МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Чорноморський національний університет імені Петра Могили

Факультет комп’ютерних наук

Кафедра інтелектуальних інформаційних систем

ДОПУЩЕНО ДО ЗАХИСТУ

Завідувач кафедри інтелектуальних
інформаційних систем
__________________Євген СІДЕНКО
«____»___________ 2026 р.

КВАЛІФІКАЦІЙНА РОБОТА

НА ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ БАКАЛАВР

РОЗРОБКА ТРЬОХВИМІРНИХ МОДЕЛЕЙ ТА ІМІТАЦІЇ ФІЗИЧНИХ ПРОЦЕСІВ У РЕАЛЬНОМУ ЧАСІ

Спеціальність 122 Комп’ютерні науки

Освітня програма «Комп’ютерні науки»

Здобувач __________ Яна МЕДВЕДЄВА

«___»_______________ 2026 р.

Керівник канд. техн. наук, доцент ____________ Анжела БОЙКО

«___»_______________ 2026 р.

Миколаїв – 2026

Чорноморський національний університет імені Петра Могили
(повне найменування закладу вищої освіти)

Факультет	Комп’ютерних наук
Кафедра	Інтелектуальних інформаційних систем
Рівень вищої освіти	Перший (бакалаврський)
Освітній ступень	Бакалавр
Спеціальність	122 Комп’ютерні науки
Освітня програма	Комп’ютерні науки

ЗАТВЕРДЖУЮ

Завідувач кафедри інтелектуальних інформаційних систем

________________ Євген СІДЕНКО

«____» _____________ 2025 р.

ЗАВДАННЯ
на кваліфікаційну роботу здобувача

Медведєва Яна Ігорівна
(прізвище, ім’я, по батькові здобувача)

1. Тема кваліфікаційної роботи: Розробка трьохвимірних моделей та імітації фізичних процесів у реальному часі

Керівник роботи: Бойко Анжела Петрівна, доцент кафедри дизайну, кандидат технічних наук, доцент.
(прізвище, ім’я, по батькові, посада, науковий ступінь, вчене звання)

Затверджена наказом ЧНУ ім. Петра Могили від «25» грудня 2025 р. № 353.

2. Строк представлення кваліфікаційної роботи «16» червня 2026 р.

3. Очікуваний результат роботи та початкові дані, якщо такі потрібні розроблений програмний комплекс, що забезпечує візуалізацію тривимірних об’єктів та динамічний прорахунок фізичних взаємодій у режимі реального часу.

4. Перелік питань, що підлягають розробці:

– дослідження основ фізичної імітації в реальному часі, аналіз законів та методів, які лягають в основу симуляції руху;

– аналіз особливостей виконання фізичних обчислень у реальному часі та огляд сучасних фізичних рушіїв;

– розробка тривимірної геометрії об'єктів сцени та реалізація методів виявлення й обробки зіткнень;

– програмна реалізація фізичного ядра та засобів візуалізації фізичних процесів;

– дослідження та впровадження підходів до підвищення продуктивності симуляцій у реальному часі;

– проведення тестування точності та продуктивності розробленої системи, аналіз отриманих результатів та оцінка ефективності реалізованих рішень.

5. Перелік графічних матеріалів:_презентація__.

Керівник роботи ____________________ ____ Анжела БОЙКО ___
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Здобувач _____________________ __ Яна МЕДВЕДЄВА __
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Дата видачі завдання «_21_» _грудня_ 2025 р.

КАЛЕНДАРНИЙ ПЛАН

кваліфікаційної роботи

Тема:__ Розробка трьохвимірних моделей та імітації фізичних процесів у реальному часі__.

№	Найменування роботи	Початок	Закінчення	Примітки
1	Отримання завдання на виконання КР	21.12.2025	24.12.2025	Виконано
2	Аналіз предметної області та постановка задачі	25.12.2026	30.01.2026	Виконано
3	Огляд літературних джерел за темою кваліфікаційної роботи, зокрема огляд публікацй та аналогічних систем щодо імітації фізичних процесів	31.01.2026	01.03.2026	Виконано
4	Огляд існуючих фізичних рушіїв для вирішення поставленої задачі	20.03.2026	01.04.2026	Виконано
5	Реалізація обраних технологій з аналізом отриманих результатів	02.04.2026	24.05.2026	Виконано
6	Перший попередній захист КР на засіданні комісії кафедри	25.05.2026	25.05.2026	Виконано
7	Корегування роботи за результатами попереднього захисту	26.05.2026	04.06.2026	Виконано
8	Другий попередній захист КР на засіданні комісії кафедри	05.06.2026	05.06.2026	Виконано
9	Доробка та остаточне оформлення КР	06.06.2026	14.06.2026	Виконано
10	Подання КРБ, її електронної копії та інших документів (відгуку, рецензії) до захисту	15.06.2026	19.06.2026	Виконано

Керівник роботи ____________________ ____Анжела БОЙКО___
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Здобувач ____________________ ___Яна МЕДВЕДЄВА__
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Дата складання календарного плану

«_29_» __січня__ 2026 р.

АНОТАЦІЯ

до кваліфікаційної роботи бакалавра студентки 402 групи ЧНУ ім. Петра Могили

Медведєвої Яни Ігорівни

на тему: «РОЗРОБКА ТРЬОХВИМІРНИХ МОДЕЛЕЙ ТА ІМІТАЦІЇ ФІЗИЧНИХ ПРОЦЕСІВ У РЕАЛЬНОМУ ЧАСІ»

Актуальність роботи обумовлена наступними чинниками:

– зростання попиту на високореалістичні інтерактивні системи. Сучасні галузі, такі як індустрія комп'ютерних ігор, віртуальна та доповнена реальність, вимагають не просто статичної 3D-графіки, а динамічного середовища, яке реагує на дії користувача за законами фізики без затримок;

– необхідність безпечного та дешевшого навчання. Створення тренажерів для пілотів, водіїв, операторів важкої техніки або медичних хірургів дозволяє відпрацьовувати критичні ситуації у безпечному віртуальному просторі. Точна імітація фізики у реальному часі є критично важливою для того, щоб отримані навички відповідали реальності;

– розвиток концепції «цифрових двійників». У сучасній промисловості та архітектурі тестування поведінки об'єктів під дією фізичних навантажень у реальному часі дозволяє оптимізувати витрати на виробництво та запобігти аваріям ще на етапі проектування;

– технологічний прогрес апаратного забезпечення. Поява потужних багатоядерних процесорів та графічних прискорювачів з підтримкою апаратного трасування променів та спеціалізованих фізичних рушіїв відкрила нові можливості для обчислення складних фізичних систем безпосередньо під час рендерингу.

Об'єктом кваліфікаційної роботи є процеси тривимірного моделювання та програмної імітації фізичних явищ у динамічних комп'ютерних середовищах..

Предметом роботи є методи та засоби побудови 3D-моделей, алгоритми імітації фізичних процесів, фізичного моделювання та інструментальні засоби розробки графіки реального часу.

Метою кваліфікаційної роботи є дослідження алгоритмів тривимірної візуалізації та розробка програмної системи імітації фізичних процесів у реальному часі з використанням сучасних методів чисельного моделювання та комп'ютерної графіки.

Робота складається зі вступу, основної частини (3 розділи), висновків та додатків. Пояснювальна записка складається зі вступу, трьох розділів, висновків та додатків. У вступі обґрунтовано актуальність роботи, визначено об'єкт, предмет, мету та завдання дослідження.

У першому розділі проводиться аналіз законів класичної механіки, які лягають в основу імітації, методів чисельного інтегрування та рушіїв, що використовуються в індустрії для побудови 3D-моделей та імітації фізичних явищ.

У другому розділі описано процес моделювання об'єкта та створення сітки, вибір фаз обробки зіткнень та використаний стек технологій, включаючи обрану мову та графічний API.

У третьому розділі описана реалізація фізичного ядра та візуалізатора, застосування методів для підвищення продуктивності, проведення тестування точності та продуктивності розробленої системи, аналіз отриманих результатів та оцінка ефективності реалізованих рішень.

Кваліфікаційна робота бакалавра містить 92 сторінки, 5 таблиць, 15 рисунків, 31 джерело та 3 додатки. Ключові слова: рендеринг, фізика, симуляція, 3D-моделювання, фізичний рушій, виявлення зіткнень, графічний API, Python, OpenGL.

ABSTRACT

to the bachelor's thesis of a student of group 402 of the Petro Mohyla Black Sea University

Medvedieva Yana Igorivna

on the topic: «DEVELOPMENT OF THREE-DIMENSIONAL MODELS AND SIMULATION OF PHYSICAL PROCESSES IN REAL TIME»

The relevance of the work is determined by the following factors:

– increasing demand for highly realistic interactive systems. Modern industries, such as the computer game industry, virtual and augmented reality, require not just static 3D graphics, but a dynamic environment that responds to user actions according to the laws of physics without delays;

– the need for safe and cheaper training. Creating simulators for pilots, drivers, heavy equipment operators or medical surgeons allows you to practice critical situations in a safe virtual space. Accurate real-time physics simulation is critical to ensuring that the skills learned match reality;

– development of the concept of «digital doubles». In modern industry and architecture, testing the behavior of objects under the influence of physical exertion in real time allows to optimize production costs and prevent accidents even at the design stage;

– technological progress of hardware. The advent of powerful multi-core processors and graphics accelerators with support for hardware ray tracing and specialized physical engines opened up new opportunities for computing complex physical systems directly during rendering.

The object of the qualification work is the processes of three-dimensional modeling and software simulation of physical phenomena in dynamic computer environments..

The subject of the work is methods and means of building 3D models, algorithms for simulating physical processes, physical modeling and tools for developing real-time graphics.

The purpose of the qualification work is the study of three-dimensional visualization algorithms and the development of a software system for simulating

physical processes in real time using modern methods of numerical modeling and computer graphics.

The work consists of an introduction, the main part (3 chapters), conclusions and appendices. The explanatory note consists of an introduction, three chapters, conclusions and appendices. The introduction substantiates the relevance of the work, defines the object, subject, purpose and tasks of the research.

The first chapter analyzes the laws of classical mechanics, which form the basis of imitation, numerical integration methods and drivers used in the industry to build 3D models and simulate physical phenomena.

The second chapter describes the process of object modeling and grid creation, the selection of collision processing phases, and the technology stack used, including the selected language and graphics API.

The third chapter describes the implementation of the physical core and visualizer, the application of methods to increase productivity, testing the accuracy and productivity of the developed system, analyzing the obtained results and evaluating the effectiveness of the implemented solutions.

The bachelor’s thesis contains 92 pages, 5 tables, 15 figures, 31 sources and 3 appendices. Keywords: rendering, physics, simulation, 3D modeling, physical engine, collision detection, graphics API, Python, OpenGL.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ 3
ВСТУП........ 4
1 ТЕОРЕТИЧНІ ОСНОВИ ТА АНАЛІЗ МЕТОДІВ ФІЗИЧНОЇ ІМІТАЦІЇ В РЕАЛЬНОМУ ЧАСІ........ 6
1.1 Огляд і аналіз законів класичної механіки, які лягають в основу імітації 6
1.2 Аналіз методів чисельного інтегрування 8
1.3 Специфіка обчислень у реальному часі........ 12
1.4 Аналіз сучасного ринку фізичних рушіїв........ 13
Висновки до розділу 1 16
2 ПРОЕКТУВАННЯ АРХІТЕКТУРИ СИСТЕМИ ТА РОЗРОБКА 3D-МОДЕЛЕЙ 17
2.1 Вибір мови програмування 17
2.2 Вибір графічного API 22
2.3 Схема взаємодії графічного ядра та фізичного рушія........ 25
2.4 Створення 3D-геометрії........ 17
2.5 Принципи та методи виявлення зіткнень об'єктів........ 19
Висновки до розділу 2 29
3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ 30
3.1 Реалізація фізичного ядра та візуалізатора 30
3.2 Підходи для підвищення продуктивності симуляцій........ 47
3.3 Тестування точності та продуктивності 49
3.4 Аналіз отриманих результатів 53
Висновки до розділу 3 58
ВИСНОВКИ........ 60
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ........ 62
ДОДАТОК А Лістинг коду модуля physics_engine.py 66
ДОДАТОК Б Лістинг коду модуля render_core.py 75
ДОДАТОК В Лістинг коду модуля world_manager.py 83

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

ААА-ігри – високобюджетні відеоігри, створені великими компаніями

Open-source – відкритий для загального доступу

3D – 3-Dimensional

AABB – Axis-Aligned Bounding Boxes

API – Application Programming Interface

CPU – Central Processing Unit

EBO – Element Buffer Object

FPS – Frames Per Second

GPGPU – General-Purpose Graphics Processing Unit

GPU – Graphics Processing Unit

LOD – Level of Detail

PC – Personal Computer

SIMD – Single Instruction, Multiple Data

VAO – Vertex Array Object

VBO – Vertex Buffer Object

VR – Virtual Reality

ВСТУП

Сучасний етап розвитку сфери інформаційних технологій характеризується стрімким збільшенням складності та інтерактивності систем тривимірної візуалізації. Імітація фізичних процесів у реальному часі є однією з найбільш наукомістких та обчислювально-складних задач у комп’ютерної графіки. Вона вимагає розв’язання складних систем диференціальних рівнянь за мілісекунди, що робить проблему оптимізації обчислень критично важливою.

Актуальність дослідження обумовлена такими чинниками:

– зростання попиту на високореалістичні інтерактивні системи. Сучасні галузі, такі як індустрія комп’ютерних ігор, віртуальна (VR) та доповнена (AR) реальність, вимагають не просто статичної 3D-графіки, а динамічного середовища, яке реагує на дії користувача за законами фізики без затримок;

– необхідність безпечного та дешевшого навчання (Симулятори). Створення тренажерів для пілотів, водіїв, операторів важкої техніки або медичних хірургів дозволяє відпрацьовувати критичні ситуації у безпечному віртуальному просторі. Точна імітація фізики (гравітація, зіткнення, гідродинаміка) у реальному часі є критично важливою для того, щоб отримані навички відповідали реальності;

– розвиток концепції «цифрових двійників». У сучасній промисловості та архітектурі тестування поведінки об’єктів під дією фізичних навантажень у реальному часі дозволяє оптимізувати витрати на виробництво та запобігти аваріям ще на етапі проектування;

– технологічний прогрес апаратного забезпечення. Поява потужних багатоядерних процесорів (CPU) та графічних прискорювачів (GPU) з підтримкою апаратного трасування променів та спеціалізованих фізичних рушіїв (наприклад, NVIDIA PhysX, Chaos, Havok) відкрила нові можливості для обчислення складних фізичних систем безпосередньо під час рендерингу.

Таким чином, розробка тривимірних моделей та ефективних алгоритмів імітації фізичних процесів у реальному часі є актуальною науково-технічною задачею.

Метою кваліфікаційної роботи є дослідження алгоритмів тривимірної візуалізації та розробка програмної системи імітації фізичних процесів у реальному часі з використанням сучасних методів чисельного моделювання та комп’ютерної графіки.

Для досягнення поставленої мети були визначені такі завдання:

– проаналізувати існуючі алгоритми тривимірної візуалізації, математичні моделі та чисельні методи інтегрування, що використовуються для обчислення траєкторій руху об'єктів та виявлення колізій;

– здійснити порівняльний аналіз сучасних фізичних рушіїв для моделювання фізичних процесів у реальному часі;

– спроектувати високопродуктивну архітектуру програмної системи, що забезпечує стабільну частоту оновлення кадрів (FPS) в умовах обмежених обчислювальних ресурсів;

– реалізувати програмні модулі імітації базових фізичних властивостей та взаємодій об’єктів (включаючи масу, силу тертя, пружність та гравітацію);

– поліпшити алгоритми виявлення зіткнень (Collision Detection) для об’єктів різної геометричної складності;

– провести тестування розробленої програмної системи.

Об’єктом дослідження є процеси тривимірного моделювання та програмної імітації фізичних явищ у динамічних комп’ютерних середовищах.

Предметом дослідження є методи та засоби побудови 3D-моделей, алгоритми імітації фізичних процесів, фізичного моделювання та інструментальні засоби розробки графіки реального часу.

Практичне значення роботи полягає у розробці програмного комплексу, який демонструє ефективне поєднання методів 3D-візуалізації та фізичних розрахунків. Результати роботи можуть бути використані як основа для подальшого розвитку фізичних рушіїв, навчальних симуляторів, інтерактивних тренажерів та інших програмних систем, що потребують реалістичного відтворення фізичних явищ.

1 ТЕОРЕТИЧНІ ОСНОВИ ТА АНАЛІЗ МЕТОДІВ ФІЗИЧНОЇ ІМІТАЦІЇ В РЕАЛЬНОМУ ЧАСІ

1.1 Огляд і аналіз законів класичної механіки, які лягають в основу імітації

В основі сучасної фізичної імітації в реальному часі лежать різноманітні закони фізики та побудова математичних моделей, що описують динаміку твердих тіл, м’яких об’єктів або рідин. Головним завданням таких моделей є апроксимація фізичних процесів навколишнього світу за допомогою обчислювальних алгоритмів, які повинні виконуватися за строго обмежений проміжок часу, зазвичай від 16 до 33 мілісекунд для забезпечення плавності при 30-60 FPS. А оскільки більшість фізичних симуляцій орієнтована на відтворення руху макроскопічних об’єктів, їх теоретичною основою виступають принципи класична механіка, оскільки саме вони забезпечують достатньо точний опис руху макроскопічних тіл за відносно невеликих обчислювальних витрат.

Класична механіка являє собою розділ фізики, який вивчає рух на основі законів Ньютона та принципу відносності Галілея, тому її часто називають «Ньютоновою механікою». Дану механіку, в свою чергу, поділяють на кінематику, що вивчає рух тіл і при цьому не бере до уваги діючі сили, динаміку, яка вивчає рух тіл під дією сил, та статику, що вивчає фізику тіл у спокої та питання їхньої рівноваги. Самі об’єкти, які вивчаються механікою, називають механічними системами. Завданням механіки є вивчення властивостей механічних систем, зокрема їхньої еволюції в часі.

Базовими поняттями класичної механіки є поняття сили, маси та руху. Маса m в класичній механіці визначається як міра інертності, здатності тіла до збереження стану спокою або рівномірного прямолінійного руху за відсутності дії на нього сил. З іншого боку, сили, які діють на тіло, змінюють стан його руху, викликаючи прискорення. Взаємодія цих двох ефектів і є головною темою механіки Ньютона.

Іншими важливими поняттями цього розділу фізики також є енергія, імпульс та момент імпульсу, які можуть передаватись між об’єктами в процесі взаємодії.

Енергія механічної системи складається з її кінетичної та потенціальної, залежної від положення тіла відносно інших тіл, енергій. Щодо цих фізичних величин діють фундаментальні закони збереження.

Основу більшості фізичних рушіїв (також physics engines) становлять закони класичної механіки, насамперед три закони Ньютона та закони збереження імпульсу та енергії, які розглядають та визначають поведінку макроскопічних об'єктів під час взаємодії.

1. **Перший закон Ньютона, закон інерції:** об'єкт залишається у стані спокою або рівномірного прямолінійного руху, доки на нього не діють зовнішні сили. У цифровому середовищі це означає, що за відсутності вхідних даних, як сил тертя, чи гравітації, об'єкт повинен зберігати свій вектор швидкості між кадрами.

2. **Другий закон Ньютона, основне рівняння динаміки:** прискорення об'єкта прямо пропорційне рівнодійній усіх сил, що діють на нього, і обернено пропорційне його масі, що математично виражається формулою:

$$\sum \vec{F} = m \times \vec{a}, \tag{1.1}$$

де $\sum \vec{F}$ – векторна сума сил, що діють на тіло, Н;

m – маса тіла, кг;

$\vec{a}$ – пришвидшення тіла, м/с^2.

Формула (1.1) є основою для більшості фізичних рушіїв і для імітації в реальному часі вона перетворюється на задачу інтегрування, де знаючи прикладені сили, потрібно обчислити прискорення a, на основі якого потрібно оновити швидкість v та позицію об'єкта x.

3. **Третій закон Ньютона, закон дії та протидії:** два тіла взаємодіють одне з одним із силами, напрямленим вздовж однієї прямої, рівними за модулем та протилежними за напрямом. Цей закон є ключовим при моделюванні зіткнень та взаємодії об'єктів, наприклад, коли м'яч вдаряється об стіну, стіна діє на м'яч із такою ж силою, з якою м'яч діє на стіну, що призводить до відскоку.

4. **Закон збереження імпульсу:** векторна сума імпульсів тіл, які становлять замкнену систему, не змінюється. Це основний принцип для вирішення зіткнень

(розв'язання задач collision resolution). При контакті двох об'єктів розподіл швидкостей після удару розраховується так, щоб сума їхніх імпульсів до і після події була однаковою, з урахуванням коефіцієнта пружності. Імпульс p визначається як:

$$p = m \times \vec{v}, \tag{1.2}$$

де m – маса тіла, кг;

$\vec{v}$ – вектор швидкості тіла, м/с.

Зміна імпульсу дозволяє обчислювати результати миттєвих подій, таких як вибухи або швидкі зіткнення, без детального моделювання деформації матеріалів. Цей закон використовується для вирішення задач зіткнень, коли необхідно розрахувати нові швидкості об'єктів після контакту, враховуючи їхні маси. Часто ефективніше використовувати поняття імпульсу в тих імітаціях, де обчислювальні ресурси обмежені, ніж безпосередньо обчислювати сили в кожний мікромомент часу.

5. **Закон збереження енергії:** якщо між тілами системи діють тільки сили тяжіння та сили пружності, механічна енергія замкненої системи тіл зберігається. Енергія не виникає з нічого і не зникає, вона лише переходить з одного виду в інший. У фізичних рушіях часто моделюють дисипацію, втрату кінетичної енергії через тертя або непружне зіткнення, щоб імітація виглядала реалістичною, але при цьому не була «вічною».

Також для повноцінної імітації твердих тіл закони Ньютона поширюються також і на обертальний рух, а саме на момент сили, аналог сили для обертання, момент інерції, аналог маси, що визначає опір тіла обертальному прискоренню, та кутовий імпульс, при збереженні якого пояснюється, чому об'єкти продовжують обертатися навколо власної осі після припинення дії зовнішніх факторів.

1.2 Аналіз методів чисельного інтегрування

Оскільки рівняння руху в класичній механіці описуються неперервними диференціальними рівняннями, їхнє точне аналітичне розв'язання можливе лише

для обмеженого кола задач. У більшості практичних випадків, особливо під час моделювання складних фізичних систем у комп'ютерній графіці та ігрових рушіях, застосовуються методи чисельного інтегрування. Їхнє основне призначення полягає в наближеному обчисленні положення, швидкості та інших параметрів об'єктів у дискретні моменти часу. Вибір конкретного методу інтегрування залежить від вимог до точності, стабільності обчислень та доступних обчислювальних ресурсів. Найбільш розповсюдженими є:

– метод Ейлера є найпростіший, але при цьому найменш стабільним, де нова позиція обчислюється та оновлюється на основі поточної швидкості та прискорення, що виражається в наступних формулах (1.3) та (1.4):

$$x_{n+1} = x_n + v_n \cdot \Delta t, \quad (1.3)$$

$$v_{n+1} = v_n + a_n \cdot \Delta t, \quad (1.4)$$

де x_{n+1} та x_n – наступне та поточне положення тіла;

v_n – вектор швидкості тіла, м/с;

a_n – пришвидшення тіла, м/с^2;

Δt – часовий крок, с;

– метод інтегрування Верле, що часто використовується в комп'ютерній фізиці та ігрових рушіях для симуляції тканин та систем частинок, оскільки він не використовує швидкості для обчислення майбутніх координат, а базується на попередніх позиціях об'єкта, що забезпечує більшу стабільність при великих кроках часу:

$$x_{t+\Delta t} = 2x_t - x_{t-\Delta t} + a_t \Delta t^2, \quad (1.5)$$

де $x_{t+\Delta t}, x_t, x_{t-\Delta t}$ – наступне, поточне та попереднє положення тіла;

a_n – пришвидшення тіла, м/с^2;

Δt – часовий крок, с;

– методи Рунге-Кутти є високоточним сімейством методів, що базується на порядках, і чим вище порядок, тим менша похибка і вища точність. Такі методи здебільшого використовуються в наукових симуляціях, де важлива мінімальна похибка, хоча вони є більш ресурсомісткими для реального часу. Найбільш

поширенішим і відомим порядком є четвертий, який балансує між складністю реалізації, точністю та стабільністю і має наступну формулу:

$$x_{n+1} = x_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4), \qquad (1.6)$$

де x_{n+1} та x_n – наступне та поточне положення тіла;

h – крок інтегрування;

k_1, k_2, k_3, k_4 – коефіцієнти для обчислення:

$k_1 = f(x_n, y_n)$;

$k_2 = f(x_n + \frac{h}{2}, y_n + \frac{k_1}{2})$;

$k_3 = f(x_n + \frac{h}{2}, y_n + \frac{k_2}{2})$;

$k_4 = f(x_n + h, y_n + k_3)$;

– методи Адамса-Башфорта є багатокроковими і використовують історію попередніх кроків для прогнозування наступного. Найбільш розповсюдженим є другий порядок, що наведено в формулі (1.7), але, оскільки дані методи багатокрокові, то перше поточне положення потрібно знаходити, використовуючи перший порядок, який подібний до метода Ейлера, згідно до формули (1.8):

$$x_{n+1} = x_n + \frac{h}{2}(3f(t_n, x_n) - f(t_{n-1}, x_{n-1})), \qquad (1.7)$$

$$x_{n+1} = x_n + hf(t_n, x_n), \qquad (1.8)$$

де x_{n+1} та x_n – наступне та поточне положення тіла;

h – крок інтегрування;

$f(t_n, x_n)$ та $f(t_{n-1}, x_{n-1})$ – значення похідної в поточному та попередньому положенні.

В таблиці 1.1 наведено порівняльна статистика описаних вище методів інтегрування.

Таблиця 1.1 – Порівняльна таблиця методів чисельного інтегрування

Критерій	**Методи**			
	Ейлер	**Верле**	**Рунге-Кутта (RK4)**	**Адамс-Башфорт**
Швидкість обчислень	Висока	Висока	Низька	Середня
Стабільність	Низька	Середня	Висока	Середня
Точність	Низька	Середня	Висока	Висока
Недоліки	Вимагає малого кроку для стабільності, інакше симуляція швидко «розлітається»	Потрібно більше пам'яті для зберігання попередньої позиції	Великі затрати для обчислення	Потребує «стартового» методу для перших кроків
Сфера застосування	Прості ігри, прототипи та візуалізації з необов'язковою точністю	Фізика тіл, тканини та мотузки, молекулярна динаміка, ігрова фізика	Наукові та інженерні симуляції, де точність важливіша за швидкість	Симуляції рідин, складні фізичні процеси з великою кількістю частинок

Вибір методу для потрібної задачі – це завжди компроміс між обчислювальною складністю, стабільністю та точністю.

Пріоритетність реального часу замість точності спостерігається в комп'ютерних іграх. У відеоіграх на фізику часто виділяється лише 2-5 мс на кадр. Методи подібні до Ейлера та Верле дозволяють отримати результат за один прохід обчислень без розв'язання складних матриць, оскільки користувачеві байдуже, чи впаде коробка на 1 см лівіше від «справжнього» значення, але при цьому важливо,

щоб вона не провалилася крізь підлогу і не полетіла в нескінченність через математичну помилку. Також прості методи легше інтегрувати з алгоритмами розв'язання контактів та обмежень.

А ось в наукових розрахунках, наприклад траєкторія супутника, навіть мікроскопічна помилка на старті призведе до катастрофічних наслідків через годину симуляції, тому тут здебільшого використовують RK4 або метод Адамса-Башфорта. Також, якщо в системі є дуже сильні пружини або високий опір, явні методи Ейлера та Верле стають нестабільними. Неявні методи дозволяють робити великі кроки за часом без втрати стабільності, хоча кожен крок потребує значних ресурсів CPU та GPU для ітераційних розв'язувань.

1.3 Специфіка обчислень у реальному часі

У підрозділі 1.2 було зазначено про часовий крок, тому орієнтуючись на даний термін потрібно також звернути увагу на процеси, в яких правильність результату залежить не лише від його логічної точності, а й від часу його отримання.

Часовий крок Δt – це дискретний проміжок часу, на який просувається симуляція під час однієї ітерації обчислень. Відрізняють фіксований крок та змінний крок. При фіксованому кроці симуляція завжди оновлюється на одну й ту саму величину, що є золотим стандартом для професійних рушіїв та стабільної фізики, оскільки він забезпечує відтворюваність результатів незалежно від потужності заліза. При змінному кроці симуляція використовує реальний час, що минув з попереднього кадру, що є зручним для реалізації деяких графічних застосувань, але може негативно впливати на фізичну достовірність моделі, оскільки якщо станеться раптове падіння продуктивності, або лаг, часовий крок може суттєво зрости, і об'єкти будуть «пролітати» крізь стіни.

Також для доброякісної симуляції потрібна стабільність – здатність системи повертатися до стану рівноваги або підтримувати енергетичний баланс без накопичення числових помилок, що призводять до «вибуху», нескінченного

зростання швидкостей або інших фізичних параметрів. Ключовими факторами, що впливають на зменшення стабільності, є:

– накопичення енергії, при якій деякі методи інтегрування, як метод Ейлера, можуть штучно додавати системі енергію, якої не існує в реальності;

– жорсткість, де системи з великими силами, як жорстокі пружини та висока в'язкість середовища, вимагають або дуже малих кроків, або спеціальних неявних методів інтегрування;

– проблема «тунелювання», де при великому часовому кроку об'єкт за один крок може опинитися по інший бік перешкоди, не зафіксувавши зіткнення.

Для усунення проблем стабільності використовують Continuous Collision Detection – метод, який перевіряє не лише фінальні позиції, а й траєкторію руху між ними. Він вимагає більше обчислень порівняно з дискретним виявленням зіткнень, але при цьому забезпечує вищу надійність.

Але ключовою проблемою для симуляції в реальному часі є обмеженість «бюджета часу». Якщо потрібно мати 60 FPS, то на весь кадр, включаючи рендеринг з фізикою та логікою, можливо лише 16.6 мс. Розрахунки твердих тіл зазвичай лягають на CPU. Однак для систем із тисячами частинок або рідин використовують паралельні обчислення на GPU за допомогою технологій CUDA або OpenCL. А оскільки фізика та рендеринг часто працюють на різних частотах, виникає потреба в інтерполяції, тобто якщо фізика оновилася раніше, ніж рендеринг, потрібно візуально згладити рух об'єкта між двома останніми станами, щоб уникнути «смикання» картинки, нерівномірного оновлення симуляції.

1.4 Аналіз сучасного ринку фізичних рушіїв

Сучасний ринок фізичних рушіїв поділений між комерційними гігантами, як NVIDIA, Microsoft чи Havok, та потужними open-source рішеннями. Вибір конкретного інструменту залежить від цільової платформи, бюджету, функціональних можливостей та вимог до продуктивності:

– NVIDIA PhysX є одним із найпоширеніших фізичних рушіїв у світі завдяки тривалій інтеграції в ігрові рушії як Unity та Unreal Engine. Він має потужну

підтримку GPU-прискорення, розвинені системи для імітації тканини, рідин та частинок, високу стабільність, величезну базу документації та підтримку практично всіх сучасних платформ, як PC, консолі та мобільні пристрої. Навіть незважаючи на те, що Unreal Engine перейшов на власний рушій Chaos, PhysX продовжує активно використовуватися для індустріальних симуляцій через платформу NVIDIA Omniverse;

– Havok Physics, розроблений компанією Microsoft, є одним із найбільш поширених фізичних рушіїв, що використовується в масштабних комерційних проєктах та AAA-іграх. Його перевагами є висока ефективність багатопотокових обчислень на центральному процесорі, швидкі алгоритми виявлення та обробки зіткнень, передбачуваність результатів моделювання, а також наявність додаткових модулів для анімації персонажів і симуляції руйнувань. Основним недоліком рушія є висока вартість ліцензування для комерційного використання;

– Bullet Physics є відомою бібліотекою з відкритим кодом. Має підтримку як жорстких, так і м’яких тіл. Загалом його використовують в кіноіндустрії для візуальних ефектів, робототехніці та в іграх. Навіть попри солідний вік, Bullet залишається фаворитом для академічних досліджень завдяки прозорості коду;

– Chaos Physics є власним рушієм компанії Epic Games для ігрового рушія Unreal Engine 5. Оскільки він спроектований для масштабних руйнувань та складної взаємодії в реальному часі, то й сама система рушія оптимізована для сучасних багатопотокових процесорів, що дозволяє обробляти велику кількість фізичних об’єктів;

– Jolt Physics був розроблений для гри Horizon Forbidden West. Наразі він демонструє значно кращу продуктивність на багатоядерних процесорах, ніж PhysX 4, і стає стандартом для користувацьких фізичних рушіїв і є чудовим вибором для розробників, які шукають швидку та просту в інтеграції альтернативу таким рішенням, як NVIDIA PhysX.

В таблиці 1.2 наведено загальна порівняльна статистика щодо описаних вище рушіїв.

Таблиця 1.2 – Порівняльна таблиця фізичних рушіїв

Критерій	**Рушій**				
	PhysX	**Havok**	**Bullet**	**Chaos**	**Jolt**
Розробник	NVIDIA	Microsoft	Open-source	Epic Games	Open-source
Ліцензія	BSD 3-Clause / Відкрита	Комерційна	Zlib / Відкрита	Комерційна, в складі Unreal Engine	MIT / Відкрита
Основна платформа	Unreal Engine 4, Unity	AAA-ігри	Blender, Cinema 4D, Godot	Unreal Engine 5	AAA-ігри
Апаратне прискорення	GPU (NVIDIA) + CPU	Переважно CPU	GPU (OpenGL) + CPU	CPU	CPU
Загальна спеціалізація	Ефекти частинок, тканина, руйнування	Стабільність твердих тіл, анімація персонажів	Наукові симуляції, VR, робото-техніка	Масштабні руйнування, інтеграція з Unreal Engine 5	Висока продуктив-ність для велико-масштабних сцен

Сучасні фізичні рушії забезпечують достатній рівень точності для більшості практичних застосувань, тому ключовими критеріями вибору дедалі частіше стають продуктивність, масштабованість та ефективність використання багатопотокових і графічних обчислень. Сучасні тенденції розвитку фізичного моделювання спрямовані на підвищення швидкодії, підтримку паралельних обчислень та забезпечення стабільності симуляції при великій кількості взаємодіючих об’єктів

Висновки до розділу 1

У першому розділі було проведено аналіз законів класичної механіки, методів чисельного інтегрування та складність обчислень для імітації фізичних процесів у реальному часі. Було зазначено переваги та недоліки різних фізичних рушіїв та обґрунтовано їх використання в різних спеціалізаціях.

Враховуючи один з ключових аспектів фізичних та ігрових рушіїв, а саме навантаження на обчислювальні ресурси, які можуть бути обмеженими, можна дійти висновку, що головною проблемою залишається пошук компромісу між фізичною достовірністю та швидкістю обчислень. Оскільки складність алгоритмів виявлення зіткнень та розв’язання обмежень зростає експоненційно з кількістю об’єктів у сцені, критично важливим є вибір оптимального методу інтегрування та просторового розбиття. Це обумовлює необхідність подальшого дослідження механізмів для удосконалення, які дозволяють підтримувати стабільну частоту кадрів без суттєвої втрати візуальної реалістичності.

2 ПРОЕКТУВАННЯ АРХІТЕКТУРИ СИСТЕМИ ТА РОЗРОБКА 3D-МОДЕЛЕЙ

2.1 Вибір мови програмування

Для реалізації системи імітації фізичних процесів у реальному часі критично важливо обрати інструменти, що забезпечують ефективність поєднання математичних обчислень та високорівневе керування з графічною підсистемою.

Мова програмування C++ вважається чудовим вибором для програмування в графіці та фізиці, дозволяючи розробнику вручну керувати пам’яттю, оскільки в імітаціях реального часу відсутність автоматичного «складальника сміття» гарантує відсутність раптових затримок. Також завдяки компіляції в машинний код та можливості використання SIMD-інструкцій, C++ добре підходить для математично інтенсивних обчислень. Також на даній мові програмування написано більшість фізичних та ігрових рушіїв, а також графічних API, як DirectX, OpenGL та Vulkan.

Мова C# є основним інструментом для швидкої ітерації та високорівневої розробки, дозволяючи зосередитися на логіці фізичних процесів, а не на роботі з пам’яттю. При чому, завдяки автоматичному керуванню пам’яттю та суворої типізації зменшується кількість критичних помилок, які можуть виникнути в C++ проектах. Також C# є мовою сценаріїв для Unity – найпопулярнішого ігрового рушія для VR/AR, віртуальної та доповненої реальності, та кросплатформних 3D-застосунків, що дає доступ до готових інструментів візуалізації та фізики «з коробки», та легко взаємодіє з сучасними веб технологіями та базами даних, що може бути корисним для мережевих симуляцій.

Python – популярна багатоцільова та універсальна мова програмування, яку можна використовувати для розробки практично будь-якої комп’ютерної програми, включаючи програми, розширення та веб-сайти. Особливої уваги він та широкого поширення він набув у сфері наукових обчислень, аналізу даних та машинного навчання. Python відомий своїм простим синтаксисом та плавністю вивчення, і тому багато розробників обирають його як свою першу мову програмування, а деякі й

надалі продовжують створювати на ній застосунки, розширення і тому подібне. Також Python – це мова з динамічною типізацією, тобто розробникам не потрібно оголошувати типи даних змінних, які вони визначають, тип змінної визначається під час виконання програми, а не на початку. При чому, Python підтримує модулі та пакети модулів, що сприяє модульності та повторному використанню коду. Важливою особливістю Python є наявність великої кількості спеціалізованих бібліотек для математичних розрахунків, чисельного моделювання та роботи зі штучним інтелектом.

Хоча для систем реального часу традиційно розглядають мову C++ та C#, вибір мови Python для даного проєкту зумовлений наступними факторами:

– сучасна фізика часто перетинається зі штучним інтелектом, а Python дозволяє легко впроваджувати нейромережі в симуляції для складних процесів;

– Python дозволяє зосередитися на логіці фізичних процесів та архітектурі системи, не витрачаючи надмірного часу на керування пам’яттю чи типах даних;

– завдяки різним бібліотекам, як NumPy та SciPy, більшість векторних та матричних операцій виконуються на рівні оптимізованого C-коду, що дозволяє досягти продуктивності, наближеної до низькорівневих мов, при збереженні «чистого» Python-коду;

– широке використання у науковому середовищі, що забезпечують доступ до великої кількості готових алгоритмів, прикладів реалізації та технічної документації;

– Python має розвинені обгортки для різноманітних графічних API та фізичних рушіїв, як PyOpenGL, ModernGL чи PyBullet.

Таким чином, вибір Python обумовлений не максимальною продуктивністю виконання окремих операцій, а поєднанням швидкості розробки, доступності наукових бібліотек та можливістю інтеграції з високопродуктивними обчислювальними модулями. Це дозволяє ефективно реалізовувати складні фізичні моделі та зосереджувати основну увагу на дослідженні процесів, а не на технічну реалізацію.

2.2 Вибір графічного API

Оскільки в минулому підрозділі 2.1 було обрано мову програмування Python, то під неї потрібно обрати графічне API, що зможе інтегрувати з даною мовою. А оскільки для Python існує сотні різних бібліотек і модулів, що облегшують роботу з процесором комп’ютера, з вебсайтами та з масивами даних, то обрати API буде не легко.

Вибір OpenGL для візуалізації фізичних процесів у 3D є класичним і обґрунтованим рішенням. Незважаючи на його 30-річну «старість» та появу новіших API, таких як Vulkan або DirectX, OpenGL залишається стандартом для багатьох розробників. Але його ж старий вік також є його перевагою, оскільки будь-яка знайдена помилка вже має рішення на просторах Інтернету. Також OpenGL працює практично на всіх операційних системах, як Windows, Linux чи macOS, на якому він хоч і вважається застарілим, але все ще підтримується.

Vulkan є альтернативою OpenGL, розроблений для мінімізації накладних витрат драйвера, має максимальний контроль над GPU, високу багатопоточність та низьке споживання CPU. Але при цьому він має екстремальну складність, бо для виведення одного трикутника на екран потрібно написати сотні рядків коду. Загалом його обирають для створення комерційного рушія або для симуляції з величезною кількістю об’єктів.

WebGPU є сучасним стандартом, який прийшов на заміну WebGL, графічного API для браузера та вебсторінок, але може працювати і як звичайний API, маючи сучасний простий дизайн та безпеку пам’яті. Він знижує навантаження на CPU і дозволяє ефективніше використовувати багатоядерні системи та виконувати не лише рендеринг, а й складні паралельні обчислення. Але оскільки WebGPU почав набирати популярність не так давно, то навчальних матеріалів по ньому не так багато.

DirectX рідко працює напряму з мовою Python, зазвичай через специфічні обгортки, також в нього немає кросплатформеності, тобто він працює тільки на системі Windows. Але при цьому ж він має і найкращу продуктивність та підтримку

функцій, а також дозволяє ефективно використовувати мультимедійні можливості комп’ютеру.

В таблиці 2.1 приведено коротке порівняння вищеописаних графічних API.

Таблиця 2.1. Порівняння графічних API

Характеристика	**Графічні API**			
	OpenGL	**Vulkan**	**WebGPU**	**DirectX**
Бібліотека на Python	PyOpenGL, ModernGL	vulkan	wgpu-py	pydx12
Рівень абстракції[1]	Середній	Дуже низький	Середній	Середній
Складність коду	Низька	Дуже висока	Середній	Середній
Крос-платформеність	Всі операційні системи	Windows, Linux, Android	Windows, Linux, macOS	Windows, Xbox
Продуктивність	Середня	Висока	Середня	Висока

Проаналізувавши наведені вище API, було вибрано OpenGL, оскільки він дозволяє мати прямий контроль над графічним конвеєром, дозволяючи зосередитися на фізичній моделі, а не на драйверах відеокарти.

2.3 Схема взаємодії графічного ядра та фізичного рушія

Архітектура сучасних систем імітації в реальному часі базується на принципі роз’єднання обчислювальної логіки та візуального представлення, що дозволяє системі бути гнучкою, бо фізичні розрахунки виконуються з однією частотою, а візуалізація – з іншою, адаптуючись до потужності обладнання. А взаємодія між фізичним рушієм та графічним ядром відбувається через посередника, через спеціальний шар синхронізації або, як в відеоіграх, через ігровий цикл.

[1] Рівень абстракції – спосіб управління складністю системи з приховуванням деталей реалізації та виділенням лише істотних характеристик об’єкта або процесу

Завданням фізичного рушія є математичне моделювання світу, де вхідні дані, як вектори сил, маси об'єктів, колізійні форми, параметри тертя та пружності, використовують для розв'язання систем диференційних рівнянь руху, описаних в підрозділі 1.2, щоб отримати оновлену матрицю трансформації, позицію та орієнтацію у просторі, для кожного активного тіла. Рушій оперує спрощеними формами об'єктів і йому не важливо, як виглядає модель, головне – її маса, форма та швидкість.

Графічне ядро відповідає за вигляд об'єкта для користувача, використовуючи такі вхідні дані, як 3D-меші, текстури, шейдери та дані трансформації від фізичного рушія, для обробки світла, тіней, ефектів заломлення та вивід фінального кадру на екран. Ядро оперує візуальними сітками, беручи «фізичну» позицію і накладаючи на неї зовнішній вигляд.

Основні етапи взаємодії можна виразити в наступних кроках (рис. 2.1):

1) ініціалізація, де фізичний рушій, який зазвичай працює на процесорі або GPU, обчислює початкові позиції, швидкість та масу об'єктів;

2) симуляція з розрахунком зіткнень між об'єктами, обчисленням гравітації, тертя та сил та визначенням нових координат та трансформацій для кожного об'єкта;

3) передача даних, де фізичний рушій передає оновлені координати, позиції та матриці трансформації для всіх фізичних тіл у графічний рушій;

4) візуалізація з отриманням даних від фізичного рушія та застосуванням трансформацій до 3D-моделей, рендеринг сцени, враховуючи освітлення, текстури та тіні;

5) відображення, де кадр виводиться на екран;

6) повторення циклу, з поверненням до кроку 2 для наступного кадру.

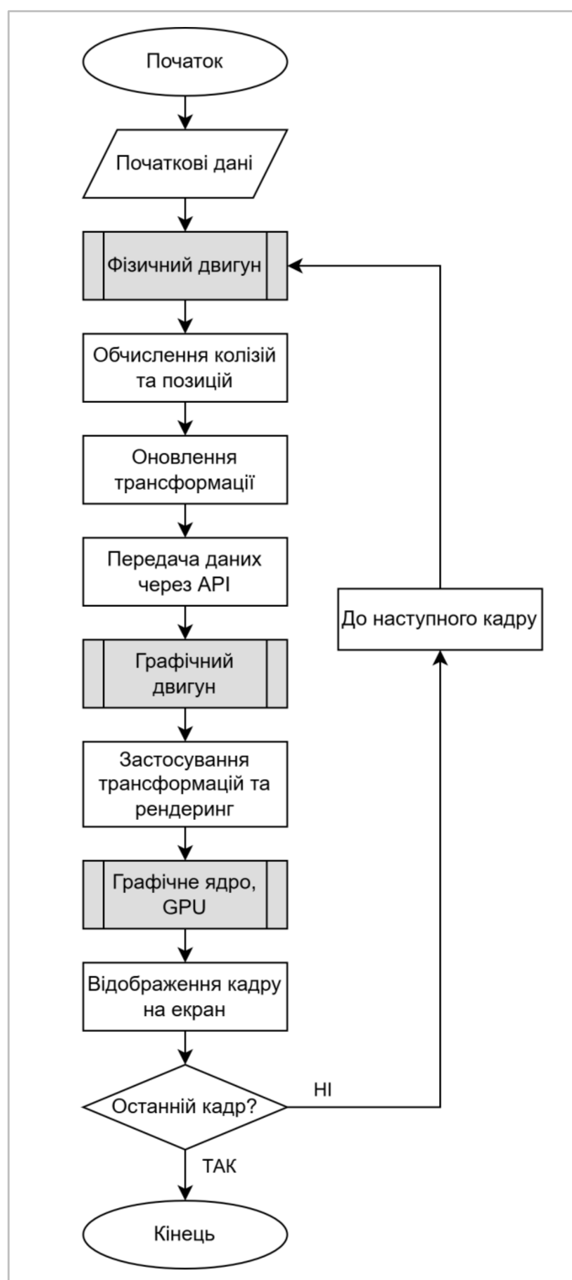


Рисунок 2.1 – Етапи взаємодії графічного ядра та фізичного рушія

Загалом, дуже важливо зберігати зв'язок графічного ядра та фізичного рушія, оскільки, якщо ці компоненти не взаємодіють правильно, будуть виникати помилки, такі як «провалювання» об'єктів крізь стіни або неправильне відображення тіней, тому вони завжди повинні бути синхронізовані в кожному кадрі.

2.4 Створення 3D-геометрії

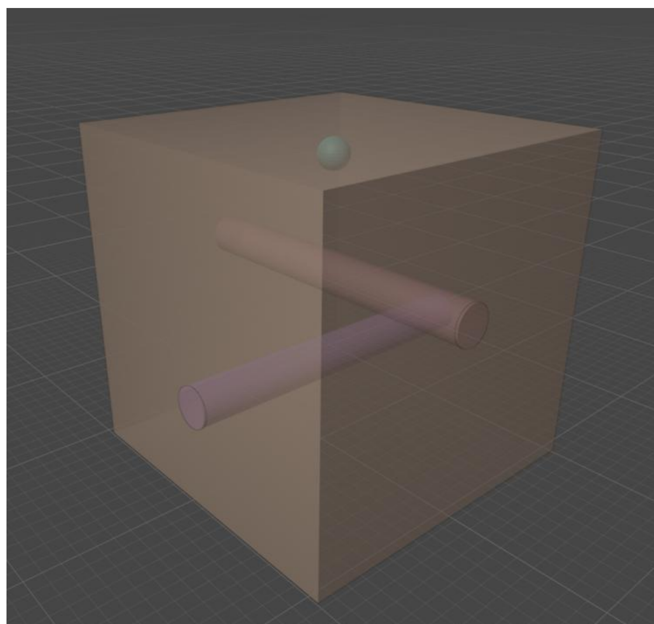
У системах імітації фізичних процесів у реальному часі 3D-геометрія виконує подвійну роль: вона є основою для візуалізації та базою для математичного

розрахунку фізичних взаємодій. Процес моделювання для таких систем має свої особливості, оскільки надмірна складність сітки може призвести до критичного падіння продуктивності, а недостатня – до некоректної поведінки фізичного рушія, як наприклад, «провалювання» об’єктів один через одного. На рис. 2.2 зображені приклади створених нескладних моделей для трьох фізичних процесів, які будуть розглядатися в даній роботі:

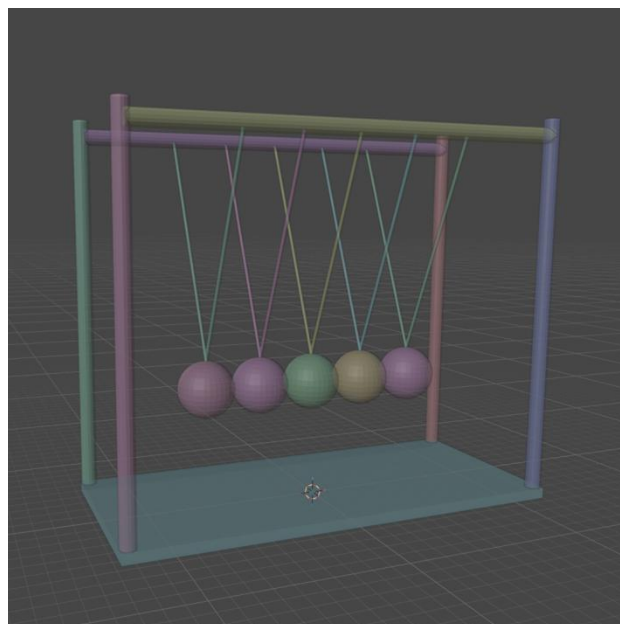
– для імітації рідини та сипких тіл, яка представлена в вигляді куба для обмеження рідини, двох циліндрів для перешкоджання потоку та сфери, з місця якої буде йде потік;

– для імітації динаміки жорстких тіл та передачі імпульсу, демонстрація збереження імпульсу та енергії на прикладі маятника Ньютона. Сам процес моделювання включає створення окремих мешів для кульок, рами та ниток, які можуть бути як жорсткими з’єднаннями, так і гнучкими обмеженнями;

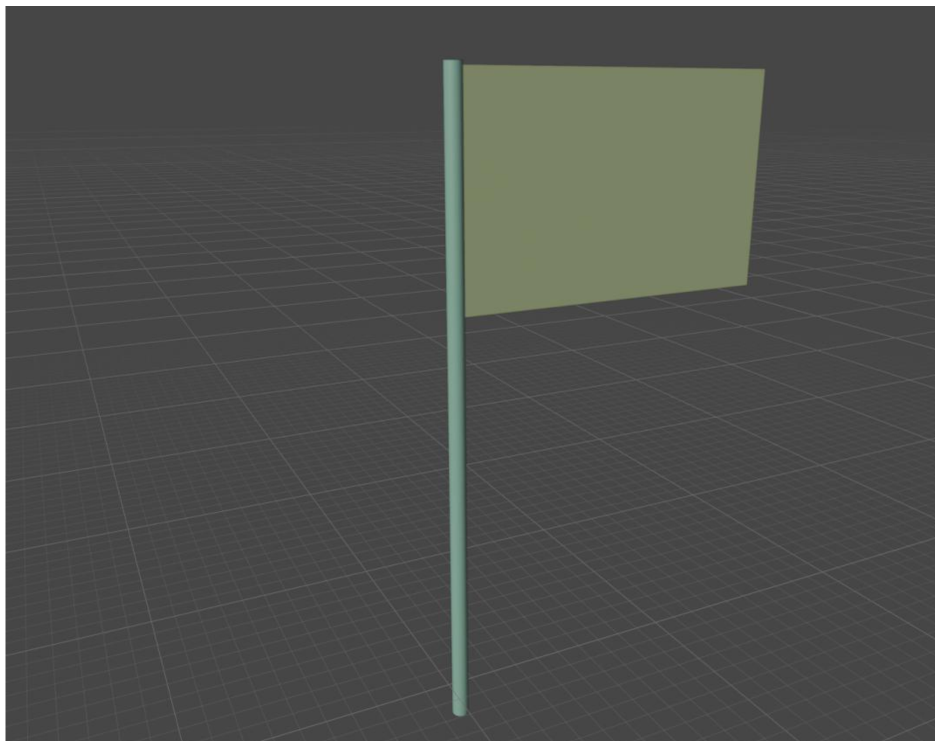
– для імітації тканини, що реагує на зовнішні сили, як гравітація або вітер, в вигляді прапора. На відміну від жорстких тіл, тут ключовим є не лише зовнішній вигляд, а й внутрішня структура сітки.



а)



б)



в)

Рисунок 2.2 – Створені моделі для імітації: а) – модель для імітації рідини та сипких тіл; б) – модель для передачі імпульсу; в) – модель для імітації тканини

Для забезпечення стабільності фізичних розрахунків у реальному часі до топології сіток потрібно висунути наступні вимоги:

– чистота топології уникненням n-гонів, багатокутників з більше, ніж чотирма вершинами, та «вироджених» трикутників, які можуть викликати помилки в алгоритмах виявлення зіткнень;

– моделі створені з мінімально необхідною кількістю полігонів, що зберігає силует, наприклад для рами маятника Ньютона на рис. 2.2.б використані спрощені циліндри з низькою кількістю граней;

– для моделі прапора на рис. 2.2.в сітка створена у вигляді рівномірної квадратної решітки. Це критично важливо, оскільки фізичний рушій розраховує сили натягу та вигину між сусідніми вершинами. Нерівномірна сітка призвела б до неприродних розривів або «смикання» тканини.

Також для підвищення продуктивності потрібно підготовити LOD-и. Сама технологія LOD передбачає створення кількох версій однієї моделі з різним

ступенем деталізації, які автоматично змінюються залежно від відстані об'єкта до віртуальної камери:

– LOD 0 – максимальна деталізація, що відображається поблизу (як повна геометрія кульок та кріплень маятника);

– LOD 1 – спрощена геометрія без дрібних фасок та другорядних деталей;

– Collision Mesh, фізична модель –окремий етап удосконалення моделі. Для розрахунку фізики часто створюється прихована від очей, максимально спрощена сітка. Наприклад, візуально кулька в колисці Ньютона може бути високополігональною для ідеальної гладкості, але для фізичного рушія вона представлена як ідеальна математична сфера або низькополігональний проксі-об'єкт.

Такий підхід дозволяє перенести основне обчислювальне навантаження з графічного конвеєра на фізичні розрахунки, що забезпечує FPS навіть при інтенсивній взаємодії об'єктів.

2.3 Принципи та методи виявлення зіткнень об'єктів

Виявлення зіткнень об'єктів є однією з найбільш ресурсомістких частин імітації фізичних процесів. Для забезпечення роботи в реальному часі алгоритми будуються за ієрархічним принципом, що дозволяє відсікати завідомо неможливі контакти ще до початку складних розрахунків. Загалом, процес виявлення зіткнень розбито на два етапи: широка фаза та вузька фаза виявлення зіткнень.

Широка фаза визначає, які геометричні об'єкти потрібно перевірити для певного запиту, швидко виключаючи віддалені об'єкти на основі обмежувальних полів або інших розділів простору. Алгоритм в цій фазі аналізує всю сцену та визначає пари об'єктів, які можуть потенційно зіткнутися, і замість перевірки кожної вершини однієї моделі з полігонами іншої використовуються максимально спрощені обмежуючі об'єми.

Вузька фаза використовується для об'єктів, які перекриваються в широкій фазі, і виконується перевірка між запитом і одним примітивним об'єктом, який представляє перешкоду. Тобто, якщо на широкій фазі було зафіксовано перекриття

обмежуючих об’ємів, система переходить до точного розрахунку. Тут обчислюються точки контакту, нормалі зіткнення та глибина проникнення, тому об’єкти часто відповідають твердим тілам.

Для полегшення та прискорення розрахунків геометрично складні моделі вписуються у прості фігури, тобто використовують примітиви зіткнень, або обмежуючі об’єми, а сам вибір об’єму залежить від необхідної точності та швидкості. Існує кілька десятків таких об’ємів, але будуть розглянуті в основному ті, які найбільше підходять для моделей з рис. 2.2:

– сфера, що є найпростішим тривимірним об’ємом, і також є найефективнішим видом примітивів зіткнень, представлена центральною точкою та радіусом. Цю інформацію можна зручно упакувати у чотирьохелементний формат вектора з плаваючою комою, який особливо добре працює з математичними бібліотеками SIMD, однією з технологій паралельних обчислень;

– капсула – це таблетоподібний об’єм, що складається з циліндра та двох напівсферичних торцевих кришок. Його можна розглядати як розгорнуту сферу, яка простежується, коли сама сфера рухається від однієї точки до іншої, хоча є деякі важливі відмінності між статичною капсулою та рухомою сферою, тому вони не ідентичні. Капсули ефективніше перетинаються, ніж циліндри чи коробки, тому їх часто використовують для моделювання об’єктів, які є приблизно циліндричними, наприклад кінцівок людського тіла;

– вирівняна по осі обмежувальна коробка AABB – прямокутний об’єм, грані якого паралельні осям системи координат. Але коробка, яка вирівняна по осі в одній системі координат, не обов’язково буде вирівняна по осі в іншій системі. Тож потрібно говорити про AABB лише у контексті конкретної координатної рамки чи кадрів, з якою вона узгоджується. Основна перевага AABB полягає в тому, що можна перевірити взаємопроникнення з іншими коробками, вирівняними по осі, високоефективним способом. Але найбільше обмеження використання AABB полягає в тому, що вони повинні залишатися вирівняними по осі завжди, якщо потрібно зберегти їхні обчислювальні переваги, бо AABB доведеться перераховувати щоразу, коли цей об’єкт обертається. Навіть якщо об’єкт схожий на

паралелепіпед, у випадку його обертання поза віссю AABB не описує належним чином його форму;

– орієнтована обмежувальна коробка OBB використовується коли вирівняна по осі коробка буде обертатися відносно її системи координат. Часто представляється трьома напіврозмірами, а саме півшириною, півглибиною та піввисотою, і перетворенням, яке позиціонує центр коробки та визначає її орієнтацію відносно осей координат. OBB є загальновживаним зіткненням примітивів, тому що вони краще справляються зі встановленням довільно орієнтованих об’єктів, хоча їх представлення доволі просте;

– дискретно орієнтований багатогранник DOP є більш загальним випадком AABB і OBB. Це опуклий багатогранник, який наближається до форми об’єкта. DOP можна побудувати, взявши кілька площин на нескінченності та ковзаючи їх уздовж нормальних векторів, доки вони не зіткнуться з об’єктом, форму якого потрібно наблизити.

Також для імітації фізики в реальному часі часто використовують комбінований підхід, наприклад, спочатку йде перевірка сферою, потім AABB, і лише в разі успіху – складні алгоритми вузької фази.

Загальне порівняння обмежуючих об’ємів наведено у таблиці 2.2.

Таблиця 2.2. Порівняння обмежуючих об’ємів та методів представлення геометрії

Тип об’єму / Форма	**Швидкість перевірки**	**Точність прилягання**	**Особливості та використання**
Сфери (Spheres)	Дуже висока	Низька	Найпростіший об’єм; інваріантний до обертання; ідеальний для швидкої грубої фільтрації
Капсули (Capsules)	Середня	Вища (для довгих об’єктів)	Дозволяє виконувати неперервні перевірки вздовж траєкторії

Кінець таблиці 2.2.

AABB (Axis-Aligned Bounding Boxes)	Висока	Середня	Паралелепіпед, вирівняний за осями координат; потребує лише порівняння координат; перераховується при обертанні об’єкта
OBB (Oriented Bounding Boxes)	Нижня	Висока	Орієнтований паралелепіпед; значно краще облягає об’єкт, але тест на перетин складніший математично
DOP (Discrete Oriented Polytopes)	Середня	Дуже висока	Опуклий багатогранник, грані якого паралельні фіксованому набору площин; точніший за AABB/OBB

Для створених моделей будуть використовуватися наступні об’єми:

– для моделі імітації рідини найважливішим є взаємодія частинок зі стінками куба та внутрішніми циліндрами, тому тут чудово підходить об’єм AABB для куба-контейнера та капсула або AABB для циліндрів;

– для моделі маятника важливо фіксувати моменти зіткнення кульок в одній точці, а оскільки вони самі є сферами, то сферичний об’єм дасть 100% точність при мінімальних витратах ресурсів, хоча AABB матиме подібний результат. Також до рами та основи маятника можна за необов’язковості додати об’єм AABB;

– для імітації тканини статичний об’єм не підійде, оскільки прапор постійно змінює свою форму, тому потрібно використати ієрархію обмежуючих об’емі, дерево AABB, – весь прапор вписується в один великий бокс, який динамічно перераховується, а всередині він ділиться на менші бокси, що охоплюють окремі ділянки тканини. Для флагштока підійде обмежуюча капсула або AABB, що дозволить розраховувати тертя та зіткнення тканини зі стовпом, уникаючи «застрягання» мешу в гострих кутах боксів.

Висновки до розділу 2

У цьому розділі було проведено проєктування архітектури програмної системи, розроблено тривимірні моделі для подальшої фізичної імітації та проаналізовано мови програмування та графічного API для обґрунтування вибору інструментів для реалізації імітацій процесів.

Серед обмежуючих об’ємів будуть використовуватися AABB та сфери, що дозволить ефективно детектувати контакти між об’єктами. Також використання алгоритмів виявлення зіткнень дозволяє системі стабільно працювати.

Серед мов програмування було розглянуто C++, C# та Python, причому серед них був обраний Python через його читабельність, велику спільноту та широкий вибір бібліотек. А аналізуючи графічні API, які можуть працювати з мовою Python, було обрано OpenGL за високу швидкість розробки, простоту коду та прямий контроль над фізичною моделлю.

Таким чином, для реалізації програмного забезпечення було обрано стек технологій: мова програмування Python, графічний API OpenGL та обмежуючі об’єми AABB та сфери.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

3.1 Реалізація фізичного ядра та візуалізатора

Оскільки вибір програмних засобів для реалізації системи зумовлений потребою в ефективній обробці фізичних процесів у реальному часі, то використання мови Python доповнюється наступними бібліотеками:

– ModernGL – бібліотека для роботи з OpenGL та пряма альтернатива іншої бібліотеки, PyOpenGL. Їхня відмінність доволі значна: PyOpenGL забезпечує прямий доступ до функцій самого API OpenGL, але може працювати повільніше через часті виклики функцій Python, при цьому ModernGL оптимізований під сучасний OpenGL та для мови Python, працює швидше та знижує ризик витоку пам’яті;

– Pyglet – бібліотека для створення вікон програм та обробки вводу користувача, при цьому має пряму підтримку використання OpenGL та простіша для роботи з візуальним контентом;

– NumPy – доволі важлива бібліотека, яка потрібна для різноманітних математичних обчислень над будь-якими масивами та векторами, в особливості над масивами координат, і її відсутність може буди критичною для продуктивності. Для цієї бібліотеки також існує додаткова бібліотека pyrr для полегшення виконання більшості математичних функцій, які потрібні для роботи з 3D.

У межах програмної реалізації було розроблено модульну структуру, що дозволяє розділити фізичне моделювання, візуалізацію та керування виконанням програми на незалежні компоненти. У результаті спрощується супровід коду, підвищується його масштабованість та зменшується залежність між окремими підсистемами. До модульної структури входить три модулі, кожен з яких має свою задачу та взаємодію:

– ***модуль physics_engine.py*** є фізичним рушієм і відповідає за оновлення стану об’єктів через фіксовані проміжки часу та перевірку зіткнень і колізій. Для розрахунку нових координат використовується метод Адамса-Башфорта для моделі

з імітацією рідини та метод Верле для інших двох моделей. Також модуль реалізовує двофазну перевірку зіткнень, де спочатку виконується перевірка через обмежуючі об'єми, і при виявленні перекриття розраховує точки контакту та вектора відштовхування;

– ***модуль render_core.py*** є візуалізатором і відповідає за передачу даних з оперативної пам'яті до відеокарти. Кожна модель завантажується у вигляді вершинних буферів, VBO, та індексних буферів, EBO, що мінімізує кількість передач даних між CPU та GPU під час кожного кадру;

– ***модуль world_manager.py*** – це сполучний модуль, що координує роботу рушія та візуалізатора. Він реалізовує фізичний цикл, де сама фізика оновлюється зі сталою частотою в 60 Гц, а рендеринг виконується зі швидкістю, наскільки дозволяє GPU. Також після кожного кроку фізики модуль оновлює матриці трансформації для графічних моделей, що дозволяє візуалізувати рух, обертання та деформації в реальному часі.

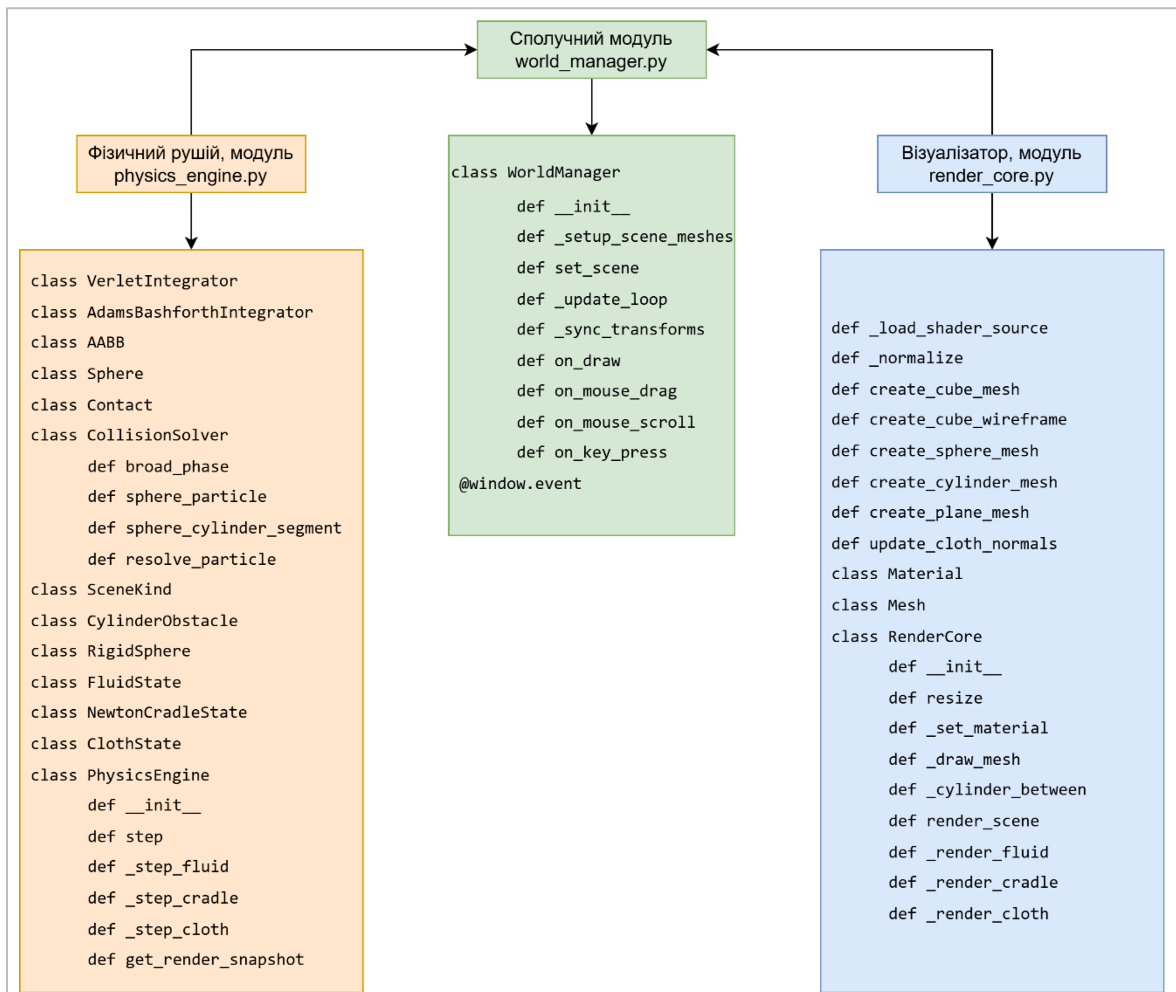


Рисунок 3.1 – Загальна структура та зв'язок модулів

Але оскільки кожен з модуль має достатньо великий об'єм, буде доречним розписати кожен з них більш детально. Повні лістингу коду до трьох модулів знаходяться в Додатках А, Б і В.

3.1.1 Фізичний рушій, модуль physics_engine.py

Даний модуль physics_engine.py розрахований на розрахунок колізій, математичних формул, алгоритмів та самої фізики, і загалом складається з тринадцяти звичайних класів та класів даних, dataclass. Також на початку модуля визначена глобальна змінна *Vec3 = numpy.ndarray*, яка буде використовувати як вектор з трьох змінних.

Спочатку йдуть класи для обчислення методів чисельного інтегрування Верле та Адамса-Башфорта другого порядку відповідно до формул (1.5), (1.7) та (1.8). В обох класах по одному статичниму методу step, що обчислюють нові позиції та швидкості об'єктів на основі прискорення та часу. При цьому клас методу Верле, оскільки він використовується для обчислення позиції тканини, також відстежує нерухомі вершини, що мають бути закріплені до флагштоку, а клас методу Адамса-Башфорта першу поточну позицію обчислює за формулою (1.8), а наступні поточні позиції за формулою (1.7).

Лістинг коду інтеграторів.

```
    class VerletIntegrator:
        @staticmethod
        def step(positions: np.ndarray, prev_positions: np.ndarray,
accelerations: np.ndarray, dt: float, mask: Optional[np.ndarray] = None,
        ) -> Tuple[np.ndarray, np.ndarray]:
            new_pos = 2*positions - prev_positions +
accelerations*(dt*dt)
            if mask is not None:
                new_pos[mask] = positions[mask]
            return new_pos, positions.copy()
```

```
    class AdamsBashforthIntegrator:
        @staticmethod
        def step(positions: np.ndarray, velocities: np.ndarray,
accelerations: np.ndarray, prev_accelerations: Optional[np.ndarray],
                 dt: float) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
            if prev_accelerations is None:
                new_vel = velocities + accelerations * dt
            else:
                new_vel = velocities + 0.5 * dt * (3.0 * accelerations -
prev_accelerations)
            new_pos = positions + new_vel * dt
            return new_pos, new_vel, accelerations.copy()
```

Далі йде класи даних AABB та Sphere для обчислення зіткнень моделей. Клас AABB має дві змінні для встановлення двох точок для побудови куба та три функції: функція overlaps, для перевірки перетину з моделями, і статичні методи from_center_extents, для створення AABB на основі центру об'єкта та його відстані від центру до граней, та from_points, що обчислює межі AABB навколо хмари точок і знаходить мінімальні та максимальні координати серед усіх переданих точок, створюючи обмежувальну рамку. Клас Sphere має подібну структуру – дві змінні для центру кулі та її радіуса, функція overlaps і статичні методи from_center_extents та from_points.

Лістинг коду класу даних AABB.

```
    @dataclass
    class AABB:
        min_corner: Vec3
        max_corner: Vec3

        def overlaps(self, other: "AABB") -> bool:
            return bool(np.all(self.max_corner >= other.min_corner) and
np.all(self.min_corner <= other.max_corner))

        @staticmethod
```

```
        def from_center_extents(center: Vec3, half_extents: float | Vec3)
-> "AABB":
            he = np.broadcast_to(half_extents, (3,))
            return AABB(center - he, center + he)

        @staticmethod
        def from_points(points: np.ndarray) -> "AABB":
            return AABB(points.min(axis=0), points.max(axis=0))
```

Далі йде клас даних Contact, що зберігає дані про зіткнення: точку контакту point, напрямок відштовхування normal та глибину проникнення penetration.

Наступний клас CollisionSolver містить статичні методи для обчислення точного контакту між різними об'єктами, але здебільшого потрібен для моделі імітації рідини:

– метод broad_phase перевіряє перетин обмежувальних об'ємів, як AABB;

– sphere_particle розраховує зіткнення двох сфер на основі відстані між їхніми центрами, але загалом потрібний для відскакування частинок від сфери-емітера, джерела, після появи;

– sphere_cylinder_segment розраховує зіткнення частинок рідини та циліндрів, при цьому метод шукає найближчу точку на циліндрі до центру сферичної частинки за допомогою проекції, а потім перевіряє, чи не проникає ця частинка в даний циліндр;

– логіка методу resolve_particle потрібна для відштовхування частинок після зіткнення з об'єктами.

Клас SceneKind є переліком для визначення режиму роботи фізичного ядра в залежності від вибраної симуляцій. Класи даних CylinderObstacle та RigidSphere відповідно описують перешкоду в вигляді циліндру, з координатами початкової та кінцевої точок центральної осі циліндру та його радіус, та кулю для маятника з поточною і попередньою позицією, радіусом, масою, точкою та довжиною підвісу та кольором для візуалізації.

Лістинг коду класів SceneKind, CylinderObstacle та RigidSphere.

```
    class SceneKind(Enum):
```

```
        FLUID = auto()
        NEWTON_CRADLE = auto()
        CLOTH = auto()

    @dataclass
    class CylinderObstacle:
        p0: Vec3
        p1: Vec3
        radius: float

    @dataclass
    class RigidSphere:
        position: Vec3
        prev_position: Vec3
        radius: float
        mass: float
        pivot: Vec3
        rod_length: float
        color_index: int = 0
```

Далі йдуть три класи даних, що зберігають параметри та поточний стан конкретної симуляції. Клас FluidState відповідальний за симуляцію рідини та сипких речовин, зберігає позиції, швидкості, маси всіх частинок рідини та параметри емітера і циліндричних перешкод, та має один класовий метод create_default, відповідальний за створення потрібних об’єктів для симуляції. Подібний до нього клас NewtonCradleState зберігає кількість сфер для маятника та коефіцієнт відновлення frame_restitution для пружного зіткнення, і також має класовий метод create_default для створення потрібних об’єктів, причому одна з крайніх сфер має початковий нахил для початку коливань.

Останній клас даних ClothState організовує сітку точок тканини у просторі та визначає їхні зв’язки для подальшої симуляції, зберігає параметри сітки, маску фіксованих точок pin_mask, параметри жорсткості та вітру. Його класовий метод create_flag також є конструктором, який генерує початкову сітку для прапора і

розраховує координати кожної частинки відносно origin і автоматично призначає фіксацію для першого стовпця частинок, щоб прапор був закріплений з одного боку. Також в класі є допоміжна функція index, яка перетворює двовимірні індекси сітки у лінійний індекс для масиву positions, що спрощує навігацію між сусідами під час розрахунку фізичних зв'язків.

Лістинг коду класів ClothState.

```
    @dataclass
    class ClothState:
        width: int
        height: int
        spacing: float
        positions: Vec3
        prev_positions: Vec3
        pin_mask: Vec3
        structural_k: float = 200.0
        damping: float = 0.98
        pole_anchor: Vec3 = np.array([0,0,0])

        @classmethod
        def create_flag(cls, cols: int = 14, rows: int = 8) -
>"ClothState":
            spacing = 0.2
            origin = np.array([0.0, 3.5, 0.0])
            positions = []
            pin = []
            for j in range(rows):
                for i in range(cols):
                    positions.append(origin + np.array([i * spacing, -j *
spacing, 0.0]))
                    pin.append(i == 0)
            pos = np.array(positions, dtype=np.float64)
            return cls(width=cols, height=rows,
                spacing=spacing, positions=pos,
                prev_positions=pos.copy(),
```

```
                pin_mask=np.array(pin, dtype=bool),
                pole_anchor=origin
            )

        def index(self, i: int, j: int) -> int:
            return j * self.width + i
```

Останній і головний клас фізичного ядра і логікою симуляції є PhysicsEngine, який інтегрує інші класи та методи модуля і в якому оновлюється симуляції з фіксованим часовим кроком для кожної з трьох моделей. В ініціалізаторі цього класу встановлюються базові фізичні параметри, такі як гравітація та вітер, ініціалізується допоміжні класи, зокрема CollisionSolver для обробки зіткнень, та створюються об’єкти для кожної зі сценарних моделей: рідини FluidState, маятника NewtonCradleState та тканини ClothState.

Лістинг коду ініціалізатора класа PhysicsEngine.

```
    def __init__(self, dt: float = 1.0 / 60.0, gravity: Optional[Vec3] = None):
        self.dt = dt
        self.gravity = gravity if gravity is not None else np.array([0.0, -9.81, 0.0])
        self.wind = np.array([4.0, 0.0, 2.5])
        self.scene = SceneKind.FLUID
        self.fluid = FluidState.create_default()
        self.cradle = NewtonCradleState.create_default()
        self.cloth = ClothState.create_flag()
        self.collision = CollisionSolver()
        self._fluid_spawn_counter = 0
        self._cloth_simulation_time = 0.0
```

Далі йде невеликий метод set_scene, що дозволяє перемикатися між трьома режимами: рідиною FLUID, маятником NEWTON_CRADLE або тканиною CLOTH. А наступний метод step є основним циклом симуляції, що викликає специфічні методи для кожної сцени, забезпечуючи оновлення позицій та швидкостей.

Одним зі специфічних методів є _step_fluid для моделі імітації рідини. Він використовує інтеграцію Адамса–Башфорта для обчислення руху частинок, впроваджує логіку емітера, що періодично створює нові частинки. Допоміжний метод _fluid_collisions для функції _step_fluid проводить перевірку зіткнень зі стінами куба AABB, циліндрами-перешкодами та самим емітером.

Для маятника специфічним методом є _step_cradle, що реалізовує інтеграцію Верле, та застосовує обмеження довжини підвісу, щоб кулі маятника завжди рухалися по дузі. Також він обробляє зіткнення між сферами з передачею імпульсу для створення поведінки маятника Ньютона та корегує позиції та швидкості кульок.

Для тканини використовується кілька методів, але основним серед них є _step_cloth, що також, як і маятник, використовує метод Верле для руху вершин прапора, враховуючи вплив вітру, що змінюється залежно від часу та положення точок тканини. Він викликає методи _cloth_collisions, що розраховує колізію тканини з підлогою та основою, та _satisfy_cloth_constraints, який ітеративно виправляє дистанції та структурні зв’язки між вузлами тканини, зберігаючи її форму, при чому, в свою чергу, він для цього викликає статичний метод _apply_distance_constraint.

Остання функція класу get_render_snapshot повертає словник із поточними даними сцени, які графічний модуль render_core.py може використати для візуалізації об’єктів, як позиції сфер, частинок або вершин прапора.

Лістинг коду функції get_render_snapshot.

```
    def get_render_snapshot(self) -> dict:
        if self.scene == SceneKind.FLUID:
            return {
                "kind": "fluid",
                "particles": self.fluid.positions.copy(),
                "particle_radius": self.fluid.particle_radius,
                "bounds": self.fluid.bounds,
                "obstacles": self.fluid.obstacles,
                "emitter": self.fluid.emitter_center,
                "emitter_radius": self.fluid.emitter_radius,
```

```
            }
        if self.scene == SceneKind.NEWTON_CRADLE:
            return {
                "kind": "cradle",
                "spheres": [ {
                        "position": s.position.copy(),
                        "radius": s.radius,
                        "pivot": s.pivot.copy(),
                        "color_index": s.color_index,
                    } for s in self.cradle.spheres ],
            }
        return {
            "kind": "cloth",
            "vertices": self.cloth.positions.copy(),
            "width": self.cloth.width,
            "height": self.cloth.height,
            "pole_anchor": self.cloth.pole_anchor.copy(),
        }
```

3.1.2 Візуалізатор, модуль render_core.py

В даному модулі render.py створюється та візуалізовуються об’єкти і світло для симуляції. Модуль загалом складається з дев’яти глобальних функцій та трьох класів.

Спочатку йде функція _load_shader_source, яка зчитує вихідний код шейдерів GLSL із додаткових файлів phong.vert та phong.frag. Наступна функція _normalize є математичною і потрібна для нормалізації векторів, приведення їх довжини до 1.

Лістинг коду функцій _load_shader_source та _normalize.

```
def _load_shader_source(name: str) -> str:
    shader_dir = Path(__file__).resolve().parent / "shaders"
    path = shader_dir / name
    return path.read_text(encoding="utf-8")
```

```
    def _normalize(v: np.ndarray) -> np.ndarray:
        n = np.linalg.norm(v)
        return v / n if n > 1e-9 else v
```

Далі йдуть функції для створення різноманітних об’єктів і генерації геометрії. Ці функції створюють набори вершин та індексів для подальшої відправки на GPU.

Функції create_cube_mesh та create_cube_wireframe створюють куби, потрібні для симуляції рідини та основи маятника, причому куб, створений create_cube_wireframe, потрібний лише в симуляції рідини для відображення ребер невидимого кубу з create_cube_mesh. Причому куб з create_cube_mesh має 24 вершини, оскільки у кожної вершини куба три нормалі, а в вершинному буфері кожна вершина може мати лише одну нормаль, тому для гарного шестигранного куба та освітлення вершини доводиться дублювати.

Функція create_sphere_mesh створює сфери для використання в вигляді частинок в симуляції рідини і кульок в маятнику Ньютона. Далі йде create_cylinder_mesh для створення циліндрів, що використовується у всіх трьох імітаціях: як перешкода для рідини, як рамка та нитки для маятника та як основа флагштоку. Функція create_plane_mesh створює в усіх трьох імітаціях площину, що потрібна лише для створення підлоги. Функція update_cloth_normals потрібна для створення та оновлення сітки тканини, а також розраховує нормалі для коректного освітлення при деформації тканини.

Лістинг коду функції update_cloth_normals.

```
    def update_cloth_normals(vertices: np.ndarray, width: int, height:
int) -> np.ndarray:
        n_verts = width * height
        normals = np.zeros((n_verts, 3))
        for j in range(height - 1):
            for i in range(width - 1):
                a = j * width + i
                b = a + 1
                c = a + width
                d = c + 1
                for tri in ((a, c, b), (b, c, d)):
```

```
                    p0, p1, p2 = vertices[tri[0]], vertices[tri[1]],
vertices[tri[2]]
                    n = np.cross(p1 - p0, p2 - p0)
                    for k in tri:
                        normals[k] += n
         mesh = np.zeros((n_verts, 8), dtype="f4")
         for k in range(n_verts):
             nn = _normalize(normals[k])
             i = k % width
             j = k // width
             p = vertices[k]
             mesh[k] = [p[0], p[1], p[2], nn[0], nn[1], nn[2], i / width,
j / height]
         return mesh.flatten()
```

Єдиний клас даних Material зберігає фізичні властивості поверхні будь-якого об’єкта, як кольори амбітного, дифузного та дзеркального освітлення (що відповідають за фоновий і основний кольори об’єкта та відблиски світла на поверхні), коефіцієнт блиску та прозорість.

Клас Mesh керує даними геометрії в пам’яті відеокарти, завантажує вершини VBO та індекси EBO у буфери ModernGL, створює VAO, Vertex Array Object, та дозволяє оновлювати вершини «на льоту» методом update_vertices, що використовується, наприклад, для деформації тканини.

Лістинг коду класів Material та Mesh.

```
@dataclass
class Material:
    ambient: Tuple[float, float, float] = (0.15, 0.15, 0.15)
    diffuse: Tuple[float, float, float] = (0.7, 0.7, 0.7)
    specular: Tuple[float, float, float] = (0.4, 0.4, 0.4)
    shininess: float = 32.0
    alpha: float = 1.0


class Mesh:
```

```
        def __init__(self, ctx, vertices: np.ndarray, indices: np.ndarray):
            self.ctx = ctx
            self.index_count = len(indices)
            self.vbo = ctx.buffer(vertices.tobytes())
            self.ebo = ctx.buffer(indices.tobytes())
            self.vao = None

        def bind(self, program):
            self.vao = self.ctx.vertex_array(program, [(self.vbo, "3f 3f 2f", "in_position", "in_normal", "in_texcoord")], self.ebo )

        def update_vertices(self, vertices: np.ndarray):
            self.vbo.write(vertices.astype("f4").tobytes())
```

Головним класом візуалізатора є RenderCore. В його ініціалізаторі налаштовується контекст ModernGL, вмикається DEPTH_TEST, тест глибини, щоб об’єкти не малювалися крізь один одного, завантажує шейдери та створюються необхідні об’єкти для симуляцій.

Далі йдуть кілька невеликих допоміжних методів для візуалізації. Метод resize коригує розміри області малювання, коли змінюється розмір вікна програми, що є критичним для того, щоб зображення не виглядало розтягнутим. Властивості матеріалу, як колір чи блиск, у поточний активний шейдер передає _set_material. Метод _draw_mesh є основним методом для малювання одного об’єкта – він налаштовує колір та матеріал, записує матриці трансформації у пам’ять GPU та викликає команду render() для конкретного вершинного масиву. Також є математичний метод _cylinder_between, що обчислює матрицю трансформації, яка «розтягує» стандартний циліндр між двома довільними точками та повертає його в просторі, та використовується для візуалізації перешкоди для рідини та рами з нитками для маятника.

Головним методом є render_scene, що очищує екран, налаштовує камеру та підлогу сцени, а потім перевіряє, який тип симуляції прийшов до змінної snapshot, і передає керування відповідному спеціальному методу з трьох.

Лістинг коду методу render_scene.

```
    def render_scene(self, snapshot: dict, camera_eye: Tuple[float,
float, float] = (0, 3.2, 11)):
        self.ctx.clear(0.4, 0.4, 0.5)
        self.view_pos = Vector3(camera_eye)
        view = Matrix44.look_at(camera_eye, (0, 1.8, 0), (0, 1, 0))
        proj = Matrix44.perspective_projection(45.0, self.width /
self.height, 0.1, 100.0)
        floor_model = Matrix44.from_translation(Vector3([0, -0.01, 0]))
        self._draw_mesh(self.mesh_plane, floor_model, view, proj,
Material(ambient=(0.2, 0.3, 0.2), diffuse=(0.7, 0.7, 0.7), alpha=1.0))
        kind = snapshot.get("kind")
        if kind == "fluid":
            self._render_fluid(snapshot, view, proj)
        elif kind == "cradle":
            self._render_cradle(snapshot, view, proj)
        elif kind == "cloth":
            self._render_cloth(snapshot, view, proj)
```

Одним з таких спеціальних методів є _render_fluid, що відповідає за візуалізацію симуляції рідини: малює межі області у вигляді невидимого куба, перешкоди в вигляді циліндрів та сферичних частинок. Колиску Ньютона візуалізовує метод _render_cradle, який малює основу, вертикальні та горизонтальні балки для рами, нитки, де кожна нитка є тонким циліндром між точкою кріплення на рамі та кулею, та самі кулі. Метод _render_cloth є третім спеціальним методом та візуалізовує тканину. Він оновлює нормалі сітки, щоб тканина виглядала об’ємною при освітленні, оновлює дані у VBO через метод update_vertices класа Mesh та малює основу флагштоку.

3.1.3 Сполучний модуль world_manager.py

Модуль world_manager.py є найменшим з трьох модулів, оскільки він є лише об’єднанням двох інших модулів, причому самою сполучною ланкою є єдиний в модулі клас WorldManager.

В класі WorldManager першою і важливою функцією є ініціалізація змінних для симуляції, що створює екземпляри фізичного рушія та візуалізатора, налаштовує початкові, мінімальні та максимальні змінні для камери, чутливість комп'ютерної миші та запускає ігровий цикл через pyglet.clock.

Лістинг коду ініціалізатора класа WorldManager.

```
def __init__(self, window: Window, ctx):
    self.window = window
    self.ctx = ctx
    self.physics = PhysicsEngine(dt=PHYSICS_DT)
    self.renderer = Renderer(ctx, window.width, window.height)
    self.accumulator = 0.0
    self.snapshot = self.physics.get_render_snapshot()
    self.model_matrices: dict = {}

    self.camera_distance = 11.0
    self.camera_height = 3.2
    self.camera_angle = 0.0
    self.mouse_sensitivity = 0.005
    self.zoom_speed = 0.5
    self.min_camera_height = 0.5
    self.max_camera_height = 10.0
    self.min_camera_distance = 2.0
    self.max_camera_distance = 30.0
    self.current_scene = SceneKind.FLUID
    self._setup_scene_meshes()
    self.set_scene(self.current_scene)
    pyglet.clock.schedule(self._update_loop)
```

Функція _setup_scene_meshes є «скелетом» для зберігання трансформацій. Вона створює початковий словник матриць для об'єктів сцени контейнерів, сфер або тканини. Функція set_scene змінює поточний тип симуляції на рідину, маятник Ньютона або тканини та в залежності від цього оновлює заголовок вікна.

Головним циклом оновлення є _update_loop, який використовує змінну-накопичувач accumulator, що накопичує час, який пройшов між кадрами, і видає

його фізичному рушію невеликими порціями по часовому кроку, що дозволяє відокремити швидкість рендерингу від частоти оновлення фізики, яка завжди оновлюється з фіксованим кроком 1/60 секунди.

Лістинг коду циклу _update_loop.

```
    def _update_loop(self, dt: float):
        self.accumulator = min(self.accumulator + dt, PHYSICS_DT * MAX_PHYSICS_STEPS)
        steps = 0
        while self.accumulator >= PHYSICS_DT and steps < MAX_PHYSICS_STEPS:
            self.physics.step(PHYSICS_DT)
            self._sync_transforms()
            self.accumulator -= PHYSICS_DT
            steps += 1
        self.snapshot = self.physics.get_render_snapshot()
```

Функція _sync_transforms оновлює матриці моделей model_matrices на основі поточних даних від класу PhysicsEngine з модуля physics_engine.py. Це дозволяє візуалізатору знати, де саме в просторі зараз знаходяться об’єкти, яка на даний момент позиція сфери, який розмір контейнера тощо.

Функція on_draw обчислює позицію камери на основі кута обертання camera_angle та викликає метод renderer.render_scene для малювання сцени на екрані.

Наступні функції on_mouse_drag та on_mouse_scroll відповідають за обертання та приближення камери за допомогою комп’ютерної миші. Також обробка введення користувача здійснюється функцією on_key_press, а саме перемикання сцен імітації води, маятника Ньютона та флагштоку за допомогою клавіш 1, 2, 3 відповідно, перезапуск сцен симуляції з початку клавішою 0 або R та зупинка і відтворення симуляції на клавіші ПРОБІЛ або P.

Після класу WorldManager йде глобальна частина з ініціалізацією самої програми та константами. Як константи були створені PHYSICS_HZ, частота оновлення фізики, а саме 60 кадрів за секунду, PHYSICS_DT, часовий крок одного фізичного кроку PHYSICS_HZ, та MAX_PHYSICS_STEPS, обмежувач для циклу

оновлення, щоб запобігти «спіралі смерті», ситуації, коли фізика забирає весь час процесора при сильному гальмуванні. Саму ініціалізацію програми виконують змінні window, шо створює вікно pyglet розміром 1280x720, gl_ctx, яка створює контекст ModernGL для виконання OpenGL команд, та manager, що створює головний об’єкт керування з використанням двох інших зиінних. Декоратори подій @window.event перенаправляють події вікна: малювання, зміна розміру, натискання клавіш та керування комп’ютерною мишою – безпосередньо методам класу WorldManager. Останній рядок pyglet.app.run() запускає головний цикл обробки подій ОС.

Лістинг коду глобальної частини з ініціалізацією програми.

```
    PHYSICS_HZ = 60.0
    PHYSICS_DT = 1.0 / PHYSICS_HZ
    MAX_PHYSICS_STEPS = 8
    window = Window(width=1280, height=720, resizable=True, caption="Physics Simulation — Fluid [1] | Cradle [2] | Flag [3]")
    gl_ctx = moderngl.create_context()
    manager = WorldManager(window, gl_ctx)

    @window.event
    def on_draw():
        manager.on_draw()
    @window.event
    def on_resize(width, height):
        manager.renderer.resize(width, height)
    @window.event
    def on_key_press(symbol, modifiers):
        manager.on_key_press(symbol, modifiers)

    @window.event
    def on_mouse_drag(x, y, dx, dy, buttons, modifiers):
        manager.on_mouse_drag(x, y, dx, dy, buttons, modifiers)
    @window.event
    def on_mouse_scroll(x, y, scroll_x, scroll_y):
```

```
        manager.on_mouse_scroll(x, y, scroll_x, scroll_y)


    pyglet.app.run()
```

3.2 Підходи для підвищення продуктивності симуляцій

Для забезпечення стабільної симуляції фізичних процесів у реальному часі на мові Python, важливим завданням є покращення обчислень, мінімізація навантаження на CPU та ефективне використання ресурсів GPU.

Одним з подібних методів є просторова розбиття або декомпозиція, де замість перевірки зіткнень кожного об’єкта з кожним, коло пошуку обмежується лише сусідніми об’єктами. Якщо говорити більш точніше, то основна ідея декомпозиції полягає в тому, щоб значно зменшити кількість об’єктів, які потрібно перевірити на перетин, шляхом поділу простору на менші ділянки. Якщо можна при цьому визначити, що два об’єкти не займають однакову площу, то їм не потрібно проходити детальну перевірку перетину.

Також різні ієрархічні схеми, такі як октодерево, дерева BSP або сферичні дерева, можна використовувати для розбиття простору для покращення виявлення зіткнень. Ці структури по-різному ділять простір, але всі мають спільний ієрархічний підхід: у корені дерева відбувається розпад на великі ділянки, які з прогресуванням в ієрархії стають меншими, поки не буде досягнуто необхідного ступеня деталізації. Потім можна буде пройти по дереву, знайти групу об’єктів, що стикаються один з одним, і перевірити, чи вони дійсно перетинаються. Через деревоподібне розбиття можна дізнатися, які об’єкти з яких паралельних гілок не можуть стикатися один з одним.

Іншим важливим методом для мінімізації навантаження використовується багатопоточність: фізичні розрахунки можна ізолювати від потоку візуалізації, і поки графічна частина рендерить поточний кадр, фізичний рушій у паралельному процесі вже обчислює стан для наступного кадру. Також використання набору процесів дозволяє паралельно обробляти незалежні фізичні групи. Але оскільки в Python є обмеження GIL, механізму з гарантією того, що в один момент часу

виконується процес тільки одного потоку, то це ускладнює ефективну роботу для багатопроцесорних систем та робить симуляцію менш ефективною. Даний механізм зазвичай обходять через сторонні бібліотеки, як NumPy та Pandas, які самі знімають блокування під час виконання важких математичних розрахунків, або через використання модуля multiprocessing, який дозволяє створювати кілька незалежних процесів, кожен з яких має власний інтерпретатор і GIL.

Але, разом з усім вище сказаним, найбільший приріст продуктивності при моделюванні є використання обчислення на GPU, або також GPGPU, що використовується для великої кількості частинок або складних сіток. Тобто замість того, щоб передавати дані будь-якої моделі з GPU на CPU для обчислення фізики в кожному кадрі, відправляються лише параметри сили, а всі розрахунки зміщення вершин моделі відбуваються безпосередньо у відеопам’яті. Також для цього потрібно уникати циклів в коді, а всі обчислення масивів координат та швидкостей виконувати через векторні операції бібліотеки NumPy, що використовують SIMD-інструкції процесора.

Для максимальної ефективності просторова декомпозиція, багатопоточність та GPGPU часто комбінують. Наприклад, алгоритмизі складною логікою розгалуження доцільно виконувати на центральному процесорі із застосуванням просторової декомпозиції та паралельної обробки, тоді як масові математичні операції над великими наборами даних ефективніше виконувати на графічному процесорі.

В вище описаних модулях підрозділу 3.1 було частково реалізовано багатопоточність та GPGPU, завдяки використанню бібліотеки NumPy, і здебільшого це можна побачити в модулі physics_engine.py. Наприклад, в головному класі PhysicEngine методи для розрахунку фізики для рідини та тканини, _step_fluid та _satisfy_cloth_constraints, хоч і використовують звичайні цикли for, але також використовують методи з бібліотеки NumPy, завдяки чому було зафіксовано незначне підвищення продуктивності в сценах симуляцій.

На жаль, повного уникнення циклів для підвищення продуктивності не є можливим через деякі особливості моделювання фізичних взаємодій та обробки

логічних залежностей між об'єктами. Аналогічно, повноцінне введення в створені модулі такого метода, як декомпозиція, не має успішного очікуваного результату – впровадження функції для даного методу в модулі physics_engine.py та його застосування для найбільш продуктивної симуляції, а саме для імітації рідини, призвело до збільшення загального навантаження на систему. З огляду на це від використання даної функції в остаточній версії було вирішено відмовитися.

3.3 Тестування точності та продуктивності

Для перевірки та оцінки коректності роботи фізичного ядра було проведено тестування реалізованих симуляцій та аналіз поведінки моделей. Метою тестування було оцінювання фізичної достовірності моделей, стабільності їх роботи у часі та продуктивності програмного комплексу при різних рівнях навантаження. На рис 3.2 – 3.10 наведено характерні стани симуляцій на різних етапах виконання.

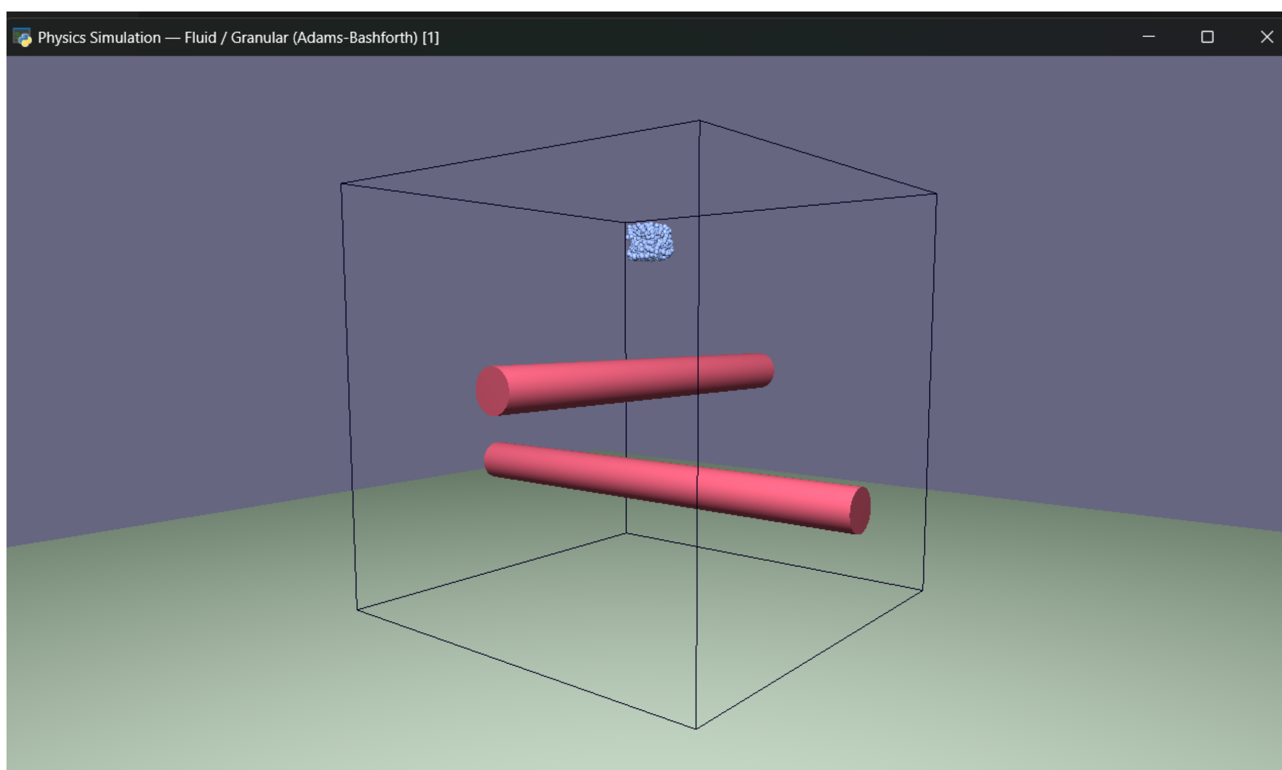


Рисунок 3.2 – Скріншот початкового стану симуляції рідини

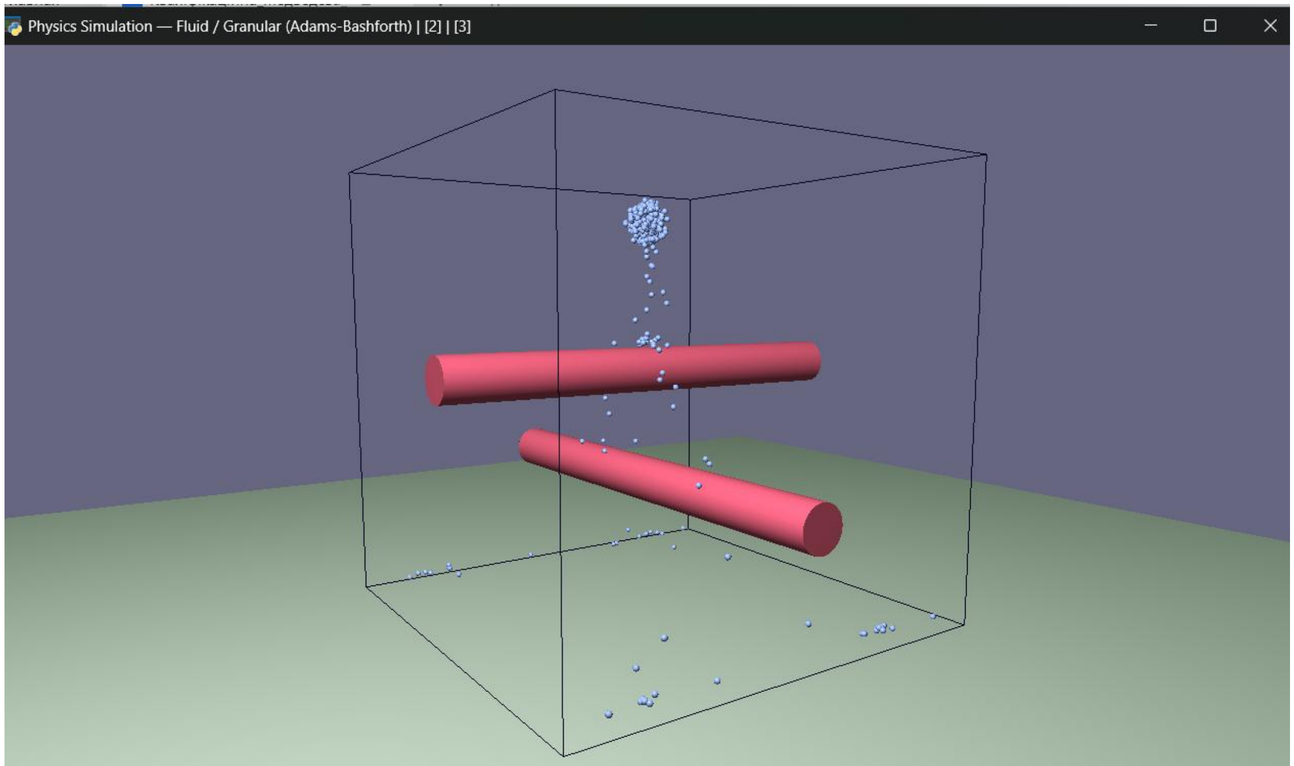


Рисунок 3.3 – Скріншот симуляції рідини через декілька секунд

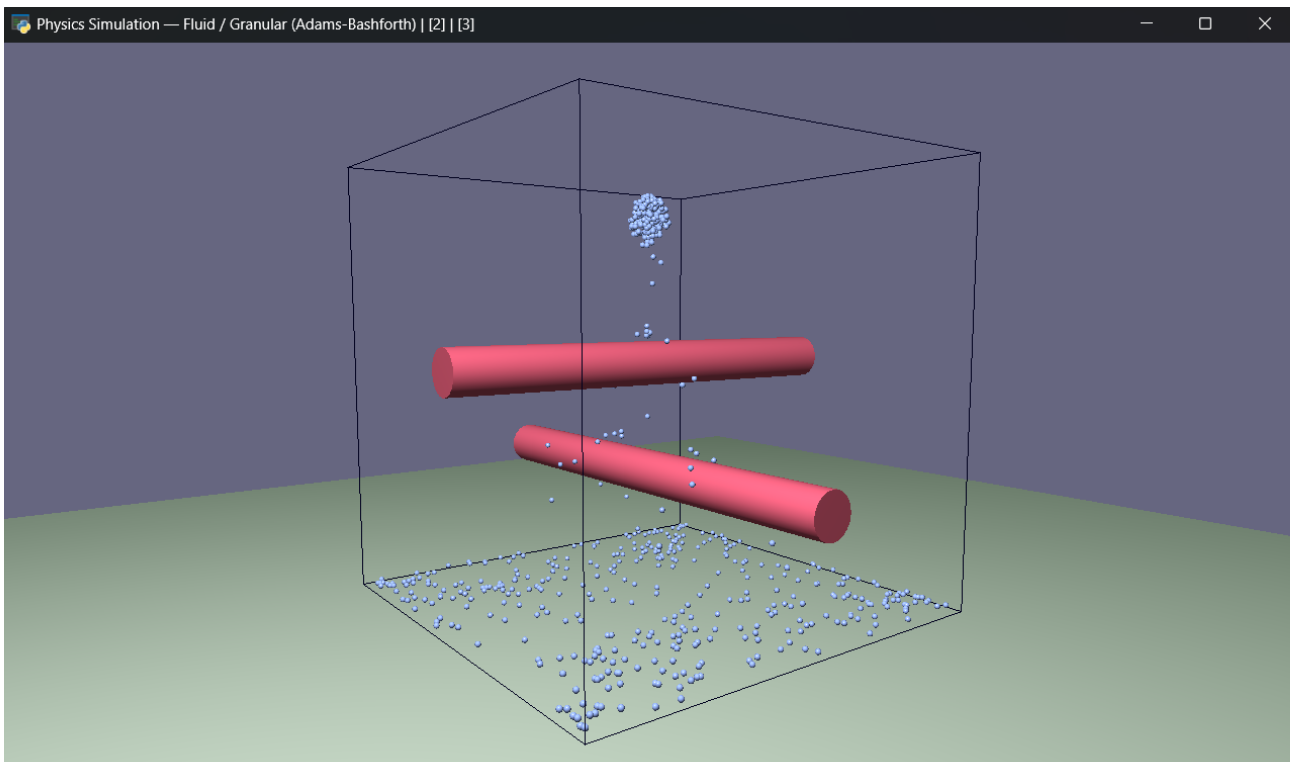


Рисунок 3.4 – Скріншот симуляції рідини через одну хвилину

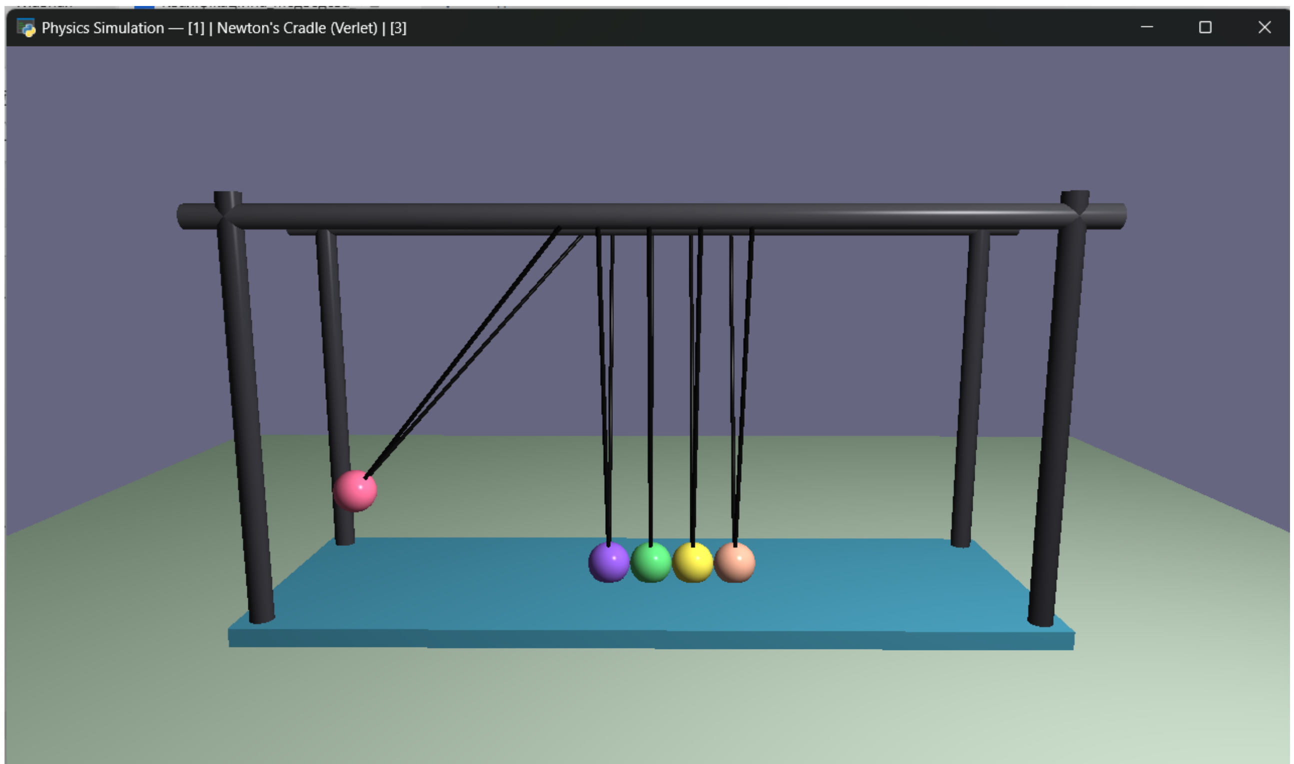


Рисунок 3.5 – Скріншот початкового стану симуляції маятника Ньютона

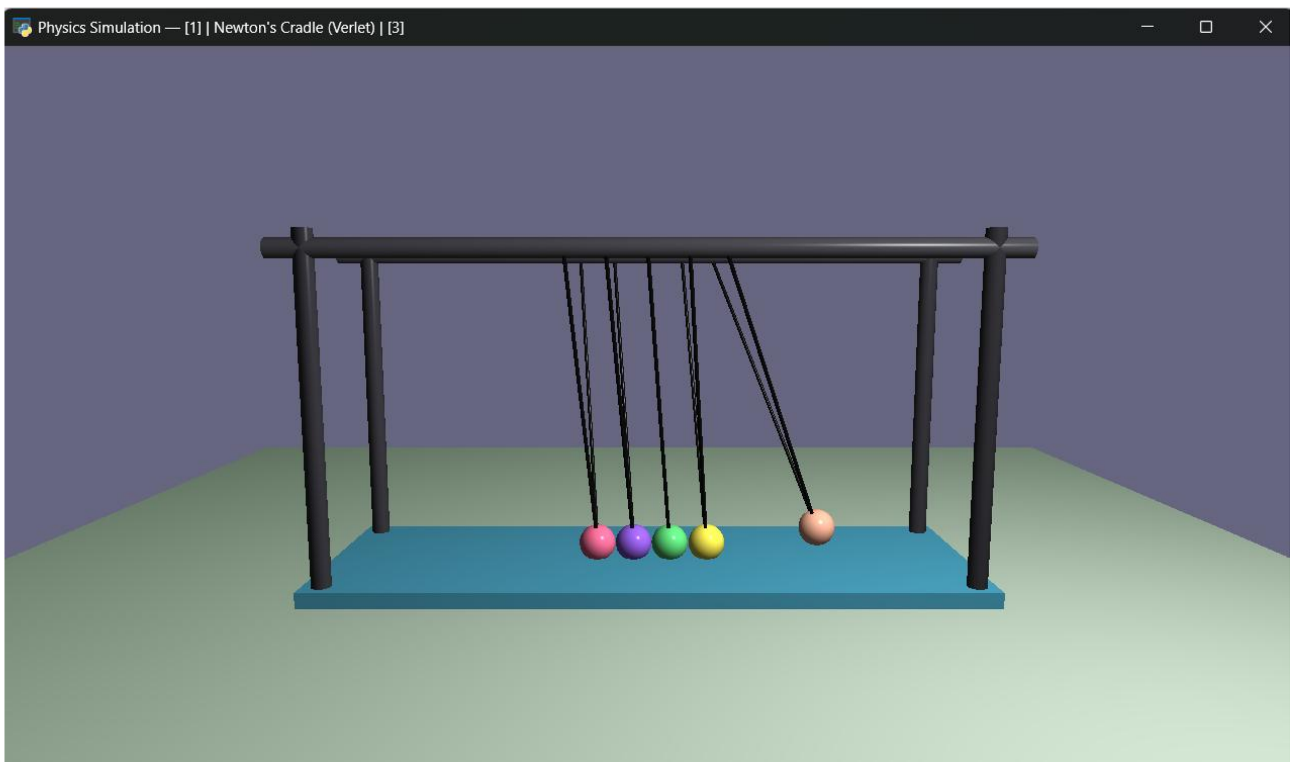


Рисунок 3.6 – Скріншот симуляції маятника Ньютона через декілька секунд

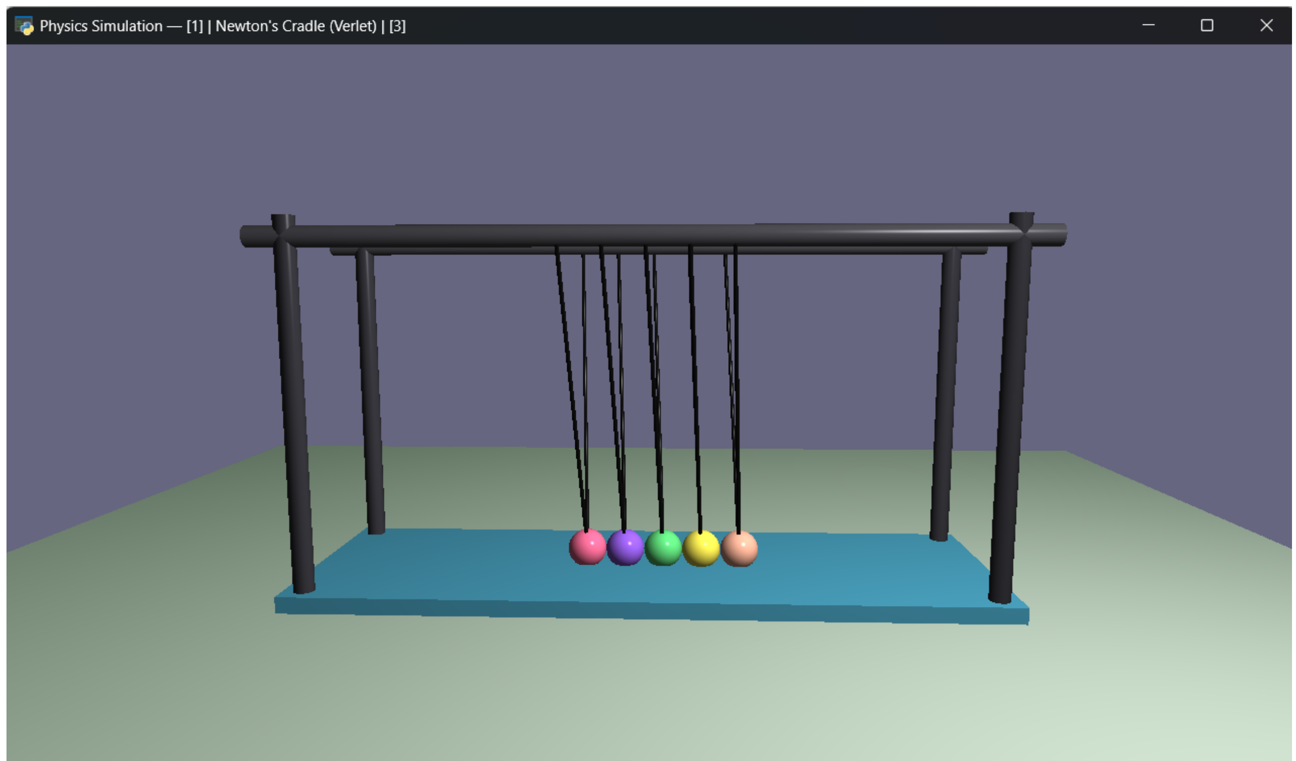


Рисунок 3.7 – Скріншот симуляції маятника Ньютона через одну хвилину

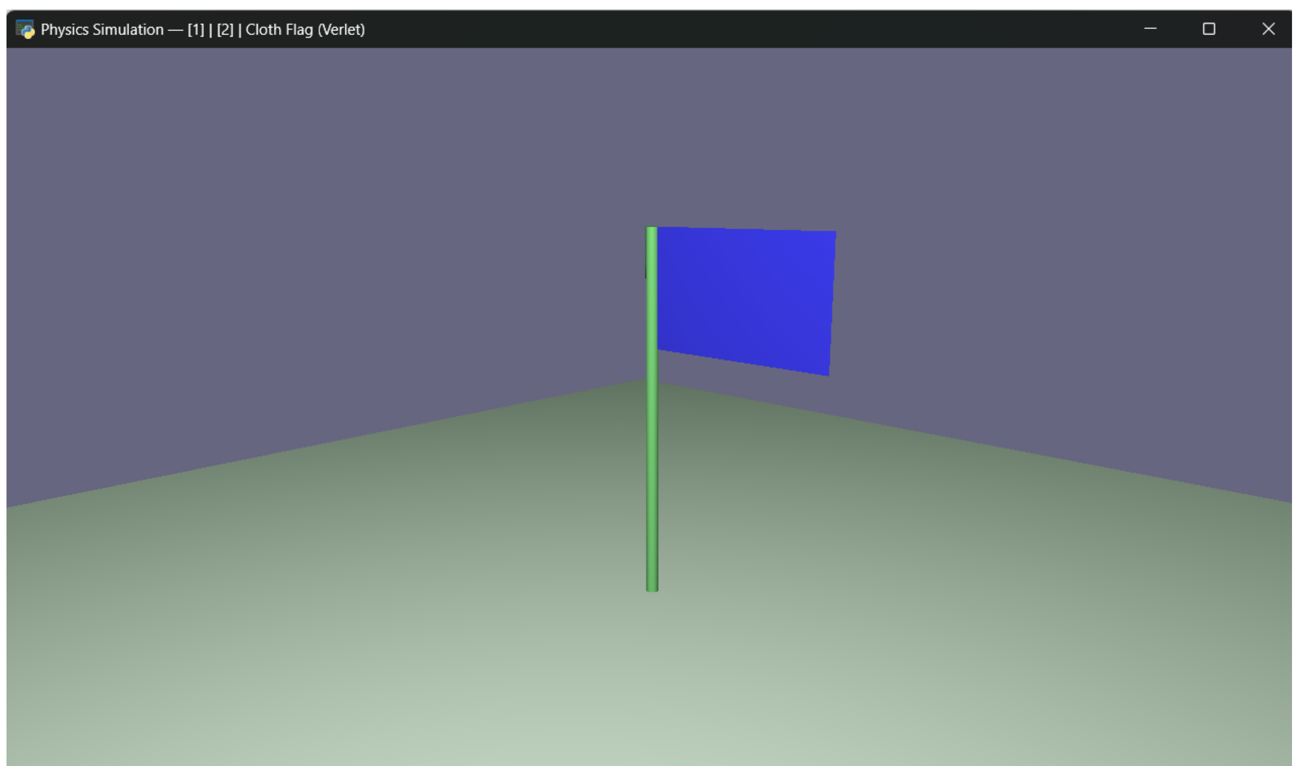


Рисунок 3.8 – Скріншот початкового стану симуляції тканини

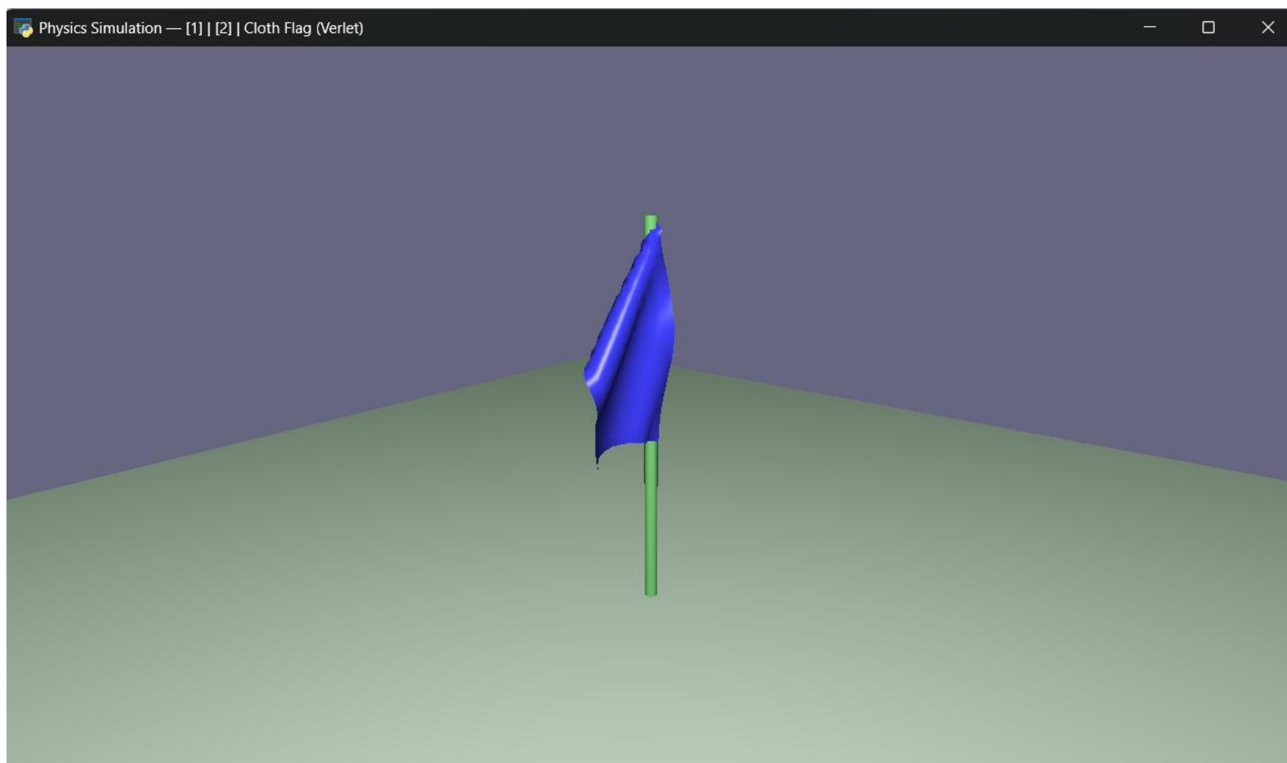


Рисунок 3.9 – Скріншот симуляції тканини через декілька секунд

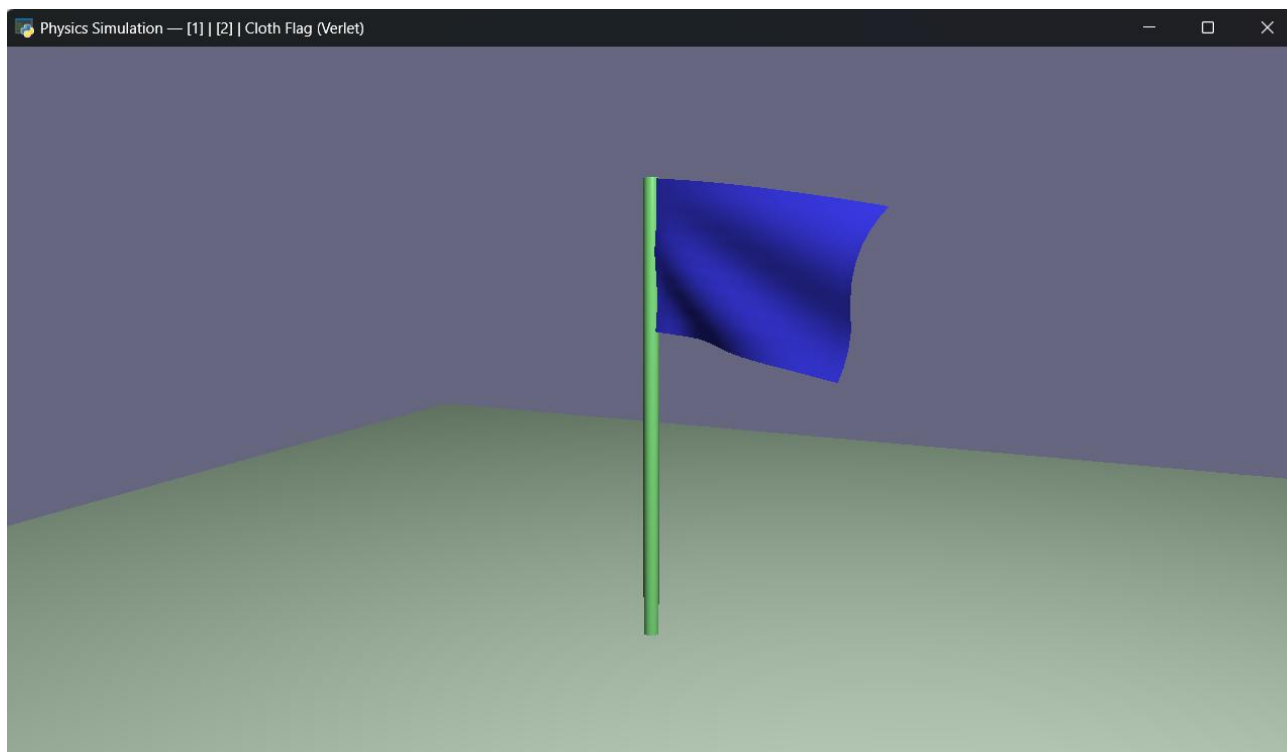


Рисунок 3.10 – Скріншот симуляції тканини через одну хвилину

Також для оцінки ефективності розробленого програмного комплексу було проведено серію тестів, спрямованих на вимірювання продуктивності при різному навантаженні, кількості об'єктів, та перевірку фізичної достовірності симуляції.

Тестування продуктивності проводилося на системі з характеристиками, вказаними в табл. 3.1. Результати вимірювань частоти кадрів для всіх трьох сцен представлені на рисунках 3.11 – 3.13.

Таблиця 3.1. Характеристики системи для тестування

Процесор, CPU	Intel® CoreTM i3-10110U CPU @ 2.10GHz
Графічний процесор, GPU	Intel® UHD Graphics
Оперативна пам’ять, RAM	8 ГБайт
Базова частота	2,1–2,5 ГГц
Ядра, Cores	2
Потоки, Threads	4
Сокети	1
Логічні процесори	4
Віртуалізація	Включено
Bus Speed	~100 МГц
Max TDP	15 W / Ват

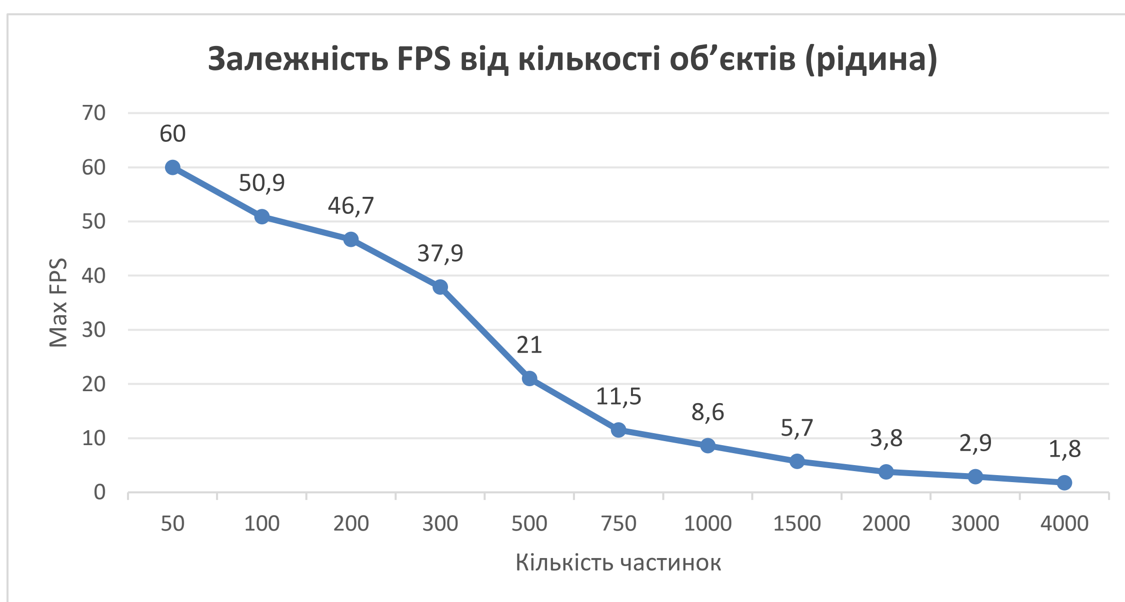


Рисунок 3.11 – Залежність продуктивності, FPS, від кількості активних фізичних об’єктів у сцені з рідиною

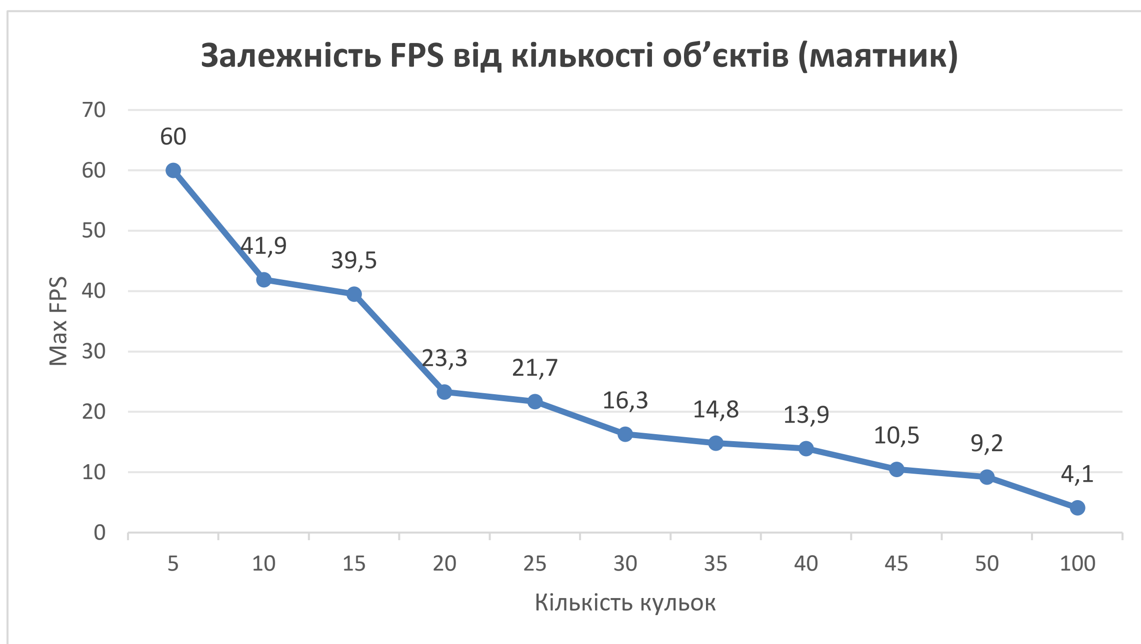


Рисунок 3.12 – Залежність продуктивності від кількості активних фізичних об’єктів у сцені з маятником

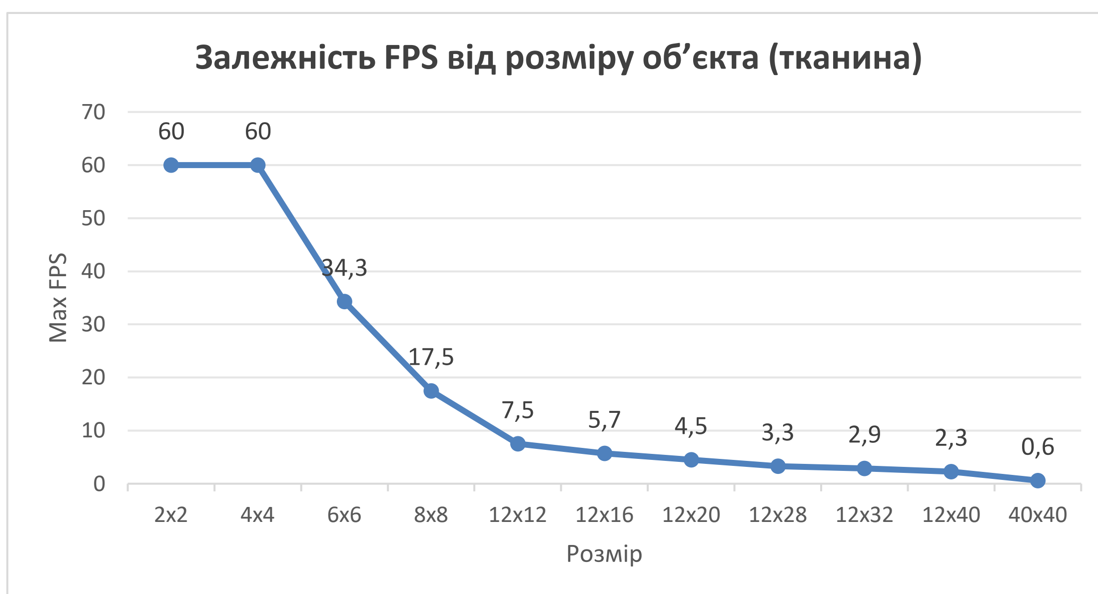


Рисунок 3.13 – Залежність продуктивності від розміру активного фізичного об’єкта у сцені з флагштоком

Як видно з графіків, при невеликій кількості та розмірі об’єктів, продуктивність залишається доволі стабільною на рівні 60-40 FPS. Подальше збільшення кількості чи розміру об’єктів призводить до поступового зниження FPS через зростання складності алгоритмів виявлення зіткнень та обмежень центрального процесора.

3.4 Аналіз отриманих результатів

Після проведення тестувань всіх трьох реалізованих імітацій можна сформулювати деякі висновки щодо отриманих результатів, продуктивності та фізичної достовірності.

Результати симуляції рідини та сипких тіл показали, що поведінка частинок більше відповідають сипким тіла, таким як пісок чи сіль, на що також був розрахунок при створенні даної моделі. При цьому, дана симуляція виконує свою задачу, демонструючи поведінку рідини та сипких тіл при взаємодії між собою та з перешкодами. Аналізуючи загалом дану модель, то її результати можна вважати задовільними і питань не задають.

Симуляція маятника Ньютона продемонструвала часткову відповідність з фізичною моделлю реального пристрою. На початку симуляції модель поводиться та коректно імітує відомий маятник, передаючи імпульс між кулями, але після кожного удару коливається не тільки куля з протилежного боку від самого удару, а й інші кулі з незначним коливанням, із-за чого частина енергії розподіляється між усіма кулями, імпульс слабішає і через декілька ударів симуляція стає майже статичною, виокремлюючи те, що всі кулі продовжують разом рухатися. Помилка такої поведінки з великою вірогідністю пов’язана з методом _step_cradle в головному класі PhysicsEngine модуля physics_engine.py, але спроби зміни та вдосконалення алгоритму в методі приводили або до погіршення помилки, або не змінювали нічого. Тому модель маятника можна вважати працездатною, хоча вона потребує подальшого вдосконалення для підвищення фізичної достовірності.

Остання симуляція з тканиною має суперечливі результати. Сама модель тканини успішно відтворює характерну поведінку гнучкого матеріалу та демонструє реалістичний рух під дією зовнішніх сил, але разом з тим були виявлені випадки, коли вона проходить крізь себе, незважаючи на наявність механізмів обробки відповідних зіткнень, що не є добре. Така поведінка свідчить про недостатню ефективність реалізованого алгоритму обробки колізій між окремими елементами тканини. Загалом дана симуляція має задовільні результати, але потрібно детальніше розглянути та опрацювати методи обробки зіткнень.

Щодо тестування продуктивності, то воно демонструє чітку залежність між кількістю об’єктів у сцені та частотою оновлення кадрів. При чому, тестування проводилося на комп’ютері з процесором Intel® Core™ i3-10110U, графічним процесором Intel® UHD Graphics та 8 ГБ оперативною пам’яттю. Отримані результати показали наступне:

– продуктивність в сцені з рідиною (рис. 3.11) залишається сталою на рівні 60-40 FPS з кількістю об’єктів до 500, але при збільшенні їх кількості FPS починає поступово зменшуватися;

– тестування сцени з маятником Ньютона (рис. 3.12) демонструє більш повільне зниження продуктивності. Навіть при 50 кульках система підтримує близько 9 FPS, а при невеликій кількості об’єктів забезпечує стабільні 60-40 FPS. Це пояснюється меншою кількістю перевірок зіткнень порівняно із симуляцією рідини;

– аналіз продуктивності моделі тканини (рис. 3.13) показав, що збільшення розміру сітки істотно впливає на швидкодію через зростання кількості вузлів та зв’язків між ними. Для сітки 40×40 частота кадрів знижується до 0,6 FPS, що практично виключає можливість роботи в режимі реального часу без додаткової оптимізації алгоритмів.

Основними причинами зниження продуктивності є збільшення кількості об’єктів, що значно ускладнює алгоритми обробки зіткнень, та апаратні обмеження тестової системи, оскільки не надто потужний процесор з двома ядрами та чотирма потоками стає помітним фактором з недостатньою кількістю можливостей і проблемою при обробці та взаємодії великої кількості фізичних моделей.

Але загалом розроблені моделі демонструють поведінку, що відповідає очікуваним фізичним закономірностям. Програмний комплекс демонструє стабільну роботу та забезпечує достатню точність для симуляції фізичних процесів, забезпечуючи при цьому комфортний рівень FPS при достатньому кількості об’єктів, що для заданої конфігурації обладнання, використаного в тестуванні, можна вважати прийнятним показником. Але водночас результати тестування вказують на необхідність подальшого вдосконалення деяких алгоритмів та методів.

Висновки до розділу 3

У третьому розділі було здійснено програмну реалізацію модульної системи для імітації трьох фізичних процесів, що включає три взаємозв'язані модулі: фізичний рушій physics_engine.py, візуалізатор render_core.py та сполучний модуль world_manager.py. Розроблена модульна архітектура дозволила розділити обчислювальну логіку та графічну візуалізацію, забезпечивши стабільне оновлення фізичних станів незалежно від швидкості рендерингу. Для реалізації даного проекту, разом з мовою програмування Python, було використано спеціалізовані бібліотеки: ModernGL для роботи з графічним API OpenGL, Pyglet для створення інтерфейсу користувача та NumPy для ефективних математичних обчислень. Зокрема, у підрозділі 3.1 було коротко описано використані бібліотеки, детально опрацьовано та описано логіку та методи кожного зробленого модуля.

У підрозділі 3.2 описано та частково реалізовано методи підвищення продуктивності для пришвидшення обчислень та візуалізації, а саме декомпозиція, багатопоточність та GPGPU. На жаль, повноцінне введення в створені модулі названі методи не мають успішного очікуваного результату. Але багатопоточність та GPGPU були частково реалізовані за допомогою математичної бібліотеки NumPy.

Підрозділ 3.3 присвячений тестуванню точності та продуктивності розробленої системи. В результаті цього, було виявлено чітку залежність між кількістю об'єктів в сцені та FPS, а саме те, що збільшення кількості або розмірів об'єктів призводить до поступового зниження FPS через зростання складності алгоритмів виявлення зіткнень та обмежень центрального процесора.

В наступному підрозділі 3.4 було зроблено аналіз та висновки тестувань з минулого підрозділу. Було виокремлено деякі недоліки та проблеми створених симуляцій, як часткова невідповідність однієї з моделі до реального аналога або проходження іншої моделі через себе. Але разом з цим розроблені моделі забезпечують достатню точність для симуляції фізичних процесів, забезпечуючи при цьому комфортний рівень FPS. Також було проаналізовано тестування продуктивності на певному процесорі.

Таким чином, у результаті виконаної роботи було реалізовано та досліджено симуляцію деяких фізичних процесів, яка виконує задану задачу, але при цьому має деякі недоліки, які потрібно дослідити та доопрацювати.

ВИСНОВКИ

У процесі виконання роботи було розглянуто теоретичні та практичні аспекти побудови систем імітації фізичних процесів у реальному часі з використанням сучасних засобів комп’ютерної графіки та чисельного моделювання. Проведено огляд та аналіз законів класичної механіки, які є фундаментом для створення більшості фізичних симуляцій, методів чисельного інтегрування, які використовуються для обчислення руху об’єктів у динамічних середовищах. Розглянуто особливості виконання фізичних розрахунків у режимі реального часу та здійснено аналіз сучасних фізичних рушіїв, їх переваг, недоліків і сфер застосування. Спроектовано архітектуру програмного комплексу, яка забезпечує взаємодію графічного ядра та фізичного рушія, а також створено тривимірні моделі об’єктів для демонстрації роботи системи.

Під час виконання кваліфікаційної роботи було виконано такі завдання:

- проаналізовано існуючі алгоритми тривимірної візуалізації, математичні моделі та чисельні методи інтегрування, що використовуються для обчислення траєкторій руху об’єктів та виявлення колізій;
- здійснено порівняльний аналіз сучасних фізичних рушіїв для моделювання фізичних процесів у реальному часі;
- спроектовано високопродуктивну архітектуру програмної системи, що забезпечує стабільну частоту оновлення кадрів в умовах обмежених обчислюваних ресурсів;
- реалізовано програмні модулі імітації базових фізичних властивостей та взаємодій об’єктів;
- поліпшено алгоритми виявлення зіткнень для об’єктів різної геометричної складності;
- проведено тестування розробленої програмної системи.

У межах практичної частини реалізовано фізичний рушій, що підтримує моделювання базових фізичних взаємодій, включаючи дію сили тяжіння, обробку зіткнень та обмежень між об’єктами. Також реалізовано візуалізацію фізичних

процесів і створено демонстраційні сцени для моделювання рідини, маятника Ньютона та тканини.

При чому, після створення програмної модульної системи було виявлено деякі недоліки, які потрібно дослідити та доопрацювати, оскільки частина з них створюють проблеми для повноцінної імітації фізичних процесів. Також було проведено тестування точності та аналіз продуктивності, які продемонстрували, що створені моделі забезпечують достатню точність для симуляції фізичних процесів при збереженні комфортного рівня FPS. Виявлено пряму залежність між складністю сцени та продуктивністю, що підтверджує важливість застосованих методів для підвищення стабільності роботи системи.

Практичне значення розробки полягає у створенні програмного комплексу, який поєднує методи 3D-візуалізації та фізичних розрахунків. Результати дослідження можуть бути використані як основа для подальшого розвитку фізичних рушіїв, навчальних симуляторів, інтерактивних тренажерів та інших програмних систем, що потребують реалістичного відтворення фізичних явищ.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Behind the Code: Applying the Laws of Motion in Game Development | by Vyshnavi Kathrine | Medium. URL: https://medium.com/@vyshnavikathrine/behind-the-code-applying-the-laws-of-motion-in-game-development-700bb31d188d (date of access: 21.04.2026)

2. Multiphysics Simulation Methods in Computer Graphics - Holz - 2025 - Computer Graphics Forum - Wiley Online Library. URL: https://onlinelibrary.wiley.com/doi/10.1111/cgf.70082 (date of access: 21.04.2026)

3. Три закони Ньютона - LibreTexts - Ukrayinska. URL: https://ukrayinska.libretexts.org/фізики/Університетська_фізика/Книга%3A_Вступна_фізика_-_побудова_моделей_для_опису_нашого_світу_(Martin_et_al.)/05%3A_Закони_Ньютона/5.01%3A_Три_закони_Ньютона (date of access: 22.04.2026)

4. Millington, Ian. Game Physics Engine Development: How to Build a Robust Commercial-Grade Physics Engine for your Game. CRC Press, 2010. 552 p.

5. Eberly D. H. Game Physics. San Francisco: Morgan Kaufmann, 2004. 835 p. URL: http://lib.ysu.am/disciplines_bk/d21285fc879e96b613640961ac802272.pdf (date of access: 21.04.2026)

6. Witkin A., Baraff D. Physically Based Modeling: Principles and Practice. Online Siggraph '97 Course Notes. 1997. URL: https://www.cs.cmu.edu/~baraff/sigcourse/ (date of access: 21.04.2026)

7. Класична механіка. URL: https://uk.wikipedia.org/wiki/Класична_механіка (дата звернення: 21.04.2026)

8. Ляшенко М.Я., Голованъ М.С. Чисельні методи: Підручник. Либідь. 1996. 288 с. URL: https://pdf.lib.vntu.edu.ua/books/2015/Lyashenko_1996_288.pdf (дата звернення: 23.04.2026)

9. Метод Рунге-Кутта: Просте Пояснення для Початківців. URL: https://www.mathros.net.ua/rozvjazuvannja-zvychajnyh-dyferencialnyh-rivnjan-metodom-runge-kutta.html (дата звернення: 24.04.2026)

10. Gregory J. Game engine architecture. Third Edition. A K Peters/CRC Press, 2017. 1240 p. URL (Download File): https://share.google/XwEKnpg7We08M4oF4 (date of access: 21.04.2026)

11. Matthias Muller, Jos Stam, Doug James, Nils Thurey. Real Time Physics Class Notes. SIGGRAPH. 2007. Page 87. URL: https://matthias-research.github.io/pages/publications/realtimeCoursenotes.pdf (date of access: 24.04.2026)

12. Fix Your Timestep! | Gaffer On Games. URL: https://gafferongames.com/post/fix_your_timestep/ (date of access: 24.04.2026)

13. Головіна Н.А., Головін М.Б. Методичні особливості моделювання фізичних явищ на прикладі взаємодіючих коливань. Фізика та освітні технології. Луцьк, Вип.2, 2021. С. 1-8. URL: http://journals.vnu.volyn.ua/index.php/physics/article/view/148/129 (дата звернення: 25.04.2026)

14. Головін М.Б. Аплікації з комп’ютерної фізики мовою Visual Python на прикладі моделювання силової взаємодії. Комп’ютерно-інтегровані технології: освіта, наука, виробництво. Луцьк, 2020. Випуск № 40 с.16-22. URL: https://evnuir.vnu.edu.ua/bitstream/123456789/19702/1/holovin_fedoniuk.pdf (дата звернення: 25.04.2026)

15. Welcome to PhysX — PhysX SDK Documentation. URL: https://nvidia-omniverse.github.io/PhysX/physx/5.6.1/ (date of access: 26.04.2026)

16. Havok Physics | Production-Proven Rigid Body Simulation for Games. URL: https://www.havok.com/havok-physics/ (date of access: 26.04.2026)

17. Bullet Physics Manual. URL: https://www.cs.kent.edu/~ruttan/GameEngines/lectures/Bullet_User_Manual (date of access: 26.04.2026)

18. Chaos Physics Overview | Unreal Engine 4.27 Documentation | Epic Developer Community. URL: https://dev.epicgames.com/documentation/unreal-engine/chaos-physics-overview?application_version=4.27 (date of access: 26.04.2026)

19. Jolt Physics: Jolt Physics. URL: https://jrouwe.github.io/JoltPhysics/ (date of access: 26.04.2026)

20. James Robert Bruce. Real-Time Motion Planning and Safe Navigation in Dynamic Multi-Robot Environments. CMU-CS-06-181. December 15, 2006. P. 49–70. URL: https://www.cs.cmu.edu/~jbruce/thesis/doc/JRB-Thesis.pdf (date of access: 28.04.2026)

21. Ґвідо ван Россум, Фред Л. Дарк. Підручник мови Python. URL: http://docs.linux.org.ua/Програмування/Python/Підручник_мови_Python/ (дата звернення: 25.04.2026)

22. Introduction to Computer Graphics - GeeksforGeeks. URL: https://www.geeksforgeeks.org/computer-graphics/introduction-to-computer-graphics/ (date of access: 30.04.2026)

23. C++ Programming Language - GeeksforGeeks. URL: https://www.geeksforgeeks.org/cpp/c-plus-plus/ (date of access: 01.05.2026)

24. Learn C# Programming - GeeksforGeeks. URL: https://www.geeksforgeeks.org/c-sharp/c-sharp-tutorial/ (date of access: 01.05.2026)

25. 3D Modeling & Animation App Using Python - GeeksforGeeks. URL: https://www.geeksforgeeks.org/python/3d-modeling-animation-app-using-python-1/ (date of access: 05.05.2026)

26. C++ дайджест #14: Graphics API — OpenGL, DirectX, Vulkan, Metal | DOU. URL: https://dou.ua/lenta/digests/plus-digest-14/ (дата звернення: 01.05.2026)

27. Getting Started - OpenGL Wiki. URL: https://wikis.khronos.org/opengl/Getting_Started (date of access: 01.05.2026)

28. Shreiner D., Selles Gr., Kessenich J., Licea-Kane B. OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3. Longman (Pearson Education), 2016. 986 p. URL: https://www.cs.utexas.edu/~fussell/courses/cs354/handouts/Addison.Wesley.OpenGL.Programming.Guide.8th.Edition.Mar.2013.ISBN.0321773039.pdf (date of access: 05.05.2026)

29. Vulkan Documentation :: Vulkan Documentation Project. URL: https://docs.vulkan.org/spec/latest/index.html (date of access: 01.05.2026)

30. WebGPU API - Web APIs | MDN. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebGPU_API (date of access: 01.05.2026)

31. Direct3D 12 programming guide - Win32 apps | Microsoft Learn. URL: https://learn.microsoft.com/en-us/windows/win32/direct3d12/directx-12-programming-guide (date of access: 01.05.2026)

ДОДАТОК А

Лістинг коду модуля physics_engine.py

```
"""Фізичний рушій: інтегратори (Адамс–Башфорт, Верле), двофазні колізії, сцени."""

from __future__ import annotations
from dataclasses import dataclass, field
from enum import Enum, auto
from typing import List, Optional, Sequence, Tuple
import numpy as np

Vec3 = np.ndarray

class VerletIntegrator:
    @staticmethod
    def step(positions: np.ndarray, prev_positions: np.ndarray,
        accelerations: np.ndarray, dt: float, mask: Optional[np.ndarray] = None,
    ) -> Tuple[np.ndarray, np.ndarray]:
        new_pos = 2*positions - prev_positions + accelerations * (dt * dt)
        if mask is not None:
            new_pos[mask] = positions[mask]
        return new_pos, positions.copy()

class AdamsBashforthIntegrator:
    @staticmethod
    def step(positions: np.ndarray, velocities: np.ndarray,
            accelerations: np.ndarray, prev_accelerations: Optional[np.ndarray],
            dt: float) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
        if prev_accelerations is None:
            new_vel = velocities + accelerations * dt
        else:
            new_vel = velocities + 0.5 * dt * (3.0 * accelerations - prev_accelerations)
        new_pos = positions + new_vel * dt
        return new_pos, new_vel, accelerations.copy()

@dataclass
class AABB:
    min_corner: Vec3
    max_corner: Vec3

    def overlaps(self, other: "AABB") -> bool:
        return bool(np.all(self.max_corner >= other.min_corner) and np.all(self.min_corner
<= other.max_corner))

    @staticmethod
    def from_center_extents(center: Vec3, half_extents: float | Vec3) -> "AABB":
        he = np.broadcast_to(half_extents, (3,))
        return AABB(center - he, center + he)
    @staticmethod
    def from_points(points: np.ndarray) -> "AABB":
        return AABB(points.min(axis=0), points.max(axis=0))
```

```
@dataclass
class Sphere:
    center: Vec3
    radius: float

    def overlaps(self, other: "Sphere") -> bool:
        distance = np.linalg.norm(self.center - other.center)
        return distance <= (other.radius + other.radius)

    @staticmethod
    def from_center_extents(center: Vec3, radius: float) -> "Sphere":
        return Sphere(center, radius)
    @staticmethod
    def from_points(points: np.ndarray) -> "Sphere":
        center = points.mean(axis=0)
        radius = np.max(np.linalg.norm(points - center, axis=1))
        return Sphere(center, radius)

@dataclass
class Contact:
    point: Vec3
    normal: Vec3
    penetration: float

class CollisionSolver:
    @staticmethod
    def broad_phase(pairs: Sequence[Tuple[AABB, AABB, int, int]]) -> List[Tuple[int,
int]]:
        hits: List[Tuple[int, int]] = []
        n = len(pairs)
        for i in range(n):
            a_aabb, b_aabb, ia, ib = pairs[i]
            if a_aabb.overlaps(b_aabb):
                hits.append((ia, ib))
        return hits

    @staticmethod
    def sphere_particle(c1: Vec3, r1: float, c2: Vec3, r2: float) -> Optional[Contact]:
        delta = c2 - c1
        dist_sq = float(np.dot(delta, delta))
        r_sum = r1 + r2
        if dist_sq >= r_sum * r_sum:
            return None
        dist = np.sqrt(dist_sq) + 1e-9
        normal = delta / dist
        penetration = r_sum - dist
        return Contact( point=c1 + normal * r1, normal=normal, penetration=penetration,)

    @staticmethod
    def sphere_cylinder_segment(
        center: Vec3, radius: float, p0: Vec3, p1: Vec3, cyl_radius: float,
```

```
    ) -> Optional[Contact]:
        axis = p1 - p0
        length_sq = float(np.dot(axis, axis)) + 1e-9
        t = np.clip(np.dot(center - p0, axis) / length_sq, 0.0, 1.0)
        closest = p0 + t * axis
        delta = center - closest
        dist = float(np.linalg.norm(delta))
        r_sum = radius + cyl_radius
        if dist >= r_sum:
            return None
        normal = delta / (dist + 1e-9)
        return Contact(point=closest, normal=normal, penetration=r_sum - dist)

    @staticmethod
    def resolve_particle(
        position: Vec3, velocity: Vec3, contact: Contact, restitution: float = 0.2,
    ) -> Tuple[Vec3, Vec3]:
        pos = position + contact.normal * contact.penetration
        vn = np.dot(velocity, contact.normal)
        if vn < 0:
            velocity = velocity - (1.0 + restitution) * vn * contact.normal
        return pos, velocity

class SceneKind(Enum):
    FLUID = auto()
    NEWTON_CRADLE = auto()
    CLOTH = auto()

@dataclass
class CylinderObstacle:
    p0: Vec3
    p1: Vec3
    radius: float
@dataclass
class RigidSphere:
    position: Vec3
    prev_position: Vec3
    radius: float
    mass: float
    pivot: Vec3
    rod_length: float
    color_index: int = 0

@dataclass
class FluidState:
    positions: np.ndarray
    velocities: np.ndarray
    prev_accelerations: Optional[np.ndarray]
    masses: np.ndarray
    bounds: AABB
    obstacles: List[CylinderObstacle]
    emitter_center: Vec3
```

```
    emitter_radius: float
    particle_radius: float
    spawn_rate: int = 1

    @classmethod
    def create_default(cls, n_particles: int = 2000) -> "FluidState":
        bounds = AABB(
            np.array([-2.0, 0.0, -2.0]), np.array([2.0, 4.0, 2.0]),
        )
        emitter = np.array([0.0, 3.5, 0.0])
        positions = emitter + np.random.uniform(-0.15, 0.15, (n_particles, 3))
        return cls(
            positions=positions.astype(np.float64),
            velocities=np.zeros((n_particles, 3)),
            prev_accelerations=None,
            masses=np.ones(n_particles),
            bounds=bounds,
            obstacles=[
                CylinderObstacle(
                    np.array([-1.0, 1.2, 0.0]), np.array([1.0, 1.2, 0.0]), 0.2),
                CylinderObstacle(
                    np.array([0.0, 2.2, -0.9]), np.array([0.0, 2.2, 0.9]), 0.2),
            ],
            emitter_center=emitter,
            emitter_radius=0.15,
            particle_radius=0.05
        )

@dataclass
class NewtonCradleState:
    spheres: List[RigidSphere]
    frame_restitution: float = 0.98
    sphere_count: int = 5

    @classmethod
    def create_default(cls) -> "NewtonCradleState":
        pivot_y = 3.5
        pivot_z = 0.0
        spacing = 0.4
        rod = 3.2
        spheres = []
        for i in range(cls.sphere_count):
            x = (i - 2) * spacing
            pivot = np.array([x, pivot_y, pivot_z])
            pos = pivot + np.array([0.0, -rod, 0.0])
            spheres.append(RigidSphere(
                    position=pos.copy(), prev_position=pos.copy(),
                    radius=0.2, mass=3.0,
                    pivot=pivot, rod_length=rod, color_index=i,
                ))
        spheres[0].position[0] -= spheres[0].pivot[0] - rod * np.sin(-45)
        spheres[0].position[1] += rod * (1 - np.cos(-45))
```

```
        spheres[0].prev_position = spheres[0].position.copy()
        return cls(spheres=spheres)

@dataclass
class ClothState:
    width: int
    height: int
    spacing: float
    positions: Vec3
    prev_positions: Vec3
    pin_mask: Vec3
    structural_k: float = 200.0
    damping: float = 0.98
    pole_anchor: Vec3 = np.array([0,0,0])

    @classmethod
    def create_flag(cls, cols: int = 20, rows: int = 12) -> "ClothState":
        spacing = 0.2
        origin = np.array([0.0, 3.5, 0.0])
        positions = []
        pin = []
        for j in range(rows):
            for i in range(cols):
                positions.append(origin + np.array([i * spacing, -j * spacing, 0.0]))
                pin.append(i == 0)
        pos = np.array(positions, dtype=np.float64)
        return cls(width=cols, height=rows, spacing=spacing, positions=pos,
            prev_positions=pos.copy(), pin_mask=np.array(pin, dtype=bool),
            pole_anchor=origin
        )

    def index(self, i: int, j: int) -> int:
        return j * self.width + i

# Головний клас фізичного ядра --------------------------------------------------
class PhysicsEngine:
    def __init__(self, dt: float = 1.0 / 60.0, gravity: Optional[Vec3] = None):
        self.dt = dt
        self.gravity = gravity if gravity is not None else np.array([0.0, -9.81, 0.0])
        self.wind = np.array([4.0, 0.0, 2.5])
        self.scene = SceneKind.FLUID
        self.fluid = FluidState.create_default()
        self.cradle = NewtonCradleState.create_default()
        self.cloth = ClothState.create_flag()
        self.collision = CollisionSolver()
        self._fluid_spawn_counter = 0
        self._cloth_simulation_time = 0.0

    def set_scene(self, kind: SceneKind):
        self.scene = kind

    def step(self, dt: Optional[float] = None):
```

```
        h = self.dt if dt is None else dt
        if self.scene == SceneKind.FLUID:
            self._step_fluid(h)
        elif self.scene == SceneKind.NEWTON_CRADLE:
            self._step_cradle(h)
        elif self.scene == SceneKind.CLOTH:
            self._step_cloth(h)

    # --- рідина / сипучі тіла ---
    def _step_fluid(self, dt: float):
        st = self.fluid
        n = len(st.positions)
        acc = np.tile(self.gravity, (n, 1))
        new_pos, new_vel, prev_acc = AdamsBashforthIntegrator.step(
            st.positions, st.velocities, acc, st.prev_accelerations, dt
        )

        self._fluid_spawn_counter += 20 * dt
        if self._fluid_spawn_counter % 3 == 0:
            idx = np.random.randint(0, n, st.spawn_rate)
            jitter = np.random.uniform(-0.08, 0.08, (st.spawn_rate, 3))
            new_pos[idx] = st.emitter_center + jitter
            new_vel[idx] = np.array([0.0, -1.5, 0.0])

        for i in range(n):
            pos = new_pos[i].copy()
            vel = new_vel[i].copy()
            pos, vel = self._fluid_collisions(pos, vel, st)
            new_pos[i] = pos
            new_vel[i] = vel
        st.positions = new_pos
        st.velocities = new_vel
        st.prev_accelerations = prev_acc

    def _fluid_collisions(self, pos: Vec3, vel: Vec3, st: FluidState) -> Tuple[Vec3,
Vec3]:
        for axis in range(3):
            if pos[axis] - st.particle_radius < st.bounds.min_corner[axis]:
                n = np.zeros(3)
                n[axis] = 1.0
                c = Contact(pos, n, st.bounds.min_corner[axis] - (pos[axis] -
st.particle_radius))
                pos, vel = self.collision.resolve_particle(pos, vel, c)
            if pos[axis] + st.particle_radius > st.bounds.max_corner[axis]:
                n = np.zeros(3)
                n[axis] = -1.0
                c = Contact( pos, n,
                    (pos[axis] + st.particle_radius) - st.bounds.max_corner[axis]
                )
                pos, vel = self.collision.resolve_particle(pos, vel, c)

        particle_aabb = AABB.from_center_extents(pos, st.particle_radius)
```

```
        broad_pairs = []
        for oi, obs in enumerate(st.obstacles):
            min_p = np.minimum(obs.p0, obs.p1) - obs.radius
            max_p = np.maximum(obs.p0, obs.p1) + obs.radius
            obs_aabb = AABB(min_p, max_p)
            broad_pairs.append((particle_aabb, obs_aabb, 0, oi))

        for _, oi_actual in self.collision.broad_phase(broad_pairs):
            obs = st.obstacles[oi_actual]
            contact = self.collision.sphere_cylinder_segment(
                pos, st.particle_radius, obs.p0, obs.p1, obs.radius
            )
            if contact:
                pos, vel = self.collision.resolve_particle(pos, vel, contact)

        emitter_contact = self.collision.sphere_particle(
            pos, st.particle_radius, st.emitter_center, st.emitter_radius
        )
        if emitter_contact:
            pos, vel = self.collision.resolve_particle(pos, vel, emitter_contact)
        return pos, vel

    # --- маятник Ньютона ---
    def _step_cradle(self, dt: float):
        spheres = self.cradle.spheres
        for s in spheres:
            new_pos = VerletIntegrator.step(s.position, s.prev_position, self.gravity, dt)
            s.prev_position = s.position.copy()
            s.position = new_pos[0]
            delta = s.position - s.pivot
            dist = np.linalg.norm(delta) + 1e-9
            s.position = s.pivot + (delta / dist) * s.rod_length

        for _ in range(self.cradle.sphere_count):
            for i in range(len(spheres) - 1):
                a, b = spheres[i], spheres[i + 1]
                dist_vec = b.position - a.position
                dist = np.linalg.norm(dist_vec)
                min_dist = a.radius + b.radius

                if dist < min_dist:
                    overlap = min_dist - dist
                    normal = dist_vec / dist
                    correction = normal * (overlap * 0.5)
                    a.position -= correction
                    b.position += correction
                    v_a = a.position - a.prev_position
                    v_b = b.position - b.prev_position
                    rel_vel = np.dot(v_b - v_a, normal)
                    if rel_vel < 0:
                        impulse = -(1.0 + self.cradle.frame_restitution) * rel_vel * 0.5
                        v_a -= impulse * normal
```

```
                    v_b += impulse * normal
                    a.prev_position = a.position - v_a
                    b.prev_position = b.position - v_b

    # --- прапор / тканина ---
    def _step_cloth(self, dt: float):
        st = self.cloth
        n = len(st.positions)
        acc = np.tile(self.gravity, (n, 1))
        wind_time_ratio = min(1.0, self._cloth_simulation_time / 4.0)
        for i, p in enumerate(st.positions):
            height_ratio = max(0.0, min(1.0, (p[1] - st.pole_anchor[1]) / (-2.0)))
            wind_factor = 3.0 + 0.3 * (2.0 - height_ratio)
            acc[i] += self.wind * wind_factor * wind_time_ratio

        self._cloth_simulation_time += dt
        new_pos, new_prev = VerletIntegrator.step(
            st.positions, st.prev_positions, acc, dt, st.pin_mask
        )
        st.prev_positions = new_prev
        st.positions = new_pos

        self._satisfy_cloth_constraints(st)
        self._cloth_collisions(st)

    def _cloth_collisions(self, st: ClothState):
        p0 = np.array([st.pole_anchor[0], 0.0, st.pole_anchor[2]])
        p1 = st.pole_anchor
        pole_radius = 0.07
        particle_radius = 0.02

        for i in range(len(st.positions)):
            if st.pin_mask[i]:  continue
            if st.positions[i, 1] < 0.0:
                st.positions[i, 1] = 0.0
                vel = st.positions[i] - st.prev_positions[i]
                st.prev_positions[i, 1] = st.positions[i, 1] + vel[1] * 0.3
            contact = self.collision.sphere_cylinder_segment(st.positions[i],
particle_radius, p0, p1, pole_radius)
            if contact:
                st.positions[i] += contact.normal * contact.penetration
                vel = st.positions[i] - st.prev_positions[i]
                vn = np.dot(vel, contact.normal)
                if vn < 0:
                    vel -= vn * contact.normal
                st.prev_positions[i] = st.positions[i] - vel * 0.9

    def _satisfy_cloth_constraints(self, st: ClothState):
        offsets = [ (1, 0, 1.0), (0, 1, 1.0),
            (1, 1, np.sqrt(2)), (-1, 1, np.sqrt(2)),
            (2, 0, 2.0), (0, 2, 2.0) ]
```

```
        for _ in range(10):
            for idx in np.where(st.pin_mask)[0]:
                j = idx // st.width
                st.positions[idx] = st.pole_anchor + np.array([0.0, -j * st.spacing, 0.0])
            for j in range(st.height):
                for i in range(st.width):
                    for dx, dy, mult in offsets:
                        ni, nj = i + dx, j + dy
                        if 0 <= ni < st.width and 0 <= nj < st.height:
                            self._apply_distance_constraint(st, st.index(i, j),
st.index(ni, nj), st.spacing * mult)

    @staticmethod
    def _apply_distance_constraint(st: ClothState, ia: int, ib: int, rest_length: float):
        pa = st.positions[ia]
        pb = st.positions[ib]
        dist = np.linalg.norm(pb - pa) + 1e-9
        diff = (dist - rest_length) / dist
        correction = 0.5 * (pb - pa) * diff
        if not st.pin_mask[ia]:
            st.positions[ia] += correction
        if not st.pin_mask[ib]:
            st.positions[ib] -= correction

    # --- дані для рендерера ---
    def get_render_snapshot(self) -> dict:
        if self.scene == SceneKind.FLUID:
            return { "kind": "fluid",
                "particles": self.fluid.positions.copy(),
                "particle_radius": self.fluid.particle_radius,
                "bounds": self.fluid.bounds,
                "obstacles": self.fluid.obstacles,
                "emitter": self.fluid.emitter_center,
                "emitter_radius": self.fluid.emitter_radius,
            }
        if self.scene == SceneKind.NEWTON_CRADLE:
            return { "kind": "cradle",
                "spheres": [ {
                        "position": s.position.copy(),
                        "radius": s.radius,
                        "pivot": s.pivot.copy(),
                        "color_index": s.color_index,
                    } for s in self.cradle.spheres ],
            }
        return { "kind": "cloth",
            "vertices": self.cloth.positions.copy(),
            "width": self.cloth.width,
            "height": self.cloth.height,
            "pole_anchor": self.cloth.pole_anchor.copy(),
        }
```

ДОДАТОК Б

Лістинг коду модуля render_core.py

```
"""Візуалізатор OpenGL 3.3+: VBO/EBO, GLSL, освітлення Фонга."""

from dataclasses import dataclass
from pathlib import Path
from typing import List, Optional, Tuple
import moderngl
import numpy as np
from pyrr import Matrix44, Vector3, quaternion

def _load_shader_source(name: str) -> str:
    shader_dir = Path(__file__).resolve().parent / "shaders"
    path = shader_dir / name
    return path.read_text(encoding="utf-8")

def _normalize(v: np.ndarray) -> np.ndarray:
    n = np.linalg.norm(v)
    return v / n if n > 1e-9 else v

def create_cube_mesh(half) -> Tuple[np.ndarray, np.ndarray]:
    s = half
    faces = [
        ([-s, -s, s], [0, 0, 1]), ([s, -s, s], [0, 0, 1]),
        ([s, s, s], [0, 0, 1]), ([-s, s, s], [0, 0, 1]),
        ([s, -s, -s], [0, 0, -1]), ([-s, -s, -s], [0, 0, -1]),
        ([-s, s, -s], [0, 0, -1]), ([s, s, -s], [0, 0, -1]),
        ([-s, s, s], [0, 1, 0]), ([s, s, s], [0, 1, 0]),
        ([s, s, -s], [0, 1, 0]), ([-s, s, -s], [0, 1, 0]),
        ([-s, -s, -s], [0, -1, 0]), ([s, -s, -s], [0, -1, 0]),
        ([s, -s, s], [0, -1, 0]), ([-s, -s, s], [0, -1, 0]),
        ([s, -s, s], [1, 0, 0]), ([s, -s, -s], [1, 0, 0]),
        ([s, s, -s], [1, 0, 0]), ([s, s, s], [1, 0, 0]),
        ([-s, -s, -s], [-1, 0, 0]), ([-s, -s, s], [-1, 0, 0]),
        ([-s, s, s], [-1, 0, 0]), ([-s, s, -s], [-1, 0, 0])
    ]
    verts = []
    for i, (pos, norm) in enumerate(faces):
        u = (i % 2) * 1.0
        v = (i // 2 % 2) * 1.0
        verts.extend([*pos, *norm, u, v])
    indices = np.array([
            0, 1, 2, 2, 3, 0,
            4, 5, 6, 6, 7, 4,
            8, 9, 10, 10, 11, 8,
            12, 13, 14, 14, 15, 12,
            16, 17, 18, 18, 19, 16,
            20, 21, 22, 22, 23, 20,
        ], dtype=np.uint32 )
    return np.array(verts, dtype="f4"), indices
```

```
def create_cube_wireframe(half) -> Tuple[np.ndarray, np.ndarray]:
    s = half
    corners = [
        (-s, -s, -s), (s, -s, -s), (s, s, -s), (-s, s, -s),
        (-s, -s, s), (s, -s, s), (s, s, s), (-s, s, s)
    ]
    verts = []
    for (x, y, z) in corners:
        verts.extend([x, y, z, 0.0, 0.0, 0.0, 0.0, 0.0])
    edges = [
        (0, 1), (1, 2), (2, 3), (3, 0),
        (4, 5), (5, 6), (6, 7), (7, 4),
        (0, 4), (1, 5), (2, 6), (3, 7),
    ]
    idx = []
    for a, b in edges:
        idx.extend([a, b])
    return np.array(verts, dtype="f4"), np.array(idx, dtype=np.uint32)

def create_sphere_mesh(radius, stacks, slices):
    verts = []
    indices = []
    for i in range(stacks + 1):
        phi = np.pi * i / stacks
        for j in range(slices + 1):
            theta = 2 * np.pi * j / slices
            x = radius * np.sin(phi) * np.cos(theta)
            y = radius * np.cos(phi)
            z = radius * np.sin(phi) * np.sin(theta)
            nx, ny, nz = _normalize(np.array([x, y, z]))
            u = j / slices
            v = i / stacks
            verts.extend([x, y, z, nx, ny, nz, u, v])
    for i in range(stacks):
        for j in range(slices):
            a = i * (slices + 1) + j
            b = a + slices + 1
            indices.extend([a, b, a + 1, a + 1, b, b + 1])
    return np.array(verts, dtype="f4"), np.array(indices, dtype=np.uint32)

def create_cylinder_mesh(radius, half_height, segments):
    verts = []
    indices = []
    for ring in (0, 1):
        y = half_height if ring else -half_height
        ny = 1.0 if ring else -1.0
        base = len(verts) // 8
        for i in range(segments):
            ang = 2 * np.pi * i / segments
            x = radius * np.cos(ang)
            z = radius * np.sin(ang)
```

```
            verts.extend([x, y, z, 0, ny, 0, i / segments, ring])
        if ring:
            for i in range(1, segments - 1):
                indices.extend([base, base + i, base + i + 1])
        else:
            for i in range(1, segments - 1):
                indices.extend([base, base + i + 1, base + i])
    side_base = len(verts) // 8
    for i in range(segments):
        ang = 2 * np.pi * i / segments
        x = radius * np.cos(ang)
        z = radius * np.sin(ang)
        nx = np.cos(ang)
        nz = np.sin(ang)
        verts.extend([x, half_height, z, nx, 0, nz, i / segments, 1])
        verts.extend([x, -half_height, z, nx, 0, nz, i / segments, 0])
    for i in range(segments):
        i2 = (i + 1) % segments
        a = side_base + i * 2
        b = side_base + i2 * 2
        indices.extend([a, a + 1, b, b, a + 1, b + 1])
    return np.array(verts, dtype="f4"), np.array(indices, dtype=np.uint32)

def create_plane_mesh(width, height):
    hw, hh = width * 0.5, height * 0.5
    verts = np.array( [
            -hw, 0, -hh, 0, 1, 0, 0, 0,
            hw, 0, -hh, 0, 1, 0, 1, 0,
            hw, 0, hh, 0, 1, 0, 1, 1,
            -hw, 0, hh, 0, 1, 0, 0, 1,
        ], dtype="f4" )
    indices = np.array([0, 1, 2, 2, 3, 0], dtype=np.uint32)
    return verts, indices

def update_cloth_normals(vertices: np.ndarray, width: int, height: int) -> np.ndarray:
    n_verts = width * height
    normals = np.zeros((n_verts, 3))
    for j in range(height - 1):
        for i in range(width - 1):
            a = j * width + i
            b = a + 1
            c = a + width
            d = c + 1
            for tri in ((a, c, b), (b, c, d)):
                p0, p1, p2 = vertices[tri[0]], vertices[tri[1]], vertices[tri[2]]
                n = np.cross(p1 - p0, p2 - p0)
                for k in tri:
                    normals[k] += n
    mesh = np.zeros((n_verts, 8), dtype="f4")
    for k in range(n_verts):
        nn = _normalize(normals[k])
        i = k % width
```

```
        j = k // width
        p = vertices[k]
        mesh[k] = [p[0], p[1], p[2], nn[0], nn[1], nn[2], i / width, j / height]
    return mesh.flatten()

@dataclass
class Material:
    ambient: Tuple[float, float, float] = (0.3, 0.3, 0.3)
    diffuse: Tuple[float, float, float] = (0.7, 0.7, 0.7)
    specular: Tuple[float, float, float] = (0.5, 0.5, 0.5)
    shininess: float = 32.0
    alpha: float = 1.0

class Mesh:
    def __init__(self, ctx, vertices: np.ndarray, indices: np.ndarray):
        self.ctx = ctx
        self.index_count = len(indices)
        self.vbo = ctx.buffer(vertices.tobytes())
        self.ebo = ctx.buffer(indices.tobytes())
        self.vao = None

    def bind(self, program):
        self.vao = self.ctx.vertex_array( program,
            [(self.vbo, "3f 3f 2f", "in_position", "in_normal", "in_texcoord")],
            self.ebo,
        )

    def update_vertices(self, vertices: np.ndarray):
        self.vbo.write(vertices.astype("f4").tobytes())

class RenderCore:
    def __init__(self, ctx, width: int, height: int):
        self.ctx = ctx
        self.width = width
        self.height = height
        self.ctx.enable(moderngl.DEPTH_TEST)
        self.prog = self.ctx.program(
            vertex_shader=_load_shader_source("phong.vert"),
            fragment_shader=_load_shader_source("phong.frag"),
        )
        self.light_pos = Vector3([4.0, 6.0, 5.0])
        self.view_pos = Vector3([0.0, 3.2, 11.0])

        cube_v, cube_i = create_cube_mesh(1.0)
        cube_wire_v, cube_wire_i = create_cube_wireframe(1.0)
        sphere_v, sphere_i = create_sphere_mesh(1.0, 16, 24)
        cyl_v_fluid, cyl_i_fluid = create_cylinder_mesh(1.0, 3.0, 20)
        cyl_v_cradle, cyl_i_cradle = create_cylinder_mesh(1.0, 1.5, 20)
        cyl_v_cloth, cyl_i_cloth = create_cylinder_mesh(1.0, 0.5, 20)
        plane_v, plane_i = create_plane_mesh(10.0, 10.0)
        part_v, part_i = create_sphere_mesh(0.5, 20, 50)
```

```
        self.mesh_cube = Mesh(ctx, cube_v, cube_i)
        self.mesh_cube_wire = Mesh(ctx, cube_wire_v, cube_wire_i)
        self.mesh_sphere = Mesh(ctx, sphere_v, sphere_i)
        self.mesh_cylinder_fl = Mesh(ctx, cyl_v_fluid, cyl_i_fluid)
        self.mesh_cylinder_cr = Mesh(ctx, cyl_v_cradle, cyl_i_cradle)
        self.mesh_cylinder_cl = Mesh(ctx, cyl_v_cloth, cyl_i_cloth)
        self.mesh_plane = Mesh(ctx, plane_v, plane_i)
        self.mesh_particle = Mesh(ctx, part_v, part_i)
        self.cloth_mesh: Optional[Mesh] = None
        self.particle_instances: List[Matrix44] = []

        for m in (self.mesh_cube, self.mesh_cube_wire, self.mesh_sphere,
self.mesh_cylinder_fl, self.mesh_cylinder_cr, self.mesh_cylinder_cl, self.mesh_plane,
self.mesh_particle):
            m.bind(self.prog)

    def resize(self, width: int, height: int):
        self.width = max(1, width)
        self.height = max(1, height)
        self.ctx.viewport = (0, 0, self.width, self.height)

    def _set_material(self, mat: Material):
        self.prog["u_ambient"].value = tuple(mat.ambient)
        self.prog["u_diffuse"].value = tuple(mat.diffuse)
        self.prog["u_specular"].value = tuple(mat.specular)
        self.prog["u_shininess"].value = mat.shininess
        self.prog["u_alpha"].value = mat.alpha
        self.prog["u_use_texture"].value = False

    def _draw_mesh(self, mesh: Mesh, model: Matrix44, view: Matrix44, proj: Matrix44, mat:
Material):
        self._set_material(mat)
        self.prog["u_model"].write(model.astype("f4").tobytes())
        self.prog["u_view"].write(view.astype("f4").tobytes())
        self.prog["u_projection"].write(proj.astype("f4").tobytes())
        self.prog["u_light_pos"].value = tuple(self.light_pos)
        self.prog["u_view_pos"].value = tuple(self.view_pos)
        if mesh.vao: mesh.vao.render()

    def _cylinder_between(self, p0: np.ndarray, p1: np.ndarray, radius: float):
        p0 = np.array(p0, dtype=float)
        p1 = np.array(p1, dtype=float)
        mid = 0.5 * (p0 + p1)
        direction = p1 - p0
        length = float(np.linalg.norm(direction))
        if length < 1e-9:
            rot = Matrix44.identity()
        else:
            direction_n = direction / length
            up = np.array([0.0, 1.0, 0.0])
            axis = np.cross(up, direction_n)
            axis_len = float(np.linalg.norm(axis))
```

```
            if axis_len > 1e-6:
                axis_n = axis / axis_len
                angle = float(np.arccos(np.clip(np.dot(up, direction_n), -1.0, 1.0)))
                q = quaternion.create_from_axis_rotation(axis_n.astype('f4'), angle)
                rot = Matrix44.from_quaternion(q)
            else:
                if np.dot(up, direction_n) < 0:
                    rot = Matrix44.from_x_rotation(np.pi)
                else:
                    rot = Matrix44.identity()

        model = (Matrix44.from_translation(Vector3(mid)) * rot
            * Matrix44.from_scale(Vector3([radius, length * 0.37, radius])) )
        return model

    def render_scene(self, snapshot: dict, camera_eye: Tuple[float, float, float] = (0,
3.2, 11)):
        self.ctx.clear(0.4, 0.4, 0.5)
        self.view_pos = Vector3(camera_eye)
        view = Matrix44.look_at(camera_eye, (0, 1.8, 0), (0, 1, 0))
        proj = Matrix44.perspective_projection(45.0, self.width / self.height, 0.1, 100.0)

        floor_model = Matrix44.from_translation(Vector3([0, -0.01, 0]))
        self._draw_mesh(self.mesh_plane, floor_model, view, proj,
            Material(ambient=(0.2, 0.3, 0.2), diffuse=(0.7, 0.7, 0.7), alpha=1.0)
        )
        kind = snapshot.get("kind")
        if kind == "fluid":
            self._render_fluid(snapshot, view, proj)
        elif kind == "cradle":
            self._render_cradle(snapshot, view, proj)
        elif kind == "cloth":
            self._render_cloth(snapshot, view, proj)

    def _render_fluid(self, snap: dict, view: Matrix44, proj: Matrix44):
        b = snap["bounds"]
        center = (b.min_corner + b.max_corner) * 0.5
        extents = (b.max_corner - b.min_corner) * 0.5
        model = Matrix44.from_translation(Vector3(center)) * Matrix44.from_scale(extents)
        self._set_material(Material(ambient=(0.05, 0.05, 0.05), diffuse=(0.06, 0.06,
0.06), alpha=0.95))
        self.prog["u_model"].write(model.astype("f4").tobytes())
        self.prog["u_view"].write(view.astype("f4").tobytes())
        self.prog["u_projection"].write(proj.astype("f4").tobytes())
        self.prog["u_light_pos"].value = tuple(self.light_pos)
        self.prog["u_view_pos"].value = tuple(self.view_pos)
        self.mesh_cube_wire.vao.render(mode=moderngl.LINES)
        self.ctx.disable(self.ctx.BLEND)

        for obs in snap["obstacles"]:
            p0, p1, r = obs.p0, obs.p1, obs.radius
            model = self._cylinder_between(p0, p1, r)
```

```
            self._draw_mesh(self.mesh_cylinder_fl, model, view, proj,
Material(diffuse=(0.85, 0.35, 0.45)))
        for p in snap["particles"]:
            pr = snap["particle_radius"]
            model = Matrix44.from_translation(Vector3(p)) *
Matrix44.from_scale(Vector3([pr, pr, pr]))
            self._draw_mesh(self.mesh_particle, model, view, proj, Material(diffuse=(0.55,
0.65, 0.9), shininess=64.0))

    def _render_cradle(self, snap: dict, view: Matrix44, proj: Matrix44):
        colors = [
            (0.9, 0.4, 0.55), (0.55, 0.35, 0.85),
            (0.35, 0.8, 0.45), (0.9, 0.85, 0.3),
            (0.9, 0.65, 0.55)
        ]
        base = (Matrix44.from_translation(Vector3([0, 0.05, 0]))
                * Matrix44.from_scale(Vector3([3.5, 0.1, 1.2])))
        self._draw_mesh(self.mesh_cube, base, view, proj, Material(diffuse=(0.25, 0.55,
0.65)))
        if len(snap["spheres"]) > 0:
            pivot_y = float(snap["spheres"][0]["pivot"][1])
        else:
            pivot_y = 3.5
        for x in [-3.3, 3.3]:
            for z in [-1, 1]:
                p0 = np.array([x, 0.05, z])
                p1 = np.array([x, pivot_y, z])
                model_post = self._cylinder_between(p0, p1, 0.1)
                self._draw_mesh(self.mesh_cylinder_cr, model_post, view, proj,
Material(diffuse=(0.18, 0.18, 0.2)))

        beam_radius = 0.1
        for z in [-1, 1]:
            p_left = np.array([-3.3, pivot_y, z])
            p_right = np.array([3.3, pivot_y, z])
            model_beam = self._cylinder_between(p_left, p_right, beam_radius)
            self._draw_mesh(self.mesh_cylinder_cr, model_beam, view, proj,
Material(diffuse=(0.26, 0.26, 0.28)))

        string_radius = 0.02
        for s in snap["spheres"]:
            pos = np.array(s["position"])
            pivot = np.array(s["pivot"])
            p_beam_front = np.array([pivot[0], pos[1], -1.0])
            p_beam_back = np.array([pivot[0], pos[1], 1.0])
            for p_start in [p_beam_front, p_beam_back]:
                model_str = self._cylinder_between(p_start, [pos[0], 3.3, pos[2]],
string_radius)
                self._draw_mesh(self.mesh_cylinder_cr, model_str, view, proj,
Material(diffuse=(0.06, 0.06, 0.06)))

        for s in snap["spheres"]:
```

```
            model = (Matrix44.from_translation(Vector3(s["position"]))
                   * Matrix44.from_scale(Vector3([s["radius"], s["radius"],
s["radius"]])))
            self._draw_mesh(self.mesh_sphere, model, view, proj,
Material(diffuse=colors[s["color_index"] % len(colors)], shininess=80.0))

    def _render_cloth(self, snap: dict, view: Matrix44, proj: Matrix44):
        w, h = snap["width"], snap["height"]
        verts_flat = update_cloth_normals(snap["vertices"], w, h)
        indices = []
        for j in range(h - 1):
            for i in range(w - 1):
                a = j * w + i
                b, c, d = a + 1, a + w, a + w + 1
                indices.extend([a, c, b, b, c, d])
        idx = np.array(indices, dtype=np.uint32)

        if self.cloth_mesh is None:
            self.cloth_mesh = Mesh(self.ctx, verts_flat, idx)
            self.cloth_mesh.bind(self.prog)
        else:
            self.cloth_mesh.update_vertices(verts_flat)

        anchor = snap["pole_anchor"]
        pole_h = float(anchor[1])
        pole_model = (Matrix44.from_translation(Vector3([anchor[0], pole_h * 0.5,
anchor[2]])) * Matrix44.from_scale(Vector3([0.06, pole_h, 0.06])))
        self._draw_mesh(self.mesh_cylinder_cl, pole_model, view, proj,
Material(diffuse=(0.45, 0.75, 0.4)))

        cloth_model = Matrix44.identity()
        self._draw_mesh(self.cloth_mesh, cloth_model, view, proj, Material(diffuse=(0.4,
0.2, 0.8), shininess=16.0))
```

ДОДАТОК В

Лістинг коду модуля world_manager.py

```
import numpy as np
import pyglet
from pyglet.window import key
from pyrr import Matrix44
import moderngl
from physics_engine import PhysicsEngine, SceneKind
from render_core import Renderer

class WorldManager:
    def __init__(self, window: Window, ctx):
        self.window = window
        self.ctx = ctx
        self.physics = PhysicsEngine(dt=PHYSICS_DT)
        self.renderer = Renderer(ctx, window.width, window.height)
        self.accumulator = 0.0
        self.snapshot = self.physics.get_render_snapshot()
        self.model_matrices: dict = {}

        self.camera_distance = 11.0
        self.camera_height = 3.2
        self.camera_angle = 0.0
        self.mouse_sensitivity = 0.005
        self.zoom_speed = 0.5
        self.min_camera_height = 0.5
        self.max_camera_height = 10.0
        self.min_camera_distance = 2.0
        self.max_camera_distance = 30.0
        self.current_scene = SceneKind.FLUID
        self._setup_scene_meshes()
        self.set_scene(self.current_scene)
        pyglet.clock.schedule(self._update_loop)

    def _setup_scene_meshes(self):
        self.model_matrices = {
            "container": Matrix44.identity(),
            "emitter": Matrix44.identity(),
            "obstacles": [], "spheres": [],
            "cloth": Matrix44.identity(),
            "pole": Matrix44.identity(),
        }

    def set_scene(self, kind: SceneKind):
        self.current_scene = kind
        self.physics.set_scene(kind)
        self.renderer.cloth_mesh = None
        self.snapshot = self.physics.get_render_snapshot()
        titles = {
            SceneKind.FLUID: "Fluid / Granular (Adams-Bashforth) | [2] | [3]",
```

```
            SceneKind.NEWTON_CRADLE: "[1] | Newton's Cradle (Verlet) | [3]",
            SceneKind.CLOTH: "[1] | [2] | Cloth Flag (Verlet)",
        }
        self.window.set_caption(f"Physics Simulation — {titles[kind]}")

    def _update_loop(self, dt: float):
        self.accumulator = min(self.accumulator + dt, PHYSICS_DT * MAX_PHYSICS)
        steps = 0
        while self.accumulator >= PHYSICS_DT and steps < MAX_PHYSICS_STEPS:
            self.physics.step(PHYSICS_DT)
            self._sync_transforms()
            self.accumulator -= PHYSICS_DT
            steps += 1
        self.snapshot = self.physics.get_render_snapshot()

    def _sync_transforms(self):
        snap = self.physics.get_render_snapshot()
        kind = snap["kind"]

        if kind == "fluid":
            b = snap["bounds"]
            center = (b.min_corner + b.max_corner) * 0.5
            extents = b.max_corner - b.min_corner
            self.model_matrices["container"] = (Matrix44.from_translation(center) *
Matrix44.from_scale(extents * 0.5))
            er = snap["emitter_radius"]
            self.model_matrices["emitter"] = Matrix44.from_translation(snap["emitter"]) *
Matrix44.from_scale([er, er, er])
        elif kind == "cradle":
            self.model_matrices["spheres"] = [
                Matrix44.from_translation(s["position"])
                * Matrix44.from_scale([s["radius"]] * 3)
                for s in snap["spheres"]
            ]
        elif kind == "cloth":
            self.model_matrices["cloth"] = Matrix44.identity()
            self.model_matrices["pole"] = Matrix44.from_translation(snap ["pole_anchor"])
* Matrix44.from_scale([0.06, 2.5, 0.06])

    def on_draw(self):
        self.window.clear()
        self.ctx.clear(0.12, 0.12, 0.14)
        eye_x = self.camera_distance * np.sin(self.camera_angle)
        eye_z = self.camera_distance * np.cos(self.camera_angle)
        eye = (eye_x, self.camera_height, eye_z)
        self.renderer.render_scene(self.snapshot, camera_eye=eye)

    def on_mouse_drag(self, x, y, dx, dy, buttons, modifiers):
        self.camera_angle -= dx * self.mouse_sensitivity
        self.camera_height -= dy * self.mouse_sensitivity * 2.0
        self.camera_height = float(np.clip(self.camera_height, self.min_camera_height,
self.max_camera_height))
```

```
    def on_mouse_scroll(self, x, y, scroll_x, scroll_y):
        self.camera_distance = float(np.clip(
            self.camera_distance - scroll_y * self.zoom_speed,
            self.min_camera_distance, self.max_camera_distance,
        ))

    def on_key_press(self, symbol, modifiers):
        sleep_time = pyglet.clock.get_sleep_time(self._update_loop)
        if symbol == key._1 or symbol == key.NUM_1:
            self.set_scene(SceneKind.FLUID)
        elif symbol == key._2 or symbol == key.NUM_2:
            self.set_scene(SceneKind.NEWTON_CRADLE)
        elif symbol == key._3 or symbol == key.NUM_3:
            self.set_scene(SceneKind.CLOTH)
        elif symbol == key.R or symbol == key._0 or symbol == key.NUM_0:
            self.physics = PhysicsEngine(dt=PHYSICS_DT)
            self.set_scene(self.current_scene)
            self.renderer.cloth_mesh = None
            self._setup_scene_meshes()
        elif symbol == key.SPACE or symbol == key.P:
            pyglet.clock.unschedule(self._update_loop) if not sleep_time else
pyglet.clock.schedule(self._update_loop)

PHYSICS_HZ = 60.0
PHYSICS_DT = 1.0 / PHYSICS_HZ
MAX_PHYSICS_STEPS = 8
window = Window(width=1280, height=720, resizable=True,
    caption="Physics Simulation — Fluid [1] | Cradle [2] | Flag [3]")
gl_ctx = moderngl.create_context()
manager = WorldManager(window, gl_ctx)

@window.event
def on_draw():
    manager.on_draw()
@window.event
def on_resize(width, height):
    manager.renderer.resize(width, height)
@window.event
def on_key_press(symbol, modifiers):
    manager.on_key_press(symbol, modifiers)

@window.event
def on_mouse_drag(x, y, dx, dy, buttons, modifiers):
    manager.on_mouse_drag(x, y, dx, dy, buttons, modifiers)
@window.event
def on_mouse_scroll(x, y, scroll_x, scroll_y):
    manager.on_mouse_scroll(x, y, scroll_x, scroll_y)

pyglet.app.run()
```