НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ
кафедра ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ

“До захисту допущено”
Завідувач кафедри ЦТЕ
_____ Наталія АУШЕВА

“___”_____2024 р.

Дипломна робота

на здобуття ступеня бакалавра
за освітньо-професійною програмою
“Цифрові технології в енергетиці”
зі спеціальності *122 “Комп’ютерні науки”*

на тему: **Паралельний метод розв’язання гравітаційної задачі N тіл**

Виконав:
студент IV курсу, групи ТР-02

ЮРИК Дмитро Миколайович _____
(прізвище, ім’я, по батькові) (підпис)

Керівник:
доцент, к.т.н., Володимир ЛАБЖИНСЬКИЙ _____
(посада, науковий ступінь, вчене звання, ім’я, ПРІЗВИЩЕ) (підпис)

Рецензент: _____
_____ _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище, ім’я, по батькові) (підпис)

Нормоконтролер: *асистент Антон ПАСІЧНЮК* _____
(посада, науковий ступінь, вчене звання, ім’я, ПРІЗВИЩЕ) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2024

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ

Кафедра ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ

Рівень вищої освіти – перший (бакалаврський)
спеціальність 122 “Комп’ютерні науки”
Освітньо-професійна програма “Цифрові технології в енергетиці”

ЗАТВЕРДЖУЮ
Завідувач кафедри ЦТЕ
_____ Наталія АУШЕВА_

«__»_____2024 р.

ЗАВДАННЯ
на дипломну роботу студенту

ЮРИКУ Дмитру Миколайовичу
(прізвище, ім’я, по батькові)

1. Тема роботи **Паралельний метод розв’язання гравітаційної задачі N тіл**

керівник роботи ***Лабжинський Володимир Анатолійович, к.т.н., доцент***
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «___»_____ 202__ р. №_____

2. Термін подання роботи студентом **07.06.2024 року**

3. Вихідні дані до роботи мова програмування C++, стандарт OpenMP та OpenGL, фреймворк OpenCL, бібліотека GLFW та Dear ImGui, середовище для складання проекту CMake, середовище розробки LazyVim

4. Перелік питань, які потрібно розробити
1) провести аналіз серед наявних аналогів
2) аргументувати вибрані інструменти, технології та алгоритми для реалізації програмного забезпечення
3) побудувати архітектуру програмного забезпечення

4) створити прикладний програмний застосунок_____
5) представити процес використання застосунку_____

5. Орієнтований перелік ілюстративного матеріалу діаграми, що демонструють роботу алгоритмів застосунку, архітектуру системи, взаємодію користувачів із системою, класи програмного забезпечення; приклади роботи з програмним продуктом._____

6. Дата видачі завдання «19» вересня 2023 р.

КАЛЕНДАРНИЙ ПЛАН

/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1.	Затвердження теми роботи		
2.	Вивчення та аналіз задачі	17-20.04.24	
3.	Розробка архітектури та загальної структури системи	21-25.04.24	
4.	Розробка частин окремих підсистем	25.04-02.05.24	
5.	Програмна реалізація системи	03-07.05.24	
6.	Оформлення пояснювальної записки	08-12.05.24	
7.	Захист програмного продукту	13-17.05.23	
8.	Передзахист	03-06.06.23	
9.	Подання готової роботи на кафедру	07.06.2024	
10.	Захист	17-21.06.2024	

Студент _____ **Дмитро ЮРИК**
(підпис) (ім'я, ПРІЗВИЩЕ)

Керівник роботи _____ **Володимир ЛАБЖИНСЬКИЙ**
(підпис) (ім'я, ПРІЗВИЩЕ)

АНОТАЦІЯ

Дипломна робота виконана на 67 сторінках, містить 11 ілюстрацій, 2 додатки, 21 джерел в переліку посилань.

Мета роботи – розроблення програмного забезпечення для розв’язання задачі N тіл з використанням паралельних методів обчислення.

Методи та засоби: прямий алгоритм розв’язання гравітаційної задачі N тіл, мова програмування C++, фреймворк OpenCl, стандарти OpenMP та OpenGL, середовище розробки LazyVim, засіб для збірки проєктів CMake.

Результати: програмне забезпечення, яке демонструє високу продуктивність та точність обчислень задачі N тіл.

Ключові слова: паралельні обчислення, OpenMP, OpenCL, CPU, GPU, C++.

ABSTRACT

The thesis is completed on 52 pages, contains 11 illustrations, 2 annex, and 21 sources in the list of references.

The aim of the work is to develop software for solving the N-body problem using parallel computing methods.

Methods and tools: direct algorithm for solving the gravitational N-body problem, C++ programming language, OpenCL framework, OpenMP and OpenGL standards, LazyVim development environment, CMake project building tool.

Results: The developed software demonstrates high performance and computational accuracy for the N-body problem.

Keywords: parallel computing, OpenMP, OpenCL, CPU, GPU, C++.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ ТА ТЕРМІНІВ

GPU (Graphics Processing Unit) – графічний процесор, призначений для обробки графічних даних, але також використовується для виконання загальних обчислень завдяки паралельній архітектурі.

CPU (Central Processing Unit) – центральний процесор, основний обчислювальний компонент комп'ютера.

API (application programming interface) – інтерфейс програмування застосунків.

OpenCL (Open Computing Language) – відкрита мова обчислень, призначена для створення паралельних програм, що виконуються як на графічному, так і на центральному процесорі.

OpenMP (Open Multi-Processing) – відкритий стандарт для розпаралелювання програм на основі потоків.

CUDA (Compute Unified Device Architecture) – програмно-апаратна архітектура паралельних обчислень, яка використовує обчислювальну продуктивність графічних процесорів фірми Nvidia.

MPI (Message Passing Interface) – інтерфейс передачі повідомлень між центральними процесорами.

OpenGL (Open Graphics Library) – відкрита графічна бібліотека.

GLFW (Graphics Library Framework) – бібліотека для створення вікон та обробки подій у графічних додатках.

ВСТУП

Розв’язання гравітаційної задачі N тіл є однією з головних проблем у сучасній астрофізиці. Її розв’язання є складним з огляду на квадратичну обчислювальну складність, оскільки кількість необхідних обчислень зростає квадратично зі збільшенням кількості тіл.

Актуальність теми зумовлена в необхідності ефективних методів обчислення імітаційних моделей систем із великою кількістю тіл. Сучасний стан обчислювальної техніки дозволяє використовувати паралельні обчислення для прискорення розв’язання таких задач. Використання технологій паралельних обчислень, таких як OpenMP і OpenCL, дає змогу значно зменшити час обробки та підвищити точність імітаційних моделей.

Метою роботи є розроблення програмного забезпечення для розв’язання задачі N тіл з використанням паралельних методів обчислення. Необхідно оптимізувати обчислювальний процес за допомогою сучасних технологій та алгоритмів, щоб забезпечити ефективне й точне моделювання гравітаційних взаємодій у системах з великою кількістю тіл.

Для досягнення поставленої мети необхідно виконати ряд завдань:

— провести аналіз підходів, методів та алгоритмів розв’язання гравітаційної задачі N тіл;

— виконати аналіз програмних засобів для реалізації програмної системи;

— розробити програмне забезпечення з використанням технологій паралельних обчислень;

— провести тестування розробленого програмного забезпечення.

1 АНАЛІЗ ТА ПОСТАНОВКА ЗАДАЧІ

Моделювання гравітаційної задачі N-тіл є складною задачею, що потребує значних обчислювальних ресурсів. Ця проблема виникає при спробі обрахунку гравітаційних сил, що діють між множиною небесних тіл, при великій їх кількості.

Завдяки впровадженню методів паралельних обчислень, стає можливим значно підвищити продуктивність та масштабованість обчислень гравітаційних взаємодій. Це досягається шляхом оптимізації алгоритмів та використання методів паралельної обробки, які дозволяють розподіляти обчислювальні завдання на декілька одночасних потоків виконання.

Метою цієї роботи є розробка застосунку для паралельного обчислення гравітаційної задачі N тіл, що дозволить покращити точність та ефективність обчислень.

Моделювання взаємодії N небесних тіл - це складна задача, що кидає виклик сучасним комп'ютерам. Її складність полягає в необхідності враховувати гравітаційні сили, що діють між кожною парою тіл у системі. Це призводить до квадратичної обчислювальної складності, що робить моделювання таких систем дедалі більш ресурсоємним зі зростанням N [1]. Ця проблема стає особливо гострою, коли мова йде про моделювання великих систем, таких як галактики або скупчення галактик. Традиційні методи моделювання стають занадто повільними та непрактичними, що обмежує наше розуміння цих складних космічних систем.

Саме тут на допомогу приходять паралельні методи обчислень. Завдяки розподілу обчислювальних завдань на декілька процесорів, ці методи можуть значно прискорити процес моделювання.

Для досягнення цієї мети необхідно виконати такі завдання:

— ознайомитись з технологіями паралельних обчислень та основними алгоритмами розв'язання задачі N тіл, такими як метод прямого обчислення сил, метод Барнса-Хата, а також швидкий метод мультиполів;

— реалізувати обраний алгоритм на мові програмування C++ з використанням таких технологій паралельних обчислень: OpenMP, OpenCL;

— виконати тестові запуски програми з різною кількістю тіл для визначення її продуктивності та правильності результатів;

— провести оптимізацію коду: вдосконалити алгоритм та код для досягнення максимальної ефективності.

Реалізація паралельного розв’язку для гравітаційної задачі N тіл в рамках дипломної роботи спрямована на поглиблення розуміння паралельних обчислень в моделюванні, сприяння дослідженням в обчислювальній астрофізиці та дозволяє проводити більш ефективне та масштабоване моделювання небесної динаміки.

2 МЕТОДИ ТА АЛГОРИТМИ РОЗВ'ЯЗАННЯ ГРАВІТАЦІЙНОЇ ЗАДАЧІ N ТІЛ

В цьому розділі буде детально розглянуто низку алгоритмів, які мають ключове значення в процесі створення програмного забезпечення для обчислення положень космічних тіл в просторі за допомогою використання паралельних обчислень.

Існує всього три алгоритми для розв’язання гравітаційної задачі: прямий алгоритм, алгоритм Барнса-Хата та швидкий метод мультиполів [2]. Прямий алгоритм, відомий також як метод грубої сили, полягає в обчисленні гравітаційних сил між кожною парою тіл у системі. Хоча цей метод є концептуально простим і точним, його обчислювальна вартість значно зростає зі збільшенням кількості тіл, що призводить до часової складності $O(n^2)$. Алгоритм Барнса-Хата і швидкий метод мультиполів є більш ефективними альтернативами прямому алгоритму.

Алгоритм Барнса-Хата використовує квадродерево для ієрархічного групування тіл за їх відносною близькістю, що дозволяє наближено обробляти віддалені взаємодії [2]. Цей метод зменшує кількість попарних обчислень, розглядаючи групи тіл як окремі об’єкти, завдяки чому, можна досягнути часової складності O(n log(n)). Однак реалізація алгоритму Барнса-Хата є досить важкою через складність керування структурою квадродерева.

Швидкий метод мультиполів використовує структуру октодерева для ефективної апроксимації взаємодії між віддаленими тілами. Швидкий метод мультиполів досягає економії обчислювальних ресурсів та має часову складність O(n), використовуючи мультипольні розширення для моделювання гравітаційного поля віддалених частинок [3]. Проте, через складність керування структурами октодерева та необхідність точного обчислення мультипольних розширень, реалізація швидкого методу мультиполів є більш складою, ніж реалізація алгоритму Барнса-Хата.

2.1 Прямий алгоритм

Прямий метод розв’язання проблеми N тіл полягає в обчисленні сил взаємодії між кількома небесними тілами на основі принципів ньютонівської гравітації. Цей метод є найбільш точним підходом до розв'язання цієї проблеми, оскільки враховує всі можливі взаємодії між тілами в системі. У цьому моделюванні кожне тіло перебуває під впливом сил тяжіння від будь-якого іншого тіла в системі, що впливає на його траєкторію та положення з плином часу [4].

Для застосування цього методу спочатку визначаються початкові умови системи, що включають в себе положення в просторі, швидкості та маси всіх N тіл. Ці початкові умови мають важливе значення, оскільки вони визначають початковий стан системи і, відповідно, впливають на подальшу еволюцію руху тіл. Наступним кроком є обчислення сил тяжіння, що діють між кожною парою тіл у системі. За допомогою цих сил, обчислюється прискорення, яких зазнають тіла, які, у свою чергу, впливають на їхні швидкості та положення під час руху в просторі. Розрахунок сил тяжіння між тілами виконується згідно з законом всесвітнього тяжіння Ньютона, який стверджує, що сила тяжіння між двома тілами пропорційна добутку їх мас і обернено пропорційна квадрату відстані між ними [5].

Ітеративно оновлюючи положення та швидкості тіл за невеликі кроки в часі, можна моделювати еволюцію системи та спостерігати, як гравітаційна взаємодія між тілами впливає на їхній рух. Це моделювання дозволяє нам вивчати складні явища, такі як орбіти планет, гравітаційні захоплення та взаємодія між небесними об’єктами. Наприклад, можна спостерігати, як планети обертаються навколо зірок, як комети проходять через Сонячну систему або як галактики зіштовхуються та зливаються.

Для двох тіл загальний вигляд обчислення сили можна записати як:

$$\hat{F_1} = -G\frac{m_1 m_2}{\left|\hat{r_1} - \hat{r_2}\right|^3}\left(\hat{r_1} - \hat{r_2}\right) \tag{2.1}$$

де $\widehat{F}$ - сила, що діє на тіло 1 з боку тіла 2,

G - гравітаційна константа,

m_1 та m_2 - маси тіл 1 та 2,

$\hat{r}_1$ та $\hat{r}_2$ - вектори, що вказують напрямок руху тіл 1 та 2.

Для обчислення прискорення частинки під дією гравітаційного поля іншої частинки використаємо другий закон Ньютона:

$$\hat{a} = \frac{\hat{F}}{m} \tag{2.2}$$

де $\hat{a}$ - прискорення.

Підставивши формулу 2.1 в формулу 2.2 отримаємо наступне:

$$\hat{a_1} = -G\frac{m_2}{\left|\hat{r_1} - \hat{r_2}\right|^3}\left(\hat{r_1} - \hat{r_2}\right)$$

У системі з N тіл на кожне з тіл діятиме сила тяжіння від інших, що виражається рівнянням:

$$\hat{a_n} = -G\sum_{k\neq n}\frac{m_k}{\left|\hat{r_n} - \hat{r_k}\right|^3}\left(\hat{r_n} - \hat{r_k}\right)$$

Для обчислення переміщення одного тіла в просторі треба розрахувати сили (формула 2.1), з якими кожне інше тіло діє на вказане. В такому випадку часова складність алгоритму буде $O(n^2)$. Для спрощення складності використовують алгоритм Барнса-Хата та швидкий метод мультиполів, що дозволяють отримати часову складність $O(n \cdot log(n))$ та $O(n)$. Іншим засобом підвищити ефективність і масштабованість імітаційної моделі є використання методів паралельних обчислень. Використання технологій, таких як OpenCL або OpenMP, дозволяє розподілити обчислювальне навантаження між кількома ядрами або процесорами, забезпечуючи більш швидкі обчислення та можливість обробки більшої кількості тіл. Це особливо важливо для імітаційних моделей з великою кількістю тіл, де обчислювальна складність зростає квадратично з кількістю тіл. Завдяки паралельним обчисленням можна значно скоротити час обчислення імітаційної моделі, що робить можливим дослідження більш складних і великих систем.

2.2 Алгоритм Барнса-Хата

Алгоритм Барнса-Хата — це один з найефективніших методів, який використовується для апроксимації сил тяжіння між декількома тілами [6]. Він дозволяє значно зменшити обчислювальну складність проблеми, знижуючи її з $O(N^2)$ до $O(N \cdot logN)$, що робить його особливо корисним для систем з великою кількістю тіл.

В основі алгоритму Барнса-Хата лежить ієрархічний підхід, реалізований через структуру даних квадродерева. Спочатку всі тіла розміщуються в кореневому вузлі квадродерева, яке представляє всю область імітаційної моделі. Далі це дерево рекурсивно ділиться на квадранти, створюючи піддерева для кожного регіону. Цей процес поділу продовжується до тих пір, поки кожен квадрант не міститиме або одне тіло, або не стане достатньо малим.

Кожен внутрішній вузол квадродерева зберігає інформацію про положення центру мас і загальну масу всіх тіл у межах свого регіону. Це дозволяє значно зменшити обсяг обчислень, оскільки гравітаційні впливи від віддалених тіл можна апроксимувати одним внутрішнім вузлом, зменшивши кількість окремих обчислень для кожного тіла. Такий підхід не тільки економить час, але й зберігає достатню точність обчислень, особливо для віддалених взаємодій.

Шляхом обходу квадродерева та використання багатополюсних розширень алгоритм обчислює сили тяжіння на окремих тілах. Це означає, що під час обходу дерева для кожного тіла визначаються сили тяжіння, що діють на нього з боку інших тіл або їх груп, представлених вузлами дерева [7]. Алгоритм збалансовує обчислювальну ефективність і точність, встановлюючи поріг для апроксимації сили, який контролює, коли використання центру мас в обчисленнях є достатньо точним.

Для ще більшого підвищення продуктивності алгоритму Барнса-Хата можна використати технології паралельних обчислень. Завдяки паралельному обчисленню, в якому різні частини квадродерева обробляються одночасно на різних процесорах або ядрах, можна значно прискорити обчислення, що

включають велику кількість тіл. Це робить алгоритм Барнса-Хата хорошим вибором для сучасних високопродуктивних обчислювальних систем.

Алгоритм Барнса-Хата пропонує більш ефективний спосіб розрахунку гравітаційних взаємодій у складних системах із N тіл порівняно з прямим методом. Він забезпечує високу обчислювальну ефективність і дозволяє досліджувати великі та складні системи, зберігаючи при цьому достатню точність результатів. Завдяки цьому алгоритму стає можливим моделювати еволюцію галактик, взаємодію зірок у скупченнях та інші складні гравітаційні процеси з високою точністю та в продуктивністю.

2.3 Швидкий метод мультиполів

Швидкий метод мультиполів є ще одним ефективним методом, який використовується для апроксимації гравітаційних взаємодій у задачі гравітації N тіл. Цей метод був розроблений для подолання обмежень традиційних методів, які стають неефективними при великій кількості тіл. Швидкий метод мультиполів працює шляхом поділу простору на комірки через ієрархічну структуру октодерева, що дозволяє ефективно організувати дані та зменшити кількість обчислень [8].

Подібно до алгоритму Барнса-Хата, швидкий метод мультиполів апроксимує взаємодію між віддаленими тілами за допомогою мультипольних розкладів. Це дозволяє значно заощадити обчислювальні ресурси за рахунок зменшення кількості обчислень необхідних прямих попарних взаємодій [9].

Швидкий метод мультиполів починається з поділу простору на комірки, створюючи структуру октодерева, де кожен вузол дерева представляє певну частину простору. Далі, для кожного вузла обчислюються мультипольні коефіцієнти, що представляють сукупний вплив всіх частинок у цій області. Шляхом рекурсивного поділу структури октодерева та обчислення мультипольних

розширень на різних рівнях, швидкий метод мультиполів може ефективно обчислювати гравітаційні сили в окремих регіонах системи [10].

Ключовою перевагою швидкого мультипольного методу є те, що він встановлює баланс між точністю та обчислювальною ефективністю. Встановлюючи певний поріг для апроксимації, метод дозволяє контролювати рівень точності, який є прийнятним для конкретної задачі. Це робить його потужним інструментом для моделювання великомасштабних задач гравітації N тіл, де необхідно обробляти величезну кількість частинок із прийнятною точністю.

Паралельні обчислення можуть бути використані для подальшого підвищення продуктивності швидкого мультипольного алгоритму. Завдяки розподілу обчислювального навантаження між кількома процесорами або ядрами, можна значно прискорити процес обробки великої кількості частинок. Це особливо корисно для сучасних високопродуктивних обчислювальних систем, які можуть виконувати величезну кількість операцій одночасно.

Загалом, швидкий мультипольний алгоритм пропонує складний підхід до обробки гравітаційних взаємодій у системах із N тіл з покращеною ефективністю порівняно з прямими методами. Він дозволяє досліджувати еволюцію великих гравітаційних систем, таких як галактики, зоряні скупчення та інші космічні об’єкти, з високою точністю та в реалістичні часові рамки. Це робить його незамінним інструментом для астрономів і фізиків, які займаються дослідженням динаміки космічних об’єктів та їх взаємодій.

2.4 Аналіз існуючих програмних рішень

Існує кілька програмних рішень, які реалізують різні аспекти гравітаційної задачі N тіл, забезпечуючи різний рівень продуктивності та функціональності. Окремо з них можна виділити такі рішення як ChaNGa, NBody та GADGET.

2.4.1 Програмне забезпечення ChaNGa

ChaNGa (Charm++ N-body Gravity) – це програмне забезпечення, спеціально розроблене для проведення моделювання проблеми N тіл без зіткнень. Вона забезпечує широкі можливості для обчислення космологічних імітаційних моделей з періодичними граничними умовами в рухомих координатах або для моделювання окремих зоряних систем [11]. Завдяки своїй універсальності, ChaNGa також підтримує розрахунок гідродинаміки за допомогою методу згладжених частинок, що дозволяє враховувати взаємодію між газом і частинками в імітаційних моделях.

Для розрахунку сили тяжіння ChaNGa використовує дерево Барнеса-Хата з гексадекапольним розширенням вузлів, що дозволяє значно підвищити точність обчислень взаємодій між тілами. Крім того, для врахування періодичних сил використовується сумування Евальда, що забезпечує коректне моделювання в умовах періодичних меж. Інтегрування за часом здійснюється за допомогою стрибкового інтегратора, який дозволяє кожній частинці використовувати індивідуальні кроки часу, що забезпечує високу точність і ефективність.

Унікальною особливістю ChaNGa є використання схеми динамічного балансування навантаження в середовищі виконання Charm++, що дозволяє досягти високої продуктивності на масово паралельних системах. Charm++ забезпечує автоматичний розподіл обчислювальних задач між доступними ресурсами, що дозволяє ефективно використовувати обчислювальні потужності та забезпечує масштабованість програми навіть при дуже великій кількості тіл у імітаційній моделі.

Переваги ChaNGa: висока продуктивність та масштабованість завдяки використанню Charm++. Це дозволяє моделювати складні системи з різними фізичними ефектами, в тому числі гідродинамічні системи. Завдяки своїм можливостям, ChaNGa є відмінним інструментом для проведення досліджень в області космології, астрономії та фізики.

Однак, використання ChaNGa має і свої недоліки. Одним з головних є дуже висока складність налаштування та використання програми. Для ефективного

використання ChaNGa необхідні глибокі знання як самої програми, так і принципів паралельних обчислень та моделей гідродинаміки. Крім того, ChaNGa потребує значних обчислювальних ресурсів, що може бути викликом для дослідників з обмеженими ресурсами. Іншим недоліком є те, що проект має закритий код, що обмежує можливості для модифікації та налаштування програми під специфічні потреби користувачів.

2.4.2 Бібліотека NBody

NBody – це бібліотека, спеціально розроблена для моделювання гравітаційної взаємодії між великою кількістю тіл з використанням графічних процесорів. Вона використовує технології CUDA або OpenCL для забезпечення високої продуктивності [12].

Використання GPU дозволяє NBody досягати значної швидкості обчислень, оскільки графічні процесори мають велику кількість обчислювальних ядер, що дозволяє паралельно виконувати велику кількість обчислень.

Однією з переваг NBody є простота інтеграції у великі проекти. Бібліотека надає зручний інтерфейс для взаємодії з графічними процесорами, що дозволяє швидко і легко включити її у власні проекти і використовувати для моделювання гравітаційних систем.

Недоліком використання NBody є необхідність наявності потужного GPU. Для досягнення високої продуктивності, в більшості випадків, необхідно мати достатньо потужний графічний процесор, що може бути обмеженням для деяких користувачів. Крім того, бібліотека має обмеження у масштабованості для дуже великої кількості тіл.

2.4.3 Програмне забезпечення GADGET

GADGET – це програмне забезпечення для обчислення космологічних імітаційних моделей, яке відоме своєю широкою популярністю в астрофізиці [13]. Воно дозволяє досліджувати еволюцію масштабних структур Всесвіту, враховуючи гравітаційну взаємодію, гідродинамічні процеси та інші фізичні

явища. GADGET може працювати у паралельному режимі обчислень, використовуючи MPI (Message Passing Interface), що дозволяє ефективно масштабувати імітаційну модель на суперкомп'ютерах.

Однією з ключових переваг GADGET є його масштабованість. Програма може обробляти великі обсяги даних та обчислювати складні фізичні процеси, може моделювати формування галактик і обраховувати взаємодії між ними, що робить її хорошим інструментом для дослідження космічних структур.

Незважаючи на ці переваги, використання GADGET може бути складним. Налаштування програми та використання її вимагають значних знань та досвіду. Крім того, програма вимагає потужне апаратне забезпечення, особливо для обчислення великих моделей, що може бути обмеженням для деяких дослідників. Для коректної конфігурації імітаційної моделі у GADGET необхідне глибоке розуміння фізичних процесів, що може бути викликом для користувачів з обмеженим досвідом у цій області.

3 ЗАСОБИ РОЗРОБЛЕННЯ ПРОГРАМНОГО ПРОДУКТУ

Використання відповідних програмних засобів є невід'ємною частиною успішної реалізації програмного забезпечення та дозволяє розробникам працювати ефективно з комп’ютерами, забезпечуючи високу продуктивність та надійність системи. Це також сприяє більш швидкому виявленню та виправленню помилок, оптимізації коду та покращенню загальної якості продукту. Завдяки сучасним інструментам розробки можлива інтеграція новітніх технологій, що значно підвищує конкурентоспроможність програмного забезпечення.

Першим засобом розробки, який буде розглянуто, є OpenMP. Ця бібліотека є потужним інструментом для паралельного програмування на мові C/C++. Вона надає широкі можливості для розподілу обчислень на декілька процесорів, що значно скорочує час розрахунку великих імітаційних моделей.

Наступним важливим компонентом є OpenCL - це відкритий стандарт для гетерогенних обчислень, який дозволяє використовувати різні обчислювальні ресурси, такі як CPU та GPU, для паралельних обчислень. Це дає можливість значно підвищити продуктивність програмного забезпечення, особливо на комп'ютерах з потужними відеокартами.

Фреймворк OpenGL, є бібліотекою для графічного програмування. Він використовується для візуалізації результатів розрахунку імітаційної моделі. OpenGL надає широкі можливості для створення 3D-графіки та анімації, що дозволяє наочно продемонструвати рух тіл під дією сил гравітаційного тяжіння.

Бібліотека GLFW, використовується для створення віконних інтерфейсів. Вона надає зручні та потужні засоби для створення і використання вікон.

Бібліотека ImGui, використовується для створення декларативних графічних інтерфейсів. Вона надає простий та зручний спосіб для створення панелей інструментів, графіків та інших елементів інтерфейсу, які дозволяють користувачеві налаштувати параметри розрахунку та візуалізувати результати.

3.1 Мова програмування C++

Одним з основних використаних засобів розробки, є мова програмування C++. C++ — це мова високого рівня, яка надає користувачеві можливості для низькорівневого програмування, що дозволяє максимально ефективно використовувати апаратні ресурси [14]. Вона забезпечує високу продуктивність та гнучкий контроль над пам'яттю комп’ютера, що є критичним для виконання складних обчислень.

Підтримка об'єктно-орієнтованого програмування (ООП) в C++ дозволяє створювати модульні, керовані та масштабовані кодові бази. ООП сприяє написанню зрозумілого та компактного коду, а також спрощує його подальше використання та обслуговування.

C++ надає розробникам глибокий контроль над пам'яттю, дозволяючи оптимізувати її використання. Це робить C++ важливим інструментом для розробки систем, які повинні ефективно використовувати обмежені ресурси пам'яті.

Можливість роботи з багатьма потоками у C++ дозволяє розпаралелювати виконання коду на багатоядерних процесорах. Це значно підвищує продуктивність програм, особливо тих, які обробляють великі обсяги даних.

Для C++ створено велику кількість сторонніх бібліотек, які охоплюють різноманітні задачі: від паралельного обчислення та роботи з мережами до обробки зображень та машинного навчання. Це дозволяє розробникам ефективно використовувати готові рішення та зосередитися на ключових аспектах своїх проектів.

3.2 Стандарт OpenMP

OpenMP (від англ. Open Multi-Processing) - це API, для паралельного програмування з розділеною пам'яттю на мовах C, C++ та Fortran [15]. Однією з

основних переваг OpenMP є його здатність ефективно підтримувати паралельне програмування на різних архітектурах. Він надає набір директив компілятора, бібліотечних функцій та змінних середовища, які дозволяють реалізувати паралельні обчислення простим способом.

За допомогою директив OpenMP розробники можуть визначити паралельні ділянки свого коду, де одночасно можуть виконуватися декілька потоків. Цей підхід дозволяє покращити продуктивність за рахунок розподілу навантаження між декількома обчислювальними блоками в межах одного вузла. Крім того, OpenMP пропонує конструкції для керування обміном даними та синхронізації між потоками, що спрощує реалізацію паралельних алгоритмів.

OpenMP підтримується широким спектром компіляторів та платформ, що робить його універсальним вибором для паралельного програмування в різних системах. Більше того програма написана з використанням OpenCL запускається навіть на процесорі, який не має підтримки багатопоточності. Простота використання та переносимість зробили його популярним у різних сферах, включаючи наукові обчислення та аналіз даних.

OpenMP надає набір директив, які є будівельними блоками для розпаралелювання коду та керування паралельним виконанням. Ці директиви пропонують структурований підхід до визначення паралельних областей, розподілу роботи між потоками та синхронізації активності потоків. Інтегруючи ці директиви у вихідний код, розробники можуть ефективно використовувати потужність багатоядерних процесорів.

Одна з основних директив OpenMP - #pragma omp parallel, яка визначає блок коду, що повинен виконуватися паралельно кількома потоками. У межах паралельної області, визначеної цією директивою, кожен потік має спільний доступ до одного адресного простору [16]. Розробники можуть керувати кількістю потоків, створених у паралельній області, за допомогою директиви OMP_NUM_THREADS, вказавши бажану кількість потоків для оптимальної продуктивності.

Інша часто використовувана директива - #pragma omp for, яка дозволяє виконувати ітерації циклу різними потоками. Додаючи цю директиву перед конструкцією циклу, розробники можуть розподілити ітерації циклу між потоками, що дозволяє ефективно розподілити навантаження та підвищити швидкість виконання циклу. Поняття schedule, пов'язане з директивою #pragma omp for, дозволяє розробникам визначати, як ітерації циклу слід розділити між потоками, пропонуючи гнучкість у стратегіях балансування навантаження.

Додатково OpenMP надає директиви синхронізації, такі як #pragma omp barrier, яка забезпечує точку синхронізації для потоків у межах паралельної області. Вставляючи директиву бар'єру, розробники можуть гарантувати, що всі потоки досягнуть певної точки в коді, перш ніж рухатися далі, запобігаючи змаганням потоків та підтримуючи цілісність даних. Ця директива особливо корисна для координації дій кількох потоків та для запобігання потенційних конфліктів у доступі до спільних даних. Також, вона допомагає уникнути небажаних станів гонки та забезпечує більш стабільну роботу багатопоточних програм. Застосування #pragma omp barrier значно спрощує управління складними паралельними обчисленнями, роблячи код більш зрозумілим та надійним.

Більше того, такі директиви, як #pragma omp critical та #pragma omp atomic, пропонують механізми для контролю доступу до спільних ресурсів у паралельних областях [16]. Директива critical позначає блок коду як критичну секцію, дозволяючи виконувати її лише одному потоку одночасно, щоб запобігти змаганню за дані. Директива atomic гарантує, що певні операції виконуються атомарно, забезпечуючи узгоджені оновлення спільних змінних без ризику переплетених модифікацій кількома потоками.

Отже, OpenMP є цінним інструментом для використання паралельних обчислень в системах із розділеною пам'яттю, пропонуючи зручний для користувача підхід до розробки паралельних програм. Широка підтримка, переваги в продуктивності та легкість впровадження роблять його привабливим вибором для розробників, які прагнуть оптимізувати свій код для багатоядерних архітектур.

3.3 Фреймворк OpenCL

OpenCL (Open Computing Language) є фреймворком для паралельного програмування на різних апаратних платформах, включаючи центральні процесори, графічні процесори та інші акселератори [17]. Розроблений Khronos Group, OpenCL надає стандартизований API для виконання обчислювальних задач паралельно, використовуючи обчислювальну потужність різноманітних пристроїв для досягнення високопродуктивних обчислювальних рішень. Завдяки своїй незалежності від платформи, OpenCL пропонує розробникам гнучкість для роботи з широким спектром апаратних архітектур, що дозволяє ефективно використовувати доступні ресурси в середовищах з різними обчислювальними можливостями.

Ключовою особливістю OpenCL є його здатність визначати ядра (kernels) - функції, які виконуються паралельно на декількох обчислювальних блоках. Створюючи код ядра на OpenCL C, розробники можуть перевантажувати обчислювально складні завдання на паралельні блоки виконання, підвищуючи продуктивність та масштабованість. Ядра в OpenCL призначені для того, щоб бути переносимими та адаптивними, дозволяючи їм ефективно працювати на різних пристроях без значних модифікацій.

OpenCL вводить концепцію обчислювальних блоків (compute units), робочих груп (work-groups) та робочих елементів (work-items) для полегшення паралельних обчислень. Обчислювальні блоки представляють окремі обчислювальні елементи, такі як ядра центрального процесора або шейдерні ядра графічного процесора, тоді як робочі групи об'єднують декілька робочих елементів разом для скоординованого виконання. Робочі елементи - це окремі одиниці виконання, які виконують певні завдання в межах робочої групи, що дозволяє досягти паралельності та ефективного використання ресурсів.

OpenCL пропонує різні типи пам'яті: глобальну пам'ять, локальну пам'ять та приватну пам'ять, кожна з яких служить певним цілям в контексті паралельного виконання. Розробники можуть ретельно керувати виділенням пам'яті та

передачею даних, щоб мінімізувати затримки та максимізувати пропускну здатність в OpenCL-додатках.

OpenCL підтримує паралельність даних, паралельність задач та конвеєрну паралельність, пропонуючи розробникам декілька підходів до ефективної паралелізації їхніх алгоритмів [18]. Використовуючи ці парадигми паралельного програмування, розробники можуть використовувати можливості паралельної обробки для різних пристроїв, досягаючи значного приросту продуктивності в широкому спектрі обчислювальних задач. Підтримка паралельності OpenCL робить його універсальним та потужним інструментом для прискорення роботи додатків на різних апаратних платформах.

Переходячи до інструментів OpenCL, розробники мають доступ до широкого спектра пакетів розробки програмного забезпечення та інтегрованих середовищ розробки, які полегшують програмування та оптимізацію OpenCL. Ці інструменти надають такі функції, як редагування коду, відладка, профілювання та аналіз продуктивності, допомагаючи розробникам оптимізувати процес розробки та підвищувати ефективність своїх OpenCL-додатків.

3.4 Стандарт OpenGL

OpenGL (Open graphic library) - це потужний міжплатформовий API для візуалізації 2D та 3D графіки на різноманітних апаратних платформах, включаючи настільні комп'ютери, мобільні пристрої та вбудовані системи [19]. Розроблений Khronos Group, OpenGL надає стандартизований інтерфейс для взаємодії з графічним апаратним забезпеченням, дозволяючи розробникам створювати візуально вражаючі програми та ігри з високою продуктивністю та переносимістю.

Одна з ключових особливостей OpenGL - це його гнучкість у підтримці різних типів конвеєрів візуалізації, включаючи негайний режим та режим із збереженням стану. Негайний режим візуалізації дозволяє розробникам надсилати

команди для візуалізації безпосередньо на графічний процесор для негайного виконання, тоді як режим із збереженням стану дозволяє створювати та керувати складними графічними сценами для більш ефективної візуалізації.

OpenGL використовує модель машинних станів для керування станами візуалізації, дозволяючи розробникам встановлювати різні параметри, такі як трансформації, освітлення та матеріали, щоб контролювати зовнішній вигляд візуалізованих об'єктів [20].

OpenGL підтримує сучасні методи візуалізації графіки, наприклад такі як програмування шейдерів, що дозволяє розробникам писати власні програми шейдерів, які виконуються на GPU для виконання складних завдань візуалізації. Програми шейдерів, написані на GLSL (мова шейдерів OpenGL), можуть реалізувати складні моделі освітлення, текстурування та ефекти постобробки, покращуючи реалістичність та візуальну якість сцен.

OpenGL підтримує об'єкти буферів вершин, текстури та буфери кадрів, дозволяючи розробникам ефективно керувати та маніпулювати даними на GPU. Об'єкти буферів забезпечують швидку передачу даних між CPU та GPU, тоді як текстури надають універсальний механізм застосування зображень та візерунків до 3D-об'єктів. Буфери кадрів дозволяють розробникам візуалізувати сцени поза екраном та виконувати складні методи візуалізації, такі як відображення тіней та ефекти постобробки.

3.5 Засіб Dear ImGui

Dear ImGui (immediate mode graphic user interface) - це легка та універсальна бібліотека GUI, призначена для створення інтерфейсів користувача в програмах та іграх реального часу. Розроблена Омаром Корню, Dear ImGui дотримується парадигми негайного режиму, де елементи GUI створюються та відображаються щоразу при оновленні кадру, що полегшує проєктування та динамічне оновлення інтерфейсів на основі стану програми та дій користувача .

Одна з ключових особливостей Dear ImGui - це її простота та легкість інтеграції, що дозволяє розробникам швидко додавати інтерфейси користувача до своїх програм із мінімальними витратами. Dear ImGui пропонує простий API для створення елементів GUI, таких як вікна, кнопки, повзунки та текстові поля, роблячи його ідеальним для прототипування дизайнів інтерфейсів.

Розробники можуть налаштувати зовнішній вигляд інтерфейсів Dear ImGui за допомогою різноманітних варіантів стилів та тем, що дозволяє їм створювати візуально привабливі та зручні інтерфейси, які відповідають естетиці їхніх програм. Dear ImGui підтримує власні віджети та розширення, дозволяючи розробникам розширювати функціональність бібліотеки додатковими елементами GUI, адаптованими до їхніх конкретних потреб.

Іншим ключовим аспектом Dear ImGui є оптимізація продуктивності, яка забезпечує ефективне відображення елементів GUI з мінімальним впливом на продуктивність програми. Dear ImGui використовує графічний процесор для прискореного рендерингу, що робить його придатним для створення складних інтерфейсів користувача з плавною анімацією та чутливими взаємодіями. Бібліотека також пропонує функції для оптимізації відображення GUI, такі як пакетна візуалізація та відсікання, щоб мінімізувати навантаження та максимізувати частоту кадрів.

Універсальність Dear ImGui поширюється і на підтримку платформ, оскільки бібліотека сумісна з широким спектром графічних API, включаючи OpenGL, DirectX, Vulkan та Metal. Розробники можуть легко інтегрувати Dear ImGui у свої програми незалежно від базової графічної технології, що дозволяє розробляти GUI з підтримкою різних платформ із послідовною поведінкою та продуктивністю.

Інструменти Dear ImGui доповнюють функціональність бібліотеки, надаючи додаткові можливості для дизайну, відладки та профілювання GUI. Такі інструменти, як ImGui Demo та ImGui Profiler, пропонують аналіз продуктивності та поведінки GUI, допомагаючи розробникам оптимізувати свої інтерфейси для чутливості та ефективності. ImGui Docking забезпечує гнучку систему стикування

вікон, дозволяючи користувачам налаштовувати розташування елементів GUI відповідно до своїх робочих уподобань.

Розширюваність Dear ImGui за допомогою плагінів та внесків спільноти додатково посилює його можливості. Розробники можуть використовувати широкий спектр додатків та розширень для покращення функціональності GUI та інтеграції з сторонніми бібліотеками та інструментами. Теми та стилі ImGui, створені спільнотою, пропонують різноманітні варіанти дизайну для налаштування зовнішнього вигляду елементів GUI, дозволяючи розробникам створювати унікальні та візуально привабливі інтерфейси у своїх програмах.

Підсумовуючи, Dear ImGui - це потужна та універсальна бібліотека GUI, яка спрощує створення інтерфейсів користувача в програмах та іграх. Завдяки підходу негайного режиму, оптимізації продуктивності, сумісності з різними платформами та можливості розширення, Dear ImGui дозволяє розробникам створювати інтерактивні та захоплюючі інтерфейси, які покращують враження користувачів та оптимізують робочі процеси програми.

3.6 Бібліотека GLFW

GLFW (Graphics Library Framework) - це легка та портативна бібліотека для керування вікнами, контекстами та введенням даних у програмах з OpenGL. Розроблена як альтернатива залежним від платформи API для вікон, GLFW спрощує процес створення та керування вікнами в різних операційних системах, дозволяючи розробникам зосередитися на програмуванні графіки [21].

За допомогою GLFW розробники можуть створювати вікна, обробляти введення користувача та керувати контекстами OpenGL за допомогою простого та універсального API. Бібліотека абстрагує залежні від платформи деталі, дозволяючи розробникам написати код, який без проблем працює в системах Windows, macOS та Linux без необхідності значних модифікацій.

Одна з ключових особливостей GLFW - це підтримка декількох вікон та контекстів, що дозволяє розробникам легко створювати складні багатовіконні програми. GLFW пропонує функції для керування вікнами та обробки подій, дозволяючи розробникам реалізовувати інтерактивні та чутливі інтерфейси користувача у своїх програмах.

На додаток до керування вікнами, GLFW пропонує підтримку роботи з пристроями вводу, такими як клавіатури, миші та геймпади. Бібліотека надає функції для опитування стану вводу, реєстрації зворотних викликів вводу та обробки подій вводу, що полегшує реалізацію взаємодії користувача в програмах OpenGL.

GLFW підтримує сучасні функції OpenGL, такі як підказки щодо створення контексту чи підтримку декількох моніторів, дозволяючи розробникам використовувати передові графічні можливості у своїх програмах. Бібліотека також пропонує утиліти для керування розширеннями OpenGL, що дозволяє розробникам отримувати доступ до найновіших графічних функцій на різних платформах.

Простота та зручність використання GLFW роблять її популярним вибором для розробників, які працюють над графічними програмами та комп’ютерними іграми. Простий API бібліотеки та чітка документація спрощують процес розробки, дозволяючи розробникам швидко налаштовувати функціональність вікон та вводу у своїх проектах.

Незалежність від платформи та сумісність GLFW сприяють її універсальності, оскільки бібліотека підтримує широкий спектр операційних систем та графічних API. Розробляючи програми для настільних комп'ютерів, мобільних пристроїв чи вбудованих платформ, розробники можуть покладатися на GLFW для забезпечення узгоджених можливостей вікон та вводу в різних середовищах.

Орієнтованість бібліотеки на продуктивність та ефективність гарантує, що програми GLFW будуть працювати плавно та швидко на різних апаратних конфігураціях. Використовуючи оптимізовані процедури обробки вікон та

введення даних GLFW, розробники можуть створювати високопродуктивні графічні програми з мінімальними витратами.

Інструменти та утиліти GLFW розширюють функціональність бібліотеки додатковими можливостями для керування вікнами, обробки вводу та обробки подій. API GLFW дозволяє розробникам опитувати та керувати пристроями відображення.

Підтримка спільноти та активна розробка GLFW гарантують, що бібліотека залишається сумісною з найновішими графічними технологіями та галузевими стандартами. Розробники можуть отримати доступ до безлічі ресурсів, навчальних матеріалів та прикладів, які допоможуть їм використовувати можливості GLFW та інтегрувати функціональність вікон та вводу у свої проекти.

4 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

У цьому розділі подано детальний опис архітектури та програмної реалізації системи, що є важливим етапом розробки будь-якого застосунку. Відповідний опис дозволяє краще зрозуміти, як саме функціонує система, які компоненти вона містить та як ці компоненти взаємодіють між собою. Програма складається з трьох незалежних модулів: модуля імітаційної моделі, графічного модуля та керуючого модуля.

Модуль імітаційної системи відповідає за розрахунки та моделювання фізичних процесів, які відбуваються у системі. У цьому модулі реалізовані прямі та паралельні обчислення для розв'язання гравітаційної задачі.

Графічний модуль відповідає за візуалізацію результатів моделювання. Він включає в себе різноманітні графічні ефекти, які допомагають краще зрозуміти поведінку імітаційної моделі.

Керуючий модуль відповідає за взаємодію користувача з програмою. Він ініціалізує всі необхідні бібліотеки та керує іншими модулями.

Загалом, ці три модулі тісно взаємодіють між собою, утворюючи цілісну систему, яка забезпечує ефективне моделювання та візуалізацію фізичних процесів.

4.1 Архітектура застосунку

Архітектура системи умовно ділиться на 3 незалежні модулі: модуль імітаційної моделі (Simulation), модуль візуалізації (Renderer) та модуль керування програмою (Application). Такий підхід дозволяє нам в майбутньому вільно додавати нові реалізації модулів для графічного відображення та математичних обчислень не змінюючи старий код та використовувати нові технології і засоби, які не передбачені на цьому етапі розробки.

Модуль Application дозволяє керувати іншими модулями програми без сильного зв’язування. В цьому модулі знаходиться точка входу програми, в якій відбувається ініціалізація всіх інших модулів програми. Та головним завданням цього модулю є керування шарами та подіями. Він містить в собі головний цикл застосунку, з якого почергово викликаються функції OnEvent, OnUpdate, OnImGuiRender для кожного шару програми.

Модуль Renderer надає нам можливість будувати графічне 3D зображення. Він складається з таких інтерфейсів: VertexBuffer, IndexBuffer, VertexArray, Texture, Shader, RendererAPI та двох класів: Renderer, RendererCommand. Для побудови графічного зображення нам потрібно реалізувати всі інтерфейси та обгорнути функції конкретної графічної бібліотеки в методи надані інтерфейсами. На даний момент реалізовано обгортку тільки для OpenGl.

4.1.1 Система шарів

Шар (Layer) - це абстрактний клас, який дозволяє нам добитися ще більшої гнучкості програми та відокремити різні елементи одне від одного, наприклад такі як графічний інтерфейс і графічне відображення результату імітаційної моделі. Це важливо для підтримки принципу розділення відповідальностей та створення масштабованих програмних рішень.

Використання абстрактного класу шара дозволяє легко додавати нові функціональні можливості до програми, не змінюючи вже існуючий код. Наприклад, можна легко додати новий шар для відображення додаткової інформації на екрані без необхідності вносити зміни в інші частини програми.

Діаграма класів Layer та SimulationLayer на рисунку 4.1 відображає взаємозв'язок між цими класами. Layer є базовим класом, від якого успадковується SimulationLayer. Це дозволяє використовувати методи та властивості класу Layer у класі SimulationLayer, що спрощує роботу з об'єктами цих класів та забезпечує їхню взаємодію у програмі.

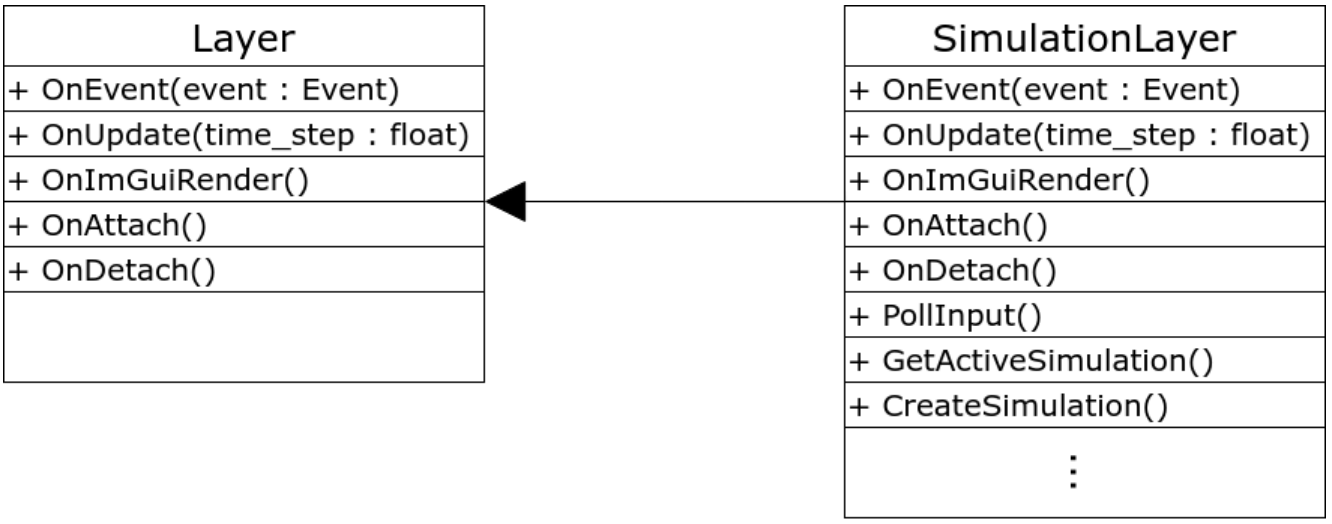

Рисунок 4.1 – Діаграма класів для системи шарів

Клас Layer не є абстрактним, тобто всі методи класу визначені, що дозволяє нам не реалізовувати деякі його функції без потреби. Однак, для використання нового шару, потрібно створити нащадка класу Layer і вже в ньому перевизначати потрібні нам методи.

У головному циклі програми виконується почергове звернення до всіх шарів, під час якого відбувається виклик основних трьох методів: OnEvent, OnUpdate, OnImGuiRender, причому виклики здійснюються в певній, заданій нами, послідовності. Це дозволяє керувати порядком виконання дій у програмі та забезпечує потрібну логіку взаємодії між шарами.

На рисунку 4.2 зображена блок-схема головного циклу програми, яка ілюструє послідовність виконання дій та взаємодію між різними шарами. Ця блок-схема допомагає краще зрозуміти логіку програми та відображає основні кроки, які виконуються під час роботи програми.

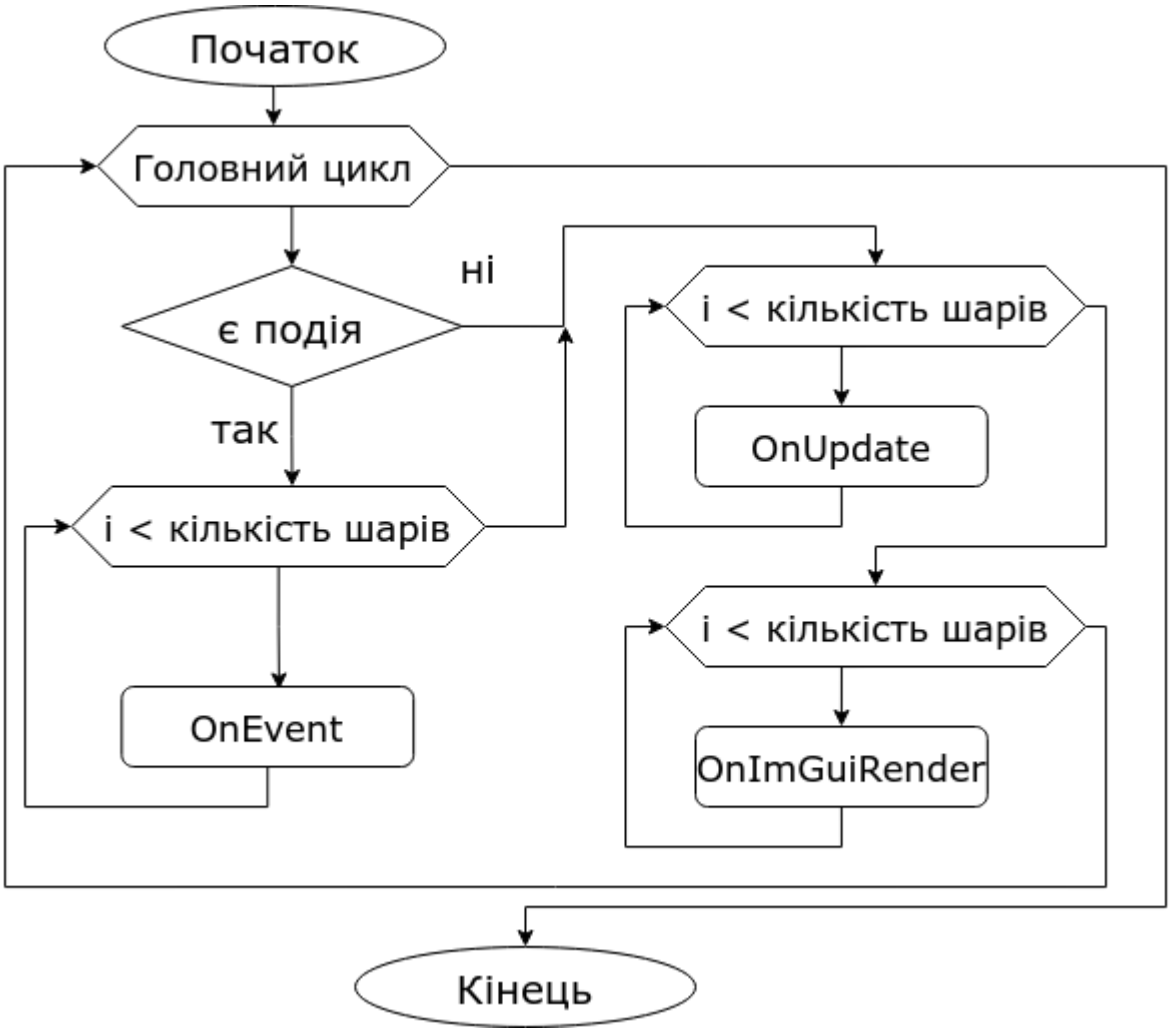

Рисунок 4.2 - Блок-схема головного циклу програми

Слід зазначити, що порядок шарів при виклику функції OnEvent, обернений відносно порядку шарів при виклику функцій OnImGuiRender та OnUpdate. Також слід відмітити те, що на даний момент відрисовка відбувається в функції OnUpdate, що не є доречним. У майбутньому відображення графіки буде відокремлене в функцію OnRender, що дозволить краще контролювати процес відображення та розрахунків.

При виклику функції OnUpdate, модуль має можливість змінити свій стан, наприклад: при виклику функції OnUpdate клас розраховує нове положення тіл. Функція OnUpdate приймає в якості аргументу час останньої ітерації головного циклу, що дозволяє нам досягти однакової швидкості програми незалежно від потужності комп’ютера, на якому запускається наша програма.

Функція OnImGuiRender використовується для звернення до бібліотеки ImGui. В цій функції немає потреби до звернення до конкретної реалізації ImGui, адже все це відбувається в шарі ImGuiLayer. Це дозволяє нам використовувати різні інструменти для графічного відображення, просто створюючи нові шари ImGui та не міняти код в середині інших шарів.

4.1.2 Система повідомлень

Система повідомлень була створена для того, щоб уникнути зв'язування всіх компонентів з бібліотекою GLFW. Це було необхідно, оскільки ця бібліотека, завдяки своїй складності, може бути менш ефективною за інші варіанти. Базовий клас Event має віртуальний деструктор і методи, які повинні бути перевизначені у підкласах. Ці методи включають GetEventType, який повертає тип події, GetName, який повертає назву події, та ToString, який може бути використаний для отримання рядкового представлення події. Поле Handled, розташоване у відкритій секції, використовується для позначення того, чи була подія оброблена. Це дозволяє уникнути проблем, пов'язаних з залежністю від конкретної бібліотеки та забезпечує більшу гнучкість та оптимізацію у роботі програми.

У зв’язку з частим використанням подій для спрощення роботи з ними створено шаблонний клас EventDispatcher. Цей клас дозволяє привести об’єкт батьківського класу до об’єкту дочірнього класу без використання великої кількості конструкцій. Його використання передбачає обробку подій за допомогою викликів обробників подій, що мають приймати в якості аргументу саму подію. При обробці подій, якщо тип події співпадає з типом аргументу обробника, цей обробник викликається, а подія позначається як оброблена. У протилежному випадку EventDispatcher просто повертає керування. Використання цього класу є дуже простим і не потребує явного використання умовних операцій та операцій приведення типів, що значно зменшує кількість коду та спрощує обробку подій.

4.1.3 Архітектура імітаційної моделі

Модуль Simulation є інтерфейсом з методами OnUpdate та GetParticles. Метод OnUpdate повинен розраховувати нові позиції тіл, а метод GetParticles дозволяє нам отримати результати останніх обчислень у вигляді масиву з характеристиками кожного з тіл. За допомогою такого рішення, можна створювати нові імітаційні моделі з використанням різних, не передбачених на даний момент технологій, наприклад CUDA та MPI. На рисунку 4.3 зображено діаграму класів цього модуля.

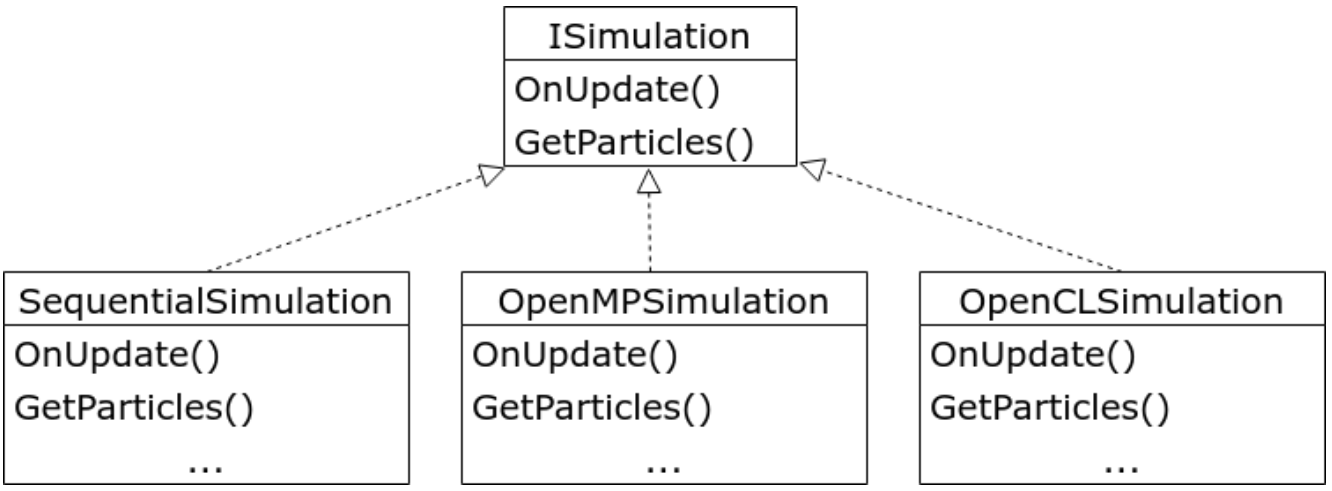

Рисунок 4.3 – Діаграма класів математичного модуля

В програмі створено 3 класи, які реалізують інтерфейс Simulation: SequentialSimulation (послідовне обчислення імітаційної моделі), OpenMPSimulation (паралельне обчислення моделі з використанням OpenMP) та OpenCLSimulation (паралельне обчислення моделі з використанням OpenCL). За допомогою такої реалізації ми можемо одночасно обчислювати імітаційну модель під час виконання програми різними методами обчислень. Якщо метод або програмний засіб для реалізації потребує деяких додаткових налаштувань, ми можемо доповнити клас-нащадок додатковою функцією та в подальшому звертатися до неї. Це забезпечує гнучкість і можливість оптимізації обчислень в залежності від конкретних потреб та обмежень.

4.2 Опис реалізації математичного модуля

Як було зазначено вище, існують 3 класи які реалізують інтерфейс Simulation: SequentialSimulation, OpenMPSimulation та OpenCLSimulation. Розглянемо детальніше роботу класу SequentialSimulation.

4.2.1 Послідовний алгоритм

Цей клас відповідає за послідовне обчислення положення частинок у просторі. Він успадковує функціонал базового класу Simulation і реалізує його віртуальні методи.

Поля класу:

— particles_: контейнер частинок;

— time_step_: крок часу.

Методи класу:

— SequentialSimulation: конструктор класу, який приймає вектор частинок та крок часу;

— OnUpdate: віртуальний метод, який викликається для оновлення стану імітаційної моделі на кожному кроці;

— GetParticles: віртуальний метод, який повертає вектор частинок;

— GetDistance: метод, який обчислює відстань між двома частинками;

— VelocityInteraction: метод, який обчислює взаємодію швидкостей двох частинок;

— ConserveMomentum: метод, який обчислює новий вектор руху при колізії двох частинок;

— Collision: метод, який виявляє колізію двох частинок;

— HasSamePosition: метод, який перевіряє, чи знаходяться два тіла в одному положенні.

Метод OnUpdate використовується для оновлення стану імітаційної моделі на кожному кроці. Для кожної пари частинок відбувається перевірка колізії та

обчислення взаємодії швидкостей. Після цього відбувається оновлення позицій частинок згідно їхніх швидкостей.

Метод Colision використовується для обчислення нової швидкості та радіусу частинки після колізії з іншою частинкою. Він використовує метод ConserveMomentum для збереження кількості руху.

Цей клас реалізує базовий функціонал для обчислення положення частинок, проте, з огляду на питому складність обчислень, можуть виникати проблеми з продуктивністю на великих об'ємах даних.

4.2.2 Паралельний алгоритм з використанням OpenMP

Клас OpenMPSimulation не сильно відрізняється від SequentialSimulation, проте дозволяє суттєво пришвидшити розрахунок сил, за допомогою технології OpenMP.

Завдяки директиві #pragma omp parallel for обробка частинок відбувається паралельно, що дозволяє використовувати більше потоків CPU для обчислень. Кожен потік відповідає за обчислення певної кількості частинок.

Використання schedule(static, 1) дозволяє рівномірно розподілити навантаження між потоками, вказуючи, що кожна частина циклу буде виконуватись одним потоком. Це допомагає уникнути перевантаження одних потоків проти інших.

Перевірка колізій і обчислення взаємодії швидкостей теж відбувається паралельно. Використання #pragma omp parallel for дозволяє кожному потоку працювати зі своєю частиною тіл. Умовна блок-схема алгоритму зображена на рисунку 4.4. Код реалізації наведений в додатку А.

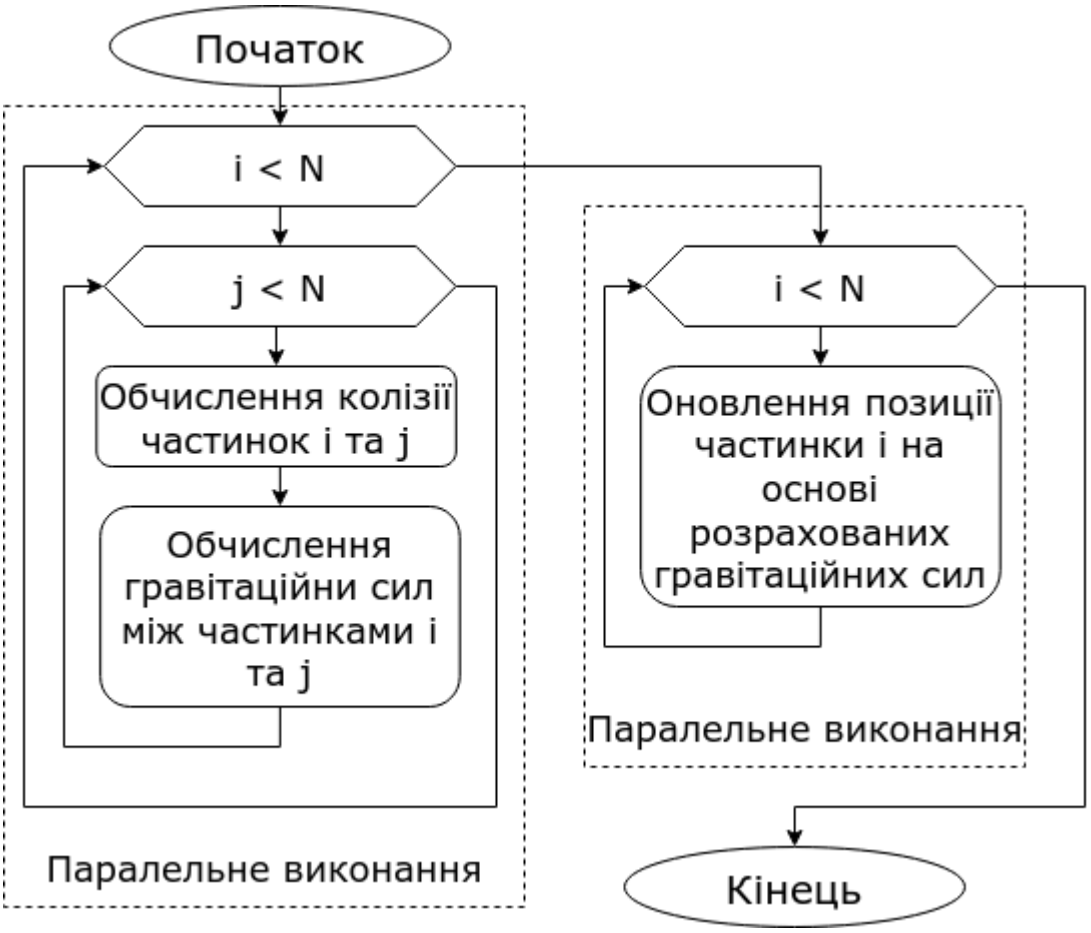

Рисунок 4.4 – Блок-схема алгоритму обчислення гравітаційної задачі

У внутрішньому циклі, який обчислює нові позиції частинок, також використовуються паралельні обчислення для покращення продуктивності, але вони виконується уже після розрахунку для того, щоб уникнути гонки за ресурсами.

4.2.3 Паралельний алгоритм з використанням OpenCL

Алгоритм паралельного обчислення за допомогою OpenCL суттєво не відрізняється від алгоритму для OpenMP. Основна відмінність в них полягає в способі реалізації. Оскільки OpenCL виконує код на GPU, то нам потрібно зробити наступні кроки [16]:

— зчитати код ядра з файлової системи;

— створити програму за допомогою функції clCreateProgram;

— скомпілювати програму ядра за допомогою функції clBuildProgram;

— створити буфер даних за допомогою функції clCreateBuffer;

— створити ядро за допомогою функції clCreateKernel;

— записати дані в пам’ять GPU за допомогою функції clSetKernelArg;

— виконати програму за допомогою функції clEnqueueNDRangeKernel;

— зчитати вихідні дані з буфера за допомогою функції clEnqueueReadBuffer.

Як можна помітити, хоч алгоритм програми суттєво не змінився, та кількість кроків значно зросла в порівнянні з OpenMP і це має свої наслідки: хоч при великій кількості даних ефективність OpenCL значно вище за OpenMP, через кількість одночасно виконуваних потоків, проте нам постійно потрібно виконувати дорогу операцію запису у пам’ять GPU, що зменшить продуктивність на невеликому наборі даних.

Код ядра не сильно відрізняється від коду для послідовного алгоритму, та все ж має деякі відмінності. Так, у коді ядра немає зовнішнього циклу, оскільки він створений для виконання одним робочим елементом процесора, то замість лічильника і, який був в зовнішньому циклі, для ітерації по масиву даних використовується глобальний ідентифікатор. Ще одна відмінність полягає у збереженні результатів виклику функції для запобігання повторних викликів, оскільки компілятор OpenGL не оптимізує код на відміну компілятора мови C чи C++. Код ядра наведений у додатку Б.

4.3 Реалізація та тестування імітаційної моделі

Клас SimulationTest призначений для часового тестування імітаційної моделі. Він містить змінні для збереження поточної ітерації, кількості ітерацій, часу виконання та стану імітаційної моделі. Конструктор класу приймає кількість ітерацій і вказівник на об'єкт Simulation, що дозволяє нам протестувати будь-яку реалізацію інтерфейсу. Методи Begin та Stop встановлюють стан моделі на "запущено" та "зупинено" відповідно. Метод OnUpdate викликає метод OnUpdate у класі Simulation і вимірює час виконання, підраховуючи загальний час

виконання тесту. Варто зазначити, щоб уникнути блокування програми, даний клас на кожній ітерації зберігає час розрахунків та передає керування програмі, після чого чекає наступного виклику OnUpdate.

Методи IsDone, GetStatus та GetResult дозволяють перевірити стан завершення тесту, отримати поточну ітерацію та кількість ітерацій, а також отримати час виконання тесту. Цей клас допомагає контролювати та вимірювати час виконання моделі в класі Simulation, що дозволяє проводити тестування та оптимізацію коду.

5 РОБОТА КОРИСТУВАЧА З ПРОГРАМОЮ

У цьому розділі детально розглянуто процес взаємодії користувача з програмною системою для розв'язання гравітаційної задачі N тіл за допомогою паралельних методів обчислень. Описано кроки, які користувач повинен виконати для ефективного використання системи, включаючи введення даних, налаштування параметрів для розрахунків, запуск паралельних алгоритмів та тестування їх продуктивності. Наведено приклади використання інтерфейсу користувача.

5.1 Системні вимоги

Впровадження передових технологій у програмному забезпеченні для забезпечення високої продуктивності вимагає відповідних системних вимог для його безперебійної роботи.

Мінімальні вимоги:

– процессор: Intel i3-2100 / AMD A6-5400K;

– операційна система: 32-розрядна версія Windows 7 або 32-розрядний дистрибутив Linux;

– оперативна пам'ять: 2 ГБ;

– об'єм вільного дискового простору: від 50 мегабайт;

– графічний процесор: Intel HD Graphics 2000.

Рекомендовані вимоги:

– процессор: Intel Core i5-6400 / AMD Ryzen 5 2400G, або вище;

– операційна система: 64-розрядна версія Windows 7 або 64-розрядний дистрибутив Linux;

– оперативна пам'ять: від 8 ГБ;

– об'єм вільного дискового простору: від 100 МБ;

– графічний процесор: Nvidia GeForce GTX 650 або вище.

Для розробників:

– встановлений 64-розрядний компілятор C++;

– встановлений CMake;

– встановлений OpenCL ICD.

5.2 Встановлення програмного забезпечення

Для встановлення програмного продукту на операційну систему Windows користувачу необхідно розпакувати архів з розширенням .zip у будь-якому зручному місці на комп'ютері та запустити файл інсталяції NBodyInstaller.exe.

Для встановлення на операційну систему Linux слід розпакувати архів з розширенням .tar у зручному місці на комп’ютері та запустити файл NBody. У разі необхідності, треба надати файлу NBody дозвіл на виконання за допомогою команди chmod +x ./NBody [21].

Додаткові інструкції щодо встановлення та використання програмного продукту містяться у файлі README.txt, який входить до складу інсталяційного архіву.

5.3 Графічний інтерфейс користувача

Програма має графічний інтерфейс та керується за допомогою комп’ютерної миші та клавіатури. Вікно програми складається з вікна імітаційної моделі (рисунок 5.1) та вікна меню. Небесні тіла у вікні представлені у вигляді червоних сфер. Натиснувши клавішу J на клавіатурі можна увійти в режим огляду і змінити положення камери. Переміщення камери керується за допомогою клавіш клавіатури W, A, S, D та комп’ютерної миші. Щоб вийти з цього режиму потрібно натиснути клавішу K.

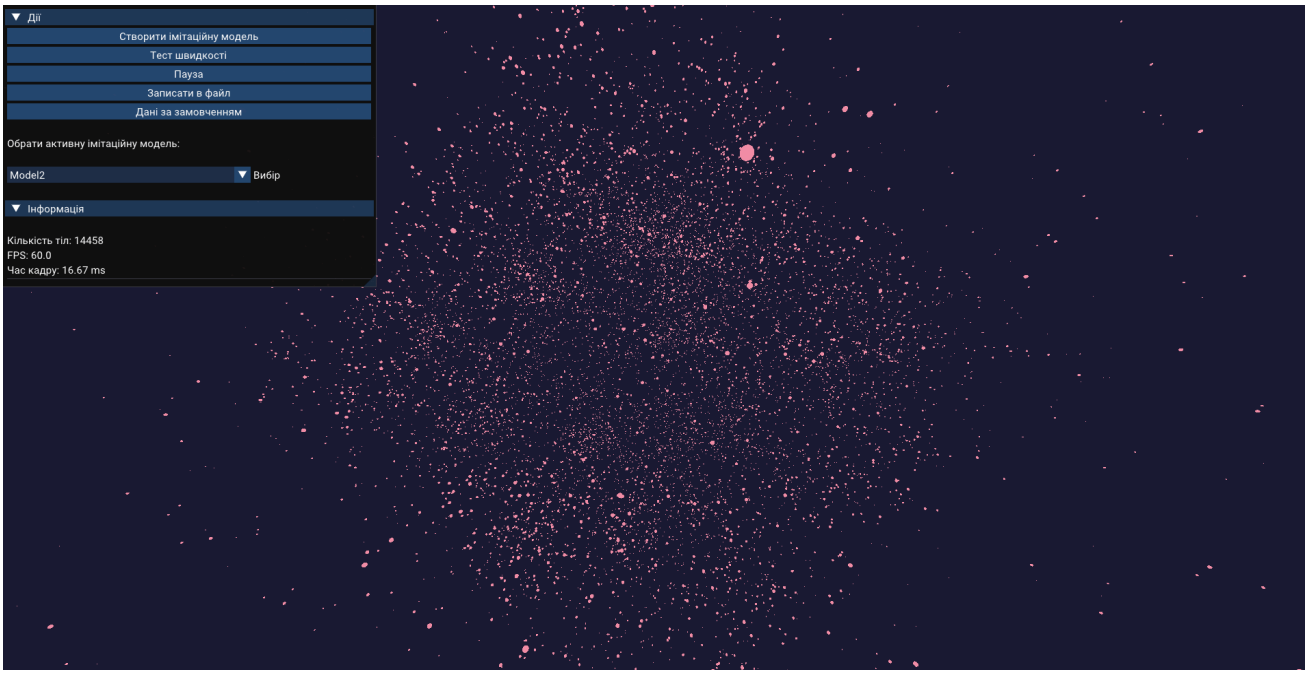

Рисунок 5.1 – Головне вікно програми

Вікно меню знаходиться у верхньому лівому кутку та зображено на рисунку 5.2. Будь-яке вікно програми можна вільно пересувати та змінювати його розміри.

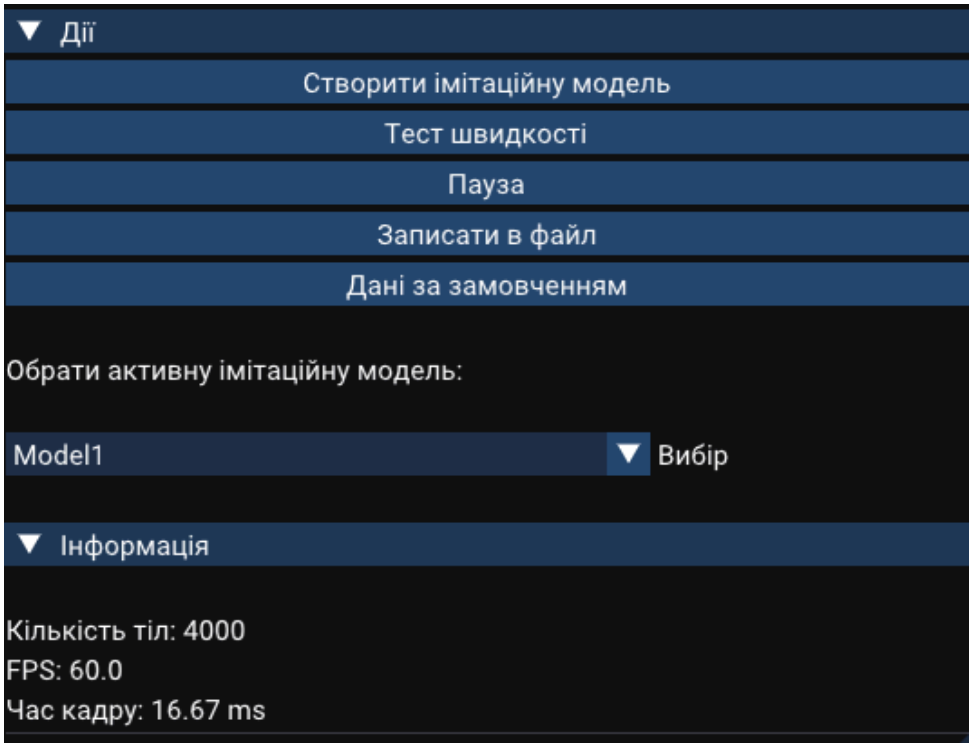

Рисунок 5.2 – Меню програми

В головному меню можна виконати такі дії:

— змінити активну імітаційну модель;

— створити нову імітаційну модель;

— запустити тест швидкодії;

— записати в файл результат обчислень поточної імітаційної моделі;

— переглянути додаткову інформацію.

При натисканні кнопки “Створити імітаційну модель” відкривається нове вікно, яке зображене на рисунку 5.3. У вікні можна обрати тип імітаційної моделі, зчитати вхідні дані з файлу або створити нову модель, вказавши початкові границі значення тіл, такі як: кількість частинок, розміри області побудови, проміжки для маси, прискорення, радіуса та часовий крок. Після натискання кнопки “Створити” створюється нова імітаційна модель і додається до списку в головному меню.

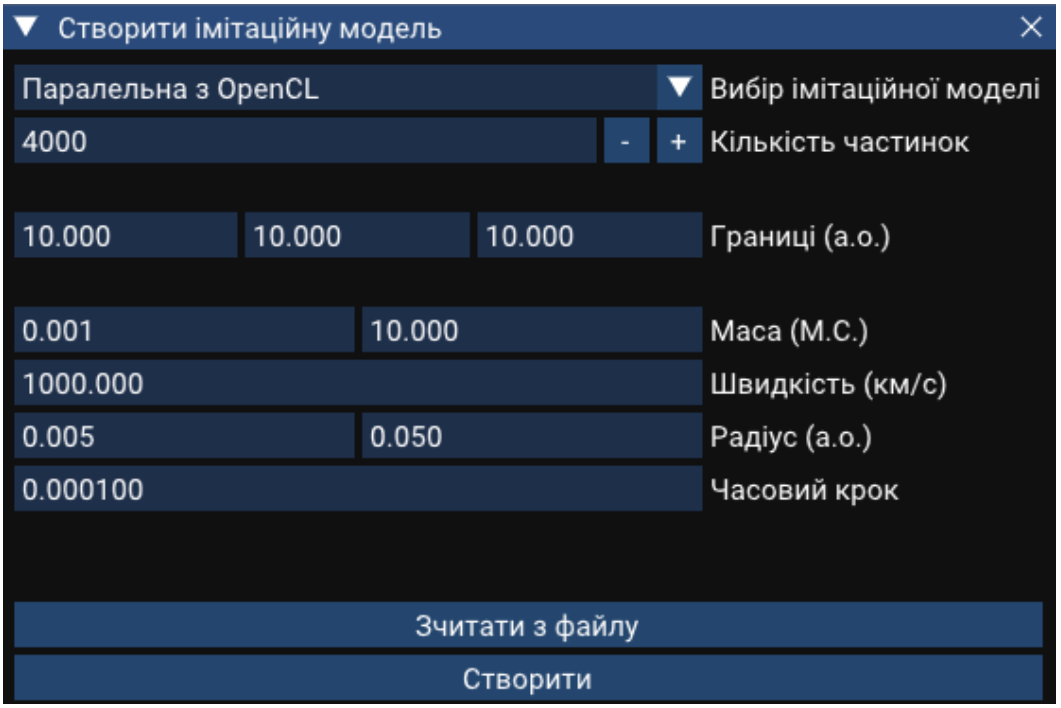

Рисунок 5.3 – Вікно створення імітаційної моделі

При натисканні на кнопку “Тест швидкості” відкривається вікно для вводу початкових значень для тесту. Це меню зображено на рисунку 5.4. При натисканні кнопки “Почати тест” відкриється вікно з результатами тестування.

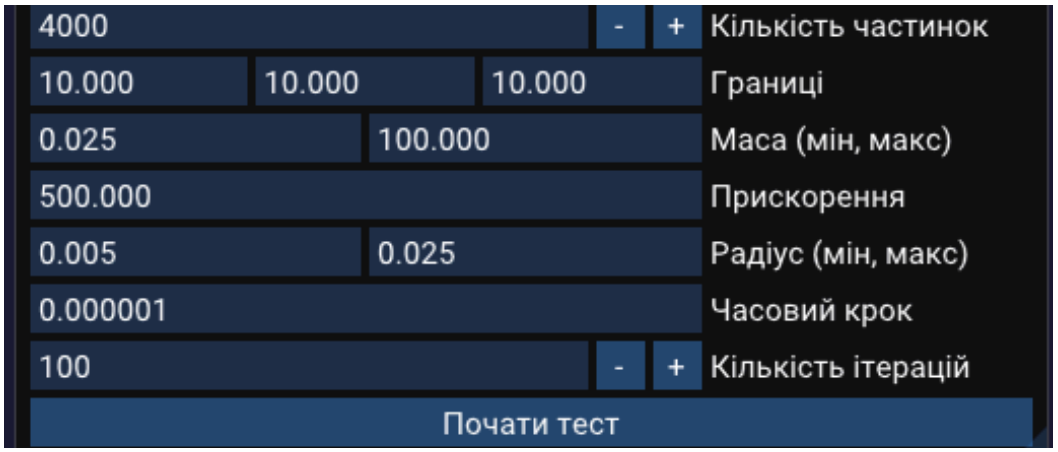

Рисунок 5.4 – Вікно тестів

Даний програмний застосунок має зручний та наочний інтерфейс. Програма не використовує додаткових важких бібліотек та може запускатися як на домашніх персональних комп’ютерах, так і на професійних машинах з продуктивними графічними картами. Додаток дозволяє створювати зразу декілька імітаційних моделей з різними способами обчислення та активно перемикатися між ними.

5.4 Результати виконання програми

Програма для паралельного розв'язання гравітаційної задачі N тіл була успішно розроблена та протестована на різних наборах даних. Програма продемонструвала високу продуктивність та стійкість, дозволяючи моделювати складні системи з великою кількістю тіл.

Програма може зчитувати дані про тіла з текстового файлу. Файл має містити інформацію про кількість тіл, їх положення, швидкість, масу та радіус. Кожен рядок у файлі відповідає одному тілу, а дані розділені комами. Результат зчитування даних 2000 тіл графічно зображений на рисунку 5.5.

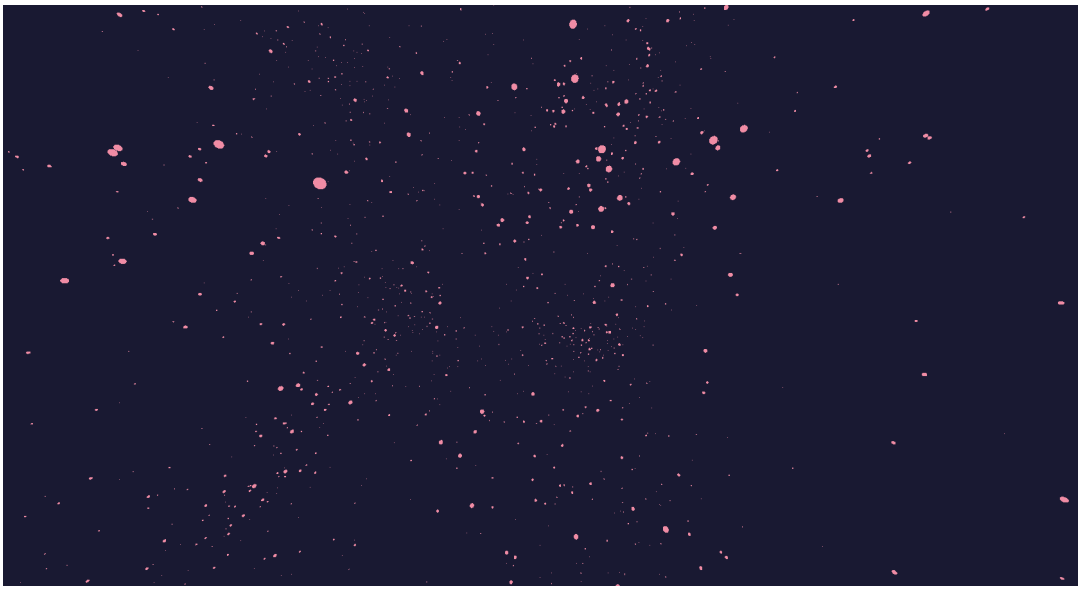

Рисунок 5.5 – Результат зчитування даних з файлу

Програма також може генерувати нову імітаційну модель, задаючи дані точок випадковим методом. В цьому випадку користувач може вказати кількість тіл, діапазон значень для їх положення, швидкості, маси та радіуса. Приклад створення імітаційної моделі, що містить 20000 тіл зображений на рисунку 5.6.

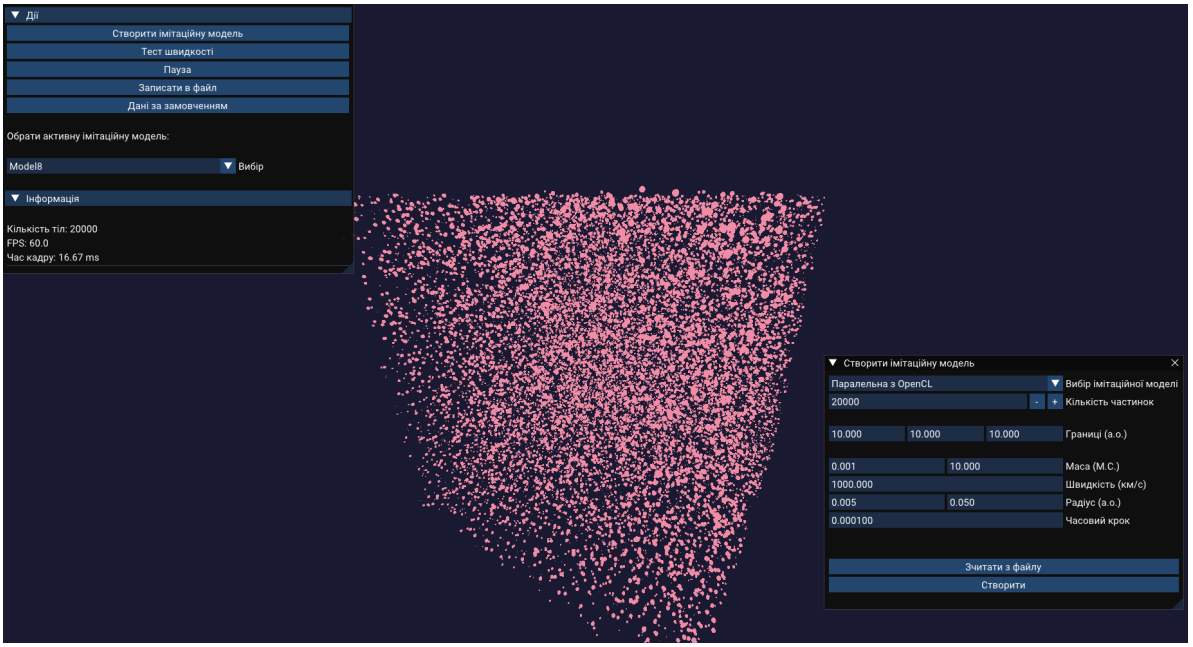

Рисунок 5.6 – Результат генерування початкових даних

Після того, як дані про тіла зчитані або згенеровані, користувач може запустити на виконання імітаційну модель. Програма буде розраховувати траєкторії тіл протягом заданого проміжку часу. Результат еволюції моделі, яка задана на рисунку 5.6, зображений на рисунку 5.7.

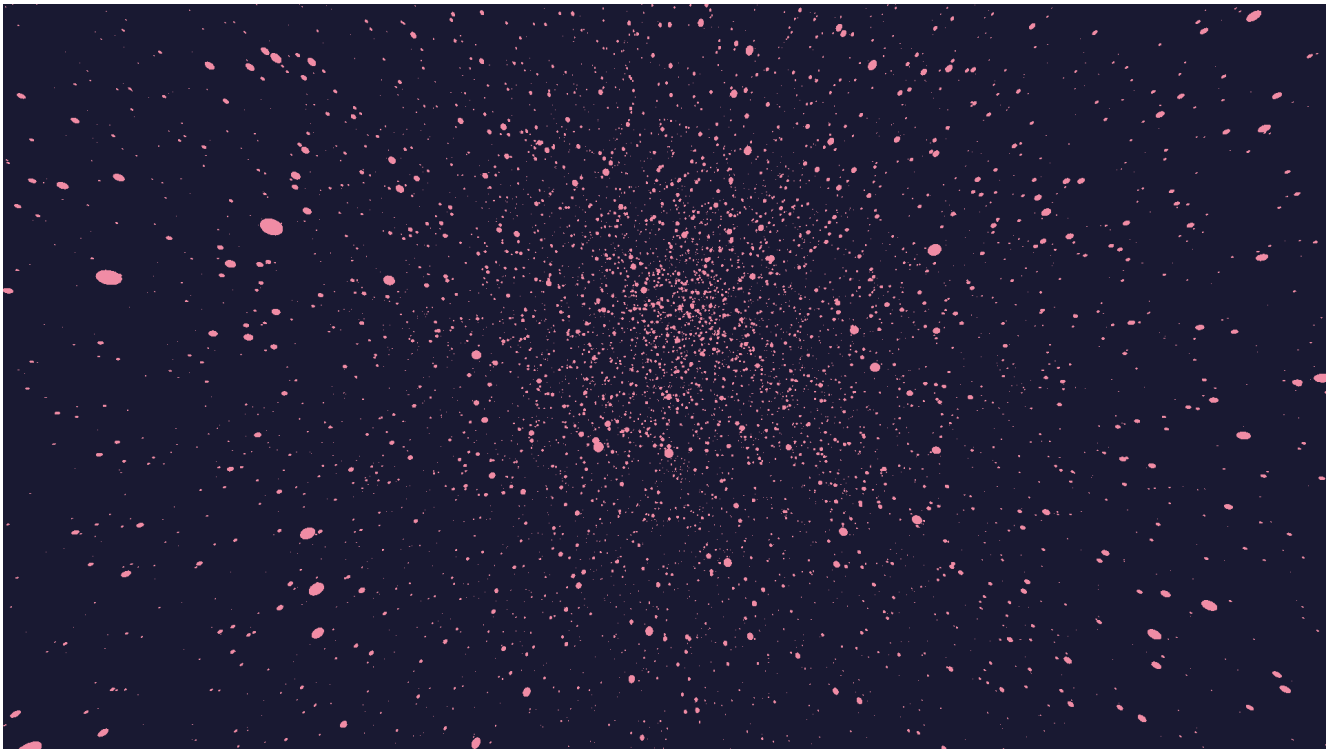

Рисунок 5.7 – Результат еволюції ітераційної моделі

Після завершення обчислення імітаційної моделі результати можуть бути записані до текстового файлу. Файл міститиме інформацію про положення, швидкість та масу кожного тіла на кожному етапі обчислення імітаційної моделі.

ВИСНОВКИ

У цій дипломній роботі було розроблено паралельний метод розв'язання гравітаційної задачі N тіл, що дозволяє ефективно моделювати взаємодію великої кількості тіл під впливом гравітаційних сил. Основна мета роботи полягає в створенні системи, яка використовує сучасні обчислювальні технології для досягнення високої продуктивності та точності обчислень.

Розробка програми включала використання бібліотек OpenMP та OpenCL для паралельного обчислення. Це дозволило значно скоротити час виконання імітаційних моделей за рахунок розподілу обчислювального навантаження між кількома процесорами та графічними процесорами. Використання OpenGL для графічного відображення результатів імітаційних моделей забезпечило наочність та інтуїтивно зрозуміле представлення даних, що є важливим для подальшого аналізу та інтерпретації результатів.

Результати роботи демонструють, що запропонована система є ефективним інструментом для моделювання гравітаційних систем. Вона забезпечує високу продуктивність і точність обчислень, що дозволяє досліджувати складні динамічні процеси у всесвіті. Розроблена система може бути використана не лише для наукових досліджень, але й в освітніх цілях, сприяючи поглибленню розуміння динаміки гравітаційних систем та принципів паралельного обчислення.

Таким чином, ця дипломна робота робить внесок у розвиток методів паралельного розв'язання гравітаційної задачі N тіл і демонструє переваги використання сучасних обчислювальних технологій для досягнення високої продуктивності та точності.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Aarseth S. J. Gravitational n-body simulations: tools and algorithms (cambridge monographs on mathematical physics). Cambridge University Press, 2003. 430 p.

2. Greengard L., Rokhlin V. A fast algorithm for particle simulations. Journal of computational physics. 1987. Vol. 73, no. 2. P. 325–348.

3. Hockney R. W. Computer simulation using particles. CRC Press, 1988.

4. Ostlie D. A., Carroll B. W. Introduction to modern astrophysics, an (2nd edition). 2nd ed. Benjamin Cummings, 2006. 1400 p.

5. Binney J. Galactic dynamics. 2nd ed. Princeton : Princeton University Press, 2008. 885 p.

6. Barnes J., Hut P. A hierarchical O(N log N) force-calculation algorithm. Nature. 1986. Vol. 324, no. 6096. P. 446–449.

7. Greengard L., Rokhlin V. A new version of the Fast Multipole Method for the Laplace equation in three dimensions. Acta numerica. 1997. Vol. 6. P. 229–269.

8. Cipra, Barry Arthur (May 16, 2000). "The Best of the 20th Century: Editors Name Top 10 Algorithms". SIAM News. 33 (4). Society for Industrial and Applied Mathematics: 2.

9. The fast multipole method. Wayback Machine. URL: https://web.archive.org/web/20110603231158/http://www-theor.ch.cam.ac.uk/people/ross/thesis/node97.html (дата звернення: 06.03.2024).

10. ChaNGa. UW Faculty Web Server. URL: http://faculty.washington.edu/trq/hpcc/tools/changa.html (дата звернення: 07.03.2024).

11. Greengard L., Rokhlin V. A new version of the Fast Multipole Method for the Laplace equation in three dimensions. Acta numerica. 1997. Vol. 6. P. 229–269.

12. Springel V. The cosmological simulation code gadget-2. Monthly notices of the royal astronomical society. 2005. Т. 364, № 4. С. 1105–1134.

13. Stroustrup B. C++ programming language. Addison Wesley, 2013. 1368 p.

14. Chapman B. Using OpenMP: portable shared memory parallel programming. Cambridge, Mass : The MIT Press, 2008. 353 p.

15. Parallel Programming in OpenMP / R. Menon та ін. Elsevier Science & Technology Books, 2000.

16. Munshi A. OpenCL programming guide. Upper Saddle River, NJ : Addison-Wesley, 2012. 603 p.

17. Heterogeneous computing with opencl 2. 0 / D. R. Kaeli et al. Elsevier Science & Technology Books, 2015. 330 p.

18. Heterogeneous computing with opencl 2. 0 / D. R. Kaeli et al. Elsevier Science & Technology Books, 2015. 330 p.

19. GitHub - ocornut/imgui: Dear ImGui: Bloat-free Graphical User interface for C++ with minimal dependencies. GitHub. URL: https://github.com/ocornut/imgui (дата звернення: 22.03.2024).

20. GLFW: introduction. An OpenGL library | GLFW. URL: https://www.glfw.org/docs/latest/ (дата звернення: 10.06.2024).

21. UsingTheTerminal - community help wiki. Official Ubuntu Documentation. URL: https://help.ubuntu.com/community/UsingTheTerminal (дата звернення: 10.05.2024).

Додаток А

Реалізація паралельного методу обчислення задачі N тіл за допомогою OpenMP

Програмні засоби:

– мова програмування – C++;

– засіб для паралелізації коду – OpenMP;

openmp_simulation.h

```
#ifndef __NBODY_OPENMP_SIMULATION_H_
#define __NBODY_OPENMP_SIMULATION_H_

#include <vector>

#include "Program/OpenCLSimulation/opencl_particle.h"
#include "simulation.h"

class OpenMPSimulation : public Simulation {

private:
 std::vector<Particle> particles_;

 float force_constant_ = 0.0000000000667f;
 float time_step_;
 float last_time_ = 0;

public:
 OpenMPSimulation(const std::vector<Particle> &particles, float time_step);

 void OnUpdate(float ts) override;
```

```
 float GetLastTime() override { return last_time_; }

 const std::vector<Particle> &GetParticles() const override;

 float GetDistance(Particle a, Particle b);
 Vec VelocityInteraction(Particle a, Particle b);

 float ConserveMomentum(float mass_a, float mass_b, float velocity_a,
                        float velocity_b);
 Particle Colision(Particle a, Particle b);

 bool HasSamePosition(Particle a, Particle b);
};

#endif
```

openmp_simulation.cpp

```
#include "openmp_simulation.h"
#include <cmath>

OpenMPSimulation::OpenMPSimulation(const std::vector<Particle> &particles,
                                   float time_step)

  : particles_(particles), time_step_(time_step) {}

const std::vector<Particle> &OpenMPSimulation::GetParticles() const {
 return particles_;
}

void OpenMPSimulation::OnUpdate(float time_step) {
 last_time_ += time_step * time_step_;

#pragma omp parallel for schedule(static, 1)
 for (int i = 0; i < particles_.size(); ++i) {
#pragma omp parallel for
   for (int j = 0; j < particles_.size(); ++j) {
     if (i != j) {
       if (HasSamePosition(particles_[i], particles_[j])) {
         particles_[i] = Colision(particles_[i], particles_[j]);
         particles_.erase(particles_.begin() + j);

       } else {
         Vec velocity = VelocityInteraction(particles_[i], particles_[j]);

         particles_[i].velocity.x -= velocity.x;
         particles_[i].velocity.y -= velocity.y;
```

```
        particles_[i].velocity.z -= velocity.z;
      }
    }
  }
}

#pragma omp parallel for
 for (int i = 0; i < particles_.size(); ++i) {
  particles_[i].position.x +=
      (particles_[i].velocity.x * time_step_ * time_step);
  particles_[i].position.y +=
      (particles_[i].velocity.y * time_step_ * time_step);
  particles_[i].position.z +=
      (particles_[i].velocity.z * time_step_ * time_step);
 }
}

float OpenMPSimulation::GetDistance(Particle a, Particle b) {
 return std::sqrt(
    ((a.position.x - b.position.x) * (a.position.x - b.position.x)) +
    ((a.position.y - b.position.y) * (a.position.y - b.position.y)) +
    ((a.position.z - b.position.z) * (a.position.z - b.position.z)));
}

Particle OpenMPSimulation::Colision(Particle a, Particle b) {
 Vec velocity = {ConserveMomentum(a.mass, b.mass, a.velocity.x, b.velocity.x),
                 ConserveMomentum(a.mass, b.mass, a.velocity.y, b.velocity.y),
                 ConserveMomentum(a.mass, b.mass, a.velocity.z, b.velocity.z)};

 return {a.position, velocity, a.mass + b.mass, std::max(a.radius, b.radius)};
```

```
}
float OpenMPSimulation::ConserveMomentum(float mass_a, float mass_b,
                                         float velocity_a, float velocity_b) {

 return ((mass_a * velocity_a) + (mass_b * velocity_b)) / (mass_a + mass_b);
}

Vec OpenMPSimulation::VelocityInteraction(Particle a, Particle b) {
 float distance = GetDistance(a, b);
 double g = (b.mass * force_constant_ / (distance * distance * distance)) *
            time_step_;

 return {float(g * (a.position.x - b.position.x)),
         float(g * (a.position.y - b.position.y)),
         float(g * (a.position.z - b.position.z))};
}

bool OpenMPSimulation::HasSamePosition(Particle a, Particle b) {
 return a.radius + b.radius > GetDistance(a, b);
}
```

Додаток Б

Реалізація паралельного методу обчислення задачі N тіл за допомогою OpenCL

Програмні засоби:

– мова програмування – C++;

– засіб для паралелізації коду – OpenCL;

opencl_simulation.h

```
#ifndef INCLUDE_OPENCLSIMULATION_OPENCL_SIMULATION_H_
#define INCLUDE_OPENCLSIMULATION_OPENCL_SIMULATION_H_

#include "../simulation.h"
#include "src/Program/OpenCLSimulation/opencl_particle.h"
#include <CL/cl.h>
#include <string>

class OpenCLSimulation : public Simulation {
private:
 cl_platform_id platform;
 cl_device_id device;
 cl_context context;
 cl_command_queue queue;
 cl_program program;
 cl_program program2;
 cl_kernel kernel;
 cl_mem particles_buffer;
 cl_int err;

 float time_step_;
```

```
 float last_time_ = 0;
 std::vector<Particle> cl_particles_;
 char *loadKernelSource(const char *fileName);

 Particle *arr;
 int arr_size;

 std::vector<Particle> output_particles;

public:
 OpenCLSimulation(std::vector<Particle> particles, float time_step);

 const std::vector<Particle> &GetParticles() const override {
   return cl_particles_;
 }

 float GetLastTime() override { return last_time_; }

 ~OpenCLSimulation() {

   clReleaseMemObject(particles_buffer);
   clReleaseKernel(kernel);
   clReleaseProgram(program);
   clReleaseCommandQueue(queue);
   clReleaseContext(context);
 }
 void OnUpdate(float ts);
};
#endif // INCLUDE_OPENCLSIMULATION_OPENCL_SIMULATION_H_
```

opencl_simulation.cpp

```
#include "./opencl_simulation.h"
#include "src/Program/OpenCLSimulation/opencl_particle.h"
#include <CL/cl.h>
#include <cstdio>

char *OpenCLSimulation::loadKernelSource(const char *fileName) {
 FILE *file = fopen(fileName, "r");
 if (!file) {
  perror("Failed to open kernel file");
  exit(EXIT_FAILURE);
 }

 fseek(file, 0, SEEK_END);
 long fileSize = ftell(file);
 rewind(file);

 char *buffer = (char *)malloc(fileSize + 1);
 if (!buffer) {
  perror("Failed to allocate memory for kernel source");
  exit(EXIT_FAILURE);
 }

 size_t bytesRead = fread(buffer, 1, fileSize, file);
 if (bytesRead != fileSize) {
  perror("Failed to read kernel source file");
  exit(EXIT_FAILURE);
 }

 buffer[fileSize] = '\0';
```

```
    fclose(file);

    return buffer;
    }

    OpenCLSimulation::OpenCLSimulation(std::vector<Particle> particles,
                                       float time_step)
      : cl_particles_(particles), time_step_(time_step),
        output_particles(particles.size()) {
    cl_particles_.reserve(particles.size());

    err = clGetPlatformIDs(1, &platform, NULL);
    err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
    context = clCreateContext(NULL, 1, &device, NULL, NULL, &err);
    queue = clCreateCommandQueue(context, device, 0, &err);

    char *kernelSource =

loadKernelSource("/home/dmitro/Documents/Projects/C/NBodyProject/Nbody/"
                         "src/Program/OpenCLSimulation/kernel.cl");


        program = clCreateProgramWithSource(context, 1, (const char
**)&kernelSource,
                                        NULL, &err);
    if (!program || err != CL_SUCCESS) {
      fprintf(stderr, "Error creating program\n");
      exit(EXIT_FAILURE);
    }
```

```
 err = clBuildProgram(program, 1, &device, NULL, NULL, NULL);
if (err != CL_SUCCESS) {
  fprintf(stderr, "Error building program\n");
  exit(EXIT_FAILURE);
}

arr = new Particle[sizeof(Particle) * cl_particles_.size()];
arr_size = cl_particles_.size();

kernel = clCreateKernel(program, "nbody", &err);
if (!kernel || err != CL_SUCCESS) {
  fprintf(stderr, "Error creating kernel\n");
  exit(EXIT_FAILURE);
}
}

void OpenCLSimulation::OnUpdate(float ts) {

float time = time_step_ * ts;
last_time_ += time;
int particles_amount = cl_particles_.size();

cl_mem particleBuffer = clCreateBuffer(
    context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
    sizeof(Particle) * cl_particles_.size(), cl_particles_.data(), &err);
if (!particleBuffer || err != CL_SUCCESS) {
  fprintf(stderr, "Error creating particle buffer\n");
  exit(EXIT_FAILURE);
}
```

```
cl_mem output_particles_buffer = clCreateBuffer(
   context, CL_MEM_READ_WRITE, sizeof(Particle) * arr_size, NULL, &err);

if (!output_particles_buffer || err != CL_SUCCESS) {
  fprintf(stderr, "Error creating particle buffer\n");
  exit(EXIT_FAILURE);
}


err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &particleBuffer);
if (err != CL_SUCCESS) {
  fprintf(stderr, "Error setting kernel argument\n");
  exit(EXIT_FAILURE);
}
clSetKernelArg(kernel, 1, sizeof(int), &particles_amount);
clSetKernelArg(kernel, 2, sizeof(float), &time);

err = clSetKernelArg(kernel, 3, sizeof(cl_mem), &output_particles_buffer);
if (err != CL_SUCCESS) {
  fprintf(stderr, "Error setting kernel argument2\n");
  exit(EXIT_FAILURE);
}


size_t globalWorkSize = cl_particles_.size();
 err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &globalWorkSize,
NULL, 0,
                        NULL, NULL);
if (err != CL_SUCCESS) {
```

```
  fprintf(stderr, "Error enqueueing kernel\n");
  exit(EXIT_FAILURE);
}
clFinish(queue);

err = clEnqueueReadBuffer(queue, output_particles_buffer, CL_TRUE, 0,
                          sizeof(Particle) * arr_size, arr, 0, NULL, NULL);
if (err != CL_SUCCESS) {
  fprintf(stderr, "Error reading particle buffer\n");
  exit(EXIT_FAILURE);
}

cl_particles_.clear();
for (int i = 0; i < arr_size; i++)
  if (arr[i].is_exist)
    cl_particles_.push_back(arr[i]);

arr_size = cl_particles_.size();
}
```

kernel.cl

```
typedef struct {
 float x;
 float y;
 float z;
} Vec;

typedef struct {
 Vec position;
 Vec velocity;
 float mass;
 float radius;
 int is_exist;  // OpenCL не підтримує тип bool
} Particle;

__kernel void nbody(__global Particle *particles, const int numParticles, const float deltaTime, __global Particle *output_particles) {
   int i = get_global_id(0);
   Particle current_particle = particles[i];
   if (!current_particle.is_exist)
      return;

   Vec total_acceleration = {0, 0, 0};
   for (int j = 0; j < numParticles; ++j) {
      if (i != j) {
         Particle other_particle = particles[j];
         if (!other_particle.is_exist)
            continue;

         if (HasSamePosition(current_particle, other_particle)) {
```

```
            current_particle.is_exist = false;
            break;  // Exit the loop if collision occurs
        } else {
            Vec acceleration = VelocityInteraction(current_particle, other_particle);
            total_acceleration.x += acceleration.x;
            total_acceleration.y += acceleration.y;
            total_acceleration.z += acceleration.z;
        }
    }
  }

  current_particle.velocity.x -= total_acceleration.x * deltaTime;
  current_particle.velocity.y -= total_acceleration.y * deltaTime;
  current_particle.velocity.z -= total_acceleration.z * deltaTime;

  current_particle.position.x += current_particle.velocity.x * deltaTime;
  current_particle.position.y += current_particle.velocity.y * deltaTime;
  current_particle.position.z += current_particle.velocity.z * deltaTime;

  output_particles[i] = current_particle;
}

bool HasSamePosition(Particle a, Particle b) {
  float distance = sqrt((a.position.x - b.position.x) * (a.position.x - b.position.x) +
                        (a.position.y - b.position.y) * (a.position.y - b.position.y) +
                        (a.position.z - b.position.z) * (a.position.z - b.position.z));
  return a.radius + b.radius > distance;
}

Particle Colision(Particle a, Particle b) {
```

```
    Vec velocity = {
            ConserveMomentum(a.mass, b.mass, a.velocity.x, b.velocity.x),
            ConserveMomentum(a.mass, b.mass, a.velocity.y, b.velocity.y),
            ConserveMomentum(a.mass, b.mass, a.velocity.z, b.velocity.z)
    };

    return {a.position, velocity, a.mass + b.mass, fmax(a.radius, b.radius), false};
}

Vec VelocityInteraction(Particle a, Particle b) {
    float distance = sqrt((a.position.x - b.position.x) * (a.position.x - b.position.x) +
                          (a.position.y - b.position.y) * (a.position.y - b.position.y) +
                          (a.position.z - b.position.z) * (a.position.z - b.position.z));
    double g = (b.mass * force_constant_ / (distance * distance * distance));

    return {(float) (g * (b.position.x - a.position.x)),
            (float) (g * (b.position.y - a.position.y)),
            (float) (g * (b.position.z - a.position.z))};
}

float ConserveMomentum(float mass_a, float mass_b, float velocity_a, float
velocity_b) {
    return ((mass_a * velocity_a) + (mass_b * velocity_b)) / (mass_a + mass_b);
}
```