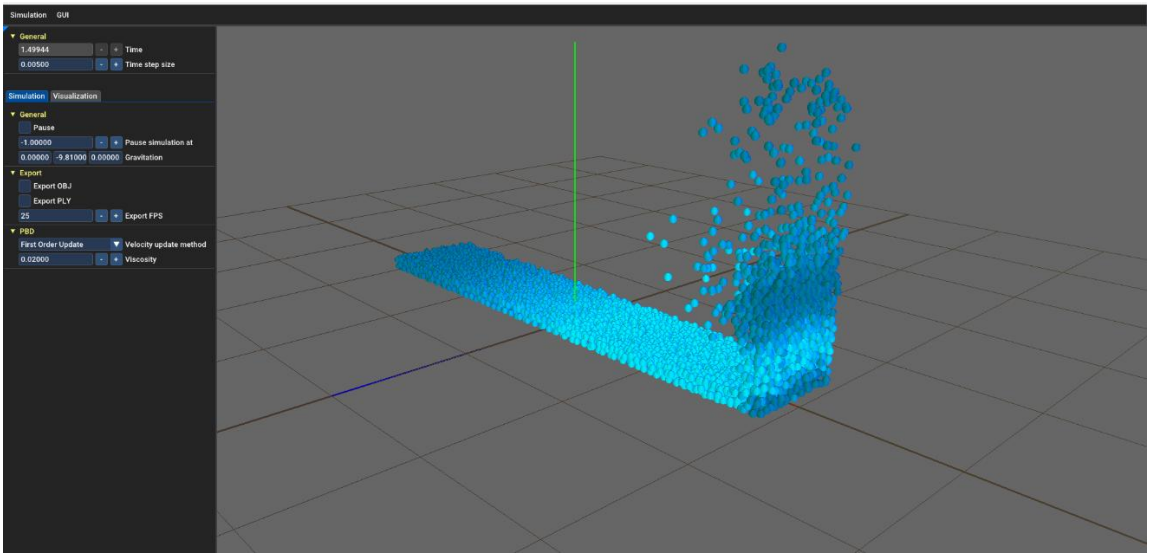

Рисунок 3.14 – Розбризкування рідини під дією гравітації

Основними параметрами, що визначають динаміку рідини, є її густота, в’язкість та розмір частинок. Густота встановлена на рівні 1000.0 кг/м³, а маса кожної частинки обчислюється як $0.8 \times (2r)^3 \times \rho_0$, відображаючи об’єм кубічного сегмента.

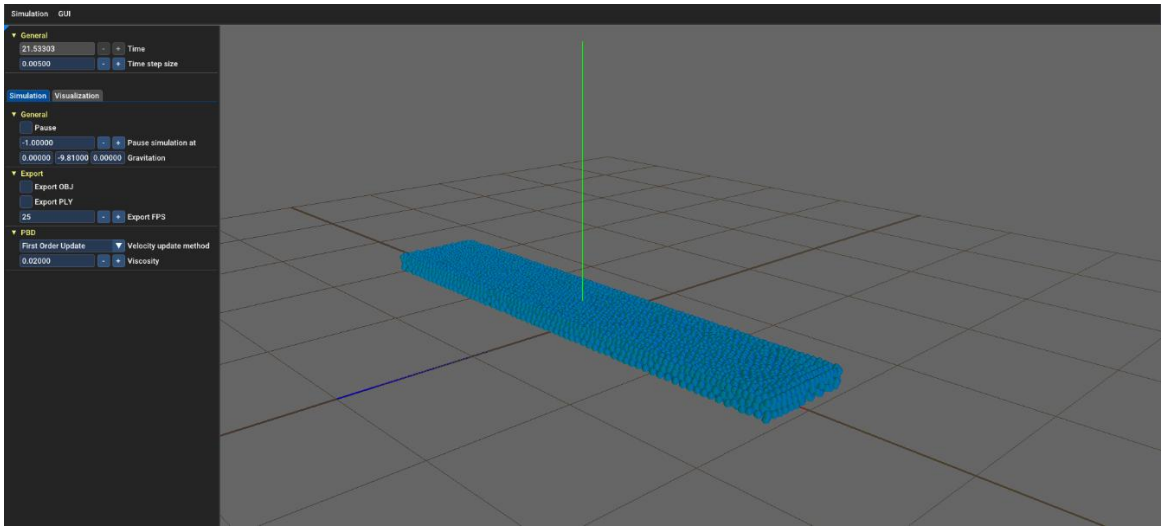

Рисунок 3.15 – Стабілізація рідини на дні контейнера

В’язкість, початково задана як 0.02, регулює текучість, тоді як часовий крок, що коливається від 0.0025 до 0.005 секунд, забезпечує плавність анімації. Якщо встановити занадто велику в’язкість (напр., 10), симуляція стане нестабільною та частинки просто розлітяться у різні боки (рис. 3.16).

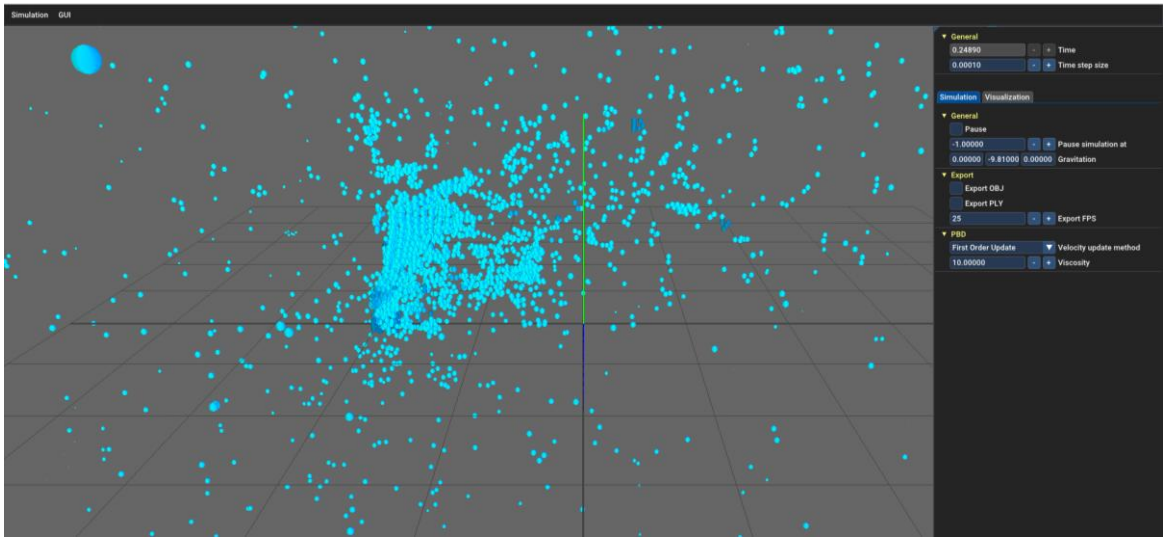

Рисунок 3.16 – Нестабільність симуляції при високій в’язкості

Наведена сцена ефективно демонструє можливості PBF у моделюванні ізольованих рідинних систем, і подальші дослідження в’язкості та розміру частинок можуть поглибити розуміння їхнього впливу на поведінку самої рідини та її вплив на інші тіла.

ВИСНОВКИ

У кваліфікаційній роботі досягнуто мету – створення інтегрованої моделі на основі XPBD і PBF для моделювання твердих тіл, м’яких тіл і рідин у реальному часі. Робота демонструє значний потенціал для застосування в комп’ютерній графіці, ігровій індустрії та системах віртуальної реальності, де висока продуктивність і реалістичність є ключовими вимогами. Теоретичний аналіз підтвердив переваги XPBD, зокрема його незалежність від часового кроку та точність у моделюванні еластичних властивостей завдяки параметру податливості, а також ефективність PBF у забезпеченні нестискуваності рідин через корекцію щільності. Ці методи виявилися значно ефективнішими порівняно з традиційними підходами, такими як FEM, які, хоча й точні, не адаптовані до інтерактивних застосувань.

Практична частина роботи включає розробку архітектури імітаційної моделі інтегрованого моделювання та реалізацію алгоритмів. Для твердих тіл використано XPBD із субкроками для оновлення позицій і орієнтацій, а для м’яких тіл – ітеративне розв’язання відстаневих, згинальних і об’ємних обмежень. PBF реалізовано для рідин із механізмами корекції позицій і обробки зіткнень. Побудована модель дозволяє поєднувати ці типи тіл у єдиному циклі симуляції, хоча повна взаємодія між рідинами та іншими типами тіл залишається нереалізованою.

Експериментальні результати, отримані на чотирьох тестових сценах підтвердили високу стабільність моделі при оптимальних параметрах. Аналіз показав, що правильне налаштування цих параметрів забезпечує реалістичну поведінку об’єктів, хоча виявлено й певні недоліки. Зокрема, штучна податливість XPBD проявляється при низькій кількості ітерацій, а точність моделі поступається офлайн-методам. Практичне значення роботи полягає в можливості її використання для розробки ігрових рушіїв, графічних застосунків і систем віртуальної реальності, де реалістична симуляція фізичних взаємодій є визначальною.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. M. Müller, B. Heidelberger, M. Hennix, and J. Ratcliff, “Position Based Dynamics,” *Journal of Visual Communication and Image Representation*, vol. 18, no. 2, pp. 109–118, Apr. 2007, doi: 10.1016/j.jvcir.2007.01.005.
2. M. Macklin, M. Müller, and N. Chentanez, “XPBD: Position-Based Simulation of Compliant Constrained Dynamics,” in *Proceedings of the 9th International Conference on Motion in Games*, in MIG ’16. New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 49–54. doi: 10.1145/2994258.2994272.
3. M. Macklin and M. Müller, “Position Based Fluids,” *ACM Trans. Graph.*, vol. 32, no. 4, p. 104:1-104:12, Jul. 2013, doi: 10.1145/2461912.2461984.
4. J. Bender, M. Müller, and M. Macklin, “A Survey on Position Based Dynamics,” in *Proceedings of the European Association for Computer Graphics: Tutorials*, in EG ’17. Goslar, DEU: Eurographics Association, Apr. 2017, pp. 1–31. doi: 10.2312/egt.20171034.
5. M. Müller, D. Charypar, and M. Gross, “Particle-Based Fluid Simulation for Interactive Applications,” in *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, in SCA ’03. Goslar, DEU: Eurographics Association, Jul. 2003, pp. 154–159.
6. M. Müller, J. Stam, D. James, and N. Thürey, “Real Time Physics: Class Notes,” in *ACM SIGGRAPH 2008 classes*, in SIGGRAPH ’08. New York, NY, USA: Association for Computing Machinery, Aug. 2008, pp. 1–90. doi: 10.1145/1401132.1401245.
7. A. Nealen, M. Müller, R. Keiser, E. Boxerman, and M. Carlson, “Physically Based Deformable Models in Computer Graphics,” *Computer Graphics Forum*, vol. 25, no. 4, pp. 809–836, 2006, doi: 10.1111/j.1467-8659.2006.01000.x.

8. M. Macklin, M. Müller, N. Chentanez, and T.-Y. Kim, “Unified Particle Physics for Real-Time Applications,” *ACM Trans. Graph.*, vol. 33, no. 4, p. 153:1-153:12, Jul. 2014, doi: 10.1145/2601097.2601152.

9. C. Yu, X. Li, L. Lan, Y. Yang, and C. Jiang, “XPBI: Position-Based Dynamics with Smoothing Kernels Handles Continuum Inelasticity,” in *SIGGRAPH Asia 2024 Conference Papers*, Dec. 2024, pp. 1–12. doi: 10.1145/3680528.3687577.

10. T. Stuyck and H. Chen, “DiffXPBD : Differentiable Position-Based Simulation of Compliant Constraint Dynamics,” Jun. 30, 2023, *arXiv*: arXiv:2301.01396. doi: 10.48550/arXiv.2301.01396.

11. M. Müller, M. Macklin, N. Chentanez, S. Jeschke, and T.-Y. Kim, “Detailed Rigid Body Simulation with Extended Position Based Dynamics,” *Computer Graphics Forum*, vol. 39, no. 8, pp. 101–112, 2020, doi: 10.1111/cgf.14105.

12. E. G. Gilbert, D. W. Johnson, and S. S. Keerthi, “A Fast Procedure for Computing the Distance Between Complex Objects in Three-Dimensional Space,” *IEEE Journal on Robotics and Automation*, vol. 4, no. 2, pp. 193–203, Apr. 1988, doi: 10.1109/56.2083.

13. M. Lin, J. Cohen, and D. Manocha, “Collision Detection: Algorithms and Applications,” *Algorithms for Robotic Motion and Manipulation*, 1997.

14. N. Akinci, M. Ihmsen, G. Akinci, B. Solenthaler, and M. Teschner, “Versatile Rigid-Fluid Coupling for Incompressible SPH,” *ACM Trans. Graph.*, vol. 31, no. 4, p. 62:1-62:8, Jul. 2012, doi: 10.1145/2185520.2185558.

15. M. Köster and A. Krüger, “Adaptive Position-Based Fluids: Improving Performance of Fluid Simulations for Real-Time Applications,” *IJCGA*, vol. 6, no. 3, pp. 01–16, Jun. 2016, doi: 10.5121/ijcga.2016.6301.

16. R. Fedkiw, J. Stam, and H. W. Jensen, “Visual Simulation of Smoke,” in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, in SIGGRAPH ’01. New York, NY, USA: Association for Computing Machinery, Aug. 2001, pp. 15–22. doi: 10.1145/383259.383260.

17. J. Yu and G. Turk, “Reconstructing Surfaces of Particle-Based Fluids Using Anisotropic Kernels,” *ACM Trans. Graph.*, vol. 32, no. 1, p. 5:1-5:12, Feb. 2013, doi: 10.1145/2421636.2421641.

18. W. J. van der Laan, S. Green, and M. Sainz, “Screen Space Fluid Rendering with Curvature Flow,” in *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, in I3D ’09. New York, NY, USA: Association for Computing Machinery, Feb. 2009, pp. 91–98. doi: 10.1145/1507149.1507164.

19. C. Keenan, L. Ignacio, and T. Sarah, “Real-Time Simulation and Rendering of 3D Fluids,” in *GPU Gems 3*, Addison-Wesley, 2008, pp. 633–675.

20. M. Ozbulut, M. Yildiz, and O. Goren, “A Numerical Investigation Into the Correction Algorithms for SPH Method in Modeling Violent Free Surface Flows,” *International Journal of Mechanical Sciences*, vol. 79, pp. 56–65, Feb. 2014, doi: 10.1016/j.ijmecsci.2013.11.021.

21. J. Bender, *PositionBasedDynamics Library*. C++, Python. [Online]. Available: https://github.com/InteractiveComputerGraphics/PositionBasedDynamics

ДОДАТОК А

Клас Engine

```
#include "Window.h"
#include "Renderer.h"
#include "Scene.h"
#include <PBD/Simulation.h>

class Engine {
public:
    Engine();
    ~Engine();

    bool Initialize();
    void Run();
    void Shutdown() {}

private:
    void Update(float deltaTime) { scene->Simulate(deltaTime); }

    std::unique_ptr<Window> window;
    std::unique_ptr<Renderer> renderer;
    std::unique_ptr<Scene> scene;
    PBD::Simulation* simulation;
};

inline Engine::Engine() : simulation(new PBD::Simulation()) {}

inline Engine::~Engine() {
    delete simulation;
}

inline bool Engine::Initialize() {
    window = std::make_unique<Window>();
    if (!window->Init("Engine", 1280, 800)) {
        return false;
    }

    renderer = std::make_unique<Renderer>();
    if (!renderer->Init()) {
        return false;
    }

    scene = std::make_unique<Scene>("test");
    simulation->init();
    return true;
}


inline void Engine::Run() {
    float deltaTime = 1.0f / 60.0f;
    while (window->IsOpen()) {
        Update(deltaTime);
        renderer->Draw(*scene);
        window->SwapBuffers();
        deltaTime = window->GetDeltaTime();
    }
}
```

ДОДАТОК Б

Клас Window

```
#include <iostream>
#include <SDL3/SDL.h>
#include <glad/glad.h>
#include <functional>
#include "Common.h"

class Window {
public:
    Window();
    ~Window();

    bool Init(const char* title, int width, int height);
    bool IsOpen() const;
    void SwapBuffers();
    float GetDeltaTime();
    void GetMousePosition(int& x, int& y) const;
    void GetWindowSize(int& width, int& height) const;
    int GetWindowWidth() const;
    int GetWindowHeight() const;
    float GetAspectRatio() const;
    SDL_Window* GetWindow();
    void SetResizeCallback(std::function<void(int, int)> callback);

private:
    SDL_Window* window;
    SDL_GLContext glContext;
    bool isRunning;
    const char* title;
    int width;
    int height;
    int mouseX;
    int mouseY;
    std::function<void(int, int)> resizeCallback;

    static bool ResizeEventWatcher(void* userdata, SDL_Event* event);
};

inline Window::Window()
  : window(nullptr),
    glContext(nullptr),
    isRunning(true),
    title("Engine"),
    width(1280),
    height(800),
    mouseX(0),
    mouseY(0) {}

inline Window::~Window() {
    SDL_GL_DestroyContext(glContext);
    SDL_DestroyWindow(window);
    SDL_Quit();
}
```

Лістинг Б – Клас Window, аркуш 1

```
inline bool Window::Init(const char* title, int width, int height) {
    this->title = title;
    this->width = width;
    this->height = height;

    if (!SDL_Init(SDL_INIT_VIDEO)) {
        std::cerr << "SDL could not initialize! SDL_Error: " <<
SDL_GetError() << std::endl;
        return false;
    }

    SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 4);
    SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 6);
    SDL_GL_SetAttribute(SDL_GL_CONTEXT_PROFILE_MASK,
SDL_GL_CONTEXT_PROFILE_CORE);
    SDL_GL_SetAttribute(SDL_GL_MULTISAMPLEBUFFERS, 1);
    SDL_GL_SetAttribute(SDL_GL_MULTISAMPLESAMPLES, 4);
    SDL_GL_SetAttribute(SDL_GL_CONTEXT_FLAGS, SDL_GL_CONTEXT_DEBUG_FLAG);

    window = SDL_CreateWindow(title, width, height, SDL_WINDOW_OPENGL);
    if (!window) {
        std::cerr << "Window could not be created! SDL_Error: " <<
SDL_GetError() << std::endl;
        return false;
    }

    glContext = SDL_GL_CreateContext(window);
    if (!glContext) {
        std::cerr << "OpenGL context could not be created! SDL_Error: " <<
SDL_GetError() << std::endl;
        return false;
    }

    SDL_AddEventWatch(ResizeEventWatcher, this);
    return true;
}

inline bool Window::IsOpen() const {
    return isRunning;
}

inline void Window::SwapBuffers() {
    SDL_GL_SwapWindow(window);
}

inline float Window::GetDeltaTime() {
    static Uint64 lastTime = SDL_GetTicks64();
    Uint64 currentTime = SDL_GetTicks64();
    float deltaTime = (currentTime - lastTime) / 1000.0f;
    lastTime = currentTime;
    return deltaTime;
}

inline void Window::GetMousePosition(int& x, int& y) const {
    x = mouseX;
    y = mouseY;
}
```

Лістинг Б – Клас Window, аркуш 2

```
inline void Window::GetWindowSize(int& width, int& height) const {
    width = this->width;
    height = this->height;
}

inline int Window::GetWindowWidth() const {
    return width;
}

inline int Window::GetWindowHeight() const {
    return height;
}

inline float Window::GetAspectRatio() const {
    return static_cast<float>(width) / height;
}

inline SDL_Window* Window::GetWindow() {
    return window;
}

inline void Window::SetResizeCallback(std::function<void(int, int)> callback)
{
    resizeCallback = callback;
}

inline bool Window::ResizeEventWatcher(void* userdata, SDL_Event* event) {
    Window* window = static_cast<Window*>(userdata);
    if (event->type == SDL_EVENT_WINDOW_RESIZED) {
        int newWidth = event->window.data1;
        int newHeight = event->window.data2;
        window->width = newWidth;
        window->height = newHeight;
        if (window->resizeCallback) {
            window->resizeCallback(newWidth, newHeight);
        }
    } else if (event->type == SDL_EVENT_QUIT) {
        window->isRunning = false;
    } else if (event->type == SDL_EVENT_MOUSE_MOTION) {
        window->mouseX = event->motion.x;
        window->mouseY = event->motion.y;
    }
    return true;
}
```

Лістинг Б – Клас Window, аркуш 3

ДОДАТОК В

Клас Renderer

```
#include <iostream>
#include <unordered_map>
#include <SDL3/SDL.h>
#include <glad/glad.h>
#include "Scene.h"
#include "Shader.h"

void APIENTRY glDebugOutput(
    GLenum source,
    GLenum type,
    unsigned int id,
    GLenum severity,
    GLsizei length,
    const char* msg,
    const void* userParams
);


class Renderer {
public:
    Renderer();
    ~Renderer();

    bool Init();
    void Draw(const Scene& scene);

private:
    std::unordered_map<std::string, Shader> shaders;

    bool InitShaders();
    bool InitDebugging();
    void SetGlobalUniforms(const std::unique_ptr<Camera>& camera, const
std::unique_ptr<Light>& light);
    void SetLocalUniforms(const std::unique_ptr<Body>& body);
};


inline Renderer::Renderer() : shaders() {}

inline Renderer::~Renderer() {
    shaders.clear();
}

inline bool Renderer::Init() {
    if (!gladLoadGLLoader((GLADloadproc)SDL_GL_GetProcAddress)) {
        std::cerr << "Failed to initialize GLAD" << std::endl;
        return false;
    }

    glEnable(GL_DEPTH_TEST);
    glEnable(GL_MULTISAMPLE);
```

Лістинг В – Клас Renderer, аркуш 1

```
    if (!InitShaders()) {
        return false;
    }


    if (!InitDebugging()) {
        return false;
    }

    return true;
}

inline void Renderer::Draw(const Scene& scene) {
    glClearColor(0.53f, 0.81f, 0.92f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    SetGlobalUniforms(scene.camera, scene.light);

    for (const auto& body : scene.bodies) {
        auto [it, inserted] = shaders.try_emplace("phong");
        if (inserted) {
            it->second.CompileShaderFile(GL_VERTEX_SHADER,
"assets/shaders/phong.vert");
            it->second.CompileShaderFile(GL_FRAGMENT_SHADER,
"assets/shaders/phong.frag");
            it->second.CompileProgram();
        }
        shaders["phong"].Begin();
        SetLocalUniforms(body);
        body->Draw();
        shaders["phong"].End();
    }
}

inline bool Renderer::InitShaders() {
    Shader basic;
    basic.CompileShaderFile(GL_VERTEX_SHADER, "assets/shaders/basic.vert");
    basic.CompileShaderFile(GL_FRAGMENT_SHADER, "assets/shaders/basic.frag");
    basic.CompileProgram();
    if (basic.IsInitalized()) {
        basic.AddAttribute("vertexPos");
        basic.AddAttribute("vertexColor");
        basic.AddUniform("model");
        basic.AddUniform("view");
        basic.AddUniform("projection");
        shaders["basic"] = std::move(basic);
    }

    Shader phong;
    phong.CompileShaderFile(GL_VERTEX_SHADER, "assets/shaders/phong.vert");
    phong.CompileShaderFile(GL_FRAGMENT_SHADER, "assets/shaders/phong.frag");
    phong.CompileProgram();
    if (phong.IsInitalized()) {
        phong.AddAttribute("vertexPos");
        phong.AddAttribute("vertexNormal");
        phong.AddAttribute("vertexTexture");
        phong.AddUniform("model");
        phong.AddUniform("view");
        phong.AddUniform("projection");
```

Лістинг В – Клас Renderer, аркуш 2

```
        phong.AddUniform("light.pos");
        phong.AddUniform("light.ambient");
        phong.AddUniform("light.diffuse");
        phong.AddUniform("light.specular");
        phong.AddUniform("cameraPos");
        phong.AddUniform("realColor");
        phong.AddUniform("shineness");
        shaders["phong"] = std::move(phong);
    }

    if (shaders.size() == 0) {
        std::cerr << "Error: Couldn't load shaders!" << std::endl;
        return false;
    }

    return true;
}

inline bool Renderer::InitDebugging() {
    GLint flags;
    glGetIntegerv(GL_CONTEXT_FLAGS, &flags);
    if (flags & GL_CONTEXT_FLAG_DEBUG_BIT) {
        glEnable(GL_DEBUG_OUTPUT);
        glEnable(GL_DEBUG_OUTPUT_SYNCHRONOUS);
        glDebugMessageCallback(glDebugOutput, nullptr);
        glDebugMessageControl(GL_DONT_CARE, GL_DONT_CARE, GL_DONT_CARE, 0,
nullptr, GL_TRUE);
        return true;
    }

    std::cerr << "Error: Couldn't enable OpenGL debugging features" <<
std::endl;
    return false;
}

inline void Renderer::SetGlobalUniforms(const std::unique_ptr<Camera>&
camera, const std::unique_ptr<Light>& light) {
    for (auto& [name, shader] : shaders) {
        shader.Begin();
        shader.SetMat4("view", camera->view);
        shader.SetMat4("projection", camera->projection);
        if (name == "phong") {
            shader.SetVec3("light.pos", light->pos);
            shader.SetVec3("cameraPos", camera->pos);
            shader.SetVec4("light.ambient", light->ambient);
            shader.SetVec4("light.diffuse", light->diffuse);
            shader.SetVec4("light.specular", light->specular);
        }
    }
}

inline void Renderer::SetLocalUniforms(const std::unique_ptr<Body>& body) {
    Shader& shader = shaders["phong"];
    shader.Begin();
    Mat4f model = body->GetTransform().cast<float>();
    shader.SetMat4("model", model);
    shader.SetVec4("realColor", body->material->color);
    shader.SetFloat("shineness", body->material->shininess);
}
```

Лістинг В – Клас Renderer, аркуш 3

```
inline void APIENTRY glDebugOutput(
    GLenum source,
    GLenum type,
    unsigned int id,
    GLenum severity,
    GLsizei length,
    const char* msg,
    const void* userParams
) {
    if (id == 131169 || id == 131185 || id == 131204 || id == 131218) {
        return;
    }


    const char* red = "\033[1;31m";
    const char* yellow = "\033[1;33m";
    const char* magenta = "\033[1;35m";
    const char* cyan = "\033[1;36m";
    const char* bold = "\033[1m";
    const char* reset = "\033[0m";

    const char* severityStr = "Unknown";
    const char* severityColor = reset;
    switch (severity) {
        case GL_DEBUG_SEVERITY_HIGH: {
            severityStr = "HIGH";
            severityColor = red;
            break;
        }
        case GL_DEBUG_SEVERITY_MEDIUM: {
            severityStr = "MEDIUM";
            severityColor = yellow;
            break;
        }
        case GL_DEBUG_SEVERITY_LOW: {
            severityStr = "LOW";
            severityColor = cyan;
            break;
        }
        case GL_DEBUG_SEVERITY_NOTIFICATION: {
            severityStr = "NOTIFICATION";
            severityColor = magenta;
            break;
        }
    }

    const char* sourceStr = "Other";
    switch (source) {
        case GL_DEBUG_SOURCE_API: sourceStr = "API"; break;
        case GL_DEBUG_SOURCE_WINDOW_SYSTEM: sourceStr = "Window System";
break;
        case GL_DEBUG_SOURCE_SHADER_COMPILER: sourceStr = "Shader Compiler";
break;
        case GL_DEBUG_SOURCE_THIRD_PARTY: sourceStr = "Third Party"; break;
        case GL_DEBUG_SOURCE_APPLICATION: sourceStr = "Application"; break;
        case GL_DEBUG_SOURCE_OTHER: sourceStr = "Other"; break;
    }
```

Лістинг В – Клас Renderer, аркуш 4

```
    const char* typeStr = "Other";
    switch (type) {
        case GL_DEBUG_TYPE_ERROR: typeStr = "Error"; break;
        case GL_DEBUG_TYPE_DEPRECATED_BEHAVIOR: typeStr = "Deprecated
Behaviour"; break;
        case GL_DEBUG_TYPE_UNDEFINED_BEHAVIOR: typeStr = "Undefined
Behaviour"; break;
        case GL_DEBUG_TYPE_PORTABILITY: typeStr = "Portability"; break;
        case GL_DEBUG_TYPE_PERFORMANCE: typeStr = "Performance"; break;
        case GL_DEBUG_TYPE_MARKER: typeStr = "Marker"; break;
        case GL_DEBUG_TYPE_PUSH_GROUP: typeStr = "Push Group"; break;
        case GL_DEBUG_TYPE_POP_GROUP: typeStr = "Pop Group"; break;
        case GL_DEBUG_TYPE_OTHER: typeStr = "Other"; break;
    }

    printf("%s[OpenGL Debug]%s ID: %u\n", bold, reset, id);
    printf("  %sMessage:%s  %s\n", bold, reset, msg);
    printf("  %sSource:%s   %s%s%s\n", bold, reset, cyan, sourceStr, reset);
    printf("  %sType:%s     %s%s%s\n", bold, reset, cyan, typeStr, reset);
    printf("  %sSeverity:%s %s%s%s\n", bold, reset, severityColor,
severityStr, reset);
    printf("---------------\n"); }
```

Лістинг В – Клас Renderer, аркуш 5

ДОДАТОК Г

Клас Scene

```
#include <memory>
#include <vector>
#include "Body.h"
#include "Collision.h"
#include <PBD/SimulationModel.h>

class Scene {
public:
    Scene();
    void AddBody(std::shared_ptr<Body> body);
    void Simulate(double dt);
    void Draw();

private:
    std::vector<std::shared_ptr<Body>> bodies;
    std::unique_ptr<PBD::SimulationModel> simModel;
    std::unique_ptr<PBD::DistanceFieldCollisionDetection> collisionDetection;
};

inline Scene::Scene() {
    simModel = std::make_unique<PBD::SimulationModel>();
    collisionDetection =
std::make_unique<PBD::DistanceFieldCollisionDetection>();
    simModel->setCollisionDetection(collisionDetection.get());
}

inline void Scene::AddBody(std::shared_ptr<Body> body) {
    bodies.push_back(body);
    if (body->rigidBody) {
        simModel->addRigidBody(body->rigidBody);
        collisionDetection->addCollisionSphere(simModel-
>getRigidBodies().size() - 1, 0, body->rigidBody-
>getGeometry().getVertexData().data(),
                                               body->rigidBody-
>getGeometry().getVertexData().size(), 1.0);
        body->collisionObject =
dynamic_cast<PBD::DistanceFieldCollisionObject*>(simModel-
>getCollisionObjects().back());
    } else if (body->tetModel) {
        simModel->addTetModel(body->tetModel);
        collisionDetection->addCollisionObjectWithoutGeometry(simModel-
>getTetModels().size() - 1, 2, body->tetModel-
>getParticleMesh().getVertexData().data(),
                                                              body->tetModel-
>getParticleMesh().numVertices(), true);
        body->collisionObject =
dynamic_cast<PBD::DistanceFieldCollisionObject*>(simModel-
>getCollisionObjects().back());
    } else if (body->triangleModel) {
        simModel->addTriangleModel(body->triangleModel);
        collisionDetection->addCollisionObjectWithoutGeometry(simModel-
>getTriangleModels().size() - 1, 1, body->triangleModel-
>getParticleMesh().getVertexData().data(),
```

Лістинг Г – Клас Scene, аркуш 1

```
                                                                    body-
>triangleModel->getParticleMesh().numVertices(), true);
        body->collisionObject =
dynamic_cast<PBD::DistanceFieldCollisionObject*>(simModel-
>getCollisionObjects().back());
    } else if (body->lineModel) {
        simModel->addLineModel(body->lineModel);
        collisionDetection->addCollisionObjectWithoutGeometry(simModel-
>getLineModels().size() - 1, 3, body->lineModel->getVertexData().data(),
                                                                    body->lineModel-
>getVertexData().size(), true);
        body->collisionObject =
dynamic_cast<PBD::DistanceFieldCollisionObject*>(simModel-
>getCollisionObjects().back());
    } else if (body->fluidModel) {
        simModel->addFluidModel(body->fluidModel);
        collisionDetection->addCollisionObjectWithoutGeometry(simModel-
>getFluidModels().size() - 1, 4, body->fluidModel-
>getParticles().getPositions().data(),
                                                                    body-
>fluidModel->getParticles().size(), true);
        body->collisionObject =
dynamic_cast<PBD::DistanceFieldCollisionObject*>(simModel-
>getCollisionObjects().back());
    }
}

inline void Scene::Simulate(double dt) {
    for (int i = 0; i < 10; i++) {
        for (auto& body : bodies) {
            body->Simulate(dt / 10.0, 1);
        }
        collisionDetection->collisionDetection(*simModel);
    }
}

inline void Scene::Draw() {
    for (auto& body : bodies) {
        body->Draw();
    }
}
```

Лістинг Г – Клас Scene, аркуш 2

ДОДАТОК Д

Клас Body

```
#include <memory>
#include <vector>
#include "Common.h"
#include "Collision.h"
#include "Constraint.h"
#include "Particle.h"
#include "Gfx/Mesh.h"
#include "Gfx/Material.h"
#include <PBD/Simulation.h>
#include <PBD/SimulationModel.h>
#include <PBD/RigidBody.h>
#include <PBD/TetModel.h>
#include <PBD/TriangleModel.h>
#include <PBD/LineModel.h>
#include <PBD/FluidModel.h>
#include <PBD/XPBD.h>
#include <PBD/PositionBasedFluids.h>
#include <PBD/ParticleData.h>

class Body {
public:
    std::shared_ptr<Mesh> mesh;
    std::shared_ptr<Material> material;
    std::unique_ptr<ParticleBody> particleBody;
    PBD::RigidBody* rigidBody;
    PBD::TetModel* tetModel;
    PBD::TriangleModel* triangleModel;
    PBD::LineModel* lineModel;
    PBD::FluidModel* fluidModel;
    PBD::DistanceFieldCollisionObject* collisionObject;

    Body() : rigidBody(nullptr), tetModel(nullptr), triangleModel(nullptr),
lineModel(nullptr), fluidModel(nullptr), collisionObject(nullptr) {}
    virtual ~Body() {}

    virtual void Simulate(double dt, int substeps);
    virtual void HandleCollisions(PBD::SimulationModel& model,
PBD::DistanceFieldCollisionDetection& cd, const Real restitutionCoeff, const
Real frictionCoeff);
    virtual Mat4d GetTransform();
    void AddConstraint(std::shared_ptr<PBD::Constraint> constraint);
    void Draw();

protected:
    std::vector<std::shared_ptr<PBD::Constraint>> constraints;
    std::vector<PBD::DistanceFieldCollisionDetection::ContactData>
contactData;
};

inline void Body::Simulate(double dt, int substeps) {
    double substepDt = dt / substeps;
    PBD::SimulationModel* simModel = PBD::Simulation::getCurrent()-
>getModel();
```

Лістинг Д – Клас Body, аркуш 1

```
     PBD::DistanceFieldCollisionDetection* cd =
    dynamic_cast<PBD::DistanceFieldCollisionDetection*>(simModel-
    >getCollisionDetection());

    if (particleBody) {
        particleBody->Simulate(dt, substeps);
        if (mesh) {
            mesh->UpdateVertices(particleBody->GetPositions());
        }
    }

    if (rigidBody) {
        if (mesh) {
            PBD::RigidBodyGeometry& geometry = rigidBody->getGeometry();
            geometry.updateMeshTransformation(rigidBody->getPosition(),
rigidBody->getRotationMatrix());
            mesh->UpdateVertices(geometry.getVertexData());
        }
    }

    if (tetModel) {
        PBD::ParticleData& particles = simModel->getParticles();
        unsigned int offset = tetModel->getIndexOffset();
        unsigned int numTets = tetModel->getParticleMesh().numTets();

        for (int i = 0; i < substeps; i++) {
            for (unsigned int j = 0; j < numTets; j++) {
                const unsigned int* tet = tetModel-
>getParticleMesh().getTets() + 4 * j;
                auto constraint = std::make_shared<PBD::FEMTetConstraint>();
                constraint->InitConstraint(*simModel, tet[0] + offset, tet[1]
+ offset, tet[2] + offset, tet[3] + offset, 1.0, 0.3);
                constraint->SolvePositionConstraint(*simModel, 1);
            }
        }

        tetModel->updateMeshNormals(particles);
        tetModel->updateVisMesh(particles);
        if (mesh) {
            mesh->UpdateVertices(tetModel->getVisVertices());
        }
    }

    if (triangleModel) {
        PBD::ParticleData& particles = simModel->getParticles();
        unsigned int offset = triangleModel->getIndexOffset();
        unsigned int numEdges = triangleModel->getParticleMesh().numEdges();

        for (int i = 0; i < substeps; i++) {
            for (unsigned int j = 0; j < numEdges; j++) {
                const unsigned int* edge = triangleModel-
>getParticleMesh().getEdges() + 2 * j;
                auto constraint =
std::make_shared<PBD::DistanceConstraint>();
                constraint->InitConstraint(*simModel, edge[0] + offset,
edge[1] + offset, 1.0);
                constraint->SolvePositionConstraint(*simModel, 1);
            }
        }
```

Лістинг Д – Клас Body, аркуш 2

```
        triangleModel->updateMeshNormals(particles);
        if (mesh) {
            mesh->UpdateVertices(particles.getPositions().data() + offset);
        }
    }

    if (lineModel) {
        PBD::ParticleData& particles = simModel->getParticles();
        unsigned int offset = lineModel->getIndexOffset();
        unsigned int numEdges = lineModel->getEdges().size();

        for (int i = 0; i < substeps; i++) {
            for (unsigned int j = 0; j < numEdges; j++) {
                const auto& edge = lineModel->getEdges()[j];
                auto constraint =
std::make_shared<PBD::DistanceConstraint>();
                constraint->InitConstraint(*simModel, edge.m_vert[0] +
offset, edge.m_vert[1] + offset, 1.0);
                constraint->SolvePositionConstraint(*simModel, 1);
            }
        }

        if (mesh) {
            mesh->UpdateVertices(particles.getPositions().data() + offset);
        }
    }

    if (fluidModel) {
        PBD::ParticleData& particles = simModel->getParticles();
        unsigned int numParticles = fluidModel->getParticles().size();
        PBD::NeighborhoodSearchSpatialHashing* neighborhoodSearch =
fluidModel->getNeighborhoodSearch();

        neighborhoodSearch->update();

        for (int i = 0; i < substeps; i++) {
            for (unsigned int j = 0; j < numParticles; j++) {
                Real densityErr = 0.0;
                Real density = 0.0;
                const unsigned int* neighbors = neighborhoodSearch-
>getNeighbors(j);
                unsigned int numNeighbors = neighborhoodSearch-
>getNumNeighbors(j);
                bool res = PBD::PositionBasedFluids::computePBFDensity(
                    j, numParticles, particles.getPositions().data(),
particles.getMasses().data(),
                    fluidModel->getBoundaryX(0), fluidModel-
>getBoundaryPsi(0), numNeighbors, neighbors,
                    fluidModel->getDensity0(), true, densityErr, density
                );
                if (res) {
                    fluidModel->setDensity(j, density);
                }

                Real lambda = 0.0;
```

Лістинг Д – Клас Body, аркуш 3

```
                res = PBD::PositionBasedFluids::computePBFLagrangeMultiplier(
                    j, numParticles, particles.getPositions().data(),
particles.getMasses().data(),
                    fluidModel->getBoundaryX(0), fluidModel-
>getBoundaryPsi(0), density,
                    numNeighbors, neighbors, fluidModel->getDensity0(), true,
lambda
                );
                if (res) {
                    fluidModel->setLambda(j, lambda);
                }
            }

            for (unsigned int j = 0; j < numParticles; j++) {
                Vector3r corr;
                const unsigned int* neighbors = neighborhoodSearch-
>getNeighbors(j);
                unsigned int numNeighbors = neighborhoodSearch-
>getNumNeighbors(j);
                bool res = PBD::PositionBasedFluids::solveDensityConstraint(
                    j, numParticles, particles.getPositions().data(),
particles.getMasses().data(),
                    fluidModel->getBoundaryX(0), fluidModel-
>getBoundaryPsi(0),
                    numNeighbors, neighbors, fluidModel->getDensity0(), true,
                    fluidModel->getLambda(0), corr
                );
                if (res) {
                    particles.getPosition(j) += corr;
                }
            }
        }

        if (mesh) {
            mesh->UpdateVertices(fluidModel->getParticles().getPositions());
        }
    }

    for (const auto& constraint : constraints) {
        constraint->Solve(simModel->getParticles(), substepDt);
    }

    if (cd && collisionObject) {
        HandleCollisions(*simModel, *cd, simModel->getRestitutionCoeff(),
simModel->getFrictionCoeff());
    }
}

inline void Body::HandleCollisions(PBD::SimulationModel& model,
PBD::DistanceFieldCollisionDetection& cd, const Real restitutionCoeff, const
Real frictionCoeff) {
    PBD::ParticleData& particles = model.getParticles();
    contactData.clear();

    if (rigidBody && collisionObject) {
        for (unsigned int i = 0; i < model.getRigidBodies().size(); i++) {
            PBD::RigidBody* otherRb = model.getRigidBodies()[i];
            if (rigidBody == otherRb) {
                continue;
            }
```

Лістинг Д – Клас Body, аркуш 4

```
            PBD::DistanceFieldCollisionObject* otherCo =
dynamic_cast<PBD::DistanceFieldCollisionObject*>(model.getCollisionObjects()[
i]);
            if (otherCo) {
std::vector<std::vector<PBD::DistanceFieldCollisionDetection::ContactData>>
contacts_mt;
                cd.collisionDetectionRigidBodies(rigidBody, collisionObject,
otherRb, otherCo, restitutionCoeff, frictionCoeff, contacts_mt);
                for (const auto& contacts : contacts_mt) {
                    contactData.insert(contactData.end(), contacts.begin(),
contacts.end());
                }
            }
        }
        for (const auto& contact : contactData) {
            auto constraint =
std::make_shared<PBD::RigidBodyContactConstraint>();
            constraint->InitConstraint(model, contact.m_index1,
contact.m_index2, contact.m_cp1, contact.m_cp2,
                                       contact.m_normal, contact.m_dist,
restitutionCoeff, 1.0, frictionCoeff);
            constraint->SolveVelocityConstraint(model, 1);
        }
    }

    // Коллизии для TetModel
    if (tetModel && collisionObject) {
        unsigned int offset = tetModel->getIndexOffset();
        unsigned int numVert = tetModel->getParticleMesh().numVertices();
        for (unsigned int i = 0; i < model.getTetModels().size(); i++) {
            PBD::TetModel* otherTm = model.getTetModels()[i];
            if (tetModel == otherTm) {
                continue;
            }
            PBD::DistanceFieldCollisionObject* otherCo =
dynamic_cast<PBD::DistanceFieldCollisionObject*>(model.getCollisionObjects()[
i]);
            if (otherCo) {
std::vector<std::vector<PBD::DistanceFieldCollisionDetection::ContactData>>
contacts_mt;
                cd.collisionDetectionSolidSolid(particles, offset, numVert,
collisionObject, otherTm, otherCo, restitutionCoeff, frictionCoeff,
contacts_mt);
                for (const auto& contacts : contacts_mt) {
                    contactData.insert(contactData.end(), contacts.begin(),
contacts.end());
                }
            }
        }
        for (const auto& contact : contactData) {
            auto constraint =
std::make_shared<PBD::ParticleTetContactConstraint>();
            constraint->InitConstraint(model, contact.m_index1,
contact.m_index2, contact.m_elementIndex2, contact.m_bary2,
                                       contact.m_cp1, contact.m_cp2,
contact.m_normal, contact.m_dist, frictionCoeff);
            constraint->SolvePositionConstraint(model, 1);
            constraint->SolveVelocityConstraint(model, 1);
        }
    }
```

Лістинг Д – Клас Body, аркуш 5

```
    if (triangleModel && collisionObject) {
        unsigned int offset = triangleModel->getIndexOffset();
        unsigned int numVert = triangleModel-
>getParticleMesh().numVertices();
        for (unsigned int i = 0; i < model.getRigidBodies().size(); i++) {
            PBD::RigidBody* otherRb = model.getRigidBodies()[i];
            PBD::DistanceFieldCollisionObject* otherCo =
dynamic_cast<PBD::DistanceFieldCollisionObject*>(model.getCollisionObjects()[
i]);
            if (otherCo) {

std::vector<std::vector<PBD::DistanceFieldCollisionDetection::ContactData>>
contacts_mt;
                cd.collisionDetectionRBSolid(particles, offset, numVert,
collisionObject, otherRb, otherCo, restitutionCoeff, frictionCoeff,
contacts_mt);
                for (const auto& contacts : contacts_mt) {
                    contactData.insert(contactData.end(), contacts.begin(),
contacts.end());
                }
            }
        }
        for (const auto& contact : contactData) {
            auto constraint =
std::make_shared<PBD::ParticleRigidBodyContactConstraint>();
            constraint->InitConstraint(model, contact.m_index1,
contact.m_index2, contact.m_cp1, contact.m_cp2,
                                       contact.m_normal, contact.m_dist,
restitutionCoeff, 1.0, frictionCoeff);
            constraint->SolveVelocityConstraint(model, 1);
        }
    }

    if (fluidModel && collisionObject) {
        unsigned int numParticles = fluidModel->getParticles().size();
        for (unsigned int i = 0; i < model.getRigidBodies().size(); i++) {
            PBD::RigidBody* otherRb = model.getRigidBodies()[i];
            PBD::DistanceFieldCollisionObject* otherCo =
dynamic_cast<PBD::DistanceFieldCollisionObject*>(model.getCollisionObjects()[
i]);
            if (otherCo) {

std::vector<std::vector<PBD::DistanceFieldCollisionDetection::ContactData>>
contacts_mt;
                cd.collisionDetectionRBSolid(particles, 0, numParticles,
collisionObject, otherRb, otherCo, restitutionCoeff, frictionCoeff,
contacts_mt);
                for (const auto& contacts : contacts_mt) {
                    contactData.insert(contactData.end(), contacts.begin(),
contacts.end());
                }
            }
        }



        for (const auto& contact : contactData) {
            auto constraint =
std::make_shared<PBD::ParticleRigidBodyContactConstraint>();
```

Лістинг Д – Клас Body, аркуш 6

```
            constraint->InitConstraint(model, contact.m_index1,
contact.m_index2, contact.m_cp1, contact.m_cp2, contact.m_normal,
contact.m_dist, restitutionCoeff, 1.0, frictionCoeff);
            constraint->SolveVelocityConstraint(model, 1);
        }
    }

    if (particleBody) {
        for (const auto& constraint : constraints) {
            constraint->Solve(particles, substepDt);
        }
    }
}

inline Mat4d Body::GetTransform() {
    Mat4d transform = Mat4d::Identity();
    if (particleBody) {
        transform = particleBody->GetTransform();
    } else if (rigidBody) {
        transform.block<3, 3>(0, 0) = rigidBody->getRotationMatrix();
        transform.block<3, 1>(0, 3) = rigidBody->getPosition();
    } else if (tetModel || triangleModel || lineModel || fluidModel) {
        PBD::SimulationModel* simModel = PBD::Simulation::getCurrent()-
>getModel();
        PBD::ParticleData& particles = simModel->getParticles();
        unsigned int offset = 0;
        unsigned int numPoints = 0;
        if (tetModel) {
            offset = tetModel->getIndexOffset();
            numPoints = tetModel->getParticleMesh().numVertices();
        } else if (triangleModel) {
            offset = triangleModel->getIndexOffset();
            numPoints = triangleModel->getParticleMesh().numVertices();
        } else if (lineModel) {
            offset = lineModel->getIndexOffset();
            numPoints = lineModel->getEdges().size();
        } else if (fluidModel) {
            numPoints = fluidModel->getParticles().size();
        }
        Vector3r center(0.0, 0.0, 0.0);
        for (unsigned int i = 0; i < numPoints; i++) {
            center += particles.getPosition(offset + i);
        }
        if (numPoints > 0) {
            center /= numPoints;
        }
        transform.block<3, 1>(0, 3) = center;
    }
    return transform;
}

inline void Body::AddConstraint(std::shared_ptr<PBD::Constraint> constraint){
    constraints.push_back(constraint);
}

inline void Body::Draw() {
    if (mesh) {
        mesh->Draw();
    }
}
```

Лістинг Д – Клас Body, аркуш 7

ДОДАТОК Е

Клас Constaint

```
#include "Common.h"
#include "MathFunctions.h"
#include <PBD/SimulationModel.h>
#include <PBD/ParticleData.h>
#include <PBD/RigidBody.h>
#include <PBD/TetModel.h>
#include <PBD/DistanceFieldCollisionDetection.h>
#include <PBD/XPBD.h>
#include <vector>
#include <array>
#include <Eigen/Dense>

namespace PBD {

class Constraint {
public:
    std::vector<unsigned int> m_bodies;

    Constraint(const unsigned int numberOfBodies) {
        m_bodies.resize(numberOfBodies);
    }
    virtual ~Constraint() {}
    virtual int& GetTypeId() const = 0;
    virtual bool InitConstraint(SimulationModel& model) { return true; }
    virtual bool UpdateConstraint(SimulationModel& model) { return true; }
    virtual bool SolvePositionConstraint(SimulationModel& model, const
unsigned int iter) { return true; }
    virtual bool SolveVelocityConstraint(SimulationModel& model, const
unsigned int iter) { return true; }
    virtual void Solve(ParticleData& p, double dts) {}
};

class DistanceConstraint : public Constraint {
public:
    static int TYPE_ID;
    Real m_restLength;
    Real m_stiffness;
    Real m_lambda;

    DistanceConstraint() : Constraint(2), m_restLength(0.0),
m_stiffness(1.0), m_lambda(0.0) {}
    virtual int& GetTypeId() const { return TYPE_ID; }

    bool InitConstraint(SimulationModel& model, const unsigned int particle1,
const unsigned int particle2, const Real stiffness) {
        m_bodies[0] = particle1;
        m_bodies[1] = particle2;
        m_stiffness = stiffness;
        ParticleData& p = model.getParticles();
        m_restLength = (p.getPosition(particle1) -
p.getPosition(particle2)).norm();
        return true;
    }
```

Лістинг Е – Клас Constraint, аркуш 1

```
    bool SolvePositionConstraint(SimulationModel& model, const unsigned int
iter) {
        ParticleData& p = model.getParticles();
        if (p.getInvMass(m_bodies[0]) < 1e-6 && p.getInvMass(m_bodies[1]) <
1e-6) {
            return false;
        }

        Vector3r delta = p.getPosition(m_bodies[0]) -
p.getPosition(m_bodies[1]);
        Real distance = delta.norm();
        if (distance < 1e-6) {
            return false;
        }

        Real C = distance - m_restLength;
        Vector3r gradC1 = delta / distance;
        Vector3r gradC2 = -gradC1;

        Real denom = p.getInvMass(m_bodies[0]) + p.getInvMass(m_bodies[1]) +
m_stiffness / (iter * iter);
        if (denom < 1e-6) {
            return false;
        }

        Real lambda = -C / denom;
        m_lambda += lambda;

        p.getPosition(m_bodies[0]) += lambda * p.getInvMass(m_bodies[0]) *
gradC1;
        p.getPosition(m_bodies[1]) += lambda * p.getInvMass(m_bodies[1]) *
gradC2;
        return true;
    }

    void Solve(ParticleData& p, double dts) {
        if (p.getInvMass(m_bodies[0]) < 1e-6 && p.getInvMass(m_bodies[1]) <
1e-6) {
            return;
        }

        Vector3r delta = p.getPosition(m_bodies[0]) -
p.getPosition(m_bodies[1]);
        Real distance = delta.norm();
        if (distance < 1e-6) {
            return;
        }

        Real C = distance - m_restLength;
        Vector3r gradC1 = delta / distance;
        Vector3r gradC2 = -gradC1;

        Real denom = p.getInvMass(m_bodies[0]) + p.getInvMass(m_bodies[1]) +
m_stiffness / (dts * dts);
        if (denom < 1e-6) {
            return;
        }

        Real lambda = -C / denom;
        m_lambda += lambda;
```

Лістинг Е – Клас Constraint, аркуш 2

```
        p.getPosition(m_bodies[0]) += lambda * p.getInvMass(m_bodies[0]) *
gradC1;
        p.getPosition(m_bodies[1]) += lambda * p.getInvMass(m_bodies[1]) *
gradC2;
    }
};

class BallJoint : public Constraint {
public:
    static int TYPE_ID;
    Eigen::Matrix<Real, 3, 4, Eigen::DontAlign> m_jointInfo;

    BallJoint() : Constraint(2) {}
    virtual int& GetTypeId() const { return TYPE_ID; }

    bool InitConstraint(SimulationModel& model, const unsigned int rbIndex1,
const unsigned int rbIndex2, const Vector3r& pos) {
        m_bodies[0] = rbIndex1;
        m_bodies[1] = rbIndex2;
        RigidBody* rb1 = model.getRigidBodies()[rbIndex1];
        RigidBody* rb2 = model.getRigidBodies()[rbIndex2];
        m_jointInfo.col(0) = pos;
        m_jointInfo.col(1) = rb1->getRotationMatrix().transpose() * (pos -
rb1->getPosition());
        m_jointInfo.col(2) = rb2->getRotationMatrix().transpose() * (pos -
rb2->getPosition());
        m_jointInfo.col(3) = Vector3r(0.0, 0.0, 0.0);
        return true;
    }

    bool UpdateConstraint(SimulationModel& model) {
        return true;
    }

    bool SolvePositionConstraint(SimulationModel& model, const unsigned int
iter) {
        RigidBody* rb1 = model.getRigidBodies()[m_bodies[0]];
        RigidBody* rb2 = model.getRigidBodies()[m_bodies[1]];
        const Vector3r& connector = m_jointInfo.col(0);
        const Vector3r r1 = m_jointInfo.col(1);
        const Vector3r r2 = m_jointInfo.col(2);

        Vector3r delta = rb1->getPosition() + rb1->getRotationMatrix() * r1 -
rb2->getPosition() - rb2->getRotationMatrix() * r2;
        Real C = delta.norm();
        if (C < 1e-6) {
            return true;
        }

        Vector3r n = delta / C;
        Real invMass1 = rb1->getInvMass();
        Matrix3r invInertia1 = rb1->getInertiaTensorInverseWorld();
        Real invMass2 = rb2->getInvMass();
        Matrix3r invInertia2 = rb2->getInertiaTensorInverseWorld();

        Matrix3r r1_hat, r2_hat;
        MathFunctions::crossProductMatrix(r1, r1_hat);
        MathFunctions::crossProductMatrix(r2, r2_hat);
        Real w = invMass1 + invMass2 + n.dot(r1_hat * invInertia1 * r1_hat *
n) + n.dot(r2_hat * invInertia2 * r2_hat * n);
```

Лістинг Е – Клас Constraint, аркуш 3

```
        if (w < 1e-6) {
            return false;
        }

        Real lambda = -C / w;
        Vector3r corr1 = lambda * invMass1 * n;
        Vector3r corr2 = -lambda * invMass2 * n;
        Vector3r angCorr1 = invInertia1 * r1_hat * n * lambda;
        Vector3r angCorr2 = -invInertia2 * r2_hat * n * lambda;

        rb1->getPosition() += corr1;
        rb2->getPosition() += corr2;
        rb1->getRotation() = rb1->getRotation() * Quaternionr(1.0, 0.5 *
angCorr1.x(), 0.5 * angCorr1.y(), 0.5 * angCorr1.z());
        rb2->getRotation() = rb2->getRotation() * Quaternionr(1.0, 0.5 *
angCorr2.x(), 0.5 * angCorr2.y(), 0.5 * angCorr2.z());
        rb1->rotationUpdated();
        rb2->rotationUpdated();
        return true;
    }
};

class HingeJoint : public Constraint {
public:
    static int TYPE_ID;
    Eigen::Matrix<Real, 4, 7, Eigen::DontAlign> m_jointInfo;

    HingeJoint() : Constraint(2) {}
    virtual int& GetTypeId() const { return TYPE_ID; }

    bool InitConstraint(SimulationModel& model, const unsigned int rbIndex1,
const unsigned int rbIndex2, const Vector3r& pos, const Vector3r& axis) {
        m_bodies[0] = rbIndex1;
        m_bodies[1] = rbIndex2;
        RigidBody* rb1 = model.getRigidBodies()[rbIndex1];
        RigidBody* rb2 = model.getRigidBodies()[rbIndex2];
        m_jointInfo.col(0) = pos;
        m_jointInfo.col(1) = rb1->getRotationMatrix().transpose() * (pos -
rb1->getPosition());
        m_jointInfo.col(2) = rb2->getRotationMatrix().transpose() * (pos -
rb2->getPosition());
        m_jointInfo.col(3) = rb1->getRotationMatrix().transpose() * axis;
        m_jointInfo.col(4) = rb2->getRotationMatrix().transpose() * axis;
        m_jointInfo.col(5) = Vector3r(0.0, 0.0, 0.0);
        m_jointInfo.col(6) = Vector3r(0.0, 0.0, 0.0);
        return true;
    }

    bool UpdateConstraint(SimulationModel& model) {
        return true;
    }

    bool SolvePositionConstraint(SimulationModel& model, const unsigned int
iter) {
        RigidBody* rb1 = model.getRigidBodies()[m_bodies[0]];
        RigidBody* rb2 = model.getRigidBodies()[m_bodies[1]];
        const Vector3r& pos = m_jointInfo.col(0);
        const Vector3r r1 = m_jointInfo.col(1);
        const Vector3r r2 = m_jointInfo.col(2);
```

Лістинг Е – Клас Constraint, аркуш 4

```
        const Vector3r axis1 = rb1->getRotationMatrix() * m_jointInfo.col(3);
        const Vector3r axis2 = rb2->getRotationMatrix() * m_jointInfo.col(4);

        Vector3r delta = rb1->getPosition() + rb1->getRotationMatrix() * r1 -
rb2->getPosition() - rb2->getRotationMatrix() * r2;
        Real C1 = delta.norm();
        if (C1 < 1e-6) {
            return true;
        }

        Vector3r n = delta / C1;
        Real invMass1 = rb1->getInvMass();
        Real invMass2 = rb2->getInvMass();
        Matrix3r invInertia1 = rb1->getInertiaTensorInverseWorld();
        Matrix3r invInertia2 = rb2->getInertiaTensorInverseWorld();

        Matrix3r r1_hat, r2_hat;
        MathFunctions::crossProductMatrix(r1, r1_hat);
        MathFunctions::crossProductMatrix(r2, r2_hat);

        Real w = invMass1 + invMass2 + n.dot(r1_hat * invInertia1 * r1_hat *
n) + n.dot(r2_hat * invInertia2 * r2_hat * n);
        if (w < 1e-6) {
            return false;
        }

        Real lambda = -C1 / w;
        Vector3r corr1 = lambda * invMass1 * n;
        Vector3r corr2 = -lambda * invMass2 * n;
        Vector3r angCorr1 = invInertia1 * r1_hat * n * lambda;
        Vector3r angCorr2 = -invInertia2 * r2_hat * n * lambda;

        rb1->getPosition() += corr1;
        rb2->getPosition() += corr2;
        rb1->getRotation() = rb1->getRotation() * Quaternionr(1.0, 0.5 *
angCorr1.x(), 0.5 * angCorr1.y(), 0.5 * angCorr1.z());
        rb2->getRotation() = rb2->getRotation() * Quaternionr(1.0, 0.5 *
angCorr2.x(), 0.5 * angCorr2.y(), 0.5 * angCorr2.z());
        rb1->rotationUpdated();
        rb2->rotationUpdated();

        Vector3r u = axis1.cross(axis2);
        if (u.norm() > 1e-6) {
            u.normalize();
            Matrix3r u_hat;
            MathFunctions::crossProductMatrix(u, u_hat);
            Vector3r t1 = u_hat * axis1;
            Vector3r t2 = u_hat * axis2;
            Real w_rot = t1.dot(invInertia1 * t1) + t2.dot(invInertia2 * t2);
            if (w_rot > 1e-6) {
                Real lambda_rot = -u.norm() / w_rot;
                Vector3r angCorr1 = invInertia1 * t1 * lambda_rot;
                Vector3r angCorr2 = -invInertia2 * t2 * lambda_rot;
                rb1->getRotation() = rb1->getRotation() * Quaternionr(1.0,
0.5 * angCorr1.x(), 0.5 * angCorr1.y(), 0.5 * angCorr1.z());
                rb2->getRotation() = rb2->getRotation() * Quaternionr(1.0,
0.5 * angCorr2.x(), 0.5 * angCorr2.y(), 0.5 * angCorr2.z());
                rb1->rotationUpdated();
                rb2->rotationUpdated();
            }
```

Лістинг Е – Клас Constraint, аркуш 5

```
        }
        return true;
    }
};

class IsometricBendingConstraint : public Constraint {
public:
    static int TYPE_ID;
    Real m_stiffness;
    Matrix4r m_Q;
    Real m_lambda;

    IsometricBendingConstraint() : Constraint(4), m_stiffness(1.0),
m_lambda(0.0) {}
    virtual int& GetTypeId() const { return TYPE_ID; }

    bool InitConstraint(SimulationModel& model, const unsigned int particle1,
const unsigned int particle2,
                        const unsigned int particle3, const unsigned int
particle4, const Real stiffness) {
        m_bodies = {particle1, particle2, particle3, particle4};
        m_stiffness = stiffness;
        ParticleData& p = model.getParticles();
        Vector3r p0 = p.getPosition(particle1);
        Vector3r p1 = p.getPosition(particle2);
        Vector3r p2 = p.getPosition(particle3);
        Vector3r p3 = p.getPosition(particle4);

        Vector3r e = p1 - p0;
        Real eLen = e.norm();
        if (eLen < 1e-6) {
            return false;
        }

        Vector3r n1 = (p2 - p0).cross(p1 - p0);
        Vector3r n2 = (p1 - p0).cross(p3 - p1);
        Real cotPhi = MathFunctions::cotTheta(n1, n2);

        Real A0 = 0.5 * n1.norm();
        Real A1 = 0.5 * n2.norm();
        Real height = A0 * 2.0 / eLen;

        m_Q.setZero();
        Real q = cotPhi * (A0 + A1) / (height * height);
        m_Q(0, 0) = q;
        m_Q(1, 1) = q;
        m_Q(2, 2) = q;
        m_Q(3, 3) = q;
        m_Q(0, 1) = -q;
        m_Q(1, 0) = -q;
        m_Q(2, 3) = -q;
        m_Q(3, 2) = -q;
        return true;
    }

    bool SolvePositionConstraint(SimulationModel& model, const unsigned int
iter) {
        ParticleData& p = model.getParticles();
        Vector3r corr[4];
```

Лістинг Е – Клас Constraint, аркуш 6

```
        bool res = XPBD::solve_IsometricBendingConstraint(
            p.getPosition(m_bodies[0]), p.getInvMass(m_bodies[0]),
            p.getPosition(m_bodies[1]), p.getInvMass(m_bodies[1]),
            p.getPosition(m_bodies[2]), p.getInvMass(m_bodies[2]),
            p.getPosition(m_bodies[3]), p.getInvMass(m_bodies[3]),
            m_Q, m_stiffness, 1.0 / (iter * iter), m_lambda,
            corr[0], corr[1], corr[2], corr[3]
        );
        if (res) {
            for (int i = 0; i < 4; i++) {
                p.getPosition(m_bodies[i]) += corr[i];
            }
        }
        return res;
    }
};

class FEMTetConstraint : public Constraint {
public:
    static int TYPE_ID;
    Real m_stiffness;
    Real m_poissonRatio;
    Real m_volume;
    Matrix3r m_invRestMat;
    Real m_lambda;

    FEMTetConstraint() : Constraint(4), m_stiffness(1.0),
m_poissonRatio(0.3), m_volume(1.0), m_lambda(0.0) {}
    virtual int& GetTypeId() const { return TYPE_ID; }

    bool InitConstraint(SimulationModel& model, const unsigned int particle1,
const unsigned int particle2,
                        const unsigned int particle3, const unsigned int
particle4, const Real stiffness, const Real poissonRatio) {
        m_bodies = {particle1, particle2, particle3, particle4};
        m_stiffness = stiffness;
        m_poissonRatio = poissonRatio;
        ParticleData& p = model.getParticles();
        Vector3r p0 = p.getPosition(particle1);
        Vector3r p1 = p.getPosition(particle2);
        Vector3r p2 = p.getPosition(particle3);
        Vector3r p3 = p.getPosition(particle4);
        Matrix3r restMat;
        restMat.col(0) = p1 - p0;
        restMat.col(1) = p2 - p0;
        restMat.col(2) = p3 - p0;
        m_invRestMat = restMat.inverse();
        m_volume = restMat.determinant() / 6.0;
        return true;
    }

    bool SolvePositionConstraint(SimulationModel& model, const unsigned int
iter) {
        ParticleData& p = model.getParticles();
        Vector3r corr[4];
```

Лістинг Е – Клас Constraint, аркуш 7

```
        bool res = XPBD::solve_FEMTetraConstraint(
            p.getPosition(m_bodies[0]), p.getInvMass(m_bodies[0]),
            p.getPosition(m_bodies[1]), p.getInvMass(m_bodies[1]),
            p.getPosition(m_bodies[2]), p.getInvMass(m_bodies[2]),
            p.getPosition(m_bodies[3]), p.getInvMass(m_bodies[3]),
            m_volume, m_invRestMat, m_stiffness, m_poissonRatio,
            true, 1.0 / iter, m_lambda,
            corr[0], corr[1], corr[2], corr[3]
        );
        if (res) {
            for (int i = 0; i < 4; i++) {
                p.getPosition(m_bodies[i]) += corr[i];
            }
        }
        return res;
    }
};

class VolumeConstraint : public Constraint {
public:
    static int TYPE_ID;
    Real m_stiffness;
    Real m_restVolume;
    Real m_lambda;

    VolumeConstraint() : Constraint(4), m_stiffness(1.0), m_restVolume(1.0),
m_lambda(0.0) {}
    virtual int& GetTypeId() const { return TYPE_ID; }

    bool InitConstraint(SimulationModel& model, const unsigned int particle1,
const unsigned int particle2,
                        const unsigned int particle3, const unsigned int
particle4, const Real stiffness) {
        m_bodies = {particle1, particle2, particle3, particle4};
        m_stiffness = stiffness;
        ParticleData& p = model.getParticles();
        Vector3r p0 = p.getPosition(particle1);
        Vector3r p1 = p.getPosition(particle2);
        Vector3r p2 = p.getPosition(particle3);
        Vector3r p3 = p.getPosition(particle4);
        Matrix3r mat;
        mat.col(0) = p1 - p0;
        mat.col(1) = p2 - p0;
        mat.col(2) = p3 - p0;
        m_restVolume = mat.determinant() / 6.0;
        return true;
    }

    bool SolvePositionConstraint(SimulationModel& model, const unsigned int
iter) {
        ParticleData& p = model.getParticles();
        Vector3r p0 = p.getPosition(m_bodies[0]);
        Vector3r p1 = p.getPosition(m_bodies[1]);
        Vector3r p2 = p.getPosition(m_bodies[2]);
        Vector3r p3 = p.getPosition(m_bodies[3]);
        Matrix3r mat;
        mat.col(0) = p1 - p0;
        mat.col(1) = p2 - p0;
        mat.col(2) = p3 - p0;
        Real volume = mat.determinant() / 6.0;
```

Лістинг Е – Клас Constraint, аркуш 8

```
        Real C = volume - m_restVolume;

        Vector3r grad[4];
        grad[0] = (p1 - p2).cross(p3 - p2) / 6.0;
        grad[1] = (p2 - p0).cross(p3 - p0) / 6.0;
        grad[2] = (p0 - p1).cross(p3 - p1) / 6.0;
        grad[3] = (p0 - p2).cross(p1 - p0) / 6.0;

        Real sumInvMass = 0.0;
        for (int i = 0; i < 4; i++) {
            sumInvMass += grad[i].squaredNorm() * p.getInvMass(m_bodies[i]);
        }
        if (sumInvMass < 1e-6) {
            return false;
        }

        Real lambda = -C * m_stiffness / sumInvMass;
        m_lambda += lambda;

        for (int i = 0; i < 4; i++) {
            p.getPosition(m_bodies[i]) += lambda * p.getInvMass(m_bodies[i])
* grad[i];
        }
        return true;
    }
};



class RigidBodyContactConstraint : public Constraint {
public:
    static int TYPE_ID;
    std::array<unsigned int, 2> m_bodies;
    Real m_stiffness;
    Real m_frictionCoeff;
    Real m_sum_impulses;
    Eigen::Matrix<Real, 3, 5, Eigen::DontAlign> m_constraintInfo;

    RigidBodyContactConstraint() : Constraint(2), m_stiffness(1.0),
m_frictionCoeff(0.1), m_sum_impulses(0.0) {}
    virtual int& GetTypeId() const { return TYPE_ID; }

    bool InitConstraint(SimulationModel& model, const unsigned int rbIndex1,
const unsigned int rbIndex2,
                        const Vector3r& cp1, const Vector3r& cp2, const
Vector3r& normal, const Real dist,
                        const Real restitutionCoeff, const Real stiffness,
const Real frictionCoeff) {
        m_bodies[0] = rbIndex1;
        m_bodies[1] = rbIndex2;
        m_stiffness = stiffness;
        m_frictionCoeff = frictionCoeff;
        m_constraintInfo.col(0) = cp1;
        m_constraintInfo.col(1) = cp2;
        m_constraintInfo.col(2) = normal;
        m_constraintInfo.col(3) = Vector3r(dist, restitutionCoeff, 0.0);
        m_constraintInfo.col(4) = Vector3r(0.0, 0.0, 0.0);
        return true;
    }
```

Лістинг Е – Клас Constraint, аркуш 9

```
    bool SolveVelocityConstraint(SimulationModel& model, const unsigned int
iter) {
        RigidBody* rb1 = model.getRigidBodies()[m_bodies[0]];
        RigidBody* rb2 = model.getRigidBodies()[m_bodies[1]];
        const Vector3r& cp1 = m_constraintInfo.col(0);
        const Vector3r& cp2 = m_constraintInfo.col(1);
        const Vector3r& normal = m_constraintInfo.col(2);
        Real dist = m_constraintInfo(0, 3);
        Real restitution = m_constraintInfo(1, 3);

        Vector3r v1 = rb1->getVelocity(cp1);
        Vector3r v2 = rb2->getVelocity(cp2);
        Vector3r relV = v1 - v2;
        Real vn = relV.dot(normal);

        if (vn > 0.0 || dist > 0.0) {
            return false;
        }

        Real invMass1 = rb1->getInvMass();
        Real invMass2 = rb2->getInvMass();
        Matrix3r invInertia1 = rb1->getInertiaTensorInverseWorld();
        Matrix3r invInertia2 = rb2->getInertiaTensorInverseWorld();
        Vector3r r1 = cp1 - rb1->getPosition();
        Vector3r r2 = cp2 - rb2->getPosition();

        Matrix3r r1_hat, r2_hat;
        MathFunctions::crossProductMatrix(r1, r1_hat);
        MathFunctions::crossProductMatrix(r2, r2_hat);
        Real w = invMass1 + invMass2 + n.dot(r1_hat * invInertia1 * r1_hat *
n) + n.dot(r2_hat * invInertia2 * r2_hat * n);
        if (w < 1e-6) {
            return false;
        }

        Real lambda = -(1.0 + restitution) * vn / w;
        m_sum_impulses += lambda;

        Vector3r impulse = lambda * normal;
        rb1->addForce(impulse, cp1);
        rb2->addForce(-impulse, cp2);

        Vector3r t = relV - vn * normal;
        Real tLen = t.norm();
        if (tLen > 1e-6) {
            t /= tLen;
            Real w_t = invMass1 + invMass2 + t.dot(r1_hat * invInertia1 *
r1_hat * t) + t.dot(r2_hat * invInertia2 * r2_hat * t);
            if (w_t > 1e-6) {
                Real lambda_t = -tLen / w_t;
                if (lambda_t > -m_frictionCoeff * lambda && lambda_t <
m_frictionCoeff * lambda) {
                    rb1->addForce(lambda_t * t, cp1);
                    rb2->addForce(-lambda_t * t, cp2);
                }
            }
        }
        return true;
    }
};
```

Лістинг Е – Клас Constraint, аркуш 10

```
class ParticleRigidBodyContactConstraint : public Constraint {
public:
    static int TYPE_ID;
    std::array<unsigned int, 2> m_bodies;
    Real m_stiffness;
    Real m_frictionCoeff;
    Real m_sum_impulses;
    Eigen::Matrix<Real, 3, 5, Eigen::DontAlign> m_constraintInfo;

    ParticleRigidBodyContactConstraint() : Constraint(2), m_stiffness(1.0),
m_frictionCoeff(0.1), m_sum_impulses(0.0) {}
    virtual int& GetTypeId() const { return TYPE_ID; }

    bool InitConstraint(SimulationModel& model, const unsigned int
particleIndex, const unsigned int rbIndex,
                        const Vector3r& cp1, const Vector3r& cp2, const
Vector3r& normal, const Real dist,
                        const Real restitutionCoeff, const Real stiffness,
const Real frictionCoeff) {
        m_bodies[0] = particleIndex;
        m_bodies[1] = rbIndex;
        m_stiffness = stiffness;
        m_frictionCoeff = frictionCoeff;
        m_constraintInfo.col(0) = cp1;
        m_constraintInfo.col(1) = cp2;
        m_constraintInfo.col(2) = normal;
        m_constraintInfo.col(3) = Vector3r(dist, restitutionCoeff, 0.0);
        m_constraintInfo.col(4) = Vector3r(0.0, 0.0, 0.0);
        return true;
    }

    bool SolveVelocityConstraint(SimulationModel& model, const unsigned int
iter) {
        ParticleData& p = model.getParticles();
        RigidBody* rb = model.getRigidBodies()[m_bodies[1]];
        const Vector3r& cp1 = m_constraintInfo.col(0);
        const Vector3r& cp2 = m_constraintInfo.col(1);
        const Vector3r& normal = m_constraintInfo.col(2);
        Real dist = m_constraintInfo(0, 3);
        Real restitution = m_constraintInfo(1, 3);

        Vector3r v1 = p.getVelocity(m_bodies[0]);
        Vector3r v2 = rb->getVelocity(cp2);
        Vector3r relV = v1 - v2;
        Real vn = relV.dot(normal);

        if (vn > 0.0 || dist > 0.0) {
            return false;
        }

        Real invMass1 = p.getInvMass(m_bodies[0]);
        Real invMass2 = rb->getInvMass();
        Matrix3r invInertia2 = rb->getInertiaTensorInverseWorld();
        Vector3r r2 = cp2 - rb->getPosition();

        Matrix3r r2_hat;
        MathFunctions::crossProductMatrix(r2, r2_hat);
        Real w = invMass1 + invMass2 + n.dot(r2_hat * invInertia2 * r2_hat *
n);
```

Лістинг Е – Клас Constraint, аркуш 11

```
        if (w < 1e-6) {
            return false;
        }

        Real lambda = -(1.0 + restitution) * vn / w;
        m_sum_impulses += lambda;

        Vector3r impulse = lambda * normal;
        p.getVelocity(m_bodies[0]) += impulse * invMass1;
        rb->addForce(-impulse, cp2);

        // friction
        Vector3r t = relV - vn * normal;
        Real tLen = t.norm();
        if (tLen > 1e-6) {
            t /= tLen;
            Real w_t = invMass1 + invMass2 + t.dot(r2_hat * invInertia2 *
r2_hat * t);
            if (w_t > 1e-6) {
                Real lambda_t = -tLen / w_t;
                if (lambda_t > -m_frictionCoeff * lambda && lambda_t <
m_frictionCoeff * lambda) {
                    p.getVelocity(m_bodies[0]) += lambda_t * t * invMass1;
                    rb->addForce(-lambda_t * t, cp2);
                }
            }
        }
        return true;
    }
};

class ParticleTetContactConstraint : public Constraint {
public:
    static int TYPE_ID;
    std::array<unsigned int, 2> m_bodies;
    unsigned int m_solidIndex;
    unsigned int m_tetIndex;
    Vector3r m_bary;
    Real m_lambda;
    Real m_frictionCoeff;
    Eigen::Matrix<Real, 3, 3, Eigen::DontAlign> m_constraintInfo;
    Real m_invMasses[4];
    std::array<Vector3r, 4> m_x;
    std::array<Vector3r, 4> m_v;

    ParticleTetContactConstraint() : Constraint(2), m_lambda(0.0),
m_frictionCoeff(0.1) {}
    virtual int& GetTypeId() const { return TYPE_ID; }

    bool InitConstraint(SimulationModel& model, const unsigned int
particleIndex, const unsigned int solidIndex,
                        const unsigned int tetIndex, const Vector3r& bary,
const Vector3r& cp1, const Vector3r& cp2,
                        const Vector3r& normal, const Real dist, const Real
frictionCoeff) {
        m_bodies[0] = particleIndex;
        m_bodies[1] = solidIndex;
        m_tetIndex = tetIndex;
        m_bary = bary;
        m_frictionCoeff = frictionCoeff;
```

Лістинг Е – Клас Constraint, аркуш 12

```
        m_constraintInfo.col(0) = cp1;
        m_constraintInfo.col(1) = cp2;
        m_constraintInfo.col(2) = normal;

        TetModel* tm = model.getTetModels()[solidIndex];
        const unsigned int* tet = tm->getParticleMesh().getTets() + 4 *
tetIndex;
        ParticleData& p = model.getParticles();
        for (int i = 0; i < 4; i++) {
            m_invMasses[i] = p.getInvMass(tet[i]);
            m_x[i] = p.getPosition(tet[i]);
            m_v[i] = p.getVelocity(tet[i]);
        }
        return true;
    }

    bool SolvePositionConstraint(SimulationModel& model, const unsigned int
iter) {
        ParticleData& p = model.getParticles();
        TetModel* tm = model.getTetModels()[m_bodies[1]];
        const Vector3r& normal = m_constraintInfo.col(2);
        Vector3r cp2 = m_bary[0] * m_x[0] + m_bary[1] * m_x[1] + m_bary[2] *
m_x[2] + m_bary[3] * m_x[3];
        Vector3r delta = p.getPosition(m_bodies[0]) - cp2;
        Real dist = delta.dot(normal);

        if (dist > 0.0) {
            return false;
        }

        Real w = p.getInvMass(m_bodies[0]);
        for (int i = 0; i < 4; i++) {
            w += m_bary[i] * m_bary[i] * m_invMasses[i];
        }
        if (w < 1e-6) {
            return false;
        }

        Real lambda = -dist / w;
        m_lambda += lambda;

        p.getPosition(m_bodies[0]) += lambda * p.getInvMass(m_bodies[0]) *
normal;
        for (int i = 0; i < 4; i++) {
            unsigned int idx = tm->getParticleMesh().getTets()[4 * m_tetIndex
+ i];
            p.getPosition(idx) -= lambda * m_invMasses[i] * m_bary[i] *
normal;
        }
        return true;
    }

    bool SolveVelocityConstraint(SimulationModel& model, const unsigned int
iter) {
        ParticleData& p = model.getParticles();
        TetModel* tm = model.getTetModels()[m_bodies[1]];
        const Vector3r& normal = m_constraintInfo.col(2);
        Vector3r v1 = p.getVelocity(m_bodies[0]);
        Vector3r v2 = m_bary[0] * m_v[0] + m_bary[1] * m_v[1] + m_bary[2] *
m_v[2] + m_bary[3] * m_v[3];
```

Лістинг Е – Клас Constraint, аркуш 13

```
        Real vn = (v1 - v2).dot(normal);
        if (vn > 0.0) {
            return false;
        }

        Real w = p.getInvMass(m_bodies[0]);
        for (int i = 0; i < 4; i++) {
            w += m_bary[i] * m_bary[i] * m_invMasses[i];
        }
        if (w < 1e-6) {
            return false;
        }

        Real lambda = -vn / w;
        m_lambda += lambda;

        Vector3r impulse = lambda * normal;
        p.getVelocity(m_bodies[0]) += impulse * p.getInvMass(m_bodies[0]);
        for (int i = 0; i < 4; i++) {
            unsigned int idx = tm->getParticleMesh().getTets()[4 * m_tetIndex
+ i];
            p.getVelocity(idx) -= impulse * m_bary[i] * m_invMasses[i];
        }

        Vector3r t = (v1 - v2) - vn * normal;
        Real tLen = t.norm();
        if (tLen > 1e-6) {
            t /= tLen;
            Real w_t = p.getInvMass(m_bodies[0]);
            for (int i = 0; i < 4; i++) {
                w_t += m_bary[i] * m_bary[i] * m_invMasses[i];
            }
            if (w_t > 1e-6) {
                Real lambda_t = -tLen / w_t;
                if (lambda_t > -m_frictionCoeff * lambda && lambda_t <
m_frictionCoeff * lambda) {
                    p.getVelocity(m_bodies[0]) += lambda_t * t *
p.getInvMass(m_bodies[0]);
                    for (int i = 0; i < 4; i++) {
                        unsigned int idx = tm->getParticleMesh().getTets()[4
* m_tetIndex + i];
                        p.getVelocity(idx) -= lambda_t * t * m_bary[i] *
m_invMasses[i];
                    }
                }
            }
        }
        return true;
    }
};
int DistanceConstraint::TYPE_ID = 0;
int BallJoint::TYPE_ID = 1;
int HingeJoint::TYPE_ID = 2;
int IsometricBendingConstraint::TYPE_ID = 3;
int FEMTetConstraint::TYPE_ID = 4;
int VolumeConstraint::TYPE_ID = 5;
int RigidBodyContactConstraint::TYPE_ID = 6;
int ParticleRigidBodyContactConstraint::TYPE_ID = 7;
int ParticleTetContactConstraint::TYPE_ID = 8;
} // namespace PBD
```

Лістинг Е – Клас Constraint, аркуш 14

ДОДАТОК Ж

Клас Collision

```
#include <memory>
#include <vector>
#include "Common.h"
#include "Constraint.h"
#include "Particle.h"
#include <PBD/Simulation.h>
#include <PBD/SimulationModel.h>
#include <PBD/RigidBody.h>
#include <PBD/TetModel.h>
#include <PBD/TriangleModel.h>
#include <PBD/LineModel.h>
#include <PBD/FluidModel.h>
#include <PBD/CollisionDetection.h>
#include <PBD/DistanceFieldCollisionDetection.h>
#include <PBD/CubicSDFCollisionDetection.h>
#include <PBD/Constraints.h>

class Collision {
public:
    Collision() : tolerance(0.01), contactCallback(nullptr),
solidContactCallback(nullptr), contactCallbackUserData(nullptr),
solidContactCallbackUserData(nullptr) {
        collisionDetection =
std::make_unique<PBD::DistanceFieldCollisionDetection>();
    }

    ~Collision() {
        Cleanup();
    }

    inline void Init() {
        collisionDetection->init();
        collisionObjects.clear();
        contactData.clear();
    }

    inline void SetTolerance(Real val) {
        tolerance = val;
        collisionDetection->setTolerance(val);
    }

    inline Real GetTolerance() const {
        return tolerance;
    }

    inline void AddCollisionObject(PBD::SimulationModel& model, unsigned int
bodyIndex, unsigned int bodyType, const Vector3r* vertices, unsigned int
numVertices) {
        if (bodyType ==
PBD::CollisionDetection::CollisionObject::RigidBodyCollisionObjectType) {
            collisionDetection->addCollisionBox(bodyIndex, bodyType,
vertices, numVertices, Vector3r(1.0, 1.0, 1.0), true, false);
        }
```

Лістинг Ж – Клас Collision, аркуш 1

```
          else if (bodyType ==
PBD::CollisionDetection::CollisionObject::TetModelCollisionObjectType) {
            collisionDetection->addCollisionObject(bodyIndex, bodyType);
        } else if (bodyType ==
PBD::CollisionDetection::CollisionObject::TriangleModelCollisionObjectType) {
            collisionDetection->addCollisionObject(bodyIndex, bodyType);
        }
        collisionObjects.push_back(collisionDetection-
>getCollisionObjects().back());
    }

    inline void AddCubicSDFCollisionObject(PBD::SimulationModel& model,
unsigned int bodyIndex, unsigned int bodyType, const Vector3r* vertices,
unsigned int numVertices, const std::string& sdfFile, const Vector3r& scale)
{
        auto* cubicCd =
dynamic_cast<PBD::CubicSDFCollisionDetection*>(collisionDetection.get());
        if (!cubicCd) {
            collisionDetection =
std::make_unique<PBD::CubicSDFCollisionDetection>();
            cubicCd =
dynamic_cast<PBD::CubicSDFCollisionDetection*>(collisionDetection.get());
        }
        cubicCd->addCubicSDFCollisionObject(bodyIndex, bodyType, vertices,
numVertices, sdfFile, scale, true, false);
        collisionObjects.push_back(collisionDetection-
>getCollisionObjects().back());
    }

    inline void UpdateAABBs(PBD::SimulationModel& model) {
        collisionDetection->updateAABBs(model);
    }

    inline void DetectCollisions(PBD::SimulationModel& model) {
        contactData.clear();
        collisionDetection->collisionDetection(model);

        PBD::ParticleData& particles = model.getParticles();

std::vector<std::vector<PBD::DistanceFieldCollisionDetection::ContactData>>
contacts_mt;

        // collisions between rigids
        for (unsigned int i = 0; i < model.getRigidBodies().size(); i++) {
            PBD::RigidBody* rb1 = model.getRigidBodies()[i];
            PBD::DistanceFieldCollisionObject* co1 =
dynamic_cast<PBD::DistanceFieldCollisionObject*>(collisionObjects[i]);
            for (unsigned int j = i + 1; j < model.getRigidBodies().size();
j++) {
                PBD::RigidBody* rb2 = model.getRigidBodies()[j];
                PBD::DistanceFieldCollisionObject* co2 =
dynamic_cast<PBD::DistanceFieldCollisionObject*>(collisionObjects[j]);
                if (co1 && co2) {
                    collisionDetection->collisionDetectionRigidBodies(rb1,
co1, rb2, co2, model.getRestitutionCoeff(), model.getFrictionCoeff(),
contacts_mt);
                    for (const auto& contacts : contacts_mt) {
                        contactData.insert(contactData.end(),
contacts.begin(), contacts.end());
        }}}}
```

Лістинг Ж – Клас Collision, аркуш 2

```
        // collisions between particles and rigid bodies
        for (unsigned int i = 0; i < model.getTriangleModels().size(); i++) {
            PBD::TriangleModel* tm = model.getTriangleModels()[i];
            unsigned int offset = tm->getIndexOffset();
            unsigned int numVert = tm->getParticleMesh().numVertices();
            PBD::DistanceFieldCollisionObject* co1 =
dynamic_cast<PBD::DistanceFieldCollisionObject*>(collisionObjects[i]);
            for (unsigned int j = 0; j < model.getRigidBodies().size(); j++)
{
                PBD::RigidBody* rb = model.getRigidBodies()[j];
                PBD::DistanceFieldCollisionObject* co2 =
dynamic_cast<PBD::DistanceFieldCollisionObject*>(collisionObjects[j]);
                if (co1 && co2) {
                    collisionDetection->collisionDetectionRBSolid(particles,
offset, numVert, co1, rb, co2, model.getRestitutionCoeff(),
model.getFrictionCoeff(), contacts_mt);
                    for (const auto& contacts : contacts_mt) {
                        contactData.insert(contactData.end(),
contacts.begin(), contacts.end());
                    }
                }
            }
        }

        // Collisions between tetrahedra
        for (unsigned int i = 0; i < model.getTetModels().size(); i++) {
            PBD::TetModel* tm1 = model.getTetModels()[i];
            unsigned int offset = tm1->getIndexOffset();
            unsigned int numVert = tm1->getParticleMesh().numVertices();
            PBD::DistanceFieldCollisionObject* co1 =
dynamic_cast<PBD::DistanceFieldCollisionObject*>(collisionObjects[i]);
            for (unsigned int j = i + 1; j < model.getTetModels().size();
j++) {
                PBD::TetModel* tm2 = model.getTetModels()[j];
                PBD::DistanceFieldCollisionObject* co2 =
dynamic_cast<PBD::DistanceFieldCollisionObject*>(collisionObjects[j]);
                if (co1 && co2) {
                    collisionDetection-
>collisionDetectionSolidSolid(particles, offset, numVert, co1, tm2, co2,
model.getRestitutionCoeff(), model.getFrictionCoeff(), contacts_mt);
                    for (const auto& contacts : contacts_mt) {
                        contactData.insert(contactData.end(),
contacts.begin(), contacts.end());
                    }
                }
            }
        }

        // FluidModel collisions
        for (unsigned int i = 0; i < model.getFluidModels().size(); i++) {
            PBD::FluidModel* fm = model.getFluidModels()[i];
            unsigned int numParticles = fm->getParticles().size();
            PBD::DistanceFieldCollisionObject* co1 =
dynamic_cast<PBD::DistanceFieldCollisionObject*>(collisionObjects[i]);
            for (unsigned int j = 0; j < model.getRigidBodies().size(); j++)
{
                PBD::RigidBody* rb = model.getRigidBodies()[j];
                PBD::DistanceFieldCollisionObject* co2 =
dynamic_cast<PBD::DistanceFieldCollisionObject*>(collisionObjects[j]);
```

Лістинг Ж – Клас Collision, аркуш 3

```
if (co1 && co2) {
                    collisionDetection->collisionDetectionRBSolid(particles,
0, numParticles, co1, rb, co2, model.getRestitutionCoeff(),
model.getFrictionCoeff(), contacts_mt);
                    for (const auto& contacts : contacts_mt) {
                        contactData.insert(contactData.end(),
contacts.begin(), contacts.end());
                    }
                }
            }
        }
    }

    // solve found collisions
    inline void ResolveCollisions(PBD::SimulationModel& model, double dt, int
substeps) {
        double substepDt = dt / substeps;
        for (int i = 0; i < substeps; i++) {
            for (const auto& contact : contactData) {
                if (contact.m_type ==
PBD::CollisionDetection::RigidBodyContactType) {
                    PBD::RigidBodyContactConstraint constraint;
                    constraint.initConstraint(model, contact.m_index1,
contact.m_index2, contact.m_cp1, contact.m_cp2,
                                              contact.m_normal,
contact.m_dist, model.getRestitutionCoeff(), 1.0, contact.m_friction);
                    constraint.solveVelocityConstraint(model, 1);
                } else if (contact.m_type ==
PBD::CollisionDetection::ParticleRigidBodyContactType) {
                    PBD::ParticleRigidBodyContactConstraint constraint;
                    constraint.initConstraint(model, contact.m_index1,
contact.m_index2, contact.m_cp1, contact.m_cp2,
                                              contact.m_normal,
contact.m_dist, model.getRestitutionCoeff(), 1.0, contact.m_friction);
                    constraint.solveVelocityConstraint(model, 1);
                } else if (contact.m_type ==
PBD::CollisionDetection::ParticleSolidContactType) {
                    PBD::ParticleTetContactConstraint constraint;
                    constraint.initConstraint(model, contact.m_index1,
contact.m_index2, contact.m_elementIndex2, contact.m_bary2,
                                              contact.m_cp1, contact.m_cp2,
contact.m_normal, contact.m_dist, contact.m_friction);
                    constraint.solvePositionConstraint(model, 1);
                    constraint.solveVelocityConstraint(model, 1);
                }
            }

            // apply user's contraints
            PBD::ParticleData& particles = model.getParticles();
            for (const auto& constraint : particleConstraints) {
                constraint->Solve(particles, substepDt);
            }
        }
    }

    // user's constraint for particle
    inline void AddParticleConstraint(std::shared_ptr<Constraint> constraint)
{
        particleConstraints.push_back(constraint);
    }
```

Лістинг Ж – Клас Collision, аркуш 4

```
    // general collision callback
    inline void
SetContactCallback(PBD::CollisionDetection::ContactCallbackFunction callback,
void* userData) {
        contactCallback = callback;
        contactCallbackUserData = userData;
        collisionDetection->setContactCallback(callback, userData);
    }

    // callback for rigid collision
    inline void
SetSolidContactCallback(PBD::CollisionDetection::SolidContactCallbackFunction
callback, void* userData) {
        solidContactCallback = callback;
        solidContactCallbackUserData = userData;
        collisionDetection->setSolidContactCallback(callback, userData);
    }

    inline void Cleanup() {
        collisionObjects.clear();
        contactData.clear();
        particleConstraints.clear();
        collisionDetection->cleanup();
    }

private:
    std::unique_ptr<PBD::CollisionDetection> collisionDetection;
    std::vector<PBD::CollisionObject*> collisionObjects;
    std::vector<PBD::DistanceFieldCollisionDetection::ContactData>
contactData;
    std::vector<std::shared_ptr<Constraint>> particleConstraints;
    Real tolerance;
    PBD::CollisionDetection::ContactCallbackFunction contactCallback;
    PBD::CollisionDetection::SolidContactCallbackFunction
solidContactCallback;
    void* contactCallbackUserData;
    void* solidContactCallbackUserData;
};
```

Лістинг Ж – Клас Collision, аркуш 5