КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ІМЕНІ ТАРАСА ШЕВЧЕНКА

ФАКУЛЬТЕТ РАДІОФІЗИКИ, ЕЛЕКТРОНІКИ ТА КОМП'ЮТЕРНИХ СИСТЕМ

Кафедра комп'ютерної інженерії

До захисту допущено: «На правах рукопису»
Завідувач кафедри _____ Юрій Бойко
« _ » _____ 2023 р.

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему:

«ФІЗИЧНА СИМУЛЯЦІЯ ІГРОВИХ ОБ'ЄКТІВ В БАГАТОЯДЕРНИХ СИСТЕМАХ»

Виконав:
студент 4-го курсу бакалаврату
денної форми навчання
спеціальності 123 Комп'ютерна інженерія
ОНП «_____»
Чернишев Олег Олександрович _____

Науковий керівник:
кандидат фізико-математичних наук, доцент
Моторна Оксана Віталіївна _____

Рецензент:
_____

Засвідчую, що у цій бакалаврській роботі
немає запозичень з праць інших авторів без
відповідних посилань
Студент _____

Робота допущена до захисту в ЕК рішенням кафедри _____
від «__»_____ 2023 р., протокол № __.

Завідувач кафедри _____,
кандидат фізико-математичних наук, доцент
Бойко Юрій Володимирович

(підпис)

Київ – 2023

РЕФЕРАТ

Випускна кваліфікаційна робота бакалавра: 51 с., 23 рис., 15 джерел.

Дослідження впливу багатопотоковості на ігрові системи з фізичною симуляцією. Створення програмного забезпечення, у якому об’єкти мають властивості твердих тіл для перевірки результатів досліджень

Ключові слова: Premake, STL, SFML, GLM, Фізичний двигун, Симуляція, Динаміка твердого тіла, Колізія, Обмеження об’єктів, Багатопотоковість, Пул потоків, Закони Ньютона, State Machine.

ЗМІСТ

ВСТУП 4

1. Опис кроків для побудови робочого програмного забезпечення 6

1.1 Система збірки 7

1.2 Етапи побудови 8

1.3 Структуризація проекту 9

1.4 Бібліотеки 10

2. Фізичні двигуни у програмуванні 12

2.1 Типові фізичні параметри 13

2.2 Функціонал 14

3. Симуляція динаміки твердого тіла 16

3.1 Опис базових фізичних параметрів 17

3.2 Обрахунок колізій при взаємодії різних об'єктів 20

3.3 Обмежений рух та реакція на колізію 25

4. Архітектура проекту 27

4.1 Розділення рендерного та логічного коду 27

4.2 Стани програми 28

5. Багатопотоковість 32

5.1 Пул потоків 32

5.2 Розпаралелення обчислень колізій та реакцій на них 33

ВИСНОВКИ 39

Список використаних джерел 41

Додаток А 42

ВСТУП

Фізична симуляція – це галузь у сфері комп'ютерних наук, головна мета якої відтворити фізичні явища за допомогою комп'ютера. Загалом, ці симуляції застосовують чисельні методи до існуючих теорій, щоб отримати результати, максимально наближені до того, що можна спостерігати в реальному світі. Це дозволяє розробникам різних сфер діяльності передбачити та ретельно проаналізувати, як об'єкт буде поводитися перш ніж створити його, що майже завжди простіше та дешевше зробити.

Діапазон застосувань фізичної симуляції є надзвичайно широким. Найперші комп'ютери вже використовувалися для виконання фізичного моделювання, наприклад, для прогнозування балістичного руху снарядів у військовій сфері діяльності. Це також важливий інструмент у цивільному та автомобільному будівництві, який висвітлює, як певні конструкції поводитимуться під час таких подій, як землетрус чи автомобільна аварія. Вона також використовується в астрофізиці, може бути доречною у теорії відносності та багато інших речей, які можна спостерігати серед явищ природи.

Слід зазначити що симуляція фізики дуже поширено використовується у відеоіграх. Багато ігор повністю покладаються на симуляцію фізики, щоб бути привабливими для гравців. Це означає, що для цих ігор потрібна стабільна симуляція, яка не буде ламатися чи сповільнюватися, а цього зазвичай непросто досягти.

У будь-якій грі цікаві лише певні фізичні ефекти. Динаміка твердого тіла – рух і взаємодія твердих, негнучких об'єктів – на сьогоднішній день є найпопулярнішим видом ефекту, що моделюється в іграх. Це тому що більшість об'єктів, з якими відбувається взаємодія в реальному житті, досить тверді, а симуляція твердих тіл є відносно простою, але все ж таки потребує

певних зусиль. Інші ігри і сфери використання вимагають симуляції більш складних об’єктів, таких як деформовані тіла, рідини, магнітні об’єкти тощо.

Як відомо комп’ютерні системи не стоять на одному місці і з кожним роком з’являються нові ідеї і шляхи покращення результатів роботи. Однією з таких ідей було розподілення обчислень на декілька процесорних одиниць, що при грамотному використанні може підвищити швидкість виконання програм. Майже кожна ігрова або неігрова система для симуляції різних ситуацій використовує вже готові фізичні двигуни, але деякі з них не використовують усі обчислювальні можливості комп’ютера, або не мають відкритого джерела коду. Слід зазначити що реалізація кожного фізичного двигуна є схожою з точки зору формул для обчислення математичних рівнянь, але різною в архітектурному плані, а отже і в застосуванні багатопотоковості. Тому дослідження способів покращення результатів виконання програми, що застосовує фізичну симуляцію, є досить доречним.

Мета даної роботи: дослідження впливу багатопотоковості на ігрові системи з фізичною симуляцією. Створення програмного забезпечення, у якому об’єкти мають властивості твердих тіл, для перевірки результатів досліджень.

1. Опис кроків для побудови робочого програмного забезпечення

Для виконання поставленої задачі потрібно обрати необхідні інструменти, які спростять процес і дозволять сфокусуватися на самій проблемі. В даній роботі буде використано мову програмування C++ [15] та відповідні до неї компоненти. Вибір на користь цієї мови був зроблений на основі таких причин:

1. Продуктивність: C++ є високопродуктивною мовою програмування, що дозволяє ефективно використовувати ресурси багатоядерних систем. Це особливо важливо для фізичної симуляції, де потрібно обчислювати багато операцій одночасно.
2. Контроль над пам'яттю: У C++ є можливість прямого керування пам'яттю, що дозволяє ефективно управляти ресурсами і мінімізувати затрати. Це особливо корисно для оптимізації фізичних обчислень, де швидкий доступ до пам'яті може бути критичним.
3. Розширюваність: C++ надає широкий набір бібліотек і фреймворків для фізичної симуляції та графіки. Можна використовувати наявні бібліотеки або розробити власні, щоб реалізувати специфічні функції.
4. Широке застосування: C++ є однією з найпоширеніших мов програмування в галузі ігрової розробки та фізичних симуляцій.
5. Об'єктно-орієнтований підхід: C++ підтримує об'єктно-орієнтований підхід до програмування, що дозволяє організувати код у вигляді класів і об'єктів. Це сприяє модульності, повторному використанню коду і полегшує розвиток складних систем.
6. Портативність: C++ є портативною мовою, що означає, що код, написаний на C++, можна виконувати на різних платформах без необхідності внесення значних змін. Це дозволяє працювати з різними системами і отримувати результати на різних платформах.

7. Інтеграція з існуючими системами: C++ має здатність легко інтегруватися з іншими мовами програмування, такими як C, Python, Java тощо.
8. Гнучкість та контроль: C++ надає гнучкість у виразності мови, дозволяючи реалізувати складні алгоритми та логіку, а також мати повний контроль над програмою, що дає можливість оптимізувати код і досягти бажаних результатів.
9. Довгоіснуюча мова програмування: C++ є довгоіснуючою мовою програмування з багатою історією і великою базою знань. Вона має широку підтримку у відкритому співтоваристві розробників.

1.1 Система збірки

Система збірки – це набір програм і супровідних текстових файлів, які разом створюють базу програмного коду. Створення кодової бази означає створення кінцевих результатів із вихідних файлів. Наприклад, для бази коду C++ кінцевими результатами можуть бути виконувані файли, динамічні або статичні бібліотеки, а мета системи збірки C++ полягає в тому, щоб створювати ці результати з вихідних файлів C++, знайдених у базі коду.

Системи збірки можуть бути автономними програмами командного рядка, такими як Make та Ninja, або частинами інтегрованих середовищ розробки, наприклад Visual Studio і XCode.

Ці системи потрібно використовувати, оскільки компіляція програми з вихідного коду вже не така проста, як виконання однієї команди компіляції, наприклад:

```
g++ -o Application main.cpp
```

Хоча це працює, воно покладається на параметри конфігурації за замовчуванням для компілятора та компонувальника. Було б набагато краще

мати більше контролю над генерацією проекту, саме для таких цілей і використовується система збірки, адже завдяки ній є змога керувати такими аспектами розробки програмного забезпечення:

- організація вихідного коду;

- взаємозалежності вихідного коду;

- керування сторонніми бібліотеками;

- параметри компіляції;

- параметри генерації коду;

- конфігурація програмного компонувальника;

- обробка після збірки;

- управління тестуванням. [2]

Для невеликих проектів як система збірки може підійти Premake, завдяки його простому синтаксису й зручності використання. Premake - це утиліта командного рядка, яка читає сценарій визначення проекту програмного забезпечення та, як правило, використовує його при створення файлів проекту для наборів інструментів, таких як Visual Studio, Xcode або GNU Make. [1]

1.2 Етапи побудови

Компіляція та компоновка коду в проектах, що застосовують різні інструменти й механізми побудови відбувається неоднаково. У випадку застосування мови програмування можна виділити такі етапи побудови:

Препроцесінг - препроцесор обробляє директиви препроцесора, такі як #include і #define. Він працює з одним вихідним файлом C++ за раз, замінюючи директиви #include вмістом відповідних файлів, виконуючи

заміну макросів і вибираючи різні частини тексту залежно від #if, #ifdef і #ifndef директив (якщо такі присутні). Токени відносяться до одиниць тексту, з якими працює препроцесор під час препроцесінгу вихідного файлу C++. Вони представляють окремі лексеми (наприклад, ключові слова, оператори, ідентифікатори, літерали тощо). Препроцесор працює з потоком токенів попередньої обробки. Макропідстановка визначається як заміна токенів іншими токенами. Після всього цього препроцесор виробляє єдиний вихід, який є потоком токенів, що є результатом перетворень, описаних вище;

Компіляція - виконується на кожному виході препроцесора. Компілятор аналізує чистий вихідний код C++ (без будь-яких директив препроцесора) і перетворює його на код асемблера;

Асемблювання - збирає цей код у машинний код, створюючи фактичний двійковий файл у певному форматі. Цей об’єктний файл містить скомпільований код у двійковій формі. Символи в об'єктних файлах називаються за назвою;

Компонування - вироблення кінцевого результату компіляції з об'єктних файлів, створених компілятором. Результат може бути динамічною або статичною бібліотекою, або виконуваним файлом. Компонувальник пов’язує всі об’єктні файли, замінюючи посилання на невизначені символи правильними адресами. Кожен із цих символів можна визначити в інших об’єктних файлах або в бібліотеках. Якщо вони визначені в бібліотеках, відмінних від стандартної бібліотеки, то потрібно повідомити компонувальнику про них.

1.3 Структуризація проекту

Важливим етапом розробки будь-якого ПЗ є структуризація файлів у проекті. Програміст створює підсвідомо зрозумілу структуру папок та файлів, щоб інша людина, дивлячись на проект, швидко орієнтувалася. Завдяки вже описаній системі збірки можна відсортовувати по папках проміжні та скомпільовані файли, що спростить орієнтацію у проекті, зробить його підтримку і тестування простішим.

Одним з варіантів форматування проекту може бути наступний:

```
Project/
├── assets/
├── cfg/
├── build/
│   ├── bin/
│   └── intermediate/
├── extern/
└── src/
```

Ці директорії застосовуються для відповідних типів файлів: assets – файли для дизайну програми, cfg – файли які відповідають за параметри, налаштування програми, build/bin – виконувані файли, build/intermediate – проміжні файли, extern – файли, що належать до сторонніх бібліотек, src – вихідний код проекту.

1.4 Бібліотеки

Для написання підтримуваного програмного забезпечення необхідно використовувати сторонні бібліотеки, адже багато потрібного функціоналу може бути вже створено. Тобто застосовуючи вже існуючі інструменти можна швидко й ефективно писати працюючий код, який був протестований іншими розробниками.

Використані бібліотеки:

Стандартна бібліотека шаблонів (STL) [3] — це набір шаблонних класів C++ з функціями загального призначення і реалізованими структурами даних, такими як списки, стеки, масиви тощо. Компоненти цієї бібліотеки є параметризованими;

Проста та швидка мультимедійна бібліотека(SFML) [4] - забезпечує простий інтерфейс для різних компонентів вашого ПК, щоб полегшити розробку ігор і мультимедійних програм. Ця бібліотека складається з п’яти модулів: системний, віконний, графічний, аудіо та мережевий. Слід зазначити що SFML пітримує OpenGL функції, адже має реалізацію свого функціоналу з допомогою цього графічного API;

OpenGL Mathematics (GLM) [5] — це лише заголовок математичної бібліотеки C++ для графічного програмного забезпечення на основі специфікацій OpenGL Shading Language (GLSL). Ця бібліотека ідеально працює з OpenGL, але також забезпечує взаємодію з іншими сторонніми бібліотеками та SDK. Це хороший кандидат для графічного програмування, обробки зображень, фізичної симуляції та будь-якого контексту розробки, який потребує простої та зручної математичної бібліотеки.

2. Фізичні двигуни у програмуванні

При моделюванні фізичної поведінки об’єктів слід притримуватися певних фізичних законів. В кожній симуляції присутні фізичні параметри, які можна змінювати за власним бажанням. Для врегулювання цих нюансів і створюються фізичні двигуни.

Фізичні двигуни у програмуванні - це спеціальні бібліотеки або фреймворки, які дозволяють моделювати фізичні ефекти, рух об'єктів та симулювати реалістичну фізику у програмних проектах. Вони надають розробникам засоби для створення інтерактивних та реалістичних сценаріїв, включаючи графічні ігри, симуляції фізичних процесів, віртуальну реальність та інші додатки.

Основна мета використання фізичних двигунів у програмуванні полягає у тому, щоб надати об'єктам у віртуальному середовищі реалістичні фізичні властивості та взаємодію. Це дозволяє створювати вражаючі візуальні ефекти, реалістичний рух об'єктів, детекцію зіткнень та вплив фізичних сил.

Фізичні двигуни надають розробникам можливість встановлювати фізичні властивості об'єктів, такі як маса, форма, геометрія, тривимірна позиція та швидкість. Вони також забезпечують симуляцію фізичних законів, таких як закони Ньютона, гравітація, тертя та інші, що дозволяє об'єктам рухатися та взаємодіяти один з одним згідно з цими законами.

Фізичні двигуни широко використовуються у галузі ігрової розробки, де вони дозволяють створювати реалістичні світи та інтерактивний ігровий процес. Вони також застосовуються у візуалізації фізичних ефектів, таких як руйнування, рух рідини, симуляція тканин та інше.

2.1 Типові фізичні параметри

Кількість параметрів може бути необмеженою, адже розробник може вигадати будь-який новий параметр, але , зазвичай, існують параметри, які присутні в більшості фізичних двигунів:

1) Маса об'єкта: визначає його інерцію та впливає на рух та взаємодію з іншими об'єктами у середовищі. Більша маса об'єкта призводить до повільнішого руху та більшої стійкості при зіткненні. Розробники можуть налаштовувати масу об'єктів для досягнення потрібної динаміки та реалістичності сценарію.
2) Форма та геометрія об'єкта: фізичні двигуни зазвичай підтримують різні типи форм об'єктів, такі як куля, паралелепіпед, капсула, меш (трикутникова сітка) та інші. Розробники можуть вибирати форму, яка найкраще відповідає їх об'єкту, забезпечуючи точніше моделювання його фізичних властивостей та зіткнень з іншими об'єктами.
3) Тривимірна позиція та швидкість об'єкта: розробники можуть встановлювати початкову позицію об'єкта у віртуальному просторі та встановлювати його початкову швидкість.
4) Еластичність: цей параметр визначає ступінь відскоку або деформації об'єкта при зіткненні з іншими об'єктами.
5) Сила вітру: дозволяє моделювати вплив вітру на об'єкти. Розробники можуть встановлювати напрямок та інтенсивність вітру, щоб об'єкти рухалися або деформувалися відповідно.
6) Рух рідини: деякі фізичні двигуни підтримують симуляцію руху рідини, такої як вода або слиз. Вони мають параметри, що визначають в'язкість, тиск, течію та інші властивості рідини.
7) Динамічні матеріали: існують фізичні двигуни, що дозволяють встановлювати матеріальні властивості об'єктів, такі як жорсткість,

пружність, розтягнення та інші. Це дозволяє розробникам моделювати різні типи матеріалів з різною поведінкою.

8) Статичне та динамічне тертя: статичне тертя відноситься до опору, з яким об'єкт стикається, коли розпочинає рух. Це опір, який треба подолати, щоб перемістити нерухомий об'єкт. Чим вище значення статичного тертя, тим важче почати рух об'єкта. Динамічне тертя відноситься до опору, який виникає під час руху об'єкта. Після того, як об'єкт починає рухатися, динамічне тертя впливає на швидкість руху об'єкта. Цей параметр визначає силу опору, з якою об'єкт зіштовхується під час руху. Чим вище значення динамічного тертя, тим швидше сповільнюється рух об'єкта. Якщо значення тертя встановлено відповідно до реальних фізичних властивостей матеріалів, то це дозволяє створювати більш реалістичні симуляції руху об'єктів. Завдяки цим параметрам розробники можуть досягти більш точного моделювання сили тертя, що впливає на рух та поведінку об'єктів у віртуальному середовищі. Враховуючи значення динамічного та статичного тертя, програмісти можуть створювати реалістичні сценарії, де об'єкти будуть взаємодіяти зі своїм оточенням так, як очікується в реальному світі.

Отже, існує безліч можливих фізичних параметрів, що можуть застосовуватися в симуляції, але головною задачею програміста є визначення конкретних параметрів, які будуть корисними у даній програмі.

2.2 Функціонал

Насамперед, ідея функціоналу фізичного двигуна є доволі простою. Він симулює сцену застосовуючи фізичні закони, такі як гравітація, колізії, тертя, динаміка тіл і так далі. Він розраховує рух об'єктів у віртуальному середовищі відповідно до цих законів. Також обраховує будь-які колізії між

об’єктами та вирішує їх подальший рух. Не дивлячись на просто ідею, сама реалізація є дуже комплексною роботою програміста. Результати відпрацювання того чи іншого функціоналу двигуна може використовуватися в інших ігрових модулях: анімації, рендеринг, тощо.

На наступній діаграмі абстрактно показано загальний принцип роботи фізичного двигуна:[6]

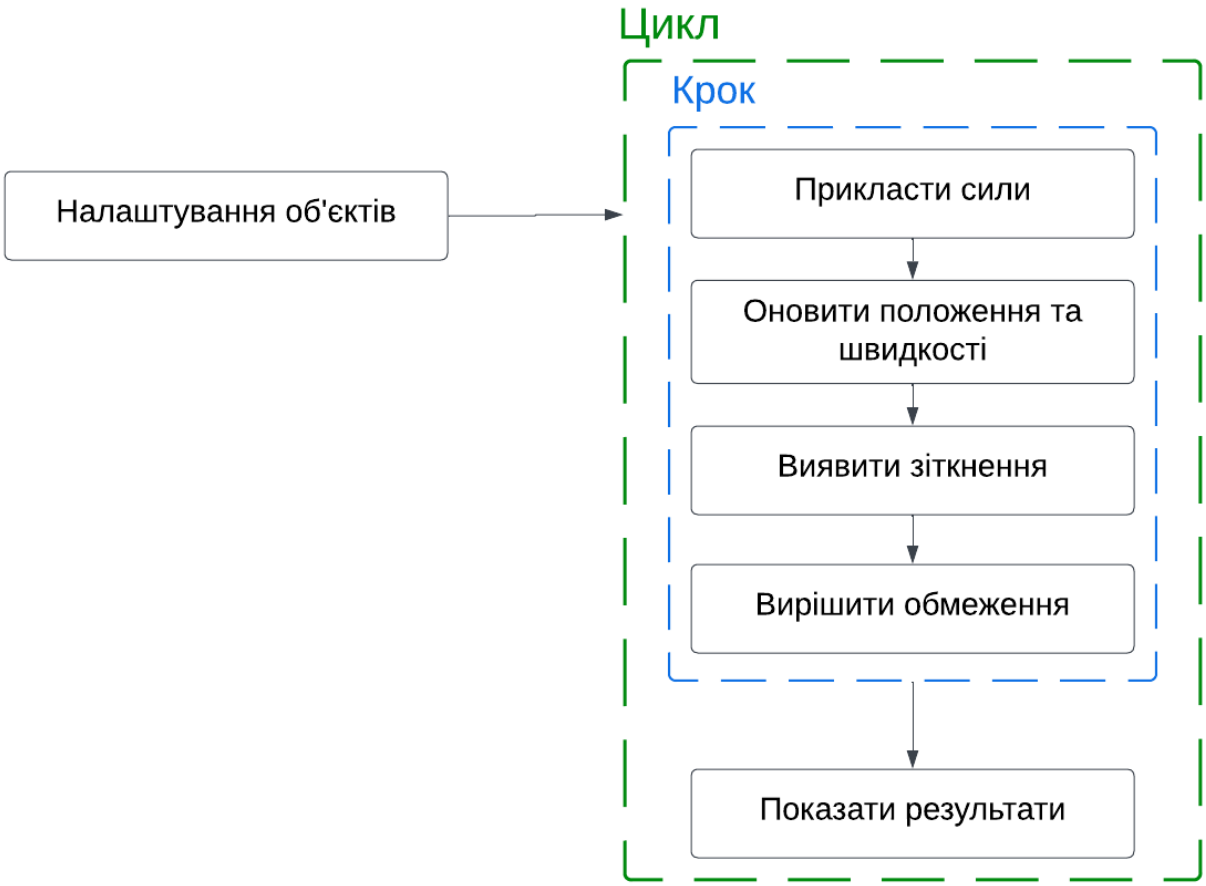

рис. 1 – діаграма принципу роботи фізичного двигуна

Слід зазначити, що на кожному кроці фізичний двигун виконує певну послідовність функцій, а далі вже передає результат до головної частини програми, де завдяки рендерингу можна буде побачити все на екрані.

3. Симуляція динаміки твердого тіла

Перш ніж замислюватись над динамікою твердого тіла слід зазначити, що будь-яка фізична симуляція в комп'ютерних системах є наближеною, тобто те що не можна рахувати ідеальним в реальному житті – можна у віртуальному світі. Тобто будь-яка модель відтворена комп'ютером буде математично описувати ідеальний варіант реального світу, але, звісно, таку модель не можна приймати за еталон.

Рух твердих тіл можна моделювати за допомогою механіки Ньютона, яка ґрунтується на трьох класичних законах:

- Закон інерції: існують такі системи відліку, в яких тіло зберігає стан спокою або рівномірного прямолінійного руху, якщо на нього не діють інші тіла, або дія цих тіл скомпенсована. Тобто, якщо на об'єкт не застосовано жодної сили, його швидкість не змінюється;

- Сила, маса та прискорення: сила, що діє на об'єкт, дорівнює масі об'єкта, помноженій на його прискорення, яке повідомляється цією силою ($F = ma$);

- Дія та реакція: Тіла взаємодіють із силами, що лежать на одній прямій, спрямованими в протилежні сторони і рівні за модулем. Іншими словами, коли одне тіло діє на інше, друге тіло діє на перше з такою ж силою протилежного напрямку.

Ґрунтуючись на цих трьох законах можна відтворити динамічну поведінку, і таким чином створити для ґравця уявлення того, що відбувається.[6]

3.1 Опис базових фізичних параметрів

Тверде тіло — це тіло, яке не може деформуватися. Таких твердих тіл не існує в реальному світі – навіть найтвердіші матеріали деформуються принаймні дуже незначно, коли до них прикладається певна сила – але тверде тіло є корисною фізичною моделлю для розробників ігор, яка спрощує вивчення динаміки твердого тіла, де деформаціями можна знехтувати.

Тверде тіло має масу, положення та швидкість. Крім того, воно має об’єм і форму, тому може обертатися. Тіло природньо обертається навколо свого центру мас, і положення твердого тіла вважається положенням його центру мас. Визначимо початковий стан твердого тіла з центром мас у початку координат і нульовим кутом повороту. Його положення та обертання в будь-який момент часу t буде зсувом початкового стану.

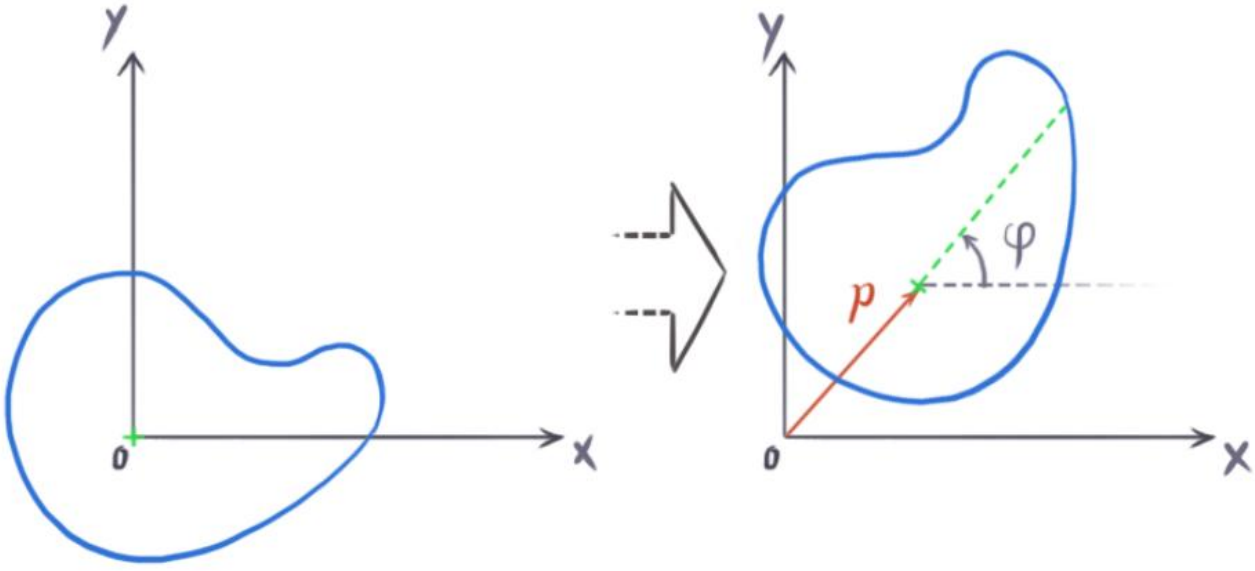

рис. 2 – зміна положення і кута повороту тіла

Центр маси - це середина розподілу мас тіла. Якщо уявити, що тверде тіло з масою M складається з N крихітних частинок, кожна з яких має масу m_i і розташована всередині тіла r_i, центр маси можна обчислити як:

$$\frac{\sum m_i r_i}{M}.$$

Ця формула показує, що центр мас є середнім значенням положень частинок, зважених за їх масою. Якщо щільність тіла рівномірна, центр мас збігається з геометричним центром форми тіла, також відомим як центроїд. Фізичні ігрові двигуни зазвичай підтримують лише рівномірну щільність, тому геометричний центр можна використовувати як центр мас.

Оскільки тверде тіло може обертатися, необхідно ввести його кутові властивості, які аналогічні лінійним властивостям частинки. У двох вимірах, наприклад, тверде тіло може обертатися лише навколо осі, що вказує за межі екрана, тому потрібен лише один скаляр, щоб представити його орієнтацію. Зазвичай використання радіан (від 0 до 2π для повного кола) як одиницю вимірювання замість кутів (від 0° до 360° для повного кола) є більш зручним, оскільки це спрощує обчислення.

Для того, щоб обертатися, твердому тілу потрібна певна кутова швидкість, яка визначається як $\frac{\Delta\theta}{\Delta t}$ і позначається грецькою літерою ω (омега). Однак, щоб отримати кутову швидкість, тіло має отримувати певну обертальну силу, яка має назву момент сили і позначається грецькою літерою τ (тау). Таким чином, другий закон Ньютона, застосований до обертання:

$\tau = I\varepsilon$, де ε - кутове прискорення, а I - момент інерції.

Для обертань момент інерції аналогічний масі для прямолінійного руху. Він визначає, наскільки важко змінити кутову швидкість твердого тіла. У двомірному просторі це скаляр і визначається як:

$I = \int_V \rho(r) r^2 dV$, де V означає, що цей інтеграл слід виконати для всіх точок по всьому об'єму тіла, r - вектор позиції кожної точки відносно осі обертання, r^2 -

фактично скалярний добуток r на себе, а ρ - функція, яка дає щільність у кожній точці тіла.

Наприклад, момент інерції двовимірної коробки масою m, шириною w і висотою h відносно її центроїда дорівнює:

$$I = \frac{m(h^2 + w^2)}{12}.$$

Коли сила прикладена до точки на твердому тілі, вона може створити крутний момент. У двомірному просторі крутний момент є скалярним, а крутний момент τ, що створюється силою f, прикладеною до точки тіла, яка має зміщений вектор r від центру мас, дорівнює:

$\tau = |r||f| \sin\theta$, де θ (тета) — найменший кут між f та r.

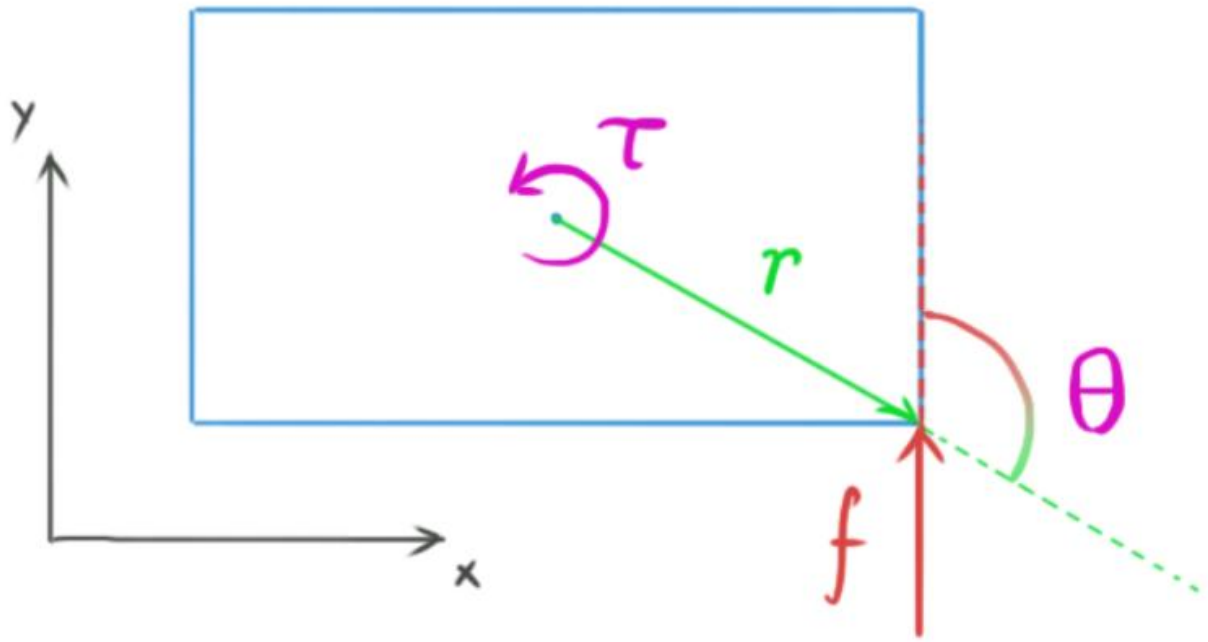

рис. 3 – обертаючий момент при прикладанні певної сили

Двовимірне моделювання можна розглядати як тривимірне моделювання, де всі тверді тіла тонкі та плоскі, і все це відбувається в площині xy, тобто немає руху по осі z. Це означає, що f і r завжди знаходяться в площині xy, тому τ завжди матиме нульові компоненти x і y, оскільки векторний добуток завжди буде перпендикулярним до площини xy. Це, у свою чергу, означає, що він завжди буде паралельним осі z. Таким чином, має значення лише компонент z векторного добутку. Це призводить до того, що обчислення крутного моменту в двох вимірах можна спростити до:

$\tau = r_x f_y - r_y f_x$. Таке спрощення є векторним добутком з нульовою координатою z : $\tau = r \times f$.

Отже, використання цих всіх параметрів на об’єктах допоможе змоделювати поведінку реального тіла, але не вистачить для повного моделювання системи тіл, бо ще не визначено як ці тіла будуть реагувати на зіткнення одне з одним, а також невизначеними є самі описи зіткнень (колізій). Тобто на рух тіл ще не накладені ніякі обмеження.[6]

3.2 Обрахунок колізій при взаємодії різних об’єктів

Щоб більш реалістично симулювати поведінку твердих об’єктів, потрібно перевіряти, чи вони стикаються один з одним кожного разу, коли рухаються, і якщо вони стикаються, є можливість щось із цим зробити, наприклад, застосувати сили, які змінюють їхні швидкості, таким чином вони

будуть рухатися в протилежному напрямку. Саме тут розуміння фізики зіткнень є особливо важливим для розробників ігор.

У контексті симуляції твердого тіла зіткнення відбувається, коли форми двох твердих тіл перетинаються або коли відстань між цими формами є меншою за вказану. Якщо ми маємо n тіл у нашій симуляції, обчислювальна складність виявлення зіткнень за допомогою парних тестів дорівнює O(n^2). Кількість попарних тестів зростає квадратично з кількістю тіл. Залежно від цілей розробника, цей процес можна оптимізувати, але це не є обов'язковим, тобто якщо для конкретної системи або дослідження така оптимізація не дасть покращення у продуктивності, то, звісно сенсу від такої оптимізації немає. Щоб оптимізувати процес виявлення зіткнень, зазвичай використовують розбиття на дві фази: широку фазу та вузьку фазу.

Широка фаза відповідає за пошук пар твердих тіл, які потенційно стикаються, і виключення пар, які точно не стикаються, з усього набору тіл, які є в симуляції. Цей крок повинен добре масштабуватись із кількістю твердих тіл, щоб переконатися, що складність перебору об’єктів буде нижчою O(n^2). Щоб досягти цього, на цьому етапі зазвичай використовується розділення простору в поєднанні з обмежувальними об’ємами, які встановлюють верхню межу для зіткнення.

Вузька фаза діє на пари твердих тіл, знайдені широкою фазою, які можуть стикатися. Це етап уточнення, на якому ми визначаємо, чи дійсно відбуваються зіткнення, і для кожного знайденого зіткнення ми обчислюємо точки контакту, які використовуватимуться для вирішення зіткнень пізніше.[7]

Для визначення колізії об’єкта однієї форми з об’єктом іншої форми потрібно використовувати різні методики, а деякі з таких методів можуть буди дуже

складними, особливо у тривимірному просторі. Для відносно простих симуляцій у двовимірному просторі можна використати наступні ідеї:

- Обмежувальна рамка, вирівняна по осі (Axis-Aligned Bounding Box) - є однією з найпростіших форм виявлення зіткнень між двома прямокутниками, які вирівняні по осі, тобто без обертання. Алгоритм працює, гарантуючи відсутність проміжків між жодною з 4 сторін прямокутника. Будь-який проміжок означає, що зіткнення не існує :[11]

рис. 4 – зіткнення двох прямокутників

```
bool Collides(rectangle1, rectangle2)
{
    return  rectangle1.Min.x <= rectangle2.Max.x &&
            rectangle1.Max.x >= rectangle2.Min.x &&
            rectangle1.Min.y <= rectangle2.Max.y &&
            rectangle1.Max.y >= rectangle2.Min.y;
}
```

- Зіткнення кола (Circle Collision) - ще одна проста форма для виявлення зіткнень – між двома колами. Цей алгоритм працює, беручи центри двох кіл і гарантуючи, що відстань між центрами менша за два радіуси, складені разом. [10]

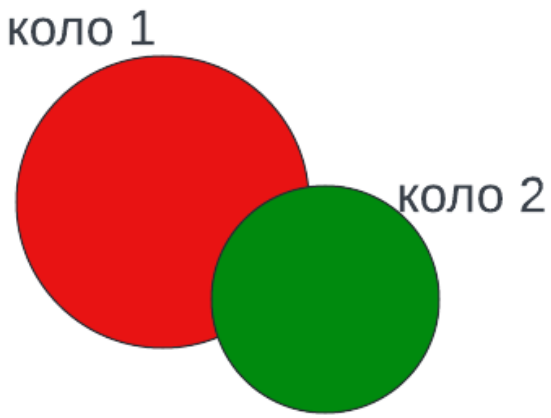

рис. 5 – зіткнення двох кіл

```
bool Collides(circle1, circle2)
{
    Vector2D distanceVec = circle2.Position – circle1.Position;
    float combinedRadius = circle1.Radius + circle2.Radius;

    return distanceVec.length() < combinedRadius;
}
```

- AABB - Circle collision detection - Виявлення зіткнень між колом і прямокутником є трохи складніше, але здійснимо: потрібно знайти точку на AABB, яка найближче до кола, і якщо відстань від кола до цієї точки менше його радіуса , маємо зіткнення. Щоб отримати цю найближчу точку *P* на AABB. На наступному зображенні показано, як обчислити цю точку для будь-якого довільного AABB і кола:

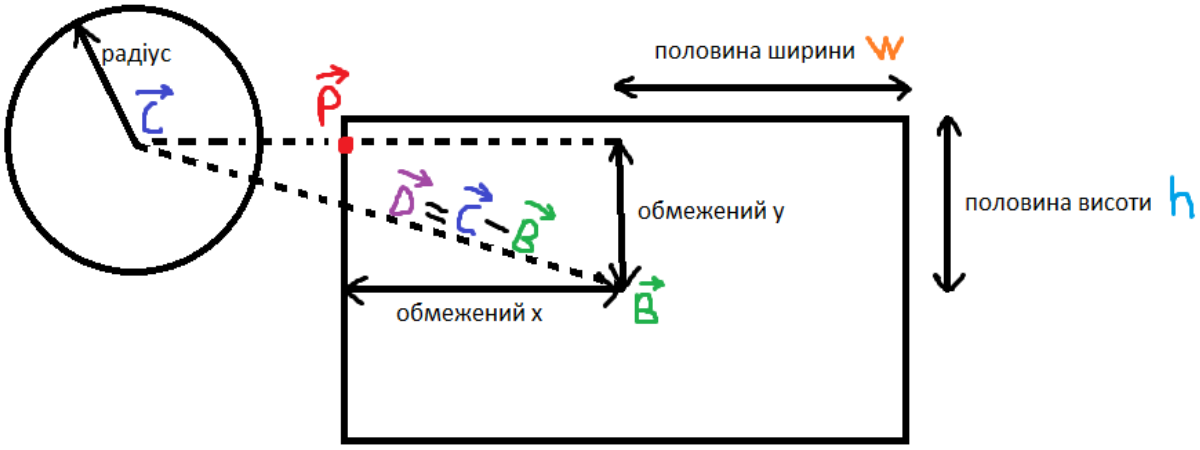

рис. 6 – знаходження точки P, найближчої до центру кола.

Спочатку потрібно отримати вектор різниці між центром C кола та центром B у прямокутника, щоб отримати D. Потім цей вектор закріплюється на половині висоті та половині ширини прямокутника. Після додавання закріпленого вектора D до цетру прямокутника (вектора B), отримується позиція вектора P, який завжди розташований десь на краю прямокутника. Далі йде обчислення нового вектор D′, який є різницею між центром кола C і вектором P.

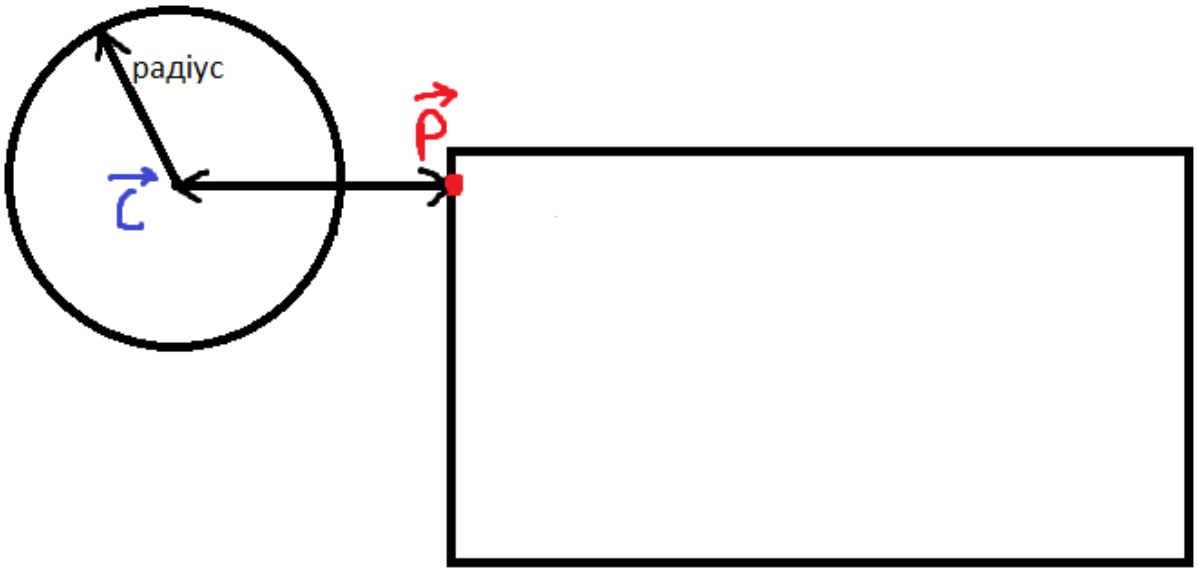

рис. 7 – відстань від центра кола до найближчої точки на прямокутнику.

Коли вектор D′ обчислений, можна порівняти його довжину з радіусом кола. Якщо довжина D′ менша за радіус кола - маємо зіткнення. [9]

3.3 Обмежений рух та реакція на колізію

Останнім кроком до моделювання реалістичних твердих об’єктів є застосування обмежень на взаємодію тіл. Після виявлення колізії об’єкти мають рухатися таким чином щоб вони не пересікалися, тобто щоб перше тіло не знаходилося всередені другого і так далі.

Обмеження — це, по суті, правила, яких необхідно дотримуватися під час моделювання, наприклад «відстань між цими двома частинками не повинна перевищувати 2» або «ці дві точки на цій парі твердих тіл мають збігатися в будь-який час». Іншими словами, обмеження позбавляє твердого тіла ступенів свободи. На кожному кроці симуляції можна обчислити коригувальні сили або імпульси, які, застосовані до тіл. Вони будуть

зближувати або розштовхувати їх, таким чином їхній рух буде обмежено, а правила, накладені обмеженнями, залишатимуться задоволеними.

На практиці обмеження визначається в термінах функції поведінки або функції обмеження C, яка приймає стан пари тіл як параметри (наприклад, положення та орієнтацію) і виводить скалярне число. Коли значення цієї функції знаходиться в прийнятному діапазоні, обмеження виконується. Таким чином, на кожному етапі моделювання необхідно прикладати сили або імпульси до твердих тіл, щоб намагатися зберегти значення C у дозволеному діапазоні.[8]

4. Архітектура проекту

Фізична симуляція, як і будь-яка інша програма потребує правильної структуризації коду, щоб підтримка і обслуговування були якомога легшими. Насамперед кожний фізичний двигун є не тільки добре спланованим, а й модульним: може виступати як статична або динамічна бібліотека, що робить його дуже корисним.

Даний проект розроблений як один суцільний виконуваний файл (без використання динамічних бібліотек), щоб спростити дослідження заданих цілей та розуміння й використання самого проекту.

4.1 Розділення рендерного та логічного коду

Будь-який проект ,пов’язаний з індустрією, в якій використовується графіка, має розділення коду на рендер та логічну частину.

Рендер частина передбачає собою усі заходи що пов’язані з відмалюванням якихось сутностей на екран монітору. Слід зазначити, що більшість сучасних технологій рендерингу використовує подвійний буфер. Тобто може існувати допоміжний буфер, а точніше “framebuffer”, у котрий будуть рендеритися окремі частини сцени, але дисплей при цьому буде відображати малюнок з попереднього кадру. Після того як до цього буферу будуть надані всі необхідні частини сцени, він буде відрендерений. Такий підхід використовується для уникнення “артефактів” на екрані. Тобто уникнення розривів або мерехтіння, які можуть виникнути, якщо візуалізація виконується безпосередньо у передній буфер під час його відображення.

Окрім підхода з допоміжним буфером існує багато інших різнів підходів рендерінгу, саме для цього і розділяють код на рендер частину. Це дозволяє ефективно перемикатися між різними техніками рендерингу та покращувати підтримку такого коду.

Логічний розділ коду відповідає за поведінку, створення й деструкцію усіх об’єктів або сутностей. Саме цьому розділу коду призначена відповідальність за якусь ігрову логіку, або фізичну симуляцію. Даний розділ має синхронізувати роботу коду при використанні різних комп’ютерів, адже різні машини можуть з різною швидкістю робити операції та команди процесору. Тому логічний розділ має враховувати різний час виконання кадру ігрового циклу. Беручи до уваги час виконання кадру кожної ітерації циклу можна досягти синхронізації: користувачі на різних комп’ютерах будуть бачити один і той самий результат. Наприклад, при симуляції падіння м’яча з якоїсь висоти, час який потрібен щоб об’єкт впав на землю – буде однаковий. Також будь-який користувач буде отримувати однакові результати незалежно від його FPS (Frames per second) – кількості кадрів в секунду.

4.2 Стани програми

В даному проекті було використано патерн “State machine”. Суть його полягає в тому що, програма має декілька станів: інтро, головне меню, симуляція, пауза та глобальний стан.

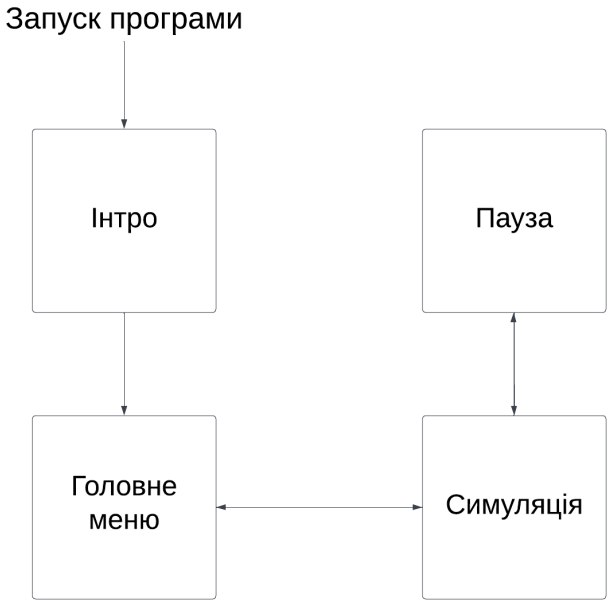

рис. 8 – схема станів програми.

Інтро стан слугує лише як простір для творчості. Зазвичай його створюють щоб надати користувачу сприятливіше відчуття, більш детально погрузитися у програму, тощо.

рис. 9 – інтро стан.

Стан головного меню слугує інтерфейсом користувача, де він може обрати наступні дії по своєму бажанню. Цей стан є необхідним оскільки він регулює усі подальші варіанти виконання програми.

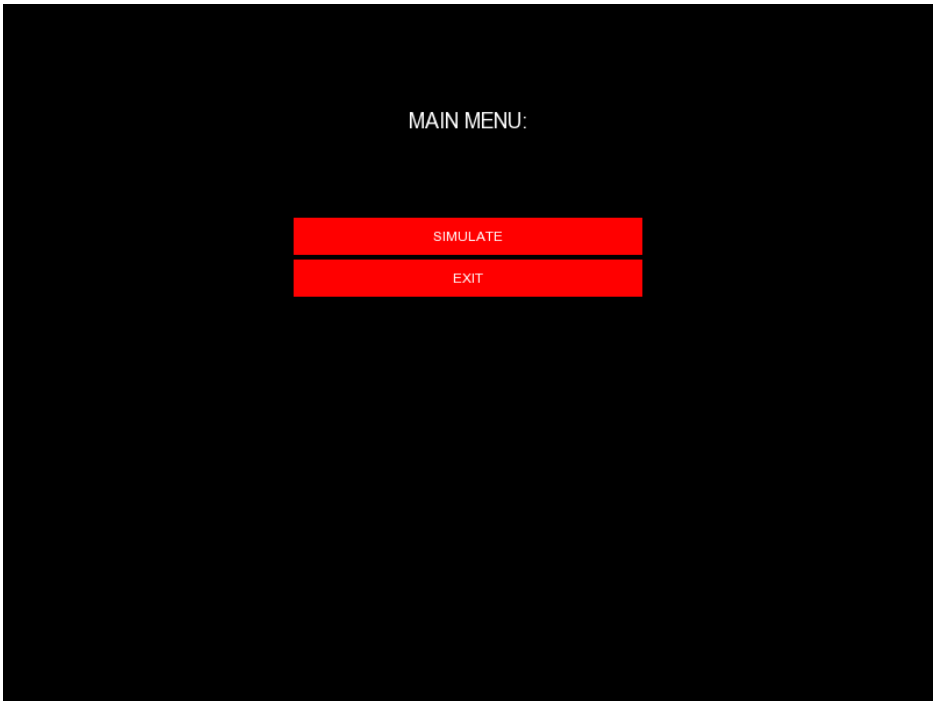

рис. 10 – стан головного меню.

У стані симуляції відбувається уся головна логіка програми: рендеринг об’єктів, що рухаються по фізичним законам, обрахунок колізії між об’єктами, вирішення колізій (реакція на них, або вирішення обмежень).

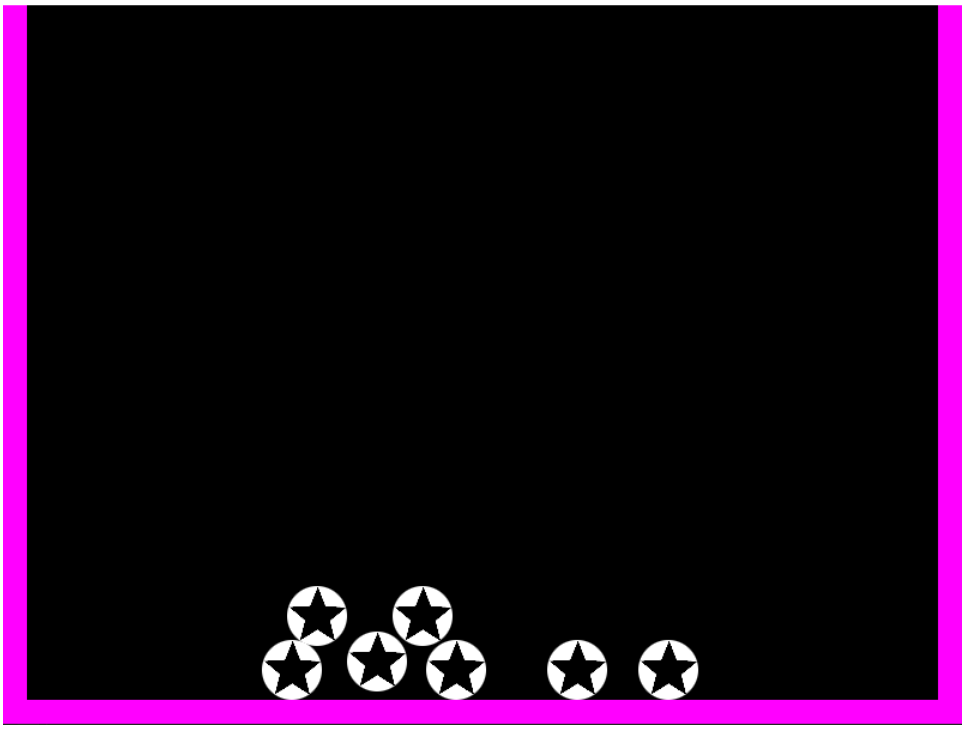

рис. 11 – стан симуляції.

Симуляцію можна поставити на паузу. Архітектура розроблена таким чином що може існувати змішення між станами, таким чином стан паузи рендериться поверх стану симуляції. Завдяки цьому об’єкти, які були в стані симуляції, так само можемо побачити і в стані паузи. Те саме не тільки може використовуватися для рендерингу, але й для оновлення позицій об’єктів. Все залежить від того як був налаштований той чи інший стан.

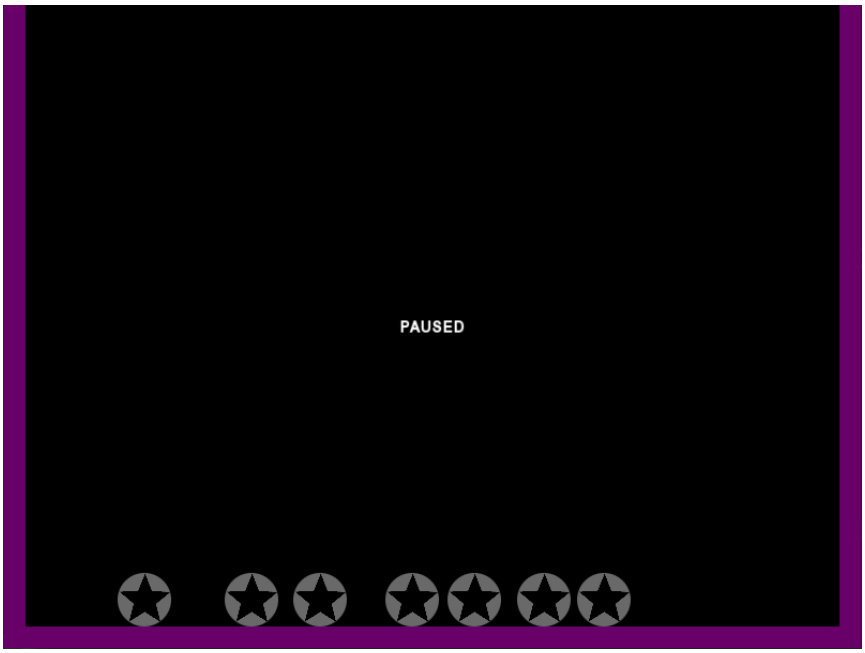

рис. 12 – стан паузи.

Слід зауважити, що в програмі існує глобальний стан, тобто він використовується постійно. До кожного стану можна під’єднати такі вхідні значення як натискання клавішей, миші і так далі. Усі стани окрім глобального матимуть змогу реагувати тільки на свої відповідні події, а глобальний стан може підписатися на подію і виконувати певні функції незалежно від інших станів.

5. Багатопотоковість

Здатність програми використовувати більше одного потоку виконання не є чимось новаторським. В наші часи, здебільшого, проблемою є не збільшення кількості цих потоків, а правильне використання. Треба чітко розуміти що потрібно даній програмі, що виконує якісь конкретні функції. Чи треба якомога пришвидшити її виконання, використовуючи усі дозволені ресурси, або використати найменше ресурсів, але отримати задовільний результат. Це все буде залежати від програмістів, які цю програму пишуть, та їхніх задач. Існує багато способів використання багатопотоковості, і головною метою людини, що збирається писати код для якоїсь системи – є знаходження оптимального рішення.

5.1 Пул потоків

Для прискорення симуляції фізичних ігрових об’єктів було використано пул потоків (thread pool). Нижче наведена схема для полегшення пояснень:

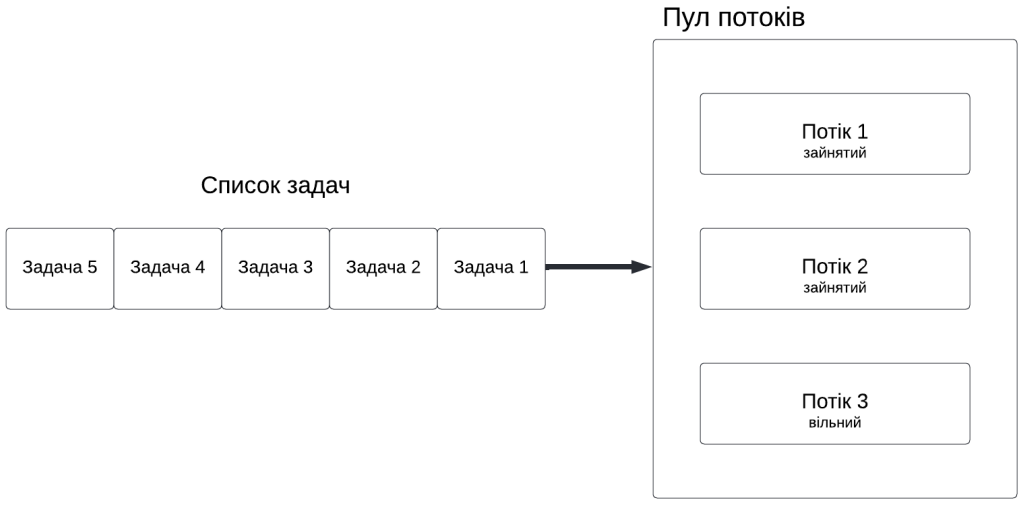

рис. 13 – спрощена схема роботи пула потоків.

При розпаралеленні задач, кожна вхідна задача (функція) буде йти у чергу с задачами, з якої будуть витягуватися задачі одна за одною. Пул потоків містить в собі фіксовану кількість потоків. Зазвичай визначення кількості залежить від характеристик самого комп’ютера та операційної системи. Не варто обирати кількість потоків більшою ніж це може підтримувати система, бо якщо процесор буде дуже часто змінювати свій контекст виконання – це може призвести до погіршення продуктивності.

Кожний вільний потік чекає на вхідну задачу, і як тільки він її отримує, то стає зайнятим. Після того як він виконав задачу, потік знову стає вільним і чекає на нову задачу. Таку поведінку маю усі потоки у пулі потоків.

5.2 Розпаралелення обчислень колізій та реакцій на них

Для дослідження впливу багатоядерності на створену систему потрібно виявити ті частини програми, що можуть бути розпаралеленні. У випадку фізичної симуляціїї найбільш підходящими варінтами є розпаралелення виявлень колізій та вирішення обмежень (при наявності колізії).

Нижче наведено графіки результатів розпаралелення вищезгаданих частин. Слід зауважити, що тестування відбувалося незалежно одне від одного. Тобто змінення кількості потоків для виявлень колізій та вирішення обмежень відбувалось почергово. Спочатку додавалися потоки для однієї частини, а інша весь час виконувалася в одному потоці та навпаки. Пул потоків створюється у програмі один раз і потім у кожній ітерації ігрового циклу сприяє розпаралелюванню вищевказаних задач.

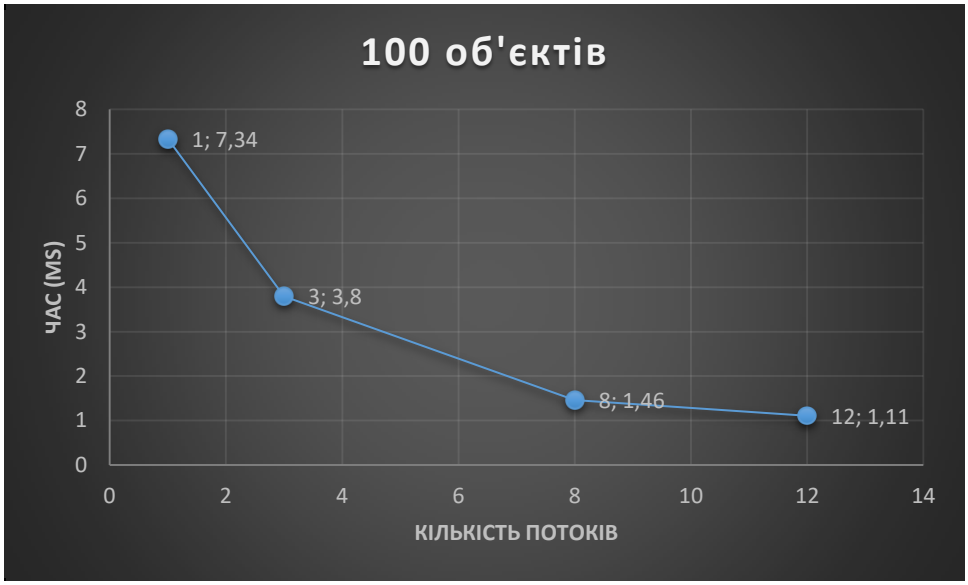

рис. 14 – графік залежності часу та кількості потоків для знаходження колізії між 100 об’єктами.

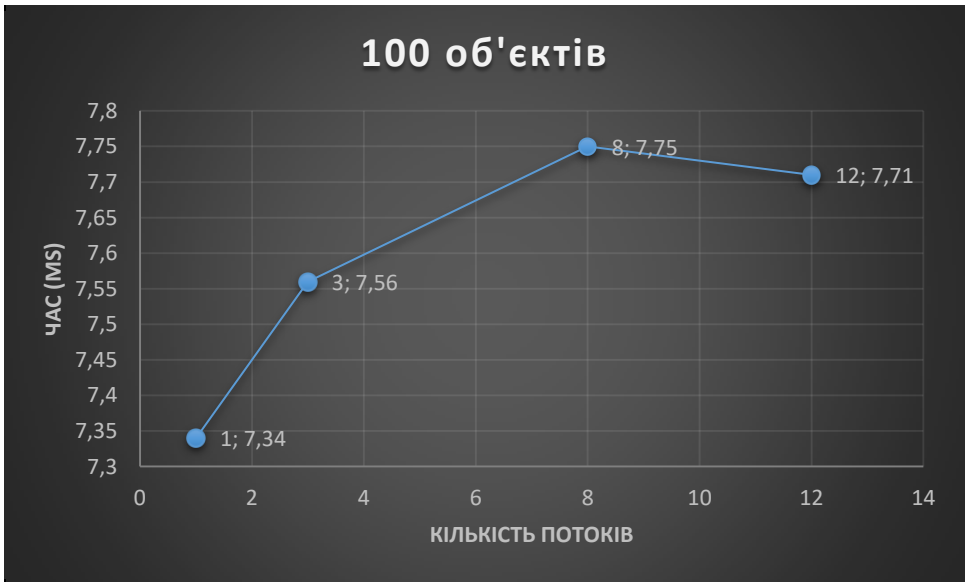

рис. 15 – графік залежності часу та кількості потоків для вирішення обмежень між 100 об’єктами.

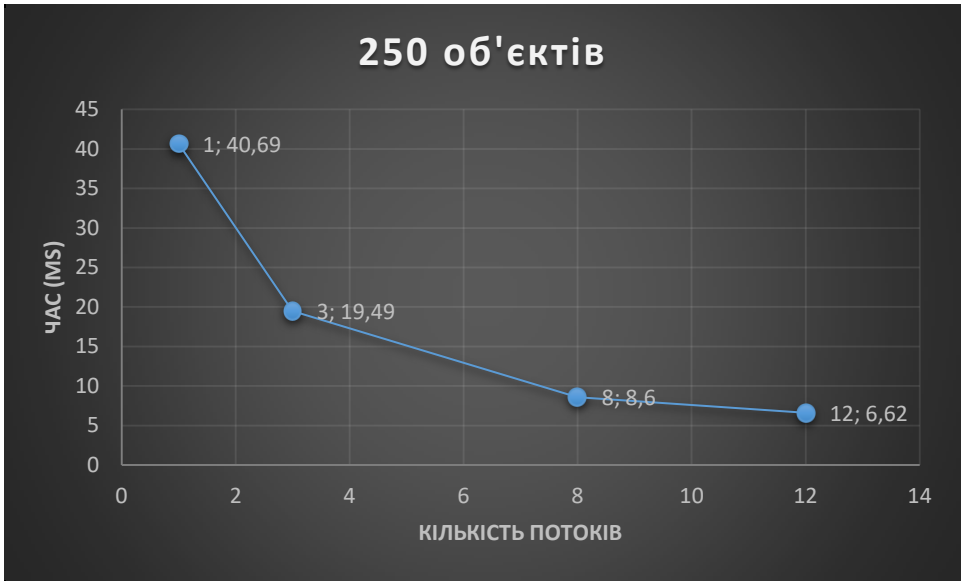

рис. 16 – графік залежності часу та кількості потоків для знаходження колізії між 250 об’єктами.

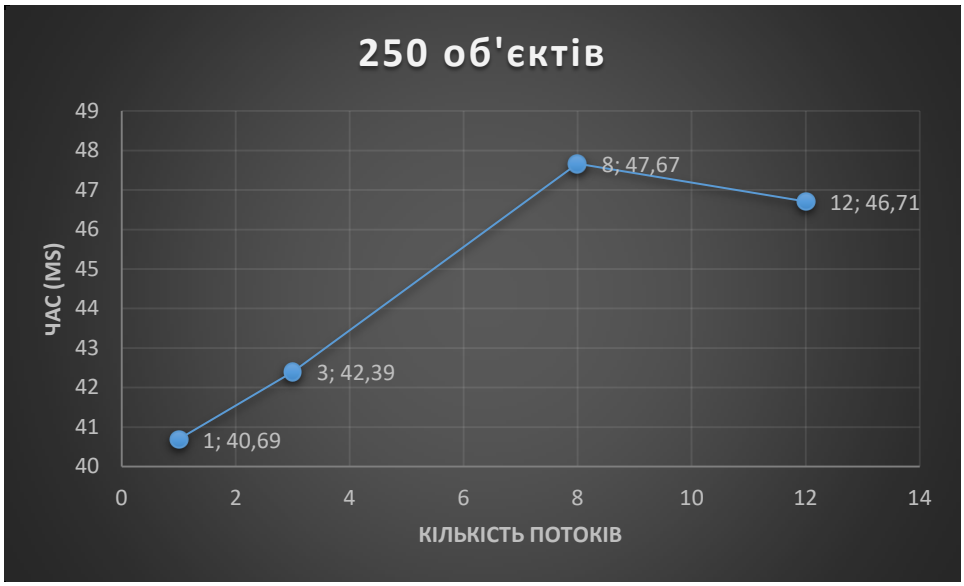

рис. 17 – графік залежності часу та кількості потоків для вирішення обмежень між 250 об’єктами.

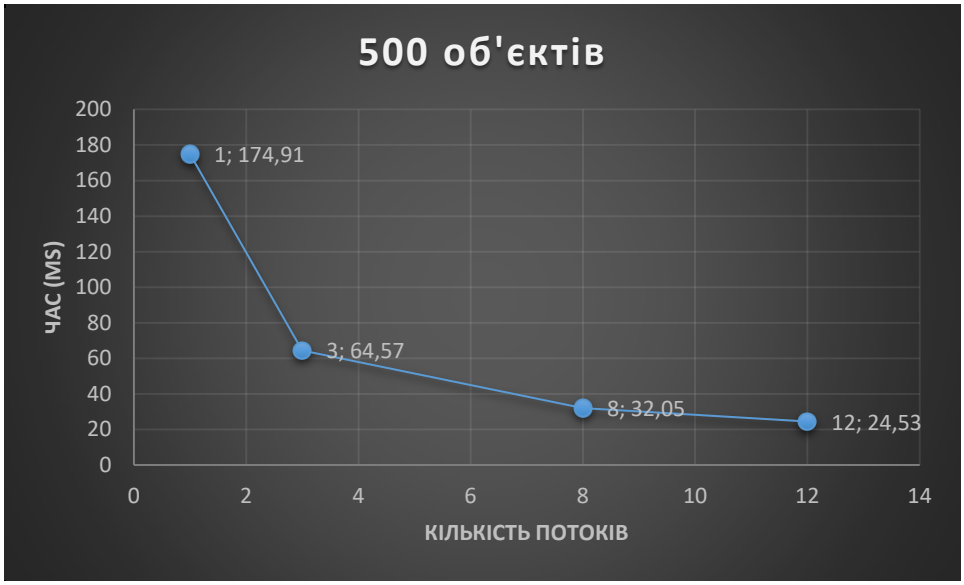

рис. 18 – графік залежності часу та кількості потоків для знаходження колізії між 500 об’єктами.

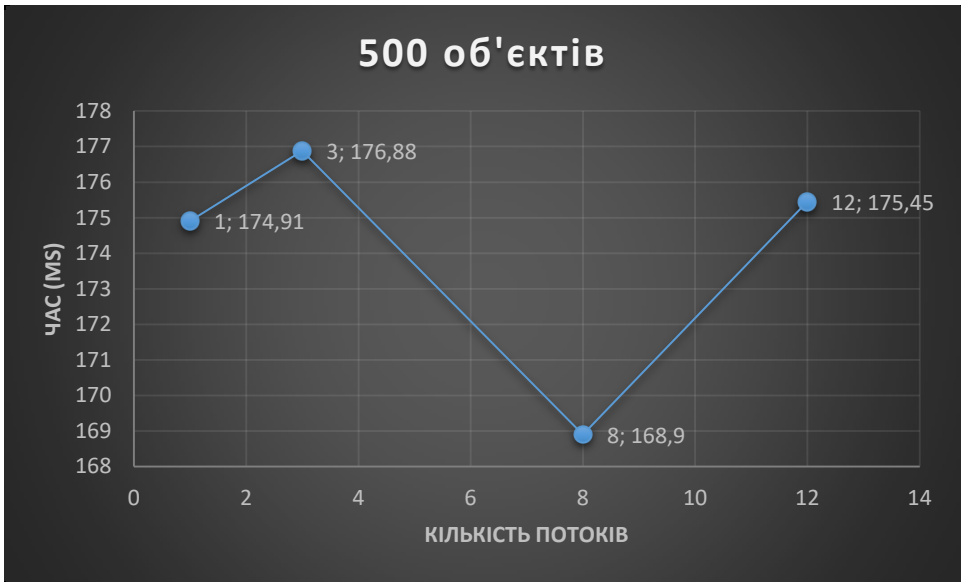

рис. 19 – графік залежності часу та кількості потоків для вирішення обмежень між 500 об’єктами.

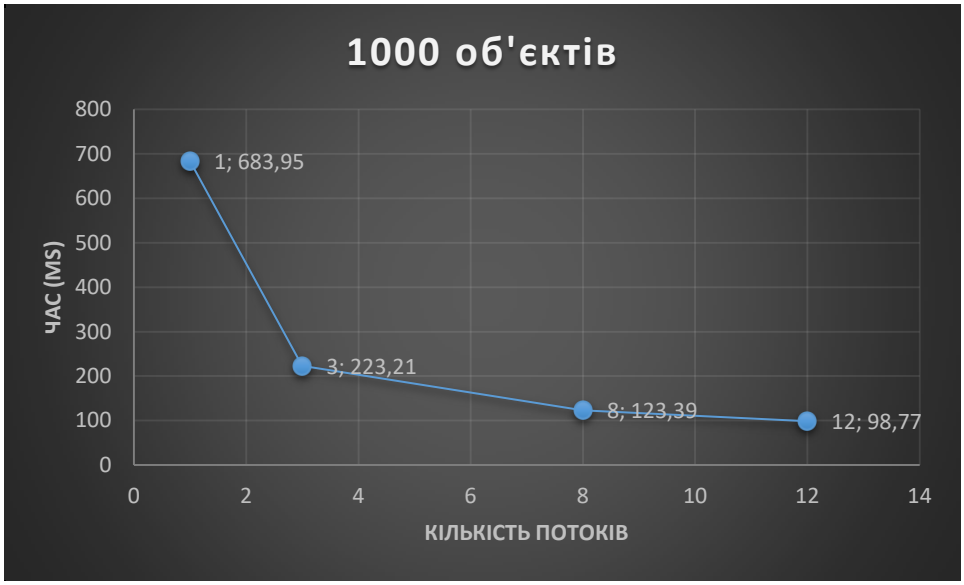

рис. 20 – графік залежності часу та кількості потоків для знаходження колізії між 1000 об’єктами.

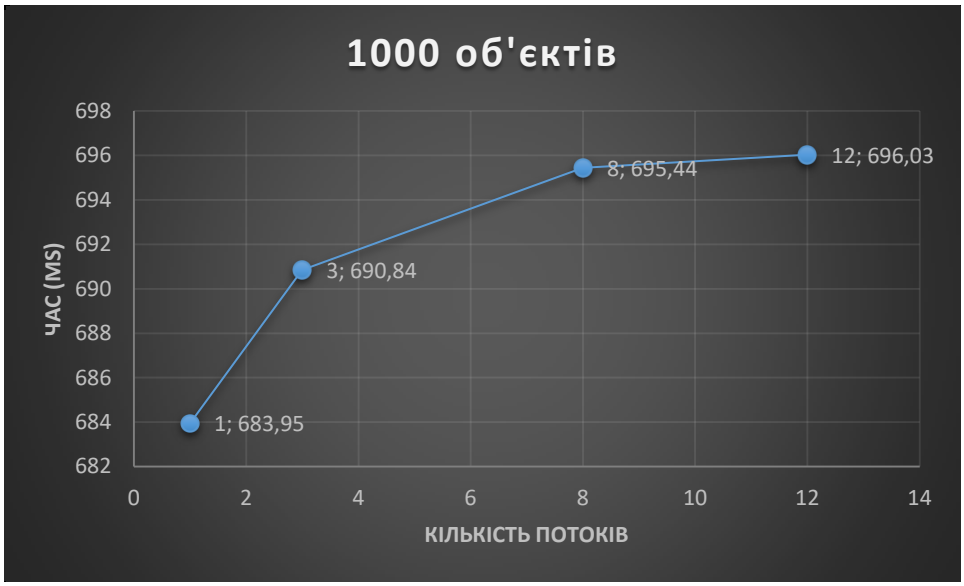

рис. 21 – графік залежності часу та кількості потоків для вирішення обмежень між 1000 об’єктами.

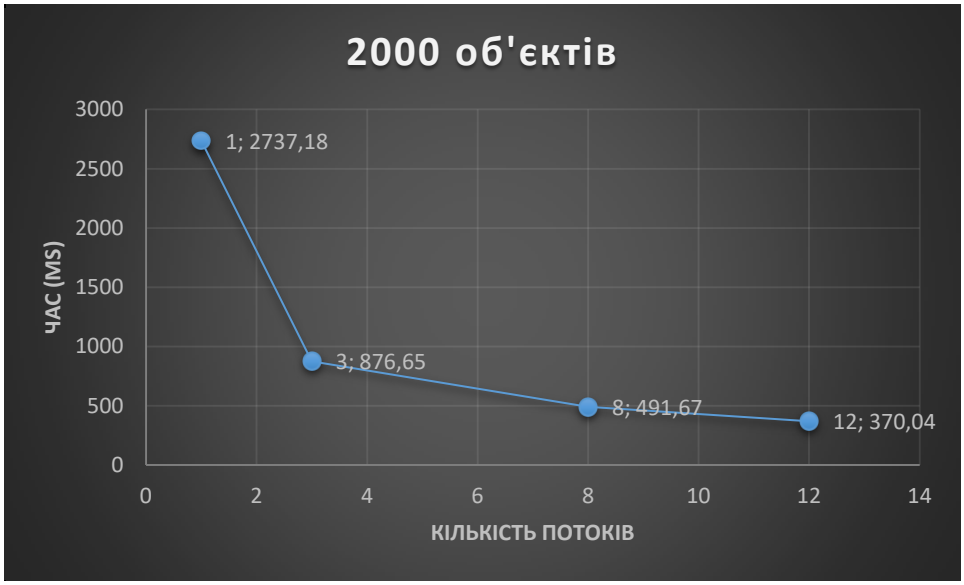

рис. 22 – графік залежності часу та кількості потоків для знаходження колізії між 2000 об’єктами.

рис. 23 – графік залежності часу та кількості потоків для вирішення обмежень між 2000 об’єктами.

Отже, можна сказати, що при зміні кількості задіяних потоків та використаних об’єктів можна як покращити, так і погіршити продуктивність програми. Зробивши правильні висновки з’явиться можливість якомога раціональніше використовувати як людські так комп’ютерні ресурси.

ВИСНОВКИ

Обрано зручні та ефективні інструменти для розробки програми, що виконує симуляцію: система збірки, графічні та математичні бібліотеки, стандартна бібліотека для роботи з мовою програмування C++, що дозволить використовувати зручні структури даних та готові алгоритми для написання більш надійної бази коду.

Було досліджено принципи симуляції твердих об’єктів. Пояснено принцип роботи фізичного двигуна з допомогою розбиття його роботи на такі етапи симуляції: застосування сил, що діють на тіло; оновлення позицій і кутів повороту тіл після застосування сил; обрахунок можливих колізій; визначення реакцій об’єктів після підтвердження колізії. Також виявлено необхідні фізичні параметри для побудови правил поведінки об’єктів. Опрацьовано методики розрахунку колізій між різними формами об’єктів: коло та коло; прямокутник та прямокутник; коло та прямокутник.

Написано програму, що має структуру з різними станами: інтро, головне меню, симуляція, пауза. Завдяки цьому підтримка такої програми стане легшою. Виконано велику частину дослідження впливу багатопотоковості на фізичну симуляцію. Пояснено та застосовано у програмі пула потоків, завдяки чому було здійснено розпаралелення виявлення колізій та вирішення обмежень об'єктів. Перевірено різні можливі ситуаціїі при використанні багатопотоковості у процесі фізичної симуляції та їх вплив на продуктивність виконання програми.

З результатів виконання можна побачити що найефективнішим є розпаралелення виявлення колізій. Це можна пояснити тим, що ця стадія містить лише математичні перевірки та не змінює стан самого об’єкту. Завдяки чому непотрібно синхронізувати критичні секції (секції, які вимагають ексклюзивного доступу до спільних ресурсів, таких як змінні, об'єкти або файли, у мультипроцесорних або багатопотокових середовищах).

При обчисленнях, що йдуть на виявлення колізій, цих секцій немає. Таким чином у вищенаведених графіках можемо побачити прискорення у 3-7 разів.

Щодо розпаралелення вирішення обмежень, то в цьому випадку ситуація є набагато гіршою, адже присутні критичні секції, що сповільнює роботу програми. Очевидної продуктивності за наданих результатів в цьому випадку немає, але можна зробити висновок, що така оптимізація потребує набагато більше зусиль, ресурсів та знань програміста.

Отже, оптимізуючи роботу фізичної симуляції завдяки багатопотоковості треба звертати увагу на кількість критичних секцій, об’єктів на сцені та доступних потоків. Потрібно мати баланс між швидкістю та використанням ресурсів. Беручи до уваги усі нюанси, можна отримати найоптимальніший варіант, який буде влаштовувати розробника.

Список використаних джерел

1. Premake. URL: [https://premake.github.io/docs/What-Is-Premake]
2. Need of build systems. URL: [https://blog.feabhas.com/2021/06/why-we-need-build-systems/]
3. STL overview. URL: [https://www.geeksforgeeks.org/the-c-standard-template-library-stl/]
4. SFML overview. URL: [https://www.sfml-dev.org/index.php]
5. GLM overview. URL: [https://github.com/g-truc/glm]
6. Вступ до динаміки твердого тіла. URL: [https://www.toptal.com/game/video-game-physics-part-i-an-introduction-to-rigid-body-dynamics]
7. Детекція колізії для твердих об’єктів. URL: [https://www.toptal.com/game/video-game-physics-part-ii-collision-detection-for-solid-objects]
8. Обмежена симуляція твердого тіла. URL: [https://www.toptal.com/game/video-game-physics-part-iii-constrained-rigid-body-simulation]
9. Детекція колізії між колом та прямокутником. URL: [https://learnopengl.com/In-Practice/2D-Game/Collisions/Collision-detection]
10. Колізія кола з колом. URL: [https://www.jeffreythompson.org/collision-detection/circle-circle.php]
11. Колізія прямокутника з прямокутником. URL: [https://www.jeffreythompson.org/collision-detection/rect-rect.php]
12. Raimondas Pupius. SFML Game Development By Example.
13. Gabor Szauer. Game Physics Cookbook.
14. Anthony Williams. C++ Concurrency in Action.
15. Bjarne Stroustrup. The C++ Programming language, fourth edition.

Додаток А

Код програми для фізичної поведінки об’єктів

```
static BS::thread_pool threadPool(std::thread::hardware_concurrency());

static std::mutex collisionMutex;


PhysicsWorld::~PhysicsWorld()
{
	for (SolverBase* solver : m_solvers)
	{
		delete solver;
	}
}

void PhysicsWorld::AddSolver(SolverBase* const object)
{
	m_solvers.emplace_back(object);
}

void PhysicsWorld::Step(const float deltaTime)
{
	if (m_objectsRef.size() == 0)
	{
		//m_SimulationTime.restart();
		return;
	}

	ApplyForcesAndUpdatePositionAndVelocity(deltaTime);

	// DetectCollisions(); // single thread

	DetectCollisionsParallelForTests();

	SolveConstraints(m_collisions, deltaTime);

	// SolveConstraintsParallel(deltaTime);

	m_collisions.clear();
}

void PhysicsWorld::ApplyForcesAndUpdatePositionAndVelocity(const float deltaTime)
{
	const float damping = 0.98f; // simulates air friction
	for (Object* const obj : m_objectsRef)
	{
		if (obj->IsStatic())
		{
```

```
                continue;
            }

            obj->AddToNetForce(obj->GetMass() * m_gravity); // gravity force component

            const glm::vec2 acceleration = obj->GetNetForce() * obj->GetInvMass();
            obj->AddVelocity(acceleration * deltaTime);

            obj->ApplyLinearDamping(damping);
            const glm::vec2 offset = obj->GetVelocity() * deltaTime;
            obj->AddToPosition(offset);

            obj->SetNetForce({ 0.0f, 0.0f });
        }
}

void PhysicsWorld::DetectCollisions()
{
        for (Object* const a : m_objectsRef)
        {
            if (!a->GetCollider())
            {
                continue;
            }

            for (Object* const b : m_objectsRef)
            {
                if (a == b)
                {
                    break; // same object
                }

                if (!b->GetCollider())
                {
                    continue; // no collider present
                }

                if (a->IsStatic() && b->IsStatic())
                {
                    continue;
                }

                CollisionManifold manifold = a->GetCollider()->Collides(*b-
>GetCollider());
                if (manifold.IsColliding)
                {
                    m_collisions.emplace_back(Collision{ a, b, manifold });
                }
            }
        }
```

```
}

void PhysicsWorld::DetectCollisionsForThread(size_t startIndex, size_t endIndex)
{
	for (size_t i = startIndex; i < endIndex; ++i)
	{
		if (!m_objectsRef[i]->GetCollider())
		{
			continue;
		}

		for (size_t j = i; j < endIndex; ++j)
		{
			if (!m_objectsRef[j]->GetCollider())
			{
				continue;
			}

			if (m_objectsRef[i]->IsStatic() && m_objectsRef[j]->IsStatic())
			{
				continue;
			}

			CollisionManifold manifold = m_objectsRef[i]->GetCollider()-
>Collides(*m_objectsRef[j]->GetCollider());
			if (manifold.IsColliding)
			{
				std::unique_lock lock(collisionMutex);
				m_collisions.emplace_back(Collision{ m_objectsRef[i],
m_objectsRef[j], manifold });
			}
		}
	}
}

void PhysicsWorld::DetectCollisionsParallelForTests()
{
	const size_t numberOfThreads = threadPool.get_thread_count();
	const size_t step = m_objectsRef.size() / numberOfThreads;
	size_t startIndex = 0;
	for (size_t i = 0; i < numberOfThreads; ++i)
	{
		size_t endIndex = ((i == (numberOfThreads - 1)) ? m_objectsRef.size() :
startIndex + step);
		threadPool.push_task(&PhysicsWorld::DetectCollisionsForThread, this,
startIndex, endIndex);
		startIndex += step;
	}

	threadPool.wait_for_tasks();
```

```
}

void PhysicsWorld::SolveConstraints(const std::vector<Collision>& collisions, const float
deltaTime)
{
	for (SolverBase* solver : m_solvers)
	{
		solver->Solve(collisions, deltaTime);
	}
}

void PhysicsWorld::SolveConstraintsParallel(float deltaTime)
{
	const int blockSize = std::thread::hardware_concurrency();
	const int elementsPerThread = m_collisions.size() / blockSize;
	if (elementsPerThread == 0)
	{
		SolveConstraints(m_collisions, deltaTime);
		return;
	}

	std::vector<std::vector<Collision>> collisionBuckets;
	for (size_t i = 0; i < m_collisions.size(); i += elementsPerThread)
	{
		auto first = m_collisions.begin() + i;
		auto last = (i + elementsPerThread < m_collisions.size()) ? first +
elementsPerThread : m_collisions.end();

		collisionBuckets.emplace_back(first, last);
	}

	for (int i = 0; i < collisionBuckets.size(); ++i)
	{
		threadPool.push_task(&PhysicsWorld::SolveConstraints, this,
collisionBuckets[i], deltaTime);
	}

	threadPool.wait_for_tasks();
}


void PositionSolver::Solve(const std::vector<Collision>& collisions, float deltaTime)
{
	//  The linear projection value indicates how much positional correction to apply. A
	//  smaller value will allow objects to penetrate more.Try to keep the value of this
	//		variable between 0.2 and 0.8
	const float LinearProjectionPercent = 0.8f;  // Determines how much to allow objects to
penetrate.This helps avoid jitter;

	//  The PenetrationSlack the larger this number, the less jitter we have in the system.Keep
	//		the value between 0.01 and 0.1
```

```
	const float PenetrationSlack = 0.1f;

	for (const Collision& collision : collisions)
	{
		const float invMassA = (collision.A->IsDynamic() ? collision.A->GetInvMass() :
0.0f);
		const float invMassB = (collision.B->IsDynamic() ? collision.B->GetInvMass() :
0.0f);
		const float invMassSum = invMassA + invMassB;
		if (invMassSum == 0.0f)
		{
			continue;
		}

		const float depth = std::fmaxf(collision.Manifold.PenetrationDepth -
PenetrationSlack, 0.0f);
		const float scalar = depth / invMassSum;
		const glm::vec2 correction = collision.Manifold.normal * scalar *
LinearProjectionPercent;

		{
			if (collision.A->IsDynamic())
			{
				std::unique_lock lock(collision.A->internalMutex);
				collision.A->AddToPosition(correction * invMassA);
			}

			if (collision.B->IsDynamic())
			{
				std::unique_lock lock(collision.B->internalMutex);
				collision.B->AddToPosition(-correction * invMassB);
			}
		}
	}
}



void ImpulseSolver::Solve(const std::vector<Collision>& collisions, float deltaTime)
{
	for (const Collision& collision : collisions)
	{
		const float invMassA = (collision.A->IsDynamic() ? collision.A->GetInvMass() :
0.0f);
		const float invMassB = (collision.B->IsDynamic() ? collision.B->GetInvMass() :
0.0f);
		const float invMassSum = invMassA + invMassB;
		if (invMassSum == 0.0f)
		{
			continue;
```

```
		}

		glm::vec2 velocityA;
		glm::vec2 velocityB;

		{
			//std::unique_lock lock(impulseMutex);

			{
				std::unique_lock lock(collision.A->internalMutex);
				velocityA = (collision.A->IsDynamic() ? collision.A->GetVelocity() : glm::vec2());
			}

			{
				std::unique_lock lock(collision.B->internalMutex);
				velocityB = (collision.B->IsDynamic() ? collision.B->GetVelocity() : glm::vec2());
			}
		}

		glm::vec2 relativeVelocity = velocityB - velocityA;
		glm::vec2 relativeNormal = collision.Manifold.normal;
		float directionMagnitude = glm::dot(relativeVelocity, relativeNormal);
		if (directionMagnitude <= 0.0f)
		{
			continue; // Moving away from each other? Do nothing!
		}

		const float e = std::fminf(collision.A->GetRestitution(), collision.B->GetRestitution());
		float numerator = (-(1.0f + e) * directionMagnitude);
		float j = numerator / invMassSum;
		if (collision.Manifold.contacts.size() > 0 && j != 0.0f)
		{
			j /= static_cast<float>(collision.Manifold.contacts.size());
		}

		// Linear Impulse
		const glm::vec2 impulse = relativeNormal * j;

		{
			//std::unique_lock lock(impulseMutex);

			if (collision.A->IsDynamic())
			{
				std::unique_lock lock(collision.A->internalMutex);
				collision.A->AddVelocity(-impulse * invMassA);
			}
```

```
                if (collision.B->IsDynamic())
                {
                        std::unique_lock lock(collision.B->internalMutex);
                        collision.B->AddVelocity(impulse * invMassB);
                }
        }

        //Friction
        {
                //std::unique_lock lock(impulseMutex);

                {
                        std::unique_lock lock(collision.A->internalMutex);
                        velocityA = (collision.A->IsDynamic() ? collision.A-
>GetVelocity() : glm::vec2());
                }

                {
                        std::unique_lock lock(collision.B->internalMutex);
                        velocityB = (collision.B->IsDynamic() ? collision.B-
>GetVelocity() : glm::vec2());
                }
        }

        relativeVelocity = velocityB - velocityA;
        relativeNormal = collision.Manifold.normal;
        directionMagnitude = glm::dot(relativeVelocity, relativeNormal);

        const glm::vec2 t = relativeVelocity - (relativeNormal * directionMagnitude);
        if (CMP(glm::length(t), 0.0f))
        {
                continue;
        }
        const glm::vec2 tNormalized = glm::normalize(t);

        numerator = -glm::dot(relativeVelocity, tNormalized);
        float jt = numerator / invMassSum;
        if (collision.Manifold.contacts.size() > 0 && jt != 0.0f)
        {
                jt /= static_cast<float>(collision.Manifold.contacts.size());
        }

        if (CMP(jt, 0.0f))
        {
                continue;
        }

        const float friction = std::sqrtf(collision.A->GetFriction() * collision.B-
>GetFriction());
        if (jt > j * friction)
```

```
                {
                        jt = j * friction;
                }
                else if (jt < -j * friction)
                {
                        jt = -j * friction;
                }

                const glm::vec2 tangentImpulse = tNormalized * jt;

                {
                        //std::unique_lock lock(impulseMutex);

                        if (collision.A->IsDynamic())
                        {
                                std::unique_lock lock(collision.A->internalMutex);
                                collision.A->AddVelocity(tangentImpulse * invMassA);
                        }

                        if (collision.B->IsDynamic())
                        {
                                std::unique_lock lock(collision.B->internalMutex);
                                collision.B->AddVelocity(-tangentImpulse * invMassB);
                        }
                }
        }
}



CollisionManifold CollisionUtility::GetCollisionManifold(const RectangleCollider& rectangle,
const CircleCollider& circle)
{
        CollisionManifold result;

        const float halfWidth = (rectangle.Max.x - rectangle.Min.x) / 2;
        const float halfHeight = (rectangle.Max.y - rectangle.Min.y) / 2;

        const glm::vec2 absolutDistance = circle.Position - rectangle.Position;
        const float clampedX = std::clamp(absolutDistance.x, -halfWidth, halfWidth);
        const float clampedY = std::clamp(absolutDistance.y, -halfHeight, halfHeight);
        const glm::vec2 closestPoint = rectangle.Position + glm::vec2{ clampedX , clampedY };

        const glm::vec2 distanceVec = circle.Position - closestPoint;
        const float distance = glm::length(distanceVec);

        if (distance > circle.Radius)
        {
                return result;
        }
```

```
	glm::vec2 normal;
	if (CMP(distance, 0.0f))
	{
		const glm::vec2 revertedDistanceVec = closestPoint - rectangle.Position;
		const float newDistance = glm::length(revertedDistanceVec);
		if (CMP(newDistance, 0.0f))
		{
			return result;
		}

		normal = glm::normalize(revertedDistanceVec);
	}
	else
	{
		normal = glm::normalize(distanceVec);
	}

	const glm::vec2 outsidePoint = circle.Position - normal * circle.Radius;
	const float doublePenetrationDistance = glm::length(closestPoint - outsidePoint);

	result.IsColliding = true;
	result.contacts.emplace_back(closestPoint + (outsidePoint - closestPoint) * 0.5f);
	result.normal = normal;
	result.PenetrationDepth = doublePenetrationDistance * 0.5f;

	return result;
}

CollisionManifold CollisionUtility::GetCollisionManifold(const CircleCollider& circle1, const
CircleCollider& circle2)
{
	CollisionManifold result;

	const glm::vec2 distanceVec = (circle2.Position - circle1.Position);
	const float combinedRadius = circle1.Radius + circle2.Radius;
	const float distance = glm::length(distanceVec);
	if (distance > combinedRadius || distance == 0.0f)
	{
		return result;
	}

	const glm::vec2 normal = glm::normalize(distanceVec);

	result.normal = normal;
	result.IsColliding = true;
	result.PenetrationDepth = std::fabsf(distance - combinedRadius) * 0.5f;

	const float distanceToIntersectionPoint = circle1.Radius - result.PenetrationDepth;
```

```
        const glm::vec2 contactPoint = circle1.Position + result.normal *
distanceToIntersectionPoint;
        result.contacts.emplace_back(contactPoint);

        return result;
}
```