

**ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ**

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка та реалізація фізичного рушія на мові
програмування C++»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

*Кваліфікаційна робота містить результати власних досліджень. Використання
ідей, результатів і текстів інших авторів мають посилання
на відповідне джерело*

_____ Олександр ЄФИМОВ
(підпис)

Виконав: здобувач вищої освіти групи ПД-41

_____ Олександр ЄФИМОВ

Керівник: _____ Андрій КОЛОДЮК
асистент кафедри

Рецензент: _____

**ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ**
Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри

Інженерії програмного забезпечення

_____ Ірина ЗАМРІЙ

« ____ » _____ 2026 р.

**ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ**

Єфимову Олександрю Євгеновичу

1. Тема кваліфікаційної роботи: «Розробка та реалізація фізичного рушія на мові програмування C++»

керівник кваліфікаційної роботи асистент кафедри ІПЗ Андрій КОЛОДЮК,
затверджені наказом Державного університету інформаційно-комунікаційних
технологій від «20» лютого 2026 року № 51.

2. Строк подання кваліфікаційної роботи «29» травня 2026 року.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про моделювання руху твердих тіл у двовимірному просторі, методи чисельного інтегрування, виявлення та обробку колізій, імпульсне вирішення зіткнень, моделювання сили тяжіння, тертя та пружності, матеріали щодо архітектури фізичних рушіїв і побудови інтерактивних демонстраційних застосунків, технічна документація бібліотек, використаних для розробки та реалізації фізичного рушія.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Аналіз предметної галузі та сучасних підходів до реалізації двовимірної фізичної симуляції.

2. Визначення вимог та проектування архітектури 2D фізичного рушія та інтерактивного демонстраційного застосунку.
 3. Розроблення програмних модулів фізичного рушія для оновлення стану тіл, виявлення контактів між об'єктами та обробки їх взаємодії.
 4. Реалізація інтерактивного демонстраційного застосунку для створення сцен, редагування параметрів тіл і візуального аналізу роботи рушія.
 4. Тестування програмного забезпечення та аналіз результатів роботи фізичного рушія в демонстраційних сценах.
5. Перелік графічного матеріалу: *презентація*
1. Аналіз аналогів.
 2. Вимоги до програмного забезпечення.
 3. Програмні засоби реалізації.
 4. Діаграма варіантів використання.
 5. Схема функціонування програми.
 6. Діаграма діяльності роботи демонстраційного застосунку.
 7. Діаграма діяльності фізичного кроку рушія.
 8. Діаграма класів.
 9. Екранні форми.
 10. Тестування застосунку.
 11. Апробація результатів дослідження.
6. Дата видачі завдання «23» лютого 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	23.02-28.02.2026	
2	Аналіз та дослідження існуючих аналогів	01.03-07.03.2026	
3	Огляд сучасних підходів до розробки фізичних рушіїв та інтерактивних testbed-застосунків	25.03-05.04.2026	
4	Проектування архітектури 2D фізичного рушія та інтерактивного демонстраційного застосунку	10.04-15.04.2026	
5	Програмна реалізація застосунку	16.04-01.05.2026	
6	Тестування застосунку	02.05-06.05.2026	
7	Оформлення роботи: вступ, висновки, реферат	07.05-09.05.2026	
8	Розробка демонстраційних матеріалів	10.05-11.05.2026	
9	Попередній захист роботи	11.05-22.05.2026	

Здобувач вищої освіти

(підпис)

Олександр ЄФИМОВ

Керівник

кваліфікаційної роботи

(підпис)

Андрій КОЛОДЮК

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 69 стор., 4 табл., 10 рис., 25 джерел.

Мета роботи – розроблення, програмна реалізація та перевірка навчально-орієнтованого 2D фізичного рушія Enjoyer2D мовою C++ з демонстраційним застосунком для візуального дослідження руху, зіткнень, тертя та пружності твердих тіл.

Об'єкт дослідження – процес симуляції фізичних взаємодій твердих тіл у середовищі реального часу.

Предмет дослідження – методи, алгоритми та програмні засоби реалізації фізичного рушія на C++.

Короткий зміст роботи: У роботі проаналізовано методи та підходи до моделювання фізичних процесів у двовимірному просторі, проаналізовано існуючі 2D фізичні рушії та демонстраційні середовища, зокрема Box2D, Chipmunk2D і Matter.js. Розроблено архітектуру власного рушія та програмно реалізовано його ключові функціональні можливості. Реалізовано інтерактивний демонстраційний застосунок для взаємодії з фізичними сценами, а також для відображення debug-візуалізації й параметрів симуляції. В роботі використано засоби мови C++ та бібліотеки для реалізації математичних обчислень, графічного відображення, користувацького інтерфейсу, тестування й серіалізації даних.

Сферою використання застосунку є навчальне дослідження принципів фізичного моделювання, створення інтерактивних демонстрацій, а також використання рушія як основи для простих ігрових або симуляційних програм.

КЛЮЧОВІ СЛОВА: 2D ФІЗИЧНИЙ РУШІЙ, ФІЗИЧНА СИМУЛЯЦІЯ, ТВЕРДІ ТІЛА, C++, ЧИСЕЛЬНЕ ІНТЕГРУВАННЯ, КОЛІЗІЇ, AABB, SDL3, ІМПУЛЬСНЕ ВИРІШЕННЯ ЗІТКНЕНЬ

ЗМІСТ

ВСТУП	9
1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ПОСТАНОВКА ЗАДАЧІ.....	12
1.1 Поняття фізичного рушія та його роль у програмному забезпеченні	12
1.2 Особливості моделювання твердих тіл у двовимірному просторі	14
1.3 Методи чисельного інтегрування руху фізичних тіл	16
1.4 Виявлення колізій та обробка зіткнень	18
1.5 Огляд існуючих 2D фізичних рушіїв.....	21
1.5.1 Box2D.....	21
1.5.2 Chipmunk2D	23
1.5.3 Matter.js	25
1.6 Порівняльний аналіз аналогів і розроблюваного рішення.....	27
1.7 Визначення вимог до програмного забезпечення	29
1.8 Висновки за результатами аналізу предметної галузі	30
2 ОГЛЯД ЗАСОБІВ РЕАЛІЗАЦІЇ	32
2.1 Мова програмування	32
2.2 Середовища розробки	34
2.3 Система збирання	35
2.4 Бібліотеки	37
2.4.1 GLM	37
2.4.2 SDL3 та SDL_Renderer	38
2.4.3 Dear ImGui	39
2.4.4 Catch2	40
2.4.5 nlohmann/json	42
3 ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	43
3.1 Загальна архітектура	43
3.2 Архітектура фізичного рушія.....	45
3.3 Архітектура демонстраційного застосунку	47
3.4 Структура фізичного світу та об'єктів симуляції	49
3.5 Діаграма варіантів використання.....	50
3.6 Схема функціонування програми	51

3.7 Діаграма діяльності демонстраційного застосунку	52
3.8 Діаграма діяльності фізичного кроку рушія	54
3.9 Діаграма класів	56
4 РЕАЛІЗАЦІЯ 2D ФІЗИЧНОГО РУШІЯ	59
4.1 Реалізація фізичного тіла та його параметрів.....	59
4.2 Реалізація форми об'єктів	60
4.3 Реалізація фізичного світу та кроку симуляції.....	60
4.4 Реалізація чисельного інтегрування	61
4.5 Реалізація виявлення колізій	61
4.6 Реалізація контактів і debug-даних	62
4.7 Реалізація імпульсного вирішення зіткнень	62
4.8 Урахування тертя та пружності.....	63
5 РЕАЛІЗАЦІЯ ДЕМОНСТРАЦІЙНОГО ЗАСТОСУНКУ	64
5.1 Структура демонстраційного застосунку	64
5.2 Реалізація вікна, рендерингу та обробки введення.....	65
5.3 Реалізація редактора фізичних тіл і параметрів сцени	65
5.4 Реалізація debug-візуалізації	66
5.5 Реалізація демонстраційних сцен	67
5.6 Збереження та завантаження сцен у форматі JSON.....	68
6 ТЕСТУВАННЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ	69
6.1 Модульне тестування фізичного рушія	69
6.2 Функціональне тестування демонстраційного застосунку	70
6.3 Перевірка демонстраційних сцен	72
6.4 Оцінювання продуктивності роботи рушія	73
6.5 Обмеження реалізованого рушія	74
ВИСНОВКИ	76
ПЕРЕЛІК ПОСИЛАНЬ.....	78
ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ	80
ДОДАТОК Б. ФРАГМЕНТИ РЕАЛІЗАЦІЇ ФІЗИЧНОГО РУШІЯ ENJOYER2D....	88
ДОДАТОК В. ФРАГМЕНТИ РЕАЛІЗАЦІЇ ДЕМОНСТРАЦІЙНОГО ЗАСТОСУНКУ	91

ВСТУП

Актуальність розробки 2D фізичного рушія зумовлена широким використанням фізичного моделювання у комп'ютерних іграх, симуляторах, навчальних системах та інтерактивних застосунках. Фізичні рушії забезпечують моделювання руху об'єктів, дію сил, виявлення колізій, обробку контактів, тертя та пружності, що дозволяє створювати більш наочну й передбачувану поведінку тіл у програмному середовищі.

Існуючі 2D фізичні рушії мають широкі функціональні можливості та можуть використовуватися у реальних програмних продуктах, але їхня внутрішня реалізація часто є складною для початкового вивчення, оскільки такі рушії орієнтовані передусім на практичне застосування, продуктивність і повноту функціоналу. Через це актуальним є створення простішого навчально-орієнтованого фізичного рушія, який дозволяє простежити основні етапи фізичної симуляції: оновлення положення тіл, виявлення колізій, формування контактів і обробку зіткнень.

Розроблення власного фізичного рушія дає змогу глибше зрозуміти принципи моделювання твердих тіл, чисельного інтегрування, виявлення колізій та імпульсного вирішення зіткнень. Поєднання рушія з інтерактивним демонстраційним застосунком (testbed) дозволяє не лише отримати результат симуляції, а й наочно досліджувати вплив маси, швидкості та сторонніх сил на поведінку тіл у двовимірному просторі. Такий програмний засіб може бути корисним як для навчання основам фізичного моделювання, так і для подальшого використання як основи простих ігрових або симуляційних програм.

Об'єктом дослідження є процес симуляції фізичних взаємодій твердих тіл у середовищі реального часу.

Предметом дослідження є методи, алгоритми та програмні засоби реалізації фізичного рушія на C++.

Метою роботи є розроблення, програмна реалізація та перевірка навчально-орієнтованого 2D фізичного рушія Enjoyer2D мовою C++ з демонстраційним застосунком для візуального дослідження руху, зіткнень, тертя та пружності твердих тіл.

Завданнями роботи є:

1. Проаналізувати теоретичні відомості про моделювання руху твердих тіл у двовимірному просторі та підходи до побудови фізичних рушіїв.
2. Дослідити існуючі 2D фізичні рушії та визначити їхні переваги, обмеження й придатність для навчального використання.
3. Обґрунтувати вибір програмних засобів, бібліотек і архітектурного підходу для створення навчально-орієнтованого фізичного рушія.
4. Сформувати структуру програмного забезпечення, яка дозволяє наочно демонструвати роботу фізичної симуляції та аналізувати її окремі етапи.
5. Провести перевірку розробленого рішення за допомогою тестування, демонстраційних сцен і debug-візуалізації.

Методи дослідження: першим етапом дослідження було вивчення особливостей фізичного моделювання у двовимірному просторі, зокрема моделювання руху твердих тіл, дії сил, виявлення колізій, обробки контактів, тертя та пружності. Далі проведено огляд існуючих 2D фізичних рушіїв і демонстраційних застосунків з метою визначення їхніх функціональних можливостей, переваг, обмежень та формування вимог до майбутнього програмного забезпечення. На основі сформованих вимог було обрано технологічні засоби для реалізації програмного забезпечення: мову програмування C++ для розробки фізичного рушія, CMake для організації збирання проєкту, GLM для роботи з векторною математикою, SDL3 та SDL_Renderer для створення вікна, обробки подій і 2D-відображення, Dear ImGui для побудови інтерфейсу демонстраційного застосунку, Catch2 для модульного тестування, а також nlohmann/json для збереження та завантаження фізичних сцен. Проєктування програмного забезпечення виконано з використанням UML-діаграм, а реалізацію

здійснено з поділом на рівень фізичного рушія, рівень сцен і рівень демонстраційного застосунку.

Перевірка роботи програмного забезпечення здійснювалася за допомогою модульного тестування окремих компонентів фізичного рушія та експериментальної перевірки в інтерактивному демонстраційному застосунку, а для оцінювання роботи рушія використано демонстраційні сцени, що відображають вільне падіння, вплив коефіцієнту пружності на поведінку тіл, стабільність нагромадження тіл одне на одне, вплив коефіцієнту тертя та роботу рушія в умовах підвищеного навантаження.

Наукова новизна роботи полягає у створенні навчально-орієнтованого 2D фізичного рушія з інтерактивним демонстраційним застосунком, який дозволяє змінювати параметри фізичних тіл і сцени в реальному часі, спостерігати debug-візуалізацію та аналізувати вплив окремих параметрів на поведінку об'єктів. Особливістю роботи є поєднання базових алгоритмів фізичної симуляції з демонстраційним середовищем, орієнтованим на поетапне вивчення внутрішньої будови рушія.

Практична значущість результатів полягає у можливості використання розробленого програмного забезпечення як навчального інструменту для демонстрації основ фізичної симуляції, а також як стартової бази для створення простих двовимірних ігрових або симуляційних програм.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Поняття фізичного рушія та його роль у програмному забезпеченні

Фізичний рушій є програмним компонентом, призначеним для моделювання поведінки об'єктів відповідно до спрощених або наближених законів фізики. У контексті комп'ютерних ігор, симуляторів та інтерактивних застосунків він відповідає за оновлення положення тіл, розрахунок швидкості, обробку сил, виявлення колізій та визначення реакції об'єктів на взаємодію між собою. На відміну від графічної підсистеми, яка лише відображає об'єкти на екрані, фізичний рушій визначає, як ці об'єкти повинні рухатися, падати, зіштовхуватися, ковзати або залишатися нерухомими в межах змодельованого середовища.

У структурі програмного забезпечення фізичний рушій зазвичай виступає окремою підсистемою, яка взаємодіє з іншими частинами застосунку: системою введення, ігровою логікою, рендерингом, редактором сцен або інтерфейсом користувача. Такий поділ дозволяє відокремити фізичну логіку від візуального представлення. Наприклад, рушій може обчислити нове положення тіла після зіткнення, а графічна підсистема лише відобразить це тіло у відповідній позиції. Подібний підхід є важливим для підтримки, тестування та розширення програмного забезпечення, оскільки фізичні алгоритми не залежать безпосередньо від способу виведення зображення на екран [2].

Основними задачами фізичного рушія є моделювання руху тіл, обробка зовнішніх сил, чисельне інтегрування, виявлення колізій та вирішення зіткнень. Моделювання руху передбачає оновлення координат і швидкостей тіл у часі. Для цього використовуються чисельні методи інтегрування, які дають змогу обчислювати новий стан об'єкта на основі його поточного положення, швидкості, маси та прикладених сил. Виявлення колізій визначає, чи перетинаються об'єкти між собою, а вирішення зіткнень відповідає за зміну їхнього стану після контакту.

Саме ці компоненти формують основу більшості фізичних рушіїв реального часу [3].

Фізичні рушії можуть мати різний рівень складності залежно від призначення програмного забезпечення. У складних ігрових або інженерних системах вони можуть підтримувати тривимірну симуляцію, з'єднання, обмеження, м'які тіла, рідини, просторове розбиття та оптимізовані алгоритми широкої фази виявлення колізій. Водночас для навчальних або демонстраційних задач достатньо реалізувати базову модель твердих тіл, прості форми об'єктів, гравітацію, тертя, пружність і базове імпульсне вирішення зіткнень. Такий підхід дозволяє зосередитися не на великій кількості функцій, а на зрозумілості внутрішньої будови рушія.

Окреме значення мають двовимірні фізичні рушії. Вони є простішими за тривимірні, оскільки працюють із координатами на площині, але при цьому дозволяють продемонструвати більшість ключових принципів фізичної симуляції. У 2D-просторі легше візуалізувати положення тіл, напрямки швидкостей, нормалі зіткнень, контактні точки та межі об'єктів, це робить двовимірний фізичний рушій зручним інструментом для навчання, налагодження алгоритмів і створення невеликих ігрових або симуляційних програм.

Роль фізичного рушія у програмному забезпеченні полягає не лише у створенні реалістичної поведінки об'єктів, а й у забезпеченні передбачуваної взаємодії між ними. У комп'ютерних іграх фізика впливає на рух персонажів, падіння предметів, зіткнення, платформену механіку та поведінку оточення, у навчальних застосунках фізичний рушій може використовуватися для демонстрації законів руху, впливу гравітації, маси, швидкості, тертя та пружності, а у симуляційних програмах він дає змогу відтворювати взаємодію об'єктів у контрольованому середовищі та аналізувати результати такої взаємодії [20, 21].

У межах цієї роботи фізичний рушій розглядається як самостійний програмний компонент, що забезпечує моделювання твердих тіл у двовимірному просторі. Його робота поєднується з демонстраційним застосунком, який дозволяє створювати сцени, додавати об'єкти, змінювати їхні параметри та спостерігати результат симуляції в реальному часі. Це дає змогу не лише реалізувати базові

фізичні алгоритми, а й зробити їх наочними для вивчення, перевірки та подальшого розширення.

1.2 Особливості моделювання твердих тіл у двовимірному просторі

Тверде тіло в моделі фізичної симуляції розглядається як об'єкт, форма якого не деформується під час руху або зіткнення з іншими об'єктами. Це означає, що відстані між точками всередині тіла вважаються сталими, а зміна стану об'єкта описується переважно через його положення, швидкість, масу, прикладені сили та параметри взаємодії з іншими тілами. Цей підхід значно спрощує моделювання порівняно з м'якими тілами або рідинами, але водночас дозволяє відтворювати велику кількість практичних ситуацій: падіння предметів, зіткнення, ковзання, нагромадження об'єктів та взаємодію з нерухомими поверхнями [3].

У двовимірному просторі стан твердого тіла описується координатами на площині і найчастіше для цього використовується вектор положення, який містить два компоненти: координату за віссю X та координату за віссю Y . У свою чергу рух тіла визначається вектором швидкості, а зміна швидкості відбувається під дією прискорення, яке може виникати внаслідок сили тяжіння або інших прикладених сил. Завдяки такій моделі фізичний рушій може поетапно оновлювати стан кожного тіла під час симуляції, розраховуючи його нове положення на кожному кроці часу. Важливою особливістю моделювання твердих тіл є поділ об'єктів на динамічні та статичні. Динамічні тіла беруть участь у симуляції повністю: вони можуть рухатися, змінювати швидкість під дією сил, стикатися з іншими об'єктами та реагувати на контакти, а статичні тіла, навпаки, не змінюють свого положення під дією сил і зазвичай використовуються для представлення підлоги, стін, перешкод або інших нерухомих елементів сцени. Подібний поділ використовується у багатьох фізичних рушіях, де тип тіла визначає спосіб його участі у симуляції [7].

Форма тіла також має суттєве значення для моделювання і тому у простих 2D рушіях найчастіше використовуються базові геометричні фігури, такі як прямокутники та кола, оскільки вони зручні для навчальної реалізації, адже мають

зрозумілий математичний опис і дозволяють відносно просто виконувати перевірку зіткнень. Коло може бути описане центром і радіусом, а прямокутник – центром і розмірами або мінімальною та максимальною точками обмежувального прямокутника. Використання простих форм зменшує складність алгоритмів і робить фізичну симуляцію більш передбачуваною для аналізу.

У 2D фізичній симуляції особливо важливою є система координат, оскільки фізичний світ зазвичай описується у власних одиницях вимірювання, які не обов'язково збігаються з пікселями екрана, це дозволяє відокремити логіку симуляції від способу її відображення. Наприклад, тіло може мати координати у світових одиницях, а під час рендерингу ці координати перетворюються у координати екрана, це дає змогу масштабувати сцену, змінювати камеру та зберігати однакову поведінку фізичної моделі незалежно від розміру вікна або налаштувань відображення.

Ще однією особливістю є дискретний характер симуляції. У реальному світі рух відбувається безперервно, але в програмному забезпеченні стан фізичного світу оновлюється кроками. На кожному кроці рушій обчислює нове положення тіл, перевіряє можливі зіткнення, формує контактні дані та змінює швидкості або положення об'єктів відповідно до результатів взаємодії. Точність і стабільність такої симуляції залежать від величини кроку часу, обраного методу інтегрування та способу обробки контактів, тому у системах реального часу часто використовується фіксований крок симуляції, що дозволяє зробити поведінку тіл більш передбачуваною.

Під час зіткнення твердих тіл рушій має не лише визначити факт перетину, а й обчислити параметри контакту: напрям нормалі, глибину проникнення та приблизну точку взаємодії. Ці дані надалі використовуються для розділення тіл і зміни їхніх швидкостей. У сучасних підходах до симуляції контактів важливу роль відіграють також тертя та пружність, які визначають, чи буде тіло ковзати, зупинятися, відскакувати або втрачати частину енергії після зіткнення [24].

У розроблюваному програмному забезпеченні моделювання твердих тіл у двовимірному просторі реалізується через набір базових фізичних параметрів:

положення, швидкість, сила, маса, форма, коефіцієнт тертя, коефіцієнт пружності та ознака статичності. Така модель не охоплює всіх можливостей професійних фізичних рушіїв, однак є достатньою для демонстрації основних принципів 2D симуляції. Вона дозволяє створювати прості сцени, аналізувати поведінку тіл під дією гравітації, спостерігати зіткнення між прямокутниками та колами, а також оцінювати вплив фізичних параметрів на результат симуляції.

1.3 Методи чисельного інтегрування руху фізичних тіл

У фізичній симуляції рух тіла описується зміною його положення, швидкості та прискорення у часі, і у реальному фізичному процесі ці величини змінюються безперервно. Однак у програмному забезпеченні симуляція виконується дискретно, тобто окремими кроками. На кожному кроці фізичний рушій обчислює новий стан тіла на основі його поточного положення, швидкості, маси, прикладених сил і проміжку часу, що минув між оновленнями, це і є чисельним інтегруванням.

Основна задача чисельного інтегрування полягає в тому, щоб наближено обчислити нове положення тіла після малого проміжку часу, оскільки якщо на тіло діє сила, вона створює прискорення відповідно до маси тіла. У спрощеному вигляді це можна подати так:

$$\text{acceleration} = \text{force} / \text{mass}$$

$$\text{velocity} = \text{velocity} + \text{acceleration} * \text{dt}$$

$$\text{position} = \text{position} + \text{velocity} * \text{dt}$$

де **dt** - крок часу симуляції і визначає, наскільки великий проміжок часу проходить за один фізичний крок. Якщо крок часу занадто великий, симуляція може втрачати точність, а тіла можуть проходити одне крізь одне або поводитися нестабільно, якщо занадто малий, симуляція стає точнішою, але потребує більше обчислень.

Одним із найпростіших методів чисельного інтегрування є метод Ейлера. Його суть полягає в тому, що новий стан тіла обчислюється на основі поточних значень швидкості та прискорення. Метод Ейлера простий для реалізації та добре підходить для навчальних фізичних рушіїв, оскільки наочно демонструє зв'язок

між силою, масою, прискоренням, швидкістю та положенням. Водночас цей метод має обмеження: за великих кроків часу або складних взаємодій він може накопичувати похибки, що впливає на стабільність симуляції [3].

У практичних фізичних рушіях часто використовується модифікований варіант методу Ейлера, який називають напівнеявним або симплектичним методом Ейлера.

У цьому підході спочатку оновлюється швидкість тіла з урахуванням прискорення, а вже потім нове положення обчислюється на основі оновленої швидкості. Такий варіант залишається простим і поводить ся стабільніше для інтерактивної симуляції, ніж базовий явний метод Ейлера.

Для двовимірного фізичного рушія цей підхід є зручним, оскільки всі основні величини можна представити як двокомпонентні вектори: положення, швидкість, сила та прискорення, вони мають компоненти за осями X і Y . Наприклад, сила тяжіння може бути подана як вектор, спрямований вниз, а швидкість тіла може одночасно містити горизонтальну та вертикальну складові. Завдяки цьому рух тіла у двовимірному просторі можна обчислювати єдиним набором векторних операцій.

Окрему роль у симуляції відіграє накопичення сил. Замість того щоб одразу змінювати швидкість тіла вручну, рушій може накопичувати сили, які діють на тіло протягом фізичного кроку. До таких сил можуть належати гравітація, зовнішні поштовхи або інші впливи. Далі сумарна сила перетворюється на прискорення, а прискорення змінює швидкість, після завершення кроку сили очищуються, щоб на наступному кроці вони могли бути сформовані заново. Це робить модель більш гнучкою, оскільки різні джерела впливу можуть додавати сили незалежно одне від одного.

Вибір кроку часу є важливим моментом, оскільки якщо фізична симуляція напряду залежить від частоти кадрів, то на різних комп'ютерах або за різного навантаження поведінка тіл може відрізнитися. Для уникнення цього у фізичних рушіях часто застосовують фіксований крок симуляції. У такому разі фізика оновлюється, наприклад, із кроком **1/60** секунди, незалежно від того, з якою швидкістю відбувається відображення кадрів. Це дозволяє зробити поведінку

об'єктів більш передбачуваною та стабільною. Фіксований крок часу також спрощує тестування фізичного рушія. Якщо симуляція щоразу виконується з однаковим значенням dt , то результати роботи інтегрування, виявлення колізій і вирішення зіткнень легше перевіряти. Це особливо важливо для навчального рушія, у якому потрібно не лише отримати візуальний результат, а й мати можливість пояснити, чому тіло перемістилося саме таким чином.

Крім методу Ейлера, у фізичній симуляції застосовуються й інші підходи, наприклад методи Верле, Рунге-Кутти та position-based. Вони можуть забезпечувати кращу стабільність або точність у певних задачах, але часто є складнішими для реалізації та пояснення. У сучасних дослідженнях також розглядаються підходи на основі position-based dynamics та extended position-based dynamics, які використовуються для підвищення стабільності симуляцій і роботи з обмеженнями [25]. Проте для базового навчального 2D рушія доцільним є використання простішого методу, який дозволяє зрозуміти фундаментальні принципи оновлення фізичного стану тіла.

1.4 Виявлення колізій та обробка зіткнень

Виявлення колізій є одним із ключових етапів роботи фізичного рушія, оскільки саме на цьому етапі визначається, чи взаємодіють фізичні тіла між собою, адже якщо інтегрування відповідає за оновлення положення та швидкості об'єктів, то система колізій визначає, чи не призвів цей рух до перетину тіл у просторі. Без цього етапу об'єкти могли б вільно проходити одне крізь одне, що робило б фізичну симуляцію некоректною та непридатною для інтерактивного використання.

У загальному випадку виявлення колізій поділяють на два основні етапи: широку фазу та вузьку фазу. Широка фаза використовується для швидкого відсіювання пар об'єктів, які точно не можуть перетинатися, що особливо важливо у сценах із великою кількістю тіл, оскільки повна перевірка кожного об'єкта з кожним має квадратичну складність і швидко стає неефективною. Для оптимізації можуть використовуватися обмежувальні об'єми, просторові сітки, дерева або інші

структури просторового розбиття. У вузькій фазі вже виконується точніша перевірка для тих пар тіл, які потенційно можуть контактувати [5].

Для двовимірних навчальних рушіїв часто використовуються прості геометричні форми: кола, прямокутники та їхні обмежувальні прямокутники і одним із найпоширеніших способів спрощення перевірки є використання AABV (Axis-Aligned Bounding Box), тобто осьово-орієнтованого обмежувального прямокутника, такий прямокутник задається мінімальною та максимальною точками за координатами X і Y . Перевірка перетину двох AABV є простою та швидкою: достатньо визначити, чи перетинаються їхні проєкції на обидві координатні осі і якщо перетину немає хоча б на одній осі, то об'єкти не стикаються.

Для кіл перевірка колізії виконується інакше, два кола перетинаються тоді, коли відстань між їхніми центрами менша або дорівнює сумі їхніх радіусів. Такий тест є зручним для навчальної реалізації, оскільки він напряду демонструє використання векторів, довжини вектора та геометричного змісту відстані між об'єктами. Для перевірки зіткнення прямокутника та кола зазвичай визначають найближчу до центра кола точку на прямокутнику, після чого порівнюють відстань від цієї точки до центра кола з радіусом.

Після визначення факту зіткнення фізичному рушію недостатньо просто знати, що два тіла перетнулися, оскільки для подальшої обробки потрібно сформувати контактні дані: нормаль зіткнення, глибину проникнення та точку контакту. Нормаль зіткнення показує напрям, у якому тіла мають бути розділені або в якому буде прикладена реакція зіткнення. Глибина проникнення показує, наскільки сильно тіла накладаються одне на одне. Точка контакту використовується для візуалізації та в складніших рушіях може впливати на обертання й момент сили. Формування коректних контактних даних є окремою складною задачею, особливо у випадках кількох одночасних контактів або складних форм тіл [6].

Обробка зіткнень складається з двох пов'язаних задач: позиційної корекції та зміни швидкостей тіл. Позиційна корекція потрібна для усунення проникнення тіл одне в одне, оскільки симуляція виконується дискретними кроками, зіткнення часто виявляється вже після того, як об'єкти частково перетнулися. У такому випадку

рушій має розсунути тіла вздовж нормалі контакту, враховуючи їхню масу та статичність. Якщо цього не зробити, об'єкти можуть залишатися всередині одне одного, що призведе до нестабільної поведінки.

Зміна швидкостей після зіткнення зазвичай виконується за допомогою імпульсного підходу. Імпульс змінює швидкості тіл таким чином, щоб вони перестали рухатися одне в одне вздовж нормалі контакту, при цьому враховується маса тіл, відносна швидкість, коефіцієнт пружності та напрям контакту. Коефіцієнт пружності визначає, наскільки сильно тіло відскакує після зіткнення: за малого значення зіткнення виглядає непружним, а за більшого тіло зберігає більше енергії руху. Імпульсний підхід широко використовується у фізичних рушіях реального часу, оскільки дає змогу достатньо швидко отримати правдоподібну реакцію тіл на зіткнення [1].

Тертя є окремою частиною обробки контактів. Якщо нормальний імпульс відповідає за розділення тіл уздовж нормалі зіткнення, то тертя діє вздовж дотичного напрямку до поверхні контакту, воно зменшує ковзання тіл відносно одне одного і дозволяє моделювати ситуації, коли об'єкт поступово сповільнюється на поверхні або по-різному рухається залежно від коефіцієнта тертя. Симуляція тертя є складнішою за просте відштовхування тіл, оскільки вона пов'язана з дотичною складовою швидкості та обмеженням сили тертя. У сучасних підходах контакт і тертя розглядаються як одна з найскладніших частин фізичної симуляції, особливо для сцен із багатьма одночасними контактами [24].

Складність обробки зіткнень зростає у випадках, коли в сцені присутня велика кількість тіл або коли багато об'єктів одночасно контактують між собою, наприклад, сцени з нагромадженням, у яких тіла стоять одне на одному, можуть демонструвати тремтіння, просідання або нестабільність, якщо вирішувач (solver) недостатньо точно обробляє контакти. У складніших моделях контактної динаміки враховуються одночасні непружні зіткнення, тертя та множинні обмеження між тілами, однак такі підходи є значно складнішими за базову навчальну реалізацію [16].

У сучасних дослідженнях виявлення колізій також розглядається як задача оптимізації, особливо для опуклих форм і складних сцен, наприклад, алгоритми на зразок GJK (Gilbert-Johnson-Keerthi) та їхні модифікації використовуються для ефективної перевірки перетину між опуклими тілами. Такі підходи важливі для професійних рушіїв, робототехніки та масштабних симуляцій, де кількість перевірок може бути дуже великою [14]. Проте для навчального 2D рушія доцільно починати з простіших перевірок, таких як AABB, коло-коло та прямокутник-коло, оскільки вони краще демонструють базову логіку виявлення колізій.

У розроблюваному фізичному рушії виявлення колізій та обробка зіткнень реалізуються для базових двовимірних форм: прямокутників і кіл, оскільки саме такий набір є достатнім для демонстрації основних принципів фізичної взаємодії: визначення факту перетину, формування контактної нормалі, обчислення глибини проникнення, корекції положення та зміни швидкості тіл після контакту. Це дозволяє наочно показати, як фізичний рушій переходить від простого факту зіткнення до повноцінної реакції об'єктів у симуляції.

1.5 Огляд існуючих 2D фізичних рушіїв

У межах цієї роботи для аналізу було обрано три відомі 2D фізичні рушії: Box2D, Chipmunk2D та Matter.js. Вони представляють різні підходи до реалізації двовимірної фізичної симуляції: Box2D є одним із найпоширеніших рушіїв для моделювання твердих тіл, Chipmunk2D позиціонується як легка й портативна бібліотека, а Matter.js орієнтований на використання у веб-застосунках. Подальший огляд цих рішень дозволяє визначити їхні функціональні особливості та сформулювати критерії порівняння з розроблюваним рушієм Enjoyer2D.

1.5.1 Box2D

Box2D є одним із найвідоміших відкритих 2D фізичних рушіїв, призначених для моделювання твердих тіл у комп'ютерних іграх та інтерактивних застосунках. Він забезпечує роботу з фізичними тілами, формами, контактами, зіткненнями, тертям, пружністю та обмеженнями. Box2D не відповідає за графіку або логіку гри,

а виконує роль окремої фізичної підсистеми, яку можна інтегрувати в інше програмне забезпечення [7].

Основними поняттями Box2D є фізичний світ, тіло, форма, контакт і вирішувач. Фізичний світ зберігає об'єкти та виконує крок симуляції, тіла можуть бути статичними, динамічними або кінематичними, що дозволяє описувати як нерухомі поверхні, так і рухомі об'єкти, форма визначає геометрію зіткнення та матеріальні властивості, зокрема тертя і пружність. Це робить архітектуру рушія гнучкою та придатною для створення складних фізичних сцен.

Box2D також має власний демонстраційний застосунок (див. рис. 1.1), у якому можна запускати різні демонстраційні сцени, перевіряти роботу тіл, контактів, з'єднань і вирішувача. Наявність такого середовища є важливою, оскільки фізичний рушій складно оцінювати лише за кодом: поведінку тіл, стабільність симуляції та коректність обробки контактів зручніше аналізувати візуально. Саме ця ідея демонстраційного середовища є близькою до підходу, використаного у розробленому програмному забезпеченні.

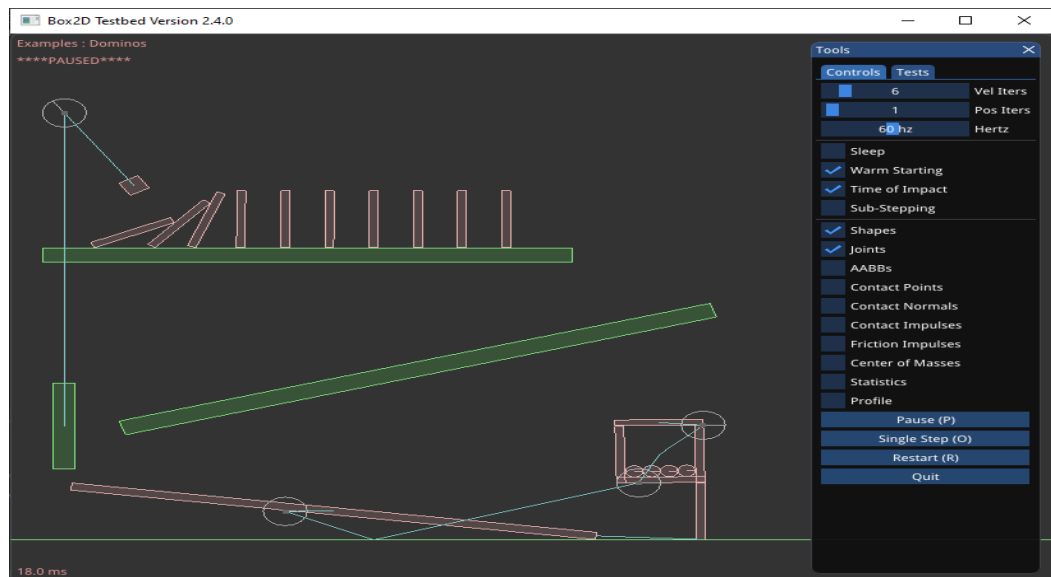


Рис. 1.1 Box2D (демонстраційний застосунок)

Імпульсний підхід до вирішення зіткнень, пов'язаний із роботами Еріна Катто, є однією з важливих основ Box2D, який дозволяє швидко й достатньо стабільно обробляти контакти між тілами у реальному часі [1]. Завдяки цьому Box2D підходить для практичної розробки ігор і симуляцій, де потрібна стабільна фізична взаємодія великої кількості об'єктів.

Основні переваги Vox2D:

- зрілість і широке використання;
- наявність документації та демонстраційного застосунку;
- підтримка статичних, динамічних і кінематичних тіл;
- підтримка контактів, обмежень, тертя та пружності;
- ефективна робота у задачах фізичної симуляції реального часу.

Обмеження Vox2D у контексті цієї роботи:

- відносно висока складність внутрішньої реалізації;
- необхідність вивчення готового API Vox2D;
- приховування більшості етапів фізичного кроку від користувача бібліотеки;
- менша придатність для поетапного навчального пояснення базової реалізації рушія.

Таким чином, Vox2D є потужним і зрілим 2D фізичним рушієм, який добре підходить для практичного використання в ігрових і симуляційних проєктах, але у межах цієї роботи він розглядається як основний аналог Enjoyer2D, який має іншу мету: не замінити Vox2D за функціональністю, а надати простішу й прозорішу навчальну реалізацію базових принципів 2D фізичної симуляції.

1.5.2 Chipmunk2D

Chipmunk2D є відкритою 2D фізичною бібліотекою, призначеною для моделювання твердих тіл у реальному часі. Вона позиціонується як швидка, портативна, чисельно стабільна та проста у використанні бібліотека, що може застосовуватися у комп'ютерних іграх та інтерактивних застосунках [8]. Chipmunk2D так само відповідає тільки за фізичну симуляцію і тому графічне відображення, введення та інші частини програми теж мають реалізовуватися окремо.

Основними об'єктами Chipmunk2D є тверді тіла, оболонки зіткнень, з'єднання і простори. Тіло зберігає фізичні параметри об'єкта, такі як маса, положення, обертання та швидкість, форма визначає геометрію зіткнень і

поверхневі властивості, зокрема тертя та пружність, простір виступає контейнером для симуляції: у нього додаються тіла, форми й обмеження, після чого він оновлюється як єдиний фізичний світ [8].

Chipmunk2D підтримує динамічні, статичні та кінематичні тіла, фільтрацію зіткнень, зворотні виклики для обробки зіткнень, обмеження, просторову індексацію та запити до сцени, це робить бібліотеку достатньо гнучкою для практичного використання. Важливою особливістю є те, що Chipmunk2D реалізований мовою C, завдяки чому його можна інтегрувати у проєкти різними мовами через відповідні обгортки (bindings).

Chipmunk2D також містить демонстраційні сцени і приклади використання, але вони більше позиціонують себе не як повноцінні демонстраційні застосунки, а як прості приклади використання рушія у невеликих проєктах, переважно ігор, і тому порівняно з Enjoyer2D та Box2D, ці демонстрації не виступають головним елементом позиціонування рушія: основний акцент зроблено на легкій C-бібліотеці, API, портативності та можливості інтеграції в різні програмні середовища [8].

Основні переваги Chipmunk2D:

- компактність і портативність;
- реалізація мовою C та можливість інтеграції з різними мовами програмування;
- підтримка динамічних, статичних і кінематичних тіл;
- підтримка форм, контактів, обмежень і просторового індексування;

Обмеження Chipmunk2D у контексті цієї роботи:

- необхідність працювати з готовим API Chipmunk2D;
- відсутність навчального пояснення внутрішньої будови рушія;
- низька наочність демонстраційного застосунку;
- приховування основних етапів фізичного кроку за внутрішньою реалізацією бібліотеки.

Chipmunk2D є легким і функціональним рішенням для інтеграції 2D фізики у програмні продукти і він є корисним аналогом для порівняння, оскільки демонструє інший підхід до організації фізичної бібліотеки: компактність, портативність і практичність. Водночас Enjoyer2D орієнтований передусім на навчальну прозорість, можливість поетапного аналізу фізичної симуляції та інтерактивну роботу з параметрами сцени.

1.5.3 Matter.js

Matter.js є 2D фізичним рушієм, реалізованим мовою JavaScript і орієнтованим на використання у веб-застосунках, він дозволяє створювати фізичні сцени безпосередньо у браузері, моделювати рух твердих тіл, обробляти зіткнення, працювати з обмеженнями та налаштовувати параметри фізичної взаємодії. На відміну від Box2D і Chipmunk2D, які частіше використовуються в нативній розробці або через обгортки, Matter.js зручно застосовувати у веб-середовищі для інтерактивних демонстрацій, невеликих ігор, навчальних прикладів і візуальних експериментів [9].

Архітектура Matter.js включає кілька основних модулів:

- **Engine** – відповідає за оновлення фізичної симуляції;
- **World або Composite** – містить фізичні об'єкти;
- **Bodies** – використовується для створення стандартних форм;
- **Render** – дозволяє швидко відобразити сцену в браузері.

Завдяки цьому Matter.js має низький поріг входу, розробник може створити просту фізичну сцену з декількома тілами без складної початкової конфігурації.

Matter.js має офіційний демонстраційний сайт (див. рис. 1.2) і набір прикладів, які працюють безпосередньо у браузері. У цих прикладах можна переглянути різні фізичні сцени, зокрема нагромадження, пружність, тертя, навантаження, обмеження, міст та інші демонстрації. Наявність таких прикладів робить Matter.js зручним не лише як бібліотеку для веб-розробки, а й як наочний інструмент для швидкого ознайомлення з можливостями 2D фізичної симуляції [9].

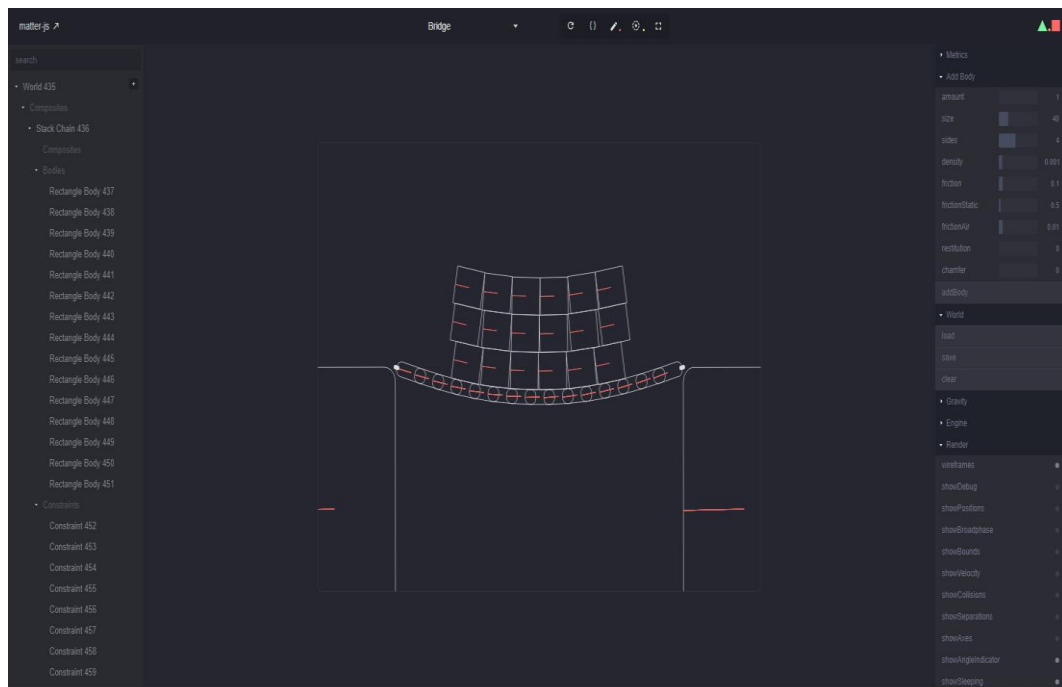


Рис. 1.2 Matter.js (демонстраційний сайт)

Основні переваги Matter.js:

- простота використання у веб-застосунках;
- можливість запуску фізичної симуляції безпосередньо у браузері;
- наявність вбудованого рендерингу для швидкого створення демонстрацій;
- велика кількість готових прикладів;
- зручність для швидкого прототипування інтерактивних фізичних сцен.

Обмеження Matter.js у контексті цієї роботи:

- орієнтація переважно на веб-середовище;
- приховування більшості внутрішніх деталей симуляції за готовим API;
- менша придатність для вивчення низькорівневої реалізації фізичного рушія;
- відмінність технологічного стеку від розроблюваного C++ рішення.

Підсумовуючи, Matter.js є зручним веб-орієнтованим аналогом, який добре підходить для створення інтерактивних фізичних сцен у браузері, але Enjoyer2D має іншу спрямованість: він реалізується мовою C++ і орієнтований на навчальне вивчення внутрішньої будови фізичного рушія, також у Enjoyer2D основні етапи

фізичного кроку, виявлення колізій, формування контактів і вирішення зіткнень реалізуються явно та можуть бути проаналізовані окремо.

1.6 Порівняльний аналіз аналогів і розроблюваного рішення

Після огляду існуючих 2D фізичних рушіїв можна зробити висновок, що всі вони є зрілими та практично орієнтованими рішеннями для моделювання фізичної взаємодії у двовимірному просторі, вони надають готові інструменти для створення фізичних тіл, обробки зіткнень, роботи з контактами, тертям, пружністю та іншими параметрами симуляції. Такі рушії доцільно використовувати у випадках, коли основною метою є швидка інтеграція фізики у гру, веб-застосунок або інше інтерактивне програмне забезпечення.

Мета розроблюваного рушія Enjoyer2D відрізняється від мети більшості готових аналогів. Його основна перевага полягає не у більшій кількості функцій, а в навчальній спрямованості, прозорій архітектурі та можливості поетапного аналізу внутрішньої будови фізичного рушія. У межах Enjoyer2D основні етапи симуляції, зокрема оновлення стану тіл, виявлення колізій, формування контактів, позиційна корекція та імпульсне вирішення зіткнень, реалізуються явно і це дозволяє використовувати рушія не лише як інструмент для симуляції, а й як основу для вивчення принципів фізичного моделювання.

Для порівняння було обрано критерії, які відображають не тільки наявність базових фізичних можливостей, а й придатність рушія до навчального аналізу, демонстрації та редагування параметрів у реальному часі. Результати порівняння наведено в таблиці 1.1.

Таблиця 1.1

Порівняльна таблиця Enjoyer2D з аналогами

Характеристика/рушій	Box2D	Matter.js	Chipmunk2D	Enjoyer2D
Повноцінне інтерактивне демонстраційне середовище	+	+	-	+
Debug-візуалізація фізичних параметрів	+	+	-	+
Зміна параметрів тіл у реальному часі	+	+	-	+
Можливість використання як готового професійного рушія	+	+	+	+/-
Орієнтація на навчальне вивчення внутрішньої будови рушія	-	-	-	+
Простота архітектури для покрокового аналізу	-	+/-	-	+
Повний контроль над реалізацією рушія	-	-	-	+

Як видно, усі розглянуті рішення мають власні переваги:

– Box2D є найбільш зрілим і технічно розвиненим рушієм серед наведених аналогів, має сильний демонстраційний застосунок і широкі можливості для фізичної симуляції.

– Matter.js є зручним рішенням для веб-середовища, оскільки має демонстраційний сайт, простий API та дозволяє швидко створювати інтерактивні фізичні сцени у браузері.

– Chipmunk2D вирізняється компактністю, портативністю та практичною спрямованістю як легка C-бібліотека для інтеграції фізики у програмні продукти.

Enjoyer2D, у свою чергу, орієнтований на інше завдання: навчальне дослідження роботи фізичного рушія та демонстрацію його внутрішніх процесів, на відміну від готових рішень, у ньому особливу увагу приділено зрозумілості структури, можливості редагування параметрів тіл у реальному часі, debug-візуалізації та демонстраційним сценам. Це робить Enjoyer2D придатним для

використання як навчальної основи, а також як стартової бази для подальшого розширення у напрямку ігрових або симуляційних програм.

1.7 Визначення вимог до програмного забезпечення

На основі аналізу предметної галузі та огляду існуючих аналогів було сформовано вимоги до Enjoyer2D.

Функціональні вимоги:

- моделювання руху твердих тіл у двовимірному просторі з урахуванням положення, швидкості, сили, маси та гравітації;
- підтримка базових форм фізичних тіл (прямокутники та кола), а також статичних і динамічних тіл;
- виявлення колізій між тілами та формування контактних даних: нормалі зіткнення, точки контакту й глибини проникнення;
- обробка зіткнень імпульсним методом з урахуванням коефіцієнтів пружності та тертя;
- використання фіксованого кроку часу для стабільнішої симуляції незалежно від частоти рендерингу;
- наявність інтерактивного демонстраційного застосунку для створення, вибору, редагування та видалення фізичних тіл;
- можливість зміни параметрів тіл і фізичного світу в реальному часі;
- підтримка керування симуляцією: пауза, продовження, скидання сцени та виконання фізичного кроку;
- наявність debug-візуалізації;
- наявність демонстраційних сцен і можливість збереження та завантаження сцен у форматі JSON.

Нефункціональні вимоги:

- програмне забезпечення має бути реалізоване мовою C++ із чітким поділом на рівень фізичного рушія, рівень сцен і рівень демонстраційного застосунку;

- основні етапи фізичного кроку мають бути логічно відокремлені: інтегрування, виявлення колізій, формування контактів, позиційна корекція та імпульсне вирішення зіткнень;
- програмне забезпечення має працювати в режимі реального часу з використанням фіксованого фізичного кроку (1/60);
- ключові компоненти фізичного рушія мають бути придатними для модульного тестування;
- реалізація має передбачати можливість подальшого розширення, зокрема додавання нових типів колізій, обертання тіл, з'єднань, просторового розбиття та оптимізацій продуктивності;
- програмне забезпечення має бути придатним для збирання та запуску на основних операційних системах: Windows, MacOS, Linux.

Окремо слід зауважити ще те, що аналіз матеріалів щодо створення власних рушіїв та програмних засобів фізичної симуляції показує, що самостійна реалізація таких систем дає змогу краще зрозуміти їхню внутрішню будову, принципи взаємодії компонентів і особливості моделювання фізичних процесів [4, 20, 21]. Тому для Enjoyer2D важливими є не лише функціональні можливості фізичного рушія, а й прозорість архітектури, наявність debug-візуалізації, можливість редагування параметрів у реальному часі та придатність програмного забезпечення для навчального аналізу.

1.8 Висновки за результатами аналізу предметної галузі

У результаті аналізу предметної галузі було розглянуто роль фізичних рушіїв у сучасному програмному забезпеченні, особливості моделювання твердих тіл у двовимірному просторі, методи чисельного інтегрування, принципи виявлення колізій та обробки зіткнень. Встановлено, що фізичний рушій є важливою підсистемою інтерактивного застосунку, яка відповідає за моделювання руху тіл, взаємодію об'єктів, обробку контактів, тертя та пружності.

Проведений огляд існуючих 2D фізичних рушіїв показав, що вони є вже зрілими та практично орієнтованими рішеннями, що забезпечують широкий набір можливостей для моделювання фізичної взаємодії, мають документацію, приклади використання та можуть застосовуватися у реальних ігрових або інтерактивних проєктах. Водночас більшість таких рушіїв орієнтована передусім на практичну інтеграцію фізики у готові застосунки, а не на поетапне навчальне вивчення внутрішньої будови фізичної симуляції.

На основі порівняльного аналізу визначено, що доцільним є створення власного 2D фізичного рушія Enjoyer2D, головною особливістю якого є навчальна спрямованість, прозора архітектура та можливість візуального аналізу роботи фізичних алгоритмів. Розроблюване програмне забезпечення має поєднувати фізичний рушій і демонстраційний застосунок, що дозволяє створювати фізичні сцени, редагувати параметри тіл, переглядати debug-візуалізацію, керувати симуляцією та аналізувати поведінку об'єктів у реальному часі.

За результатами аналізу були також сформовані функціональні та нефункціональні вимоги до програмного забезпечення.

Отримані результати аналізу є основою для вибору засобів реалізації, проєктування архітектури та подальшої програмної реалізації фізичного рушія й демонстраційного застосунку.

2 ОГЛЯД ЗАСОБІВ РЕАЛІЗАЦІЇ

2.1 Мова програмування

Для реалізації розроблюваного фізичного рушія було обрано мову програмування C++. Такий вибір зумовлений тим, що C++ широко використовується у сфері розробки ігрових рушіїв, симуляцій, графічних застосунків та програмного забезпечення, де важливими є продуктивність, контроль над структурою даних і можливість низькорівневої оптимізації. Для фізичного рушія ці властивості є важливими, оскільки симуляція потребує регулярного оновлення стану тіл, перевірки колізій, обробки контактів і виконання обчислень у режимі реального часу [2, 3].

C++ поєднує можливості високорівневого та низькорівневого програмування. З одного боку, мова підтримує об'єктно-орієнтований підхід, шаблони, стандартні контейнери та інструменти для організації великих програмних проєктів, а з іншого боку дає розробнику контроль над пам'яттю, типами даних, передачею об'єктів за значенням або посиланням, що є важливим під час реалізації продуктивних систем.

У межах розробки Enjoyer2D мова C++ використовується для реалізації таких компонентів:

- фізичний рушій, який містить тіла, форми, фізичний світ, контакти та алгоритми обробки зіткнень;
- демонстраційний застосунок, що забезпечує взаємодію користувача з фізичною сценою;
- система збереження та завантаження сцен;
- модульні тести для перевірки основних компонентів рушія;
- інтеграція допоміжних бібліотек.

Важливою перевагою C++ для цього проєкту є можливість чіткого поділу програмного забезпечення на окремі модулі. Фізичний рушій може бути реалізований як незалежна частина, яка не залежить від графічного інтерфейсу або

бібліотек рендерингу. Це дозволяє окремо тестувати його логіку, використовувати рушій у різних застосунках і надалі розширювати його без повної перебудови архітектури.

Для розробки фізичного рушія також корисною є підтримка структур і класів. Наприклад, прості об'єкти даних можуть бути представлені структурами, а компоненти з інкапсульованою логікою – класами. Такий підхід дозволяє зберігати баланс між простотою коду та контролем над внутрішнім станом об'єктів і у випадку Enjoyer2D це є важливим для навчальної спрямованості проєкту, оскільки код має бути не лише працездатним, а й зрозумілим для поетапного аналізу.

Ще однією перевагою C++ є наявність стандартної бібліотеки, яка надає готові контейнери й інструменти для роботи з даними. У проєкті можуть використовуватися такі засоби, як `std::vector`, `std::string`, файлові потоки, алгоритми та інші стандартні механізми. Це зменшує потребу у власній реалізації базових структур даних і дозволяє зосередитися на фізичній логіці рушія.

Водночас використання C++ має певні складнощі, оскільки мова потребує уважного ставлення до життєвого циклу об'єктів, посилань, копіювання даних і організації залежностей між модулями. Помилки у роботі з пам'яттю або неправильна передача об'єктів можуть призводити до складних для виявлення проблем, тому під час реалізації Enjoyer2D важливо було дотримуватися простих і зрозумілих архітектурних рішень, уникаючи зайвої складності та перевіряти ключову логіку за допомогою тестів.

Мова C++ є доцільним вибором для реалізації Enjoyer2D, оскільки забезпечує продуктивність, гнучкість, модульність і можливість точного контролю над реалізацією фізичного рушія. Саме ці властивості відповідають меті даного проєкту.

2.2 Середовища розробки

Під час реалізації Enjoyer2D використовувалися два середовища розробки: CLion та Microsoft Visual Studio. Основним середовищем розробки був CLion, а Microsoft Visual Studio застосовувалося для перевірки працездатності проєкту в операційній системі Windows.

CLion – потужне, кросплатформене інтегроване середовище розробки (IDE) від компанії JetBrains. Воно було обрано як основне середовище розробки через зручну підтримку CMake проєктів, навігацію по структурі коду, роботу з декількома цілями збірки та інтеграцію з інструментами налагодження і загалом через свою зручну та ефективну роботу з мовою C++.

У межах даної роботи CLion використовувався для:

- написання та редагування коду фізичного рушія;
- організації структури проєкту;
- роботи з CMake файлами;
- запуску демонстраційного застосунку;
- запуску модульних тестів;
- навігації між файлами, класами, структурами та функціями.

Microsoft Visual Studio використовувався лише як інструмент для перевірки збирання та запуску проєкту на Windows. Це було необхідно, оскільки однією з нефункціональних вимог до програмного забезпечення є можливість роботи на основних настільних операційних системах. Перевірка у Visual Studio дала змогу переконатися, що CMake-конфігурація, підключення бібліотек і збірка демонстраційного застосунку коректно працюють не лише на macOS та Linux, а й на Windows.

CLion забезпечив зручне основне середовище для щоденної розробки Enjoyer2D, а Microsoft Visual Studio використовувалося як додатковий засіб перевірки кросплатформеності та працездатності проєкту на Windows. Це дозволило зменшити ризик помилок, пов'язаних із збіркою проєкту на різних операційних системах, та підвищити надійність програмного забезпечення.

2.3 Система збирання

Для організації збирання проєкту було використано систему CMake. CMake є кросплатформеним інструментом, який дозволяє описувати структуру проєкту, цілі збірки, залежності, параметри компіляції та зв'язки між окремими модулями, його використання є майже невід'ємною частиною розробки сучасних C++ застосунків.

У межах даного проєкту CMake використовувався для збирання таких компонентів:

- бібліотеки фізичного рушія;
- демонстраційного застосунку;
- модульних тестів;
- допоміжної бібліотеки для інтеграції Dear ImGui;
- зовнішніх залежностей, необхідних для роботи проєкту.

Однією з важливих переваг CMake є його можливість описати проєкт незалежно від конкретного середовища розробки, тобто один і той самий CMake проєкт може бути відкритий у CLion, зібраний через командний рядок, або використаний у Microsoft Visual Studio. Це є важливим, оскільки дає можливість легко розгортати проєкти на будь-якій операційній системі.

Для підключення зовнішніх бібліотек у проєкті використовується модуль **FetchContent**, який дозволяє автоматично завантажувати необхідні залежності під час конфігурації проєкту, що спрощує процес підготовки середовища розробки (див. рис. 2.1). Завдяки цьому користувачу не потрібно вручну встановлювати всі бібліотеки окремо: достатньо виконати конфігурацію CMake, після чого потрібні залежності будуть отримані та підключені до відповідних цілей збірки [10].

```

FetchContent_Declare(
    glm
    URL https://github.com/g-truc/glm/archive/refs/tags/1.0.1.tar.gz
)

FetchContent_Declare(
    SDL3
    URL https://github.com/libsdl-org/SDL/archive/refs/tags/release-3.2.10.tar.gz
)

FetchContent_Declare(
    Catch2
    URL https://github.com/catchorg/Catch2/archive/refs/tags/v3.8.0.tar.gz
)

FetchContent_Declare(
    nlohmann_json
    URL https://github.com/nlohmann/json/archive/refs/tags/v3.11.3.tar.gz
)

FetchContent_MakeAvailable(glm SDL3 Catch2 nlohmann_json)

```

Рис. 2.1 Автоматичне завантаження бібліотек через CMake

Використання CMake також дозволяє чітко розділити логіку збирання різних частин проєкту. Наприклад, фізичний рушій може бути оформлений як окрема бібліотека, демонстраційний застосунок - як виконуваний файл, а тести - як окрема тестова ціль. Такий підхід відповідає архітектурі Enjoyer2D, у якій фізична логіка відокремлена від демонстраційного інтерфейсу та може перевірятися незалежно.

Основні функції CMake у проєкті:

- визначення стандарту мови C++;
- створення цілих збірок для рушія, тестів і демонстраційного застосунку;
- підключення зовнішніх бібліотек;
- налаштування include директорії;
- забезпечення зв'язування модулів між собою;
- запуск модульних тестів через CTest;
- спрощення збірки проєкту на різних операційних системах.

Для демонстраційного застосунку важливою є також коректна робота із зовнішніми runtime залежностями. Наприклад, під час збирання на Windows необхідно забезпечити доступність динамічних бібліотек, які потрібні для запуску виконуваного файлу. Налаштування CMake дозволяє автоматизувати копіювання

таких файлів у директорію зі зібраним застосунком, що спрощує запуск проєкту після збірки.

Таким чином, система збирання CMake у проєкті виконує не лише роль системи збирання, а й засобу організації всієї структури програмного забезпечення. Його використання забезпечує кросплатформеність, зручне підключення залежностей, поділ проєкту на окремі модулі та можливість збирати фізичний рушій, демонстраційний застосунок і тести як незалежні частини єдиного програмного рішення.

2.4 Бібліотеки

Використання бібліотек дозволяє не реалізовувати з нуля допоміжну інфраструктуру, а зосередитися на основній логіці проєкту, фізичному рушії, обробці колізій, демонстраційних сценах і взаємодії користувача з симуляцією.

У проєкті *Enjoyer2D* бібліотеки застосовуються як допоміжні інструменти, тоді як фізична логіка рушії реалізується самостійно. Це дозволяє поєднати практичність готових засобів розробки з навчальною цінністю власної реалізації ключових фізичних алгоритмів.

2.4.1 GLM

GLM – C++ бібліотека для роботи з математичними типами та операціями, що часто використовуються у графічних і симуляційних застосунках, вона надає зручні векторні та матричні типи, зокрема `glm::vec2`, який використовується для представлення двовимірних величин: положення, швидкості, сили, нормалі зіткнення та інших фізичних параметрів [18].

У проєкті GLM використовується переважно для векторної математики у фізичному рушії. Завдяки цій бібліотеці можна зручно виконувати операції додавання, віднімання, множення векторів на скаляр, обчислення довжини вектора, нормалізації та скалярного добутку. Такі операції є необхідними для оновлення

руху тіл, визначення відстані між об'єктами, обчислення контактних нормалей і розрахунку імпульсів під час зіткнень.

Основні задачі GLM:

- зберігання положення, швидкості та сили фізичних тіл;
- обчислення напрямків руху та нормалей зіткнення;
- визначення відстані між об'єктами;
- робота з AABB, контактними точками та debug-візуалізацією;
- виконання базових векторних операцій у фізичному рушії.

Використання GLM дозволяє уникнути самотійної реалізації базових математичних типів і зосередитися на логіці фізичної симуляції. При цьому бібліотека не приховує фізичні алгоритми, а лише надає зручний інструмент для роботи з векторами, що відповідає навчальній спрямованості Enjoyer2D.

2.4.2 SDL3 та SDL_Renderer

SDL3 є кросплатформеною бібліотекою, яка надає низькорівневий доступ до вікна, подій введення, клавіатури, миші та графічного обладнання, вона підтримує основні настільні операційні системи, зокрема Windows, macOS і Linux, що є важливим для створення демонстраційного застосунку Enjoyer2D [11].

SDL3 використовується для створення вікна, обробки подій користувача та організації головного циклу застосунку. Через систему подій SDL3 демонстраційний застосунок отримує інформацію про натискання клавіш, рух і кліки миші, закриття вікна та інші дії користувача. Це дозволяє реалізувати керування камерою, вибір інструментів, створення тіл у сцені та взаємодію з симуляцією.

SDL_Renderer у свою чергу є частиною SDL3 та використовується для двовимірного відображення фізичної сцени. За його допомогою у застосунку відображаються контури тіл, кола, прямокутники, AABB, вектори швидкості, контактні точки, нормалі зіткнень і траєкторії руху. Такий підхід добре підходить для навчального демонстраційного застосунку, оскільки дозволяє просто й наочно

показувати результати роботи фізичного рушія без використання складнішого графічного API.

Основні задачі SDL3 та SDL_Renderer:

- створення та керування вікном демонстраційного застосунку;
- обробка введення з клавіатури й миші;
- організація основного циклу застосунку;
- відображення сцени, фізичних тіл і debug-візуалізації.

Важливо, що SDL3 використовується лише на рівні демонстраційного застосунку, а не всередині фізичного рушія. Завдяки цьому рушій Enjoyer2D залишається незалежним від графічної бібліотеки та може бути протестований або використаний окремо від візуального інтерфейсу.

2.4.3 Dear ImGui

Dear ImGui – це бібліотека для створення графічного інтерфейсу користувача у C++ застосунках, вона працює за принципом immediate mode GUI, тобто елементи інтерфейсу описуються безпосередньо в коді та оновлюються кожен кадр. Цей підхід добре підходить для debug-панелей, редакторів, інструментів розробника та демонстраційних застосунків [12].

Enjoyer2D використовує Dear ImGui для побудови інтерфейсу демонстраційного застосунку, за його допомогою реалізовано панелі керування симуляцією, редагування параметрів фізичних тіл, зміну налаштувань фізичного світу, вибір інструментів, перемикання debug-візуалізації та роботу зі збереженням і завантаженням сцен.

На відміну від інших зовнішніх залежностей, Dear ImGui підключена до проєкту як vendor-бібліотека, тобто її необхідні вихідні файли розміщено безпосередньо в структурі проєкту. Таке рішення зумовлене тим, що Dear ImGui часто інтегрується у C++ проєкти саме на рівні вихідних файлів, а для роботи з SDL3 та SDL_Renderer потрібні конкретні backend файли бібліотеки. Розміщення необхідних файлів безпосередньо в проєкті спрощує контроль над версією ImGui, дає змогу додавати власний функціонал, зменшує залежність від зовнішнього

способу встановлення та робить збірку демонстраційного застосунку більш передбачуваною.

Основні задачі Dear ImGui:

- створення панелей керування демонстраційним застосунком;
- редагування параметрів тіл у реальному часі;
- зміна гравітації, кількості ітерацій вирішувача та інших параметрів симуляції;
- вмикання й вимикання debug-візуалізації;
- вибір інструментів для створення або виділення тіл;
- керування зберіганням і завантаженням сцен.

Dear ImGui використовується лише на рівні демонстраційного застосунку і не є частиною фізичного рушія. Це робить рушій незалежним від інтерфейсу користувача, а вся GUI логіка винесена в окремий рівень програми, що відповідає загальній архітектурі проєкту, де фізична симуляція, відображення та редагування сцени розділені між різними модулями.

2.4.4 Catch2

Catch2 є фреймворком для модульного тестування C++ коду, він дозволяє створювати окремі тестові сценарії, перевіряти очікувану поведінку функцій і класів, а також автоматизувати перевірку працездатності програмних компонентів. Згідно з документацією, Catch2 орієнтований на unit-тестування C++ застосунків і надає зручний синтаксис для написання перевірок за допомогою тестових макросів (див. рис. 2.2) [13].

```
TEST_CASE("a is correctly added to b") {
    const int a {5};
    const int b {7};

    REQUIRE(a+b == 12);
}
```

Рис. 2.2 Приклад тестового макроса Catch2

Enjoyer2D використовує Catch2 для тестування фізичного рушія окремо від демонстраційного застосунку. Це важливо, оскільки фізична логіка не повинна залежати від графічного інтерфейсу, SDL3 або Dear ImGui. Завдяки тестам можна перевіряти роботу базових компонентів рушія без запуску вікна та без ручної взаємодії з демонстраційним застосунком.

Основні задачі Catch2:

- перевірка створення фізичних тіл;
- перевірка роботи форм об'єктів;
- тестування AABB та виявлення колізій;
- перевірка формування контактів;
- тестування фізичного кроку рушія;
- перевірка поведінки тіл після зіткнення;
- перевірка роботи окремих функцій після змін у коді.

Catch2 підвищує надійність розробки, оскільки дозволяє швидко виявляти помилки у фізичній логіці. Для навчального рушія це особливо важливо: якщо виявлення колізій, інтегрування або обробка контактів працюють некоректно, помилку може бути складно відрізнити від звичайної нестабільності симуляції,

а модульні тести дають змогу перевіряти такі частини ізольовано та впевненіше змінювати реалізацію рушія під час подальшого розвитку проєкту.

2.4.5 nlohmann/json

nlohmann/json – це C++ бібліотека для роботи з форматом JSON, яка надає зручний API для створення JSON об'єктів, доступу до їхніх полів, перетворення даних, читання та запису JSON структур. Бібліотека має можливості парсингу, серіалізації, доступу до елементів, роботи з масивами, об'єктами та перетвореннями користувацьких типів [19].

JSON формат є зручним для цього проєкту, оскільки дозволяє представити сцену у вигляді зрозумілої текстової структури: список тіл, їхні положення, швидкості, форми, маси, коефіцієнти тертя, пружності та статичності. Завдяки цьому сцени можна зберігати у файлах, повторно завантажувати та використовувати для перевірки або демонстрації роботи рушія.

Основні задачі nlohmann/json:

- збереження параметрів сцени у файл;
- завантаження сцени з JSON файлу;
- відновлення тіл, форм і фізичних параметрів після завантаження;
- створення набору користувацьких або демонстраційних сцен.

Його використання дозволяє зробити демонстраційний застосунок більш гнучким. Користувач може створити сцену, налаштувати параметри тіл, зберегти її у файл і пізніше завантажити знову, тим самим порівнювати поведінку різних сцен і поступово формувати набір прикладів для демонстрації можливостей фізичного рушія.

3 ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Загальна архітектура

Програмне забезпечення Enjoyer2D спроектовано з поділом на окремі логічні рівні. Такий підхід дозволяє відокремити фізичну логіку від демонстраційного інтерфейсу, спростити тестування окремих компонентів і забезпечити можливість подальшого розширення проєкту. Поділ програмного забезпечення на підсистеми є типовим підходом для побудови ігрових і симуляційних застосунків, оскільки кожна частина відповідає за власну область відповідальності [2].

Основні рівні архітектури Enjoyer2D:

- **фізичний рушій** – відповідає за моделювання руху тіл, обробку сил, виявлення колізій, формування контактів і вирішення зіткнень;
- **сцени** – відповідає за створення демонстраційних сцен, збереження, завантаження та підготовку даних для фізичного світу;
- **демонстраційний застосунок** – відповідає за вікно, введення користувача, рендеринг, інтерфейс редактора та debug-візуалізацію.

Центральним елементом програми є фізичний рушій, який реалізований як незалежний модуль, він не залежить від SDL3, Dear ImGui або інших засобів відображення. Завдяки цьому фізичний рушій може тестуватися окремо від демонстраційного застосунку, а його логіка залишається придатною для повторного використання в інших програмах.

Демонстраційний застосунок взаємодіє з рушієм через рівень сцен і спільний стан демонстраційного середовища. Користувач може створювати об'єкти, змінювати параметри тіл, керувати симуляцією та переглядати debug-візуалізацію, при цьому сам рушій не знає про інтерфейс користувача: він лише отримує дані фізичного світу, виконує фізичний крок і повертає оновлений стан тіл та контактів.

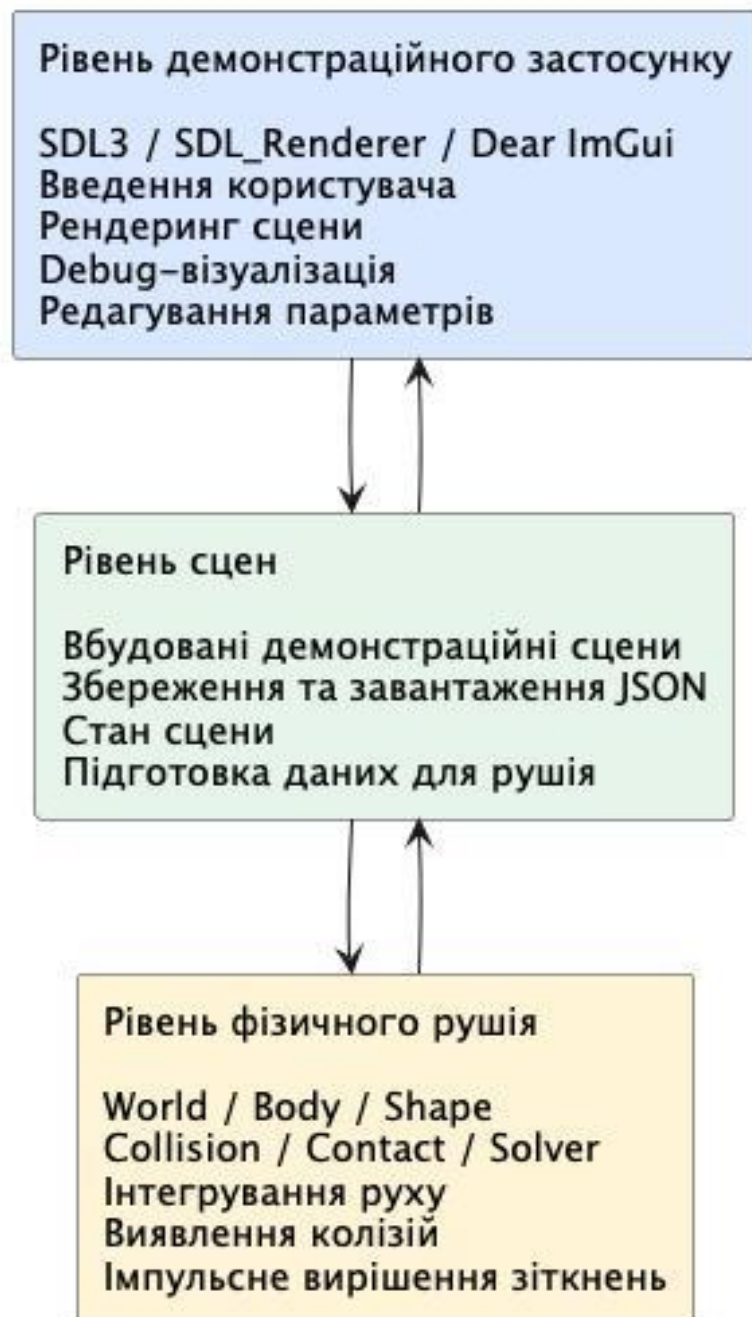


Рис. 3.1 Загальна архітектура Enjoyer2D

Наведений архітектурний підхід забезпечує модульність, зрозумілість і навчальну придатність проєкту, оскільки кожен рівень має чітко визначену роль, а взаємодія між ними дозволяє наочно простежити шлях даних: від дій користувача в демонстраційному застосунку до оновлення фізичного світу та відображення результату симуляції на екрані.

3.2 Архітектура фізичного рушія

Фізичний рушій Enjoyer2D є основним модулем програмного забезпечення, який відповідає за моделювання руху твердих тіл, виявлення колізій, формування контактів і вирішення зіткнень. Такий набір компонентів відповідає типовій структурі фізичних рушіїв реального часу, у яких окремо розглядаються інтегрування руху, виявлення колізій і відповідні на них реакції [3, 5, 24]. Фізичний рушій реалізований окремо від демонстраційного застосунку, що дозволяє використовувати його незалежно від графічного інтерфейсу та тестувати основну фізичну логіку окремо, що відповідає загальним принципам модульної архітектури програмного забезпечення [2].

Архітектура рушія побудована навколо класу World, який виступає центральним об'єктом симуляції, він зберігає фізичні тіла, контакти, глобальні параметри, зокрема гравітацію, а також виконує фізичний крок. Це дозволяє зосередити керування симуляцією в одному місці, а окремі структури та функції використовувати для представлення тіл, форм, контактів і перевірки колізій.

Основні компоненти фізичного рушія:

- **World** – фізичний світ, який зберігає тіла, контакти, параметри симуляції та виконує фізичний крок;
- **Body** – фізичне тіло з положенням, швидкістю, силою, масою, формою, тертям і пружністю;
- **BodyDef** – структура для початкового опису тіла перед його створенням у фізичному світі;
- **Shape** – опис форми тіла, зокрема прямокутника або кола;
- **AABB** – осьово-орієнтована обмежувальна область, що використовується для перевірки перетинів;
- **Contact** – структура, яка зберігає дані про зіткнення між двома тілами;
- **Collision** – набір функцій для виявлення колізій між підтримуваними формами;

– **WorldStats** – структура для збереження службової інформації про стан симуляції.

Фізичний крок рушія складається з послідовності етапів. Спочатку для динамічних тіл виконується інтегрування, під час якого оновлюються швидкість і положення тіла з урахуванням прикладених сил і гравітації. Після цього рушій перевіряє можливі зіткнення між тілами та формує список контактів. Далі виконується позиційна корекція, яка зменшує або усуває проникнення тіл одне в одне. Після цього застосовується імпульсне вирішення зіткнень, що змінює швидкості тіл відповідно до нормалі контакту, пружності та тертя.

Послідовність роботи фізичного кроку можна подати так:

1. Оновлення швидкостей і положень тіл.
2. Виявлення колізій між тілами.
3. Формування контактних даних.
4. Корекція проникнення.
5. Імпульсне вирішення зіткнень.
6. Очищення накопичених сил.

Особливістю архітектури рушія є те, що він не містить коду, пов'язаного з рендерингом, обробкою подій або графічним інтерфейсом. Демонстраційний застосунок лише передає команди до фізичного світу, читає оновлений стан тіл і відображає його. Завдяки цьому фізичний рушій залишається самостійним модулем, який можна розглядати, тестувати та розширювати незалежно від інших частин програмного забезпечення.

Така архітектура відповідає навчальній спрямованості проєкту: кожен етап симуляції має окрему роль, а основні компоненти рушія можна аналізувати поетапно. Це спрощує розуміння внутрішньої будови фізичного рушія та дозволяє надалі додавати нові можливості, зокрема обертання тіл, складніші типи колізій, з'єднання або оптимізацію широкої фази виявлення зіткнень.

3.3 Архітектура демонстраційного застосунку

Демонстраційний застосунок Enjoyer2D призначений для візуальної перевірки роботи фізичного рушія, створення фізичних сцен, редагування параметрів тіл і спостереження за результатами симуляції в реальному часі, він виступає окремим рівнем програмного забезпечення і взаємодіє з фізичним рушієм через його публічний інтерфейс. Це дозволяє не змішувати фізичну логіку з кодом рендерингу, інтерфейсу користувача та обробки введення.

Задачі демонстраційного застосунку:

- створення вікна програми;
- обробка введення з клавіатури та миші;
- керування камерою та масштабом сцени;
- запуск, пауза, скидання та покрокове виконання симуляції;
- створення, вибір, редагування та видалення фізичних тіл;
- перемикання демонстраційних сцен;
- збереження та завантаження сцен;
- відображення фізичних тіл і debug-візуалізації;
- виведення службової інформації про стан симуляції.

Архітектура демонстраційного застосунку складається з кількох логічних частин:

- **TestbedApp** відповідає за загальний життєвий цикл програми: ініціалізацію SDL3, створення вікна, запуск головного циклу, оновлення симуляції, рендеринг і завершення роботи.
- **InputHandler** обробляє події користувача та перетворює їх у зміни стану застосунку.
- **Renderer** відповідає за відображення фізичної сцени, тіл, контактів, AABB, векторів швидкості та інших debug-елементів.
- **Модулі редактора** відповідають за побудову інтерфейсу Dear ImGui та зміну параметрів тіл і фізичного світу.

Важливим компонентом є **TestbedState**, який зберігає поточний стан демонстраційного застосунку. До нього входять фізичний світ, налаштування симуляції, стан редактора, debug-параметри, дані про вибране тіло, налаштування камери та інформація про сцени. Завдяки цьому різні частини застосунку можуть працювати з єдиним станом, не змішуючи між собою фізичну логіку, обробку введення та рендеринг.

Для побудови демонстраційного застосунку використовуються SDL3, SDL_Renderer та Dear ImGui. SDL3 відповідає за вікно, події користувача та основний цикл програми, SDL_Renderer використовується для 2D-відображення фізичної сцени, а Dear ImGui - для створення панелей керування, редактора параметрів і виведення службової інформації [11, 12]. При цьому ці бібліотеки залишаються лише частиною демонстраційного рівня і не використовуються всередині фізичного рушія.

Послідовність етапів роботи демонстраційного застосунку виглядає так:

1. Ініціалізація вікна, рендерера та інтерфейсу.
2. Обробка подій користувача.
3. Оновлення стану редактора та команд симуляції.
4. Виконання фізичного кроку рушія.
5. Оновлення debug-даних.
6. Відображення сцени та інтерфейсу.
7. Повторення циклу до завершення програми.

Це дозволяє використовувати демонстраційний застосунок не лише як засіб запуску фізичного рушія, а як інтерактивне середовище для його аналізу. Користувач може змінювати параметри тіл, спостерігати контактні точки, нормалі, AABB, траєкторії руху та оцінювати вплив фізичних параметрів на поведінку сцени.

3.4 Структура фізичного світу та об'єктів симуляції

Фізичний світ у Enjoyer2D є контейнером, у якому зберігаються всі об'єкти, що беруть участь у симуляції і який містить набір фізичних тіл, список поточних контактів, глобальні параметри сцени та налаштування вирішувача. Саме фізичний світ виконує оновлення симуляції, застосовує гравітацію, перевіряє зіткнення між тілами та змінює їхній стан під час фізичного кроку.

Основні дані фізичного світу:

- список фізичних тіл;
- список контактів між тілами;
- вектор гравітації;
- кількість ітерацій для обробки швидкостей;
- кількість ітерацій для позиційної корекції;
- службова статистика симуляції.

Фізичне тіло представляє окремий об'єкт сцени, який у межах рушія зберігає фізичний стан і параметри, необхідні для моделювання руху та взаємодії з іншими об'єктами. До таких параметрів належать положення, швидкість, сила, маса, форма, коефіцієнт тертя, коефіцієнт пружності та ознака статичності. Динамічні тіла беруть участь у симуляції повністю, а статичні використовуються для нерухомих об'єктів, наприклад підлоги, стін або перешкод.

Форма тіла визначає його геометрію для виявлення колізій. У поточній реалізації підтримуються дві базові форми:

- прямокутник, який задається шириною та висотою;
- коло, яке задається радіусом.

Такий набір форм є достатнім для демонстрації базових принципів 2D фізичної симуляції, зокрема падіння тіл, зіткнень, нагромадження, тертя та пружності, також він спрощує реалізацію виявлення колізій, оскільки для прямокутників і кіл можна застосовувати зрозумілі геометричні перевірки.

Контакт у свою чергу описує факт зіткнення між двома фізичними тілами, він містить посилання на тіла, які беруть участь у зіткненні, нормаль контакту, точку

контакту та глибину проникнення. Ці дані використовуються під час позиційної корекції та імпульсного вирішення зіткнення, крім того, контактні дані передаються до демонстраційного застосунку для debug-візуалізації.

Окремо від фізичного рушія зберігаються дані, пов'язані з відображенням і аналізом симуляції, наприклад, траєкторії руху тіл, налаштування debug-візуалізації, стан вибраного об'єкта та параметри інструментів редактора належать до рівня демонстраційного застосунку. Вони не є частиною ядра фізики, оскільки не впливають безпосередньо на фізичний розрахунок.

3.5 Діаграма варіантів використання

Діаграма варіантів використання (див. рис. 3.2) відображає основні способи взаємодії користувача з демонстраційним застосунком Enjoyer2D. Єдиним актором у системі є користувач, який запускає демонстраційний застосунок, обирає демонстраційні сцени, створює фізичні тіла, редагує параметри сцени й об'єктів, керує симуляцією та працює зі збереженням і завантаженням сцен.

Основна взаємодія користувача з програмним забезпеченням зосереджена навколо створення й редагування фізичної сцени, керування ходом симуляції та аналізу її результатів за допомогою debug-візуалізації й статистики. Це відповідає навчальній спрямованості проєкту, оскільки користувач може не лише запускати готові сцени, а й змінювати параметри об'єктів та спостерігати вплив цих змін на фізичну поведінку тіл.

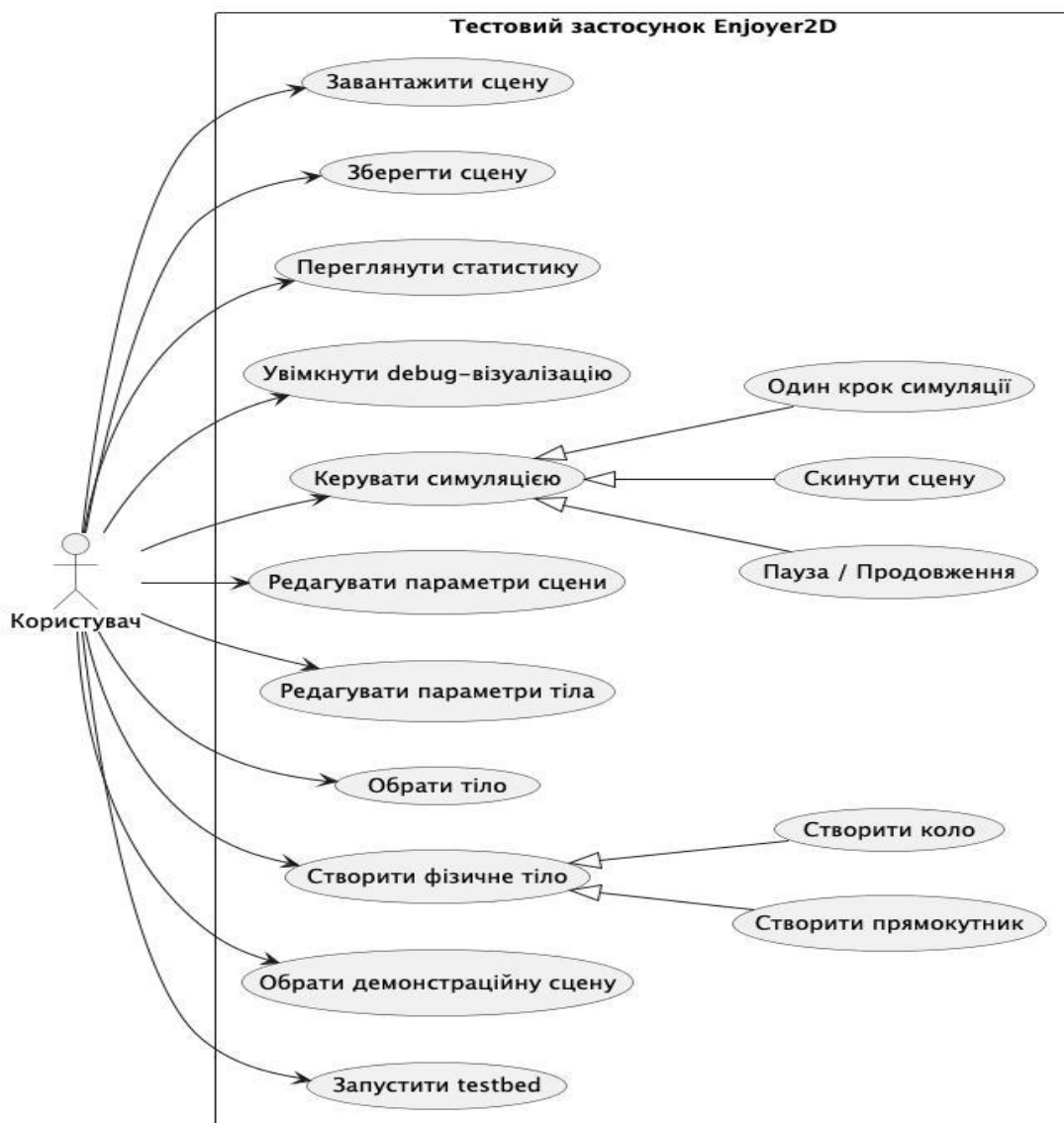


Рис. 3.2 Діаграма варіантів використання демонстраційного застосунку Enjoyer2D

3.6 Схема функціонування програми

Схема функціонування програми відображає взаємодію основних рівнів програмного забезпечення Enjoyer2D. Система поділена на три основні частини: рівень демонстраційного застосунку, рівень сцен і рівень фізичного рушія (див. рис. 3.3).

Рівень демонстраційного застосунку відповідає за взаємодію з користувачем. До нього належать обробка введення, графічний інтерфейс редактора, збереження стану застосунку та відображення фізичної сцени. Рівень сцен забезпечує роботу з

вбудованими демонстраційними сценами та JSON-збереженням/завантаженням. Рівень фізичного рушія виконує основну симуляцію: зберігає фізичний світ, обробляє тіла й форми, виявляє колізії, формує контакти та виконує імпульсне вирішення зіткнень. Ці дії змінюють **TestbedState**, який передає команди до рівня сцен або безпосередньо до фізичного світу. Після виконання фізичного кроку оновлений стан тіл, контактів і статистики передається назад до рендер-модуля, який відображає сцену на екрані. Така схема забезпечує чіткий поділ відповідальності між інтерфейсом, сценами та фізичним рушієм.

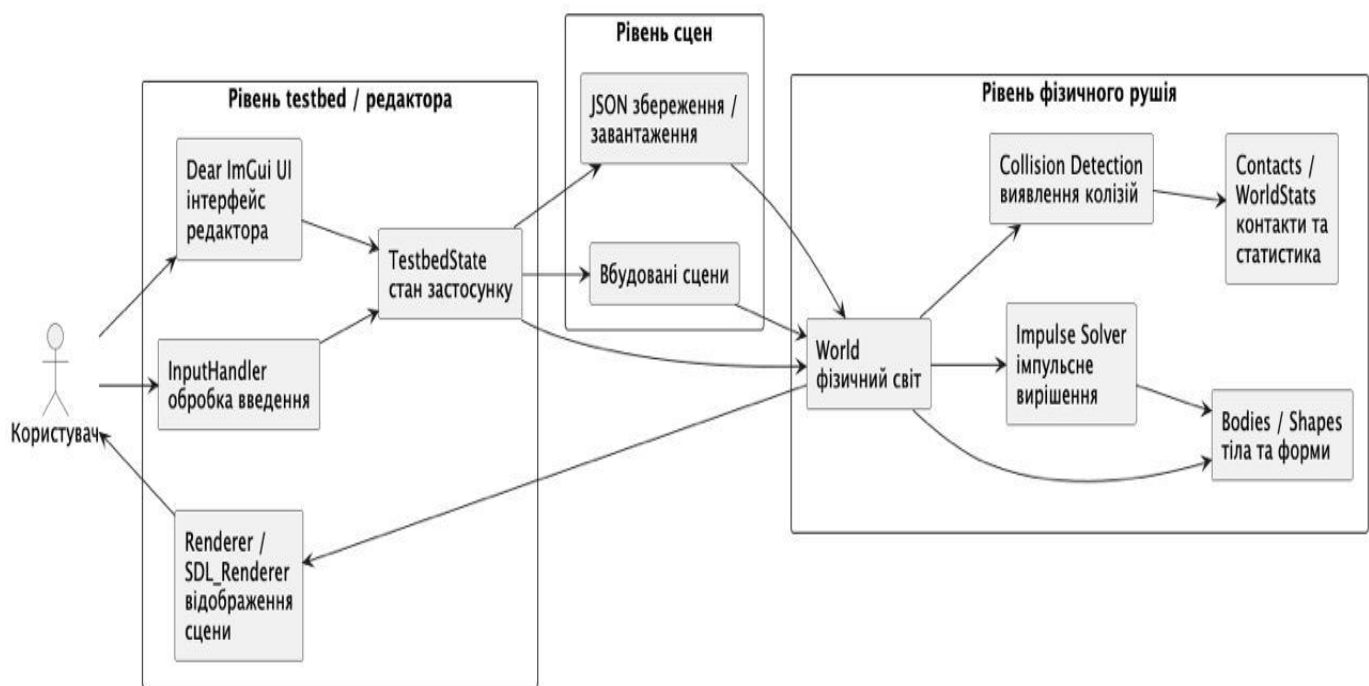


Рис. 3.3 Схема функціонування програми Enjoyer2D

3.7 Діаграма діяльності демонстраційного застосунку

Діаграма діяльності демонстраційного застосунку (див. рис. 3.4) відображає послідовність дій, які виконуються протягом одного кадру. Основний цикл починається з обробки подій SDL3, після чого оновлюється стан введення, редактора та обчислюється значення delta time.

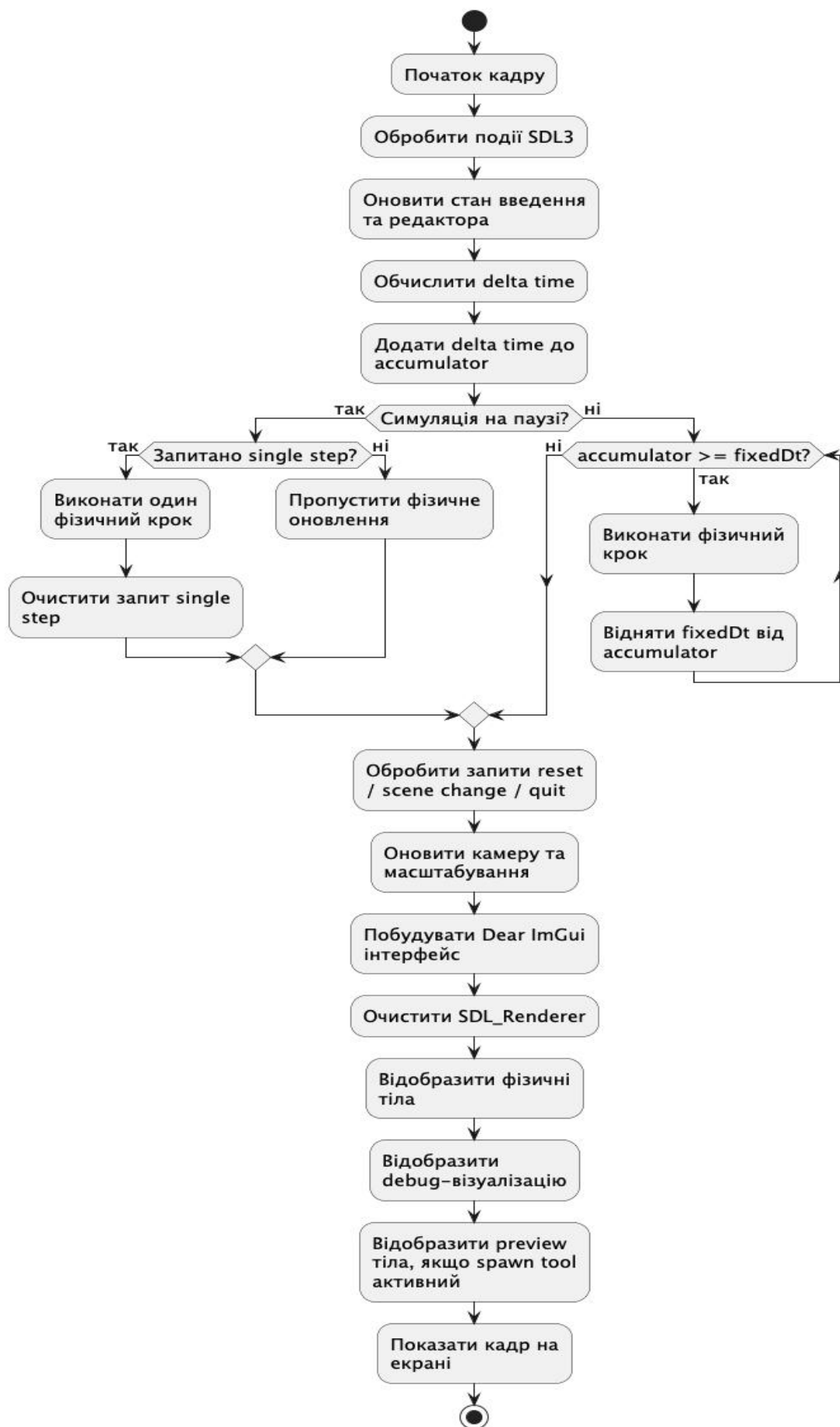


Рис. 3.4 Діаграма діяльності демонстраційного застосунку Enjoyer2D

Важливою частиною роботи застосунку є використання **accumulator**, який дозволяє відокремити частоту рендерингу від частоти оновлення фізичної симуляції. Якщо симуляція не перебуває на паузі, фізичний крок виконується один або кілька разів із фіксованим значенням **fixedDt**, якщо симуляція зупинена, користувач може виконати один фізичний крок за допомогою режиму **single step**.

Після обробки фізичного оновлення застосунок перевіряє запити на скидання сцени, зміну сцени або завершення роботи. Далі оновлюються параметри камери й масштабування, будується інтерфейс Dear ImGui, очищується рендерер і відображається фізична сцена. Наприкінці кадру відображаються debug-елементи, прев'ю створюваного тіла та готовий кадр виводиться на екран.

3.8 Діаграма діяльності фізичного кроку рушія

Діаграма діяльності фізичного кроку рушія (див. рис. 3.5) відображає послідовність операцій, які виконуються під час виклику **World::step(dt)**. Фізичний крок є основною процедурою оновлення симуляції, під час якої рушій змінює стан тіл, визначає контакти та обробляє зіткнення.

На початку кроку перевіряється значення dt , якщо крок часу є некоректним або меншим чи рівним нулю, оновлення симуляції не виконується. Після цього очищуються попередні контактні дані, оскільки на кожному новому кроці контакти мають бути сформовані заново відповідно до поточного положення тіл.

Далі виконується інтегрування руху динамічних тіл. Для кожного тіла обчислюється прискорення на основі гравітації та накопиченої сили, після чого оновлюються швидкість і положення. Далі рушій перевіряє пари тіл на наявність колізій і, у разі зіткнення, створює контактні дані. Контакт містить посилання на два тіла, точку контакту, нормаль зіткнення та глибину проникнення.

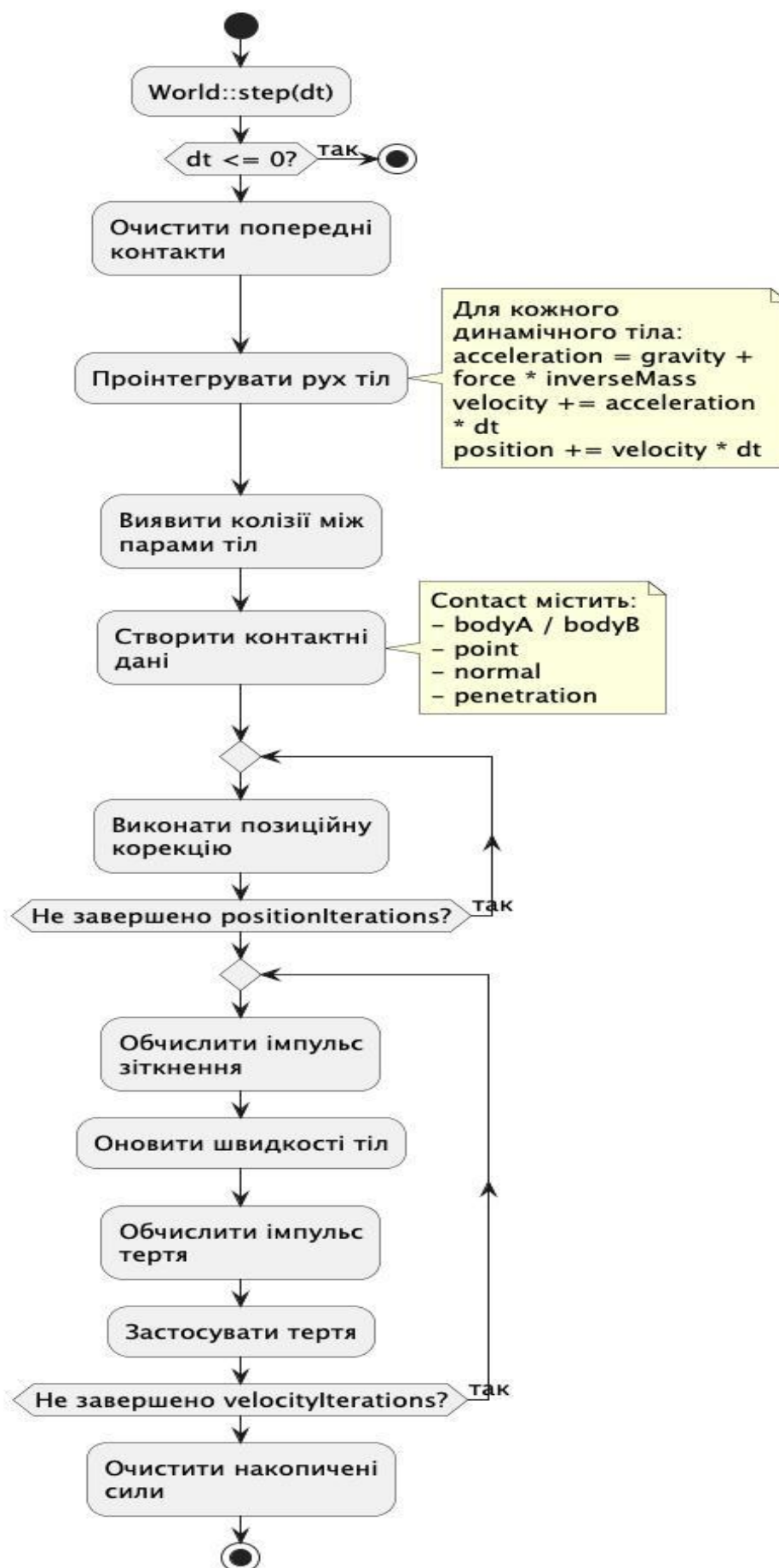


Рис. 3.5 Діаграма діяльності фізичного кроку рушія Enjoyer2D

Після формування контактів виконується позиційна корекція, вона потрібна для зменшення або усунення проникнення тіл одне в одне, яке виникає через дискретний характер симуляції. Позиційна корекція виконується кілька разів відповідно до значення **positionIterations**.

Наступним етапом є імпульсне вирішення зіткнень. Для кожного контакту обчислюється імпульс зіткнення, після чого оновлюються швидкості тіл. Додатково обчислюється і застосовується імпульс тертя, який зменшує ковзання тіл уздовж поверхні контакту. Цей етап також виконується кілька разів відповідно до значення **velocityIterations**.

Після завершення фізичного кроку рушій очищує накопичені сили. Це потрібно для того, щоб сили, прикладені на поточному кроці, не впливали на тіло повторно в наступних оновленнях, якщо вони не були додані знову.

3.9 Діаграма класів

Діаграма класів (див. рис. 3.6) використовується для уточнення структури фізичного рушія **Enjoyer2D** на рівні класів, структур, їхніх полів, методів і зв'язків. На відміну від попередніх підрозділів, де було описано загальну архітектуру та роль основних компонентів, у цьому підрозділі показано, як ці компоненти пов'язані між собою в реалізації.

Центральним елементом діаграми є клас **World**, який зберігає колекції **Body** та **Contact**, формує статистику **WorldStats** і використовує функції **Collision** для виявлення зіткнень. Структура **BodyDef** застосовується для створення нового фізичного тіла, а **Body** містить об'єкт **Shape**, який визначає геометрію тіла. **Shape**, у свою чергу, пов'язаний із **CircleShape**, **BoxShape** та **ShapeType**, що дозволяє описувати підтримувані типи форм.

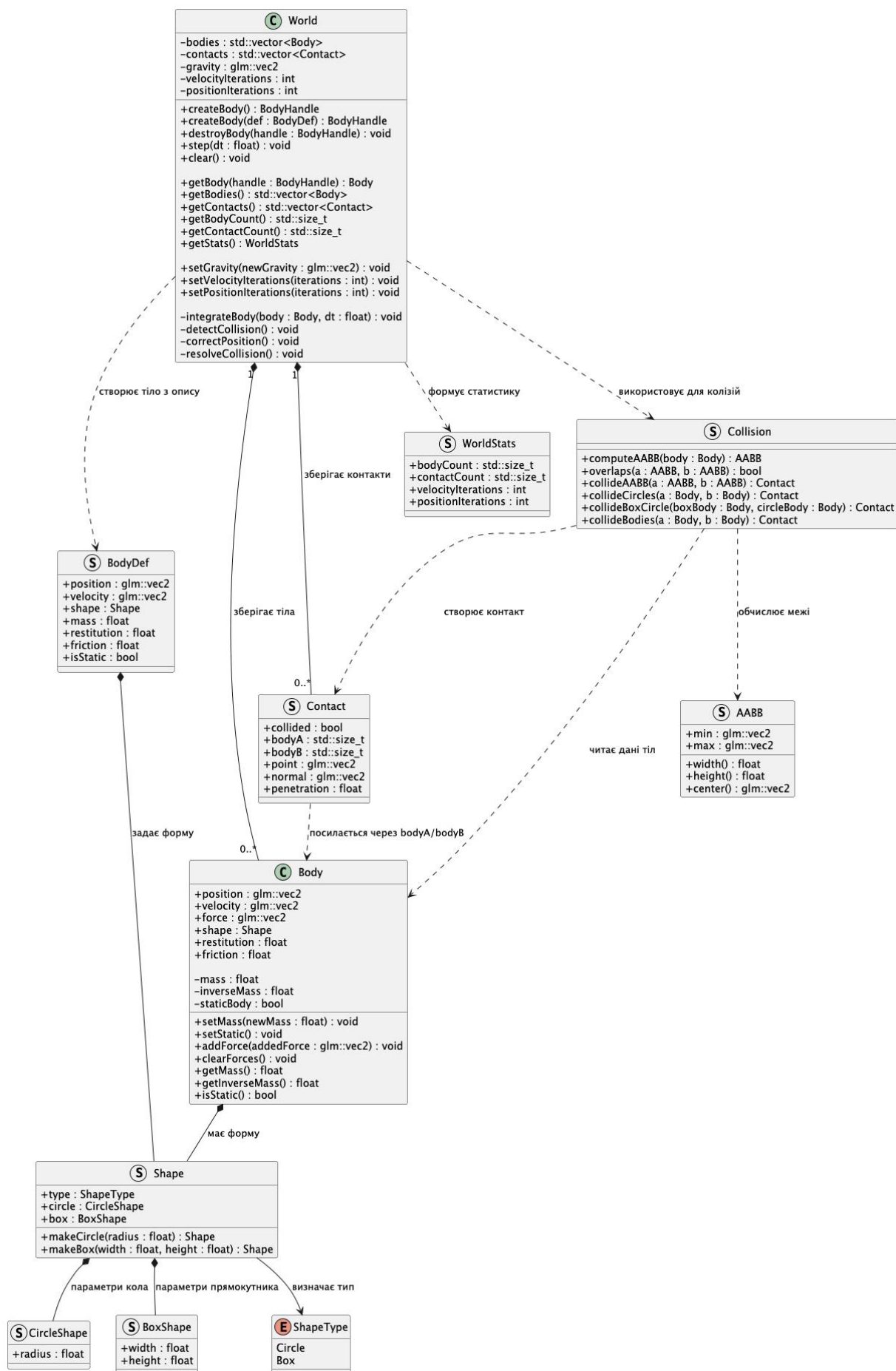


Рис. 3.6 Діаграма класів фізичного рушія Enjoyer2D

Діаграма також демонструє, що **Collision** не є окремим об'єктом фізичного світу, а виступає набором функцій для роботи з даними тіл, AABV областями та контактами. Контакти зберігають індекси тіл, між якими виникла взаємодія, а не прямі посилання на об'єкти, що спрощує збереження результатів перевірки колізій у межах фізичного світу.

Таким чином, діаграма класів показує, що ядро рушія побудоване навколо невеликої кількості взаємопов'язаних сутностей. Така структура є достатньо простою для навчального аналізу, але водночас дозволяє реалізувати основні етапи фізичної симуляції: створення тіл, опис форм, виявлення колізій, формування контактів і виконання фізичного кроку.

4 РЕАЛІЗАЦІЯ 2D ФІЗИЧНОГО РУШІЯ

4.1 Реалізація фізичного тіла та його параметрів

Фізичне тіло в Enjoyer2D є основною сутністю симуляції, воно зберігає положення, швидкість, силу, форму, масу, тертя, пружність і ознаку статичності. Поділ тіла та форми відповідає типовому підходу у фізичних рушіях, де фізичний стан об'єкта відокремлюється від його геометрії для колізій [3, 7].

Для створення фізичних тіл використовується окрема структура початкового опису. Вона містить параметри, з якими об'єкт додається до фізичного світу: початкове положення, швидкість, форму, масу, коефіцієнти тертя та пружності, а також ознаку статичності. Це дозволяє відокремити етап створення тіла від його подальшого оновлення під час симуляції.

Після створення тіло зберігає поточний фізичний стан і бере участь у всіх основних етапах роботи рушія: інтегруванні руху, виявленні колізій, формуванні контактів і вирішенні зіткнень. Положення та швидкість використовуються для оновлення руху, сила впливає на прискорення, а маса визначає, наскільки тіло реагує на зовнішні дії та імпульси. Коефіцієнти тертя й пружності застосовуються під час обробки контактів і визначають, чи буде тіло ковзати, сповільнюватися або відскакувати після зіткнення.

Окремо в рушії передбачено підтримку статичних тіл. Такі об'єкти не змінюють свого положення під дією сил і використовуються для нерухомих елементів сцени, наприклад підлоги, стін або обмежувачів. Це спрощує побудову демонстраційних сцен і дає змогу поєднувати рухомі тіла з нерухомим середовищем.

Фрагмент реалізації фізичного тіла та структури його початкового опису наведено в додатку Б.

4.2 Реалізація форми об'єктів

Форма фізичного тіла визначає його геометрію для виявлення колізій. У Enjoyer2D підтримуються дві базові форми: коло та прямокутник. Такий набір є достатнім для демонстрації основних принципів 2D фізичної симуляції та спрощує реалізацію виявлення колізій [5, 7].

Геометричний опис тіла відокремлений від його фізичного стану. Форма не зберігає положення, швидкість або масу, а лише визначає тип об'єкта та його розміри. Завдяки цьому фізичний стан обробляється на рівні тіла, а форма використовується лише для перевірки меж і зіткнень.

Для вибору потрібного алгоритму колізії використовується тип форми. Для кола зберігається радіус, а для прямокутника - ширина та висота. Створення форм через окремі допоміжні методи спрощує ініціалізацію об'єктів і зменшує ризик некоректного заповнення параметрів.

Фрагмент реалізації форм фізичних тіл наведено в додатку Б.

4.3 Реалізація фізичного світу та кроку симуляції

Фізичний світ **World** зберігає список тіл, поточні контакти, гравітацію та налаштування кількості ітерацій вирішувача. Основна робота фізичного світу виконується під час кроку симуляції. За один такий крок рушій оновлює стан тіл, формує список контактів, виконує позиційну корекцію та змінює швидкості тіл відповідно до імпульсного вирішення зіткнень. Така послідовна обробка є типовою для фізичних рушіїв реального часу, де симуляція просувається вперед дискретними часовими кроками [3].

Важливою особливістю реалізації є те, що фізичний світ працює лише з фізичними даними. Він не містить логіки рендерингу, обробки введення або побудови інтерфейсу користувача. Завдяки цьому рушій залишається незалежним від демонстраційного застосунку, а його роботу можна перевіряти окремо за допомогою модульних тестів.

Фрагмент реалізації фізичного світу та кроку симуляції наведено в додатку Б.

4.4 Реалізація чисельного інтегрування

У Enjoyer2D для оновлення руху тіл використовується напівнеявний метод Ейлера. У цьому підході спочатку оновлюється швидкість тіла з урахуванням прискорення, а потім положення змінюється на основі вже оновленої швидкості, такий метод є простим для реалізації та достатнім для навчального 2D рушія.

Перед інтегруванням перевіряється, чи є тіло статичним. Якщо об'єкт не повинен рухатися, його положення не змінюється, а накопичені сили очищуються. Для динамічних тіл прискорення визначається на основі глобальної гравітації, прикладених сил і маси тіла. Це дозволяє враховувати як постійну дію гравітації, так і зовнішні сили, що можуть бути додані під час симуляції.

Оновлення відбувається у два етапи: спочатку змінюється швидкість, після чого оновлюється положення тіла. Після завершення інтегрування накопичені сили очищуються, щоб одноразово прикладена сила не впливала на тіло повторно на наступному фізичному кроці.

Фрагмент реалізації чисельного інтегрування наведено в додатку Б.

4.5 Реалізація виявлення колізій

Виявлення колізій у Enjoyer2D реалізовано для двох базових форм: прямокутника та кола. Для прямокутників використовується AABB перевірка, а для кіл – перевірка відстані між центрами. Такі методи є простими, швидкими та добре підходять для навчальної реалізації базового 2D рушія [5]. У складніших системах для оптимізації виявлення колізій можуть застосовуватися просторові структури, зокрема BVH (Bounding Volume Hierarchy), однак у поточній реалізації рушія використовується пряме попарне порівняння тіл [15].

AABB задається двома точками: мінімальною та максимальною межею об'єкта. Для прямокутника ці межі визначаються на основі його центра, ширини та висоти, а для кола формується квадратна область навколо центра з урахуванням радіуса. Такі межі використовуються як для перевірки перетину, так і для debug-візуалізації.

Перевірка колізій між тілами виконується залежно від типів їхніх форм. Для пари прямокутників застосовується AABV перевірка, для двох кіл - аналіз відстані між центрами, а для пари прямокутник-коло - пошук найближчої точки прямокутника до центра кола. Результатом перевірки є контактні дані, які далі використовуються під час вирішення зіткнення.

Фрагмент реалізації виявлення колізій наведено в додатку Б.

4.6 Реалізація контактів і debug-даних

Після виявлення зіткнення рушію необхідно зберегти дані про контакт між тілами. Контакт містить не лише факт зіткнення, а й інформацію, потрібну для подальшої обробки: індекси тіл, точку контакту, нормаль і глибину проникнення. Подібні контактні дані використовуються у фізичних рушіях для позиційної корекції та обчислення реакції зіткнення [6, 24].

Формування контактів відбувається під час перевірки пар тіл у фізичному світі. Якщо для певної пари виявлено зіткнення, рушій створює контактні дані та додає їх до списку активних контактів. Надалі цей список використовується вирішувачем для розділення тіл і зміни їхніх швидкостей.

Контактні дані застосовуються не лише для фізичних розрахунків, а й для debug-візуалізації в демонстраційному застосунку. Наприклад, точка контакту може бути відображена на сцені, а нормаль – у вигляді вектора. Це допомагає візуально перевіряти правильність виявлення колізій і краще розуміти, як рушій обробляє зіткнення.

Фрагмент реалізації контактних даних наведено в додатку Б.

4.7 Реалізація імпульсного вирішення зіткнень

Імпульсне вирішення зіткнень використовується для зміни швидкостей тіл після контакту. Основна ідея полягає в тому, щоб обчислити імпульс уздовж нормалі зіткнення та застосувати його до обох тіл з урахуванням їхньої оберненої маси.

Під час обробки контакту спочатку визначається відносна швидкість тіл. Якщо об'єкти вже рухаються одне від одного, додатковий імпульс не застосовується. Якщо ж тіла продовжують зближуватися, рушій обчислює величину імпульсу вздовж нормалі контакту та змінює швидкості тіл.

Коефіцієнт пружності визначає, наскільки сильно тіло відскакує після зіткнення. У реалізації використовується менше значення пружності з двох тіл, щоб реакція контакту не була “пружнішою”, ніж дозволяє один з об'єктів. Маса також впливає на результат: важчі тіла отримують меншу зміну швидкості, а статичні тіла не змінюють швидкість взагалі.

Фрагмент реалізації імпульсного вирішення зіткнень наведено в додатку Б.

4.8 Урахування тертя та пружності

У Enjoyer2D під час обробки зіткнень враховуються коефіцієнти пружності та тертя. Пружність визначає, наскільки сильно тіло відскакує після контакту, а тертя зменшує ковзання тіл уздовж поверхні зіткнення. Обробка контактів із тертям є однією зі складних частин фізичної симуляції, особливо у сценах із багатьма одночасними контактами [16, 24].

Пружність використовується під час обчислення основного імпульсу зіткнення. Вона впливає на те, яку частину швидкості тіло зберігає після удару. Для контакту використовується менше значення пружності з двох тіл, що дозволяє уникнути надмірного відскоку в ситуації, коли один з об'єктів має низьку пружність.

Тертя обробляється після основного імпульсу зіткнення. Для цього визначається дотичний напрям до контакту, після чого обчислюється імпульс, який зменшує рух тіл уздовж поверхні. Коефіцієнт тертя контакту враховує властивості обох тіл, тому поведінка залежить не від одного об'єкта, а від пари поверхонь, що взаємодіють.

Фрагмент реалізації врахування тертя та пружності наведено в додатку Б.

5 РЕАЛІЗАЦІЯ ДЕМОНСТРАЦІЙНОГО ЗАСТОСУНКУ

5.1 Структура демонстраційного застосунку

Демонстраційний застосунок **Enjoyer2D**, призначений для візуальної роботи з фізичним рушієм, дозволяє запускати симуляцію, створювати тіла, редагувати параметри сцени, переглядати debug-візуалізацію, перемикає демонстраційні сцени та зберігати або завантажувати власні сцени.

Структура демонстраційного застосунку поділена на кілька основних модулів:

- **app** – головна логіка застосунку, обробка стану, введення та оновлення симуляції;
- **editor** – інтерфейс редактора, панелі керування, інструменти створення й редагування тіл;
- **rendering** – відображення фізичних тіл, debug-елементів, камери та перетворення координат;
- **scenes** – створення вбудованих сцен, збереження, завантаження та пошук JSON-файлів сцен;
- **main.cpp** – точка входу, яка запускає демонстраційний застосунок.

Основним класом демонстраційного застосунку є **TestbedApp**, він відповідає за ініціалізацію SDL3, створення вікна, запуск головного циклу, оновлення стану застосунку, рендеринг і завершення роботи. Поточний стан демонстраційного застосунку зберігається у структурі **TestbedState**, яка об'єднує фізичний світ, стан сцени, налаштування редактора, debug-параметри та статистику.

Це дозволяє не зберігати всю логіку в одному файлі та відокремити різні частини демонстраційного застосунку, наприклад, рендеринг не відповідає за створення тіл, редактор не виконує фізичну симуляцію, а модуль сцен не обробляє введення користувача. Завдяки цьому код демонстраційного застосунку залишається більш зрозумілим і зручним для подальшого розширення.

5.2 Реалізація вікна, рендерингу та обробки введення

Вікно, рендеринг і введення у демонстраційному застосунку реалізовано за допомогою SDL3 та SDL_Renderer. SDL3 відповідає за створення вікна, обробку подій клавіатури й миші, а SDL_Renderer використовується для відображення фізичної сцени та debug-елементів [11].

Головний цикл застосунку побудовано за типовою схемою інтерактивної програми реального часу: обробка подій, оновлення стану та рендеринг кадру. На етапі обробки введення враховуються дії користувача: переміщення камери, масштабування, пауза, скидання сцени, одиничний крок симуляції, перемикання сцен, вибір тіл і створення нових об'єктів.

Перед обробкою події передаються до Dear ImGui, щоб інтерфейс міг захоплювати введення під час взаємодії з панелями редактора. Завдяки цьому натискання на елементи інтерфейсу не призводить до випадкового вибору або створення тіл у фізичній сцені.

Під час рендерингу кадру відображається фізичний світ, debug-візуалізація, попередній перегляд створюваного тіла та інтерфейс Dear ImGui. Такий поділ дозволяє не змішувати фізичну симуляцію з логікою введення та відображення.

Фрагмент реалізації вікна, обробки введення та рендерингу наведено в додатку В.

5.3 Реалізація редактора фізичних тіл і параметрів сцени

Редактор демонстраційного застосунку реалізовано за допомогою Dear ImGui. Він дозволяє змінювати параметри фізичного світу, обирати інструмент роботи зі сценою, створювати нові тіла та редагувати вже наявні об'єкти. Це зручно для демонстраційного застосунку, оскільки параметри симуляції можна змінювати без перекомпіляції програми [12].

Основні можливості редактора:

- зміна гравітації;

- налаштування кількості **velocity** та **positionIterations**;
- вибір інструмента роботи зі сценою;
- налаштування параметрів тіла перед створенням;
- вибір тіла в сцені;
- редагування положення, швидкості, сили, маси, тертя, пружності та розміру тіла;
- видалення вибраного тіла.

Глобальні параметри фізичного світу змінюються через панель налаштувань симуляції. Для створення нових тіл користувач обирає відповідний інструмент і задає початкові параметри майбутнього об'єкта. Якщо тіло вже створене та вибране у сцені, його властивості редагуються через інспектор тіла.

Для прямокутників додатково редагуються ширина та висота, а для кіл - радіус. Завдяки цьому користувач може змінювати фізичні та геометричні параметри об'єктів у реальному часі й одразу спостерігати вплив цих змін на симуляцію.

Фрагмент реалізації редактора фізичних тіл і параметрів сцени наведено в додатку В.

5.4 Реалізація debug-візуалізації

Debug-візуалізація використовується для наочного аналізу роботи фізичного рушія, вона дозволяє бачити не лише самі тіла, а й допоміжні фізичні дані: центри тіл, AABV області, вектори швидкості, контактні точки, контактні нормалі та траєкторії руху.

Налаштування debug-візуалізації зберігаються окремо від фізичного світу. Під час рендерингу сцени демонстраційний застосунок перевіряє ці параметри та відображає лише ті елементи, які увімкнені користувачем. Завдяки цьому можна швидко перемикатися між звичайним виглядом сцени та режимами аналізу фізичних даних.

Контактні точки та нормалі беруться зі списку контактів, який формується фізичним рушієм під час виявлення колізій. Їх відображення допомагає перевіряти правильність роботи виявлення колізій і краще розуміти, як саме рушій реагує на зіткнення.

Debug-візуалізація не впливає на симуляцію, а лише показує її внутрішні дані. Це важливо для навчального характеру Enjoyer2D, оскільки користувач може бачити, де виникають контакти, які межі мають тіла та як змінюється їхній рух у часі.

Фрагмент реалізації debug-візуалізації наведено в додатку В.

5.5 Реалізація демонстраційних сцен

Демонстраційні сцени використовуються для перевірки окремих можливостей фізичного рушія та наочного показу його роботи. Кожна сцена створює набір тіл, задає гравітацію, параметри вирішувача та початкові властивості об'єктів.

У проєкті реалізовано такі вбудовані сцени:

- **Free fall** – демонстрація падіння тіл під дією гравітації;
- **Bounce** – демонстрація впливу коефіцієнта пружності на поведінку тіл;
- **Stacking** – перевірка стабільності піраміди з прямокутників;
- **Friction** – демонстрація впливу різних коефіцієнтів тертя;
- **Stress** – перевірка роботи рушія за великої кількості тіл.

Під час завантаження сцени фізичний світ очищується, після чого створюється новий набір об'єктів відповідно до вибраного сценарію. Для спрощення цього процесу використовуються допоміжні засоби створення прямокутників і кіл, які задають форму, масу, тертя, пружність, статичність і початкову швидкість тіла.

Такий підхід дозволяє швидко формувати різні фізичні ситуації без ручного додавання кожного тіла через інтерфейс. Завдяки цьому демонстраційний

застосунок можна використовувати як середовище для перевірки й порівняння поведінки рушія в різних умовах.

Фрагмент реалізації демонстраційних сцен наведено в додатку В.

5.6 Збереження та завантаження сцен у форматі JSON

Для збереження та повторного використання фізичних сцен у Enjoyer2D реалізовано серіалізацію у форматі JSON. Такий формат зручний для опису сцени, оскільки дозволяє зберігати параметри фізичного світу та список тіл у текстовому вигляді. Для роботи з JSON у проєкті використовується бібліотека `nlohmann/json` [19].

Під час збереження у файл записуються глобальні параметри фізичного світу та дані всіх тіл сцени. Для кожного тіла зберігаються положення, швидкість, маса, статичність, коефіцієнти тертя й пружності, а також тип і параметри форми. Для прямокутника записуються ширина та висота, а для кола - радіус.

Під час завантаження сцени фізичний світ очищується, після чого з JSON-файлу відновлюються параметри світу та створюються нові тіла. Це дозволяє зберігати підготовлені сцени, повторно їх запускати й використовувати для демонстрації або тестування поведінки рушія.

Фрагмент реалізації збереження та завантаження сцен наведено в додатку В.

6 ТЕСТУВАННЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

6.1 Модульне тестування фізичного рушія

Модульне тестування фізичного рушія було спрямоване на перевірку коректності роботи основних компонентів рушія окремо від демонстраційного застосунку. Це важливо, оскільки фізична логіка не повинна залежати від SDL3, SDL_Renderer або Dear ImGui, а має перевірятися як самостійний програмний модуль.

Для написання тестів використано фреймворк Catch2, який дає змогу описувати окремі перевірки у вигляді незалежних **TEST_CASE** та перевіряти очікувані результати за допомогою макросів **REQUIRE** і **CHECK** [13]. Тести підключені до CMake проєкту та можуть виконуватися окремо від запуску демонстраційного застосунку.

Таблиця 6.1

Основні модульні тести фізичного рушія Enjoyer2D

Об'єкт тестування	Що перевіряється	Очікуваний результат	Отриманий результат
BodyDef, Body	Створення динамічного і статичного тіла	Тіло отримує задані параметри маси, форми, швидкості та стану	Успіх
World	Дія гравітації, сил і маси	Динамічні тіла змінюють швидкість і положення відповідно до фізичних параметрів	Успіх
Статичні тіла	Поведінка тіл з нульовою рухливістю	Статичні тіла не зміщуються під дією гравітації або зіткнень	Успіх
AABB	Обчислення меж для прямокутників і кіл	Межі тіла відповідають його положенню та розмірам	Успіх

Продовження таблиці 6.1

Основні модульні тести фізичного рушія Enjoyer2D

Об'єкт тестування	Що перевіряється	Очікуваний результат	Отриманий результат
ААВВ колізії	Виявлення зіткнень між прямокутниками	Для тіл, що перетинаються, створюється контакт із правильною нормаллю	Успіх
Колізії кіл	Перетин, відсутність перетину та діагональна нормаль	Алгоритм коректно визначає факт зіткнення і напрям нормалі	Успіх
Колізії прямокутник-коло	Контакт збоку, у куті та зміна порядку тіл	Контакт створюється незалежно від порядку тіл у перевірці	Успіх
Імпульсне вирішення	Розділення тіл після проникнення	Після кроку симуляції тіла не залишаються глибоко вкладеними одне в одне	Успіх
Пружність і тертя	Відскок, сповільнення ковзання та нульове тертя	Швидкість змінюється відповідно до коефіцієнтів restitution і friction	Успіх
Статистика світу	Кількість тіл, контактів і параметри вирішувача	WorldStats повертає актуальні службові дані симуляції	Успіх

Таким чином, модульне тестування підтверджує працездатність базових фізичних операцій рушія: створення тіл, інтегрування руху, виявлення колізій, формування контактів, позиційну корекцію, імпульсне вирішення зіткнень, облік тертя та збирання статистики симуляції.

6.2 Функціональне тестування демонстраційного застосунку

Функціональне тестування демонстраційного застосунку проводилося з метою перевірки того, чи може користувач повноцінно взаємодіяти з фізичним рушієм через графічний інтерфейс. На відміну від модульних тестів, які

перевіряють окремі частини рушія, функціональне тестування оцінює роботу застосунку в цілому: обробку введення, відображення сцени, зміну параметрів, створення тіл та керування симуляцією.

Таблиця 6.2

Результати функціонального тестування демонстраційного застосунку

Функція	Сценарій перевірки	Очікуваний результат	Отриманий результат
Запуск застосунку	Запустити enjoyer2d_testbed	Відкривається вікно з демонстраційною сценою	Успіх
Вибір сцени	Обрати іншу сцену через панель інтерфейсу	Поточний фізичний світ очищується та завантажується нова сцена	Успіх
Пауза симуляції	Натиснути кнопку паузи або відповідну клавішу	Фізичне оновлення зупиняється, але рендеринг продовжується	Успіх
Одиничний крок симуляції	У режимі паузи виконати single step	Світ оновлюється рівно на один фізичний крок	Успіх
Скидання сцени	Натиснути кнопку reset	Сцена повертається до початкового стану	Успіх
Створення тіла	Обрати інструмент створення прямокутника або кола та натиснути на сцену	У вибраній точці з'являється нове фізичне тіло	Успіх
Попередній перегляд тіла	Увімкнути інструмент створення тіла і переміщати курсор	На сцені відображається preview майбутнього тіла	Успіх
Вибір тіла	Натиснути на існуюче фізичне тіло	Тіло виділяється, а його параметри з'являються в інспекторі	Успіх
Редагування параметрів тіла	Змінити масу, швидкість, тертя, пружність або розмір	Зміни застосовуються до вибраного тіла в реальному часі	Успіх

Продовження таблиці 6.2

Результати функціонального тестування демонстраційного застосунку

Функція	Сценарій перевірки	Очікуваний результат	Отриманий результат
Debug-візуалізація	Увімкнути центри тіл, AABV, нормалі, точки контактів або траєкторії	На сцені відображаються відповідні допоміжні елементи	Успіх
Збереження сцени	Зберегти поточну сцену у JSON файл	Файл сцени створюється у відповідній директорії	Успіх
Завантаження сцени	Обрати раніше збережену сцену	Фізичний світ відновлюється з JSON-файлу	Успіх

За результатами функціонального тестування встановлено, що демонстраційний застосунок забезпечує основні можливості, необхідні для роботи з фізичним рушієм: керування симуляцією, створення та редагування тіл, перегляд debug-інформації, використання демонстраційних сцен, а також збереження і повторне завантаження сцени.

6.3 Перевірка демонстраційних сцен

Перевірка демонстраційних сцен проводилася для оцінки роботи фізичного рушія в різних типових ситуаціях: падіння тіл, зіткнення, нагромадження об'єктів, вплив тертя та робота з великою кількістю тіл. Кожна сцена демонструє окремий аспект фізичної симуляції.

Таблиця 6.3

Перевірка демонстраційних сцен Enjoyer2D

Сцена	Мета перевірки	Очікувана поведінка
Free Fall	Гравітація та інтегрування руху	Динамічні тіла падають, статична підлога залишається нерухомою
Bounce	Пружність зіткнень	Тіла з більшим restitution сильніше відстрибують
Stacking	Стабільність накопичення тіл	Конструкція з прямокутників не розлітається одразу після запуску
Friction	Вплив тертя	Тіла з різним friction по-різному сповільнюються
Stress	Робота з великою кількістю тіл	Симуляція залишається працездатною при значній кількості об'єктів

Найважливішими для перевірки стабільності є сцени **Stacking** та **Stress**. У **Stacking** виникає багато одночасних контактів між тілами, тому ця сцена показує якість позиційної корекції та роботи вирішувача. У реальних фізичних рушіях стабільність стеків є складною задачею, оскільки невеликі похибки позиційної корекції та імпульсного вирішувача можуть накопичуватися і викликати тремтіння об'єктів [22]. Сцена **Stress**, у свою чергу, дозволяє оцінити роботу рушія при збільшенні кількості тіл і контактів, оскільки навіть у сучасних дослідженнях масштабованої симуляції твердих тіл значна увага приділяється саме оптимізації виявлення колізій та обробки великої кількості контактів у реальному часі [23].

Отже, реалізований набір сцен покриває основні можливості Enjoyer2D: гравітацію, зіткнення, пружність, тертя, нагромадження тіл та роботу з великою кількістю об'єктів.

6.4 Оцінювання продуктивності роботи рушія

Оцінювання продуктивності проводилося для перевірки того, чи здатний рушій працювати в режимі реального часу при різній кількості тіл і контактів. У

демонстраційному застосунку для цього відображаються FPS, час фізичного кроку, кількість тіл, кількість контактів та параметри вирішувача.

Оскільки рушій використовує фіксований крок часу, важливо, щоб один фізичний крок виконувався швидше, ніж триває проміжок між оновленнями симуляції. Для частоти 60 фізичних кроків на секунду цей проміжок становить приблизно **16,67 мс (1/60)**.

Основними показниками продуктивності є:

- **FPS** – оцінювання плавності відображення;
- **Physics step time** – час виконання одного фізичного кроку;
- **Body count** – кількість тіл у сцені;
- **Contact count** – кількість активних контактів;
- **Solver iterations** – кількість ітерацій вирішення зіткнень.

Найбільше навантаження створюють сцени **Stacking** та **Stress**. У першому випадку складність пов'язана з великою кількістю одночасних контактів між тілами, а в другому — з великою кількістю об'єктів і перевірок колізій. У поточній реалізації виявлення колізій виконується попарно, що є простим і зрозумілим для навчального рушія, але гірше масштабується при збільшенні кількості тіл. Для оптимізації таких задач у фізичних рушіях можуть використовуватися етап грубої перевірки (broad phase), просторове розбиття або BVH-структури [14, 15].

За результатами перевірки можна зробити висновок, що Enjoyer2D має достатню продуктивність для навчальних і демонстраційних сцен. Подальше покращення продуктивності доцільно пов'язати з оптимізацією етапу виявлення колізій, оскільки саме він найбільше залежить від кількості тіл у фізичному світі [17, 23].

6.5 Обмеження реалізованого рушія

Реалізований фізичний рушій Enjoyer2D орієнтований насамперед на навчальне використання та демонстрацію базових принципів 2D фізичної

симуляції, тому в ньому свідомо реалізовано не всі можливості, які характерні для складніших фізичних рушіїв або сучасних систем симуляції твердих тіл.

Основні обмеження поточної реалізації:

- відсутнє обертання тіл, тому симуляція враховує лише поступальний рух;
- прямокутники є осьово-орієнтованими, без підтримки OBB (Oriented Bounding Box) та SAT (Separating Axis Theorem) для опуклих полігонів;
- не реалізовано з'єднання та обмеження між фізичними тілами;
- виявлення колізій виконується попарно, без оптимізації етапом грубої перевірки;
- відсутнє безперервне виявлення колізій для швидких тіл;
- вирішувач є базовим, тому при складному нагромаджені тіл можливе тремтіння або нестабільність тіл;
- не використовується GPU-прискорення або спеціальні підходи для масштабних симуляцій.

Обмеження, пов'язані з нагромадженням, контактами та тертям, є типовими для фізичних рушіїв, оскільки одночасна обробка великої кількості контактів потребує складніших математичних моделей і стабільніших вирішувачів. У сучасних роботах для цього розглядаються підходи до роботи з множинними контактами, поширенням ударних імпульсів у стеках тіл та розширеними методами симуляції твердих тіл [16, 22, 25].

Також окремою задачею є масштабування симуляції для великої кількості об'єктів. У поточній реалізації Enjoyer2D використовується простий попарний пошук колізій, що добре підходить для навчального аналізу, але не є оптимальним для великих сцен. У сучасних рішеннях для масштабних симуляцій застосовуються оптимізовані структури виявлення колізій та, за потреби, GPU-прискорення [23].

Таким чином, вказані обмеження визначають напрями подальшого розвитку проекту. У майбутньому рушій може бути розширений підтримкою обертання тіл, SAT для OBB і полігонів, оптимізацією методом грубої перевірки, з'єднаннями, безперервним виявленням колізій та покращеним вирішувачем контактів.

ВИСНОВКИ

У межах виконання кваліфікаційної роботи було створено 2D фізичний рушій Enjoyer2D та демонстраційний застосунок для наочного дослідження поведінки твердих тіл у двовимірному просторі. Розроблене програмне забезпечення забезпечує моделювання руху тіл під дією гравітації та зовнішніх сил, виявлення зіткнень, формування контактних даних, імпульсне вирішення зіткнень, урахування пружності й тертя, а також відображення debug-інформації під час роботи симуляції.

Під час роботи було проаналізовано особливості побудови 2D фізичних рушіїв, принципи чисельного інтегрування, підходи до виявлення колізій та методи обробки зіткнень. Огляд існуючих рішень дав змогу визначити вимоги до власного рушія та обґрунтувати навчальну спрямованість Enjoyer2D.

Для реалізації проєкту використано мову програмування C++, систему збирання CMake та набір допоміжних бібліотек. Архітектура програмного забезпечення побудована з розділенням на рівень фізичного рушія, рівень сцен і рівень демонстраційного застосунку, що спрощує тестування, супровід і подальше розширення системи.

У межах реалізації фізичного рушія створено структури фізичних тіл, форм, контактів і фізичного світу. Рушій підтримує прямокутні та круглі тіла, статичні й динамічні об'єкти, AABB виявлення колізій, зіткнення між прямокутниками, колами та парами прямокутник-коло. Для оновлення руху використано чисельне інтегрування з фіксованим кроком часу, а для обробки контактів – позиційну корекцію та імпульсний метод з урахуванням маси, пружності й тертя.

Демонстраційний застосунок надає користувачу можливість запускати готові сцени, створювати нові фізичні тіла, редагувати їх параметри, змінювати параметри симуляції, керувати паузою, скиданням і покроковим виконанням, а також переглядати debug-візуалізацію. Додатково реалізовано збереження та

завантаження сцен у JSON-форматі, що робить застосунок придатним не лише для демонстрації, а й для експериментів з різними конфігураціями фізичного світу.

Працездатність програмного забезпечення підтверджено модульним і функціональним тестуванням. Також виконано перевірку демонстраційних сцен, оцінювання продуктивності та запуск проєкту на MacOS і Windows.

Розроблене рішення має навчальну спрямованість і не претендує на повну заміну промислових фізичних рушіїв. Поточна реалізація не підтримує багатьох особливостей зрілих рушіїв, водночас ці обмеження визначають подальші напрями розвитку: розширення системи колізій, покращення вирішувача контактів, додавання обертання, оптимізація роботи з великою кількістю тіл і розвиток демонстраційного застосунку до повноцінного редактора сцен.

Таким чином, поставлену мету досягнуто: забезпечення можливості наочного дослідження, перевірки та навчального пояснення поведінки твердих тіл у двовимірному просторі. Enjoyer2D може використовуватися як навчальна основа для вивчення принципів фізичної симуляції та як стартова точка для подальшої розробки ігрових або симуляційних програм.

Робота пройшла апробацію. За її результатами було опубліковано наступні тези доповідей:

1. Єфимов О.Є., Колодюк А.В. Визначення вимог до фізичного рушія. Матеріали II Всеукраїнської науково-технічної конференції “Виклики та рішення в програмній інженерії”, 26 листопада 2025 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. – К.: ДУІКТ, 2025, С. 392 – 393.

2. Єфимов О.Є., Колодюк А.В. Розробка та 2D реалізація фізичного рушія на мові програмування C++. Матеріали VII Всеукраїнської науково-технічної конференції “Застосування програмного забезпечення в інформаційно-комунікаційних технологіях”, 23 квітня 2026 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. –К.: ДУІКТ, 2026, С. 135 – 137.

ПЕРЕЛІК ПОСИЛАНЬ

1. Catto E. Sequential Impulses. Game Developers Conference. 2006. URL: https://box2d.org/files/ErinCatto_SequentialImpulses_GDC2006.pdf
2. Gregory J. Game Engine Architecture. 3rd ed. Boca Raton: CRC Press, 2018. 1240 p.
3. Millington I. Game Physics Engine Development: How to Build a Robust Commercial-Grade Physics Engine for your Game. 2nd ed. CRC Press, 2010. 524 p.
4. Власний Game Engine: як і навіщо створювати ігровий рушій // GameDev DOU. URL: <https://gamedev.dou.ua/blogs/how-and-why-to-create-a-game-engine/>
5. Ericson C. Real-Time Collision Detection. Boca Raton: CRC Press, 2005. 632 p.
6. Gregorius D. Physics for Game Programmers: Robust Contact Creation for Physics Simulation. Game Developers Conference. 2015. URL: <https://www.gdcvault.com/play/1022194/Physics-for-Game-Programmers-Robust>
7. Box2D Documentation. URL: <https://box2d.org/documentation/>
8. Chipmunk2D Manual. URL: <https://chipmunk-physics.net/release/ChipmunkLatest-Docs/>
9. Matter.js Documentation. URL: <https://brm.io/matter-js/docs/>
10. CMake. FetchContent documentation. URL: <https://cmake.org/cmake/help/latest/module/FetchContent.html>
11. SDL3 Documentation. URL: <https://wiki.libsdl.org/SDL3/FrontPage>
12. Dear ImGui official repository. URL: <https://github.com/ocornut/imgui>
13. Catch2 documentation. URL: <https://catch2-temp.readthedocs.io/en/latest/>
14. Montaut L., Le Lidec Q., Petrik V., Sivic J., Carpentier J. Collision Detection Accelerated: An Optimization Perspective. Robotics: Science and Systems, 2022. URL: <https://www.roboticsproceedings.org/rss18/p039.pdf>
15. Li X., Song S., Yao J., Zhang H., Zhou R., Hong Q. Efficient collision detection using hybrid medial axis transform and BVH for rigid body simulation. Graphical Models. 2023. Vol. 126. Article 101180.

URL: <https://www.sciencedirect.com/science/article/pii/S1524070323000103>

16. Halm M., Posa M. Set-valued rigid-body dynamics for simultaneous, inelastic, frictional impacts. The International Journal of Robotics Research. 2024. Vol. 43, № 10. P. 1594–1628. URL: <https://journals.sagepub.com/doi/10.1177/02783649241236860>

17. Rubanova Y., Lopez-Guevara T., Allen K. R., Whitney W. F., Stachenfeld K., Pfaff T. Learning rigid-body simulators over implicit shapes for large-scale scenes and vision. NeurIPS, 2024.

URL: https://papers.nips.cc/paper_files/paper/2024/hash/e3abc125ecacb71786cefb9f67b08c5d-Abstract-Conference.html

18. GLM Documentation. URL: <https://glm.g-truc.net/>

19. nlohmann/json Documentation. URL: <https://json.nlohmann.me/>

20. Бернатович О., Стеценко І. Методи та програмні засоби фізичної симуляції. 2023. URL: <https://ela.kpi.ua/items/5f8605f9-6ca5-4743-8026-cf9b37815ccb>

21. Вовк Л., Недавні Д. Фізика в іграх. Створення ігрового двигуна на основі фізичних процесів.

URL: <https://openarchive.nure.ua/server/api/core/bitstreams/be670038-8ef7-4b27-b6cf-79c82c4c93ef/content>

22. Xiong Z., Leach A., Thomas G., Sueda S. Two-Pass Shock Propagation for Stable Stacking with Gauss-Seidel. Proceedings of the ACM on Computer Graphics and Interactive Techniques. 2025. Vol. 8, № 4. URL: <https://dl.acm.org/doi/10.1145/3747867>

23. Sung J., Hong M. Toward Real-Time Scalable Rigid-Body Simulation Using GPU-Optimized Collision Detection and Response. Mathematics. 2025. Vol. 13, № 19. URL: <https://www.mdpi.com/2227-7390/13/19/3230>

24. Andrews S., Erleben K., Ferguson Z. Contact and Friction Simulation for Computer Graphics. SIGGRAPH Courses, 2022.

URL: https://siggraphcontact.github.io/assets/files/SIGGRAPH22_friction_contact_notes.pdf

25. Seabra R., Lopes P., Pereira J. Survey of Rigid Body Simulation with Extended Position Based Dynamics. arXiv, 2023. URL: <https://arxiv.org/abs/2311.09327>

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ



ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ



Розробка та реалізація фізичного рушія на мові програмування C++

Виконав студент 4 курсу
групи ПД-41
Єфимов Олександр Євгенович
Керівник роботи

асистент кафедри ІПЗ Колодюк Андрій Васильович

Київ – 2026

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: розроблення, програмна реалізація та перевірка навчально-орієнтованого 2D фізичного рушія Enjoyer2D мовою C++ з демонстраційним застосунком для візуального дослідження руху, зіткнень, тертя та пружності твердих тіл.

Об'єкт дослідження: процес симуляції фізичних взаємодій твердих тіл у середовищі реального часу.

Предмет дослідження: методи, алгоритми та програмні засоби реалізації фізичного рушія на C++.

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Проаналізувати теоретичні відомості про моделювання руху твердих тіл у двовимірному просторі та підходи до побудови фізичних рушіїв.
2. Дослідити існуючі 2D фізичні рушії та визначити їхні переваги, обмеження й придатність для навчального використання.
3. Обґрунтувати вибір програмних засобів, бібліотек і архітектурного підходу для створення навчально-орієнтованого фізичного рушія.
4. Сформувати структуру програмного забезпечення, яка дозволяє наочно демонструвати роботу фізичної симуляції та аналізувати її окремі етапи.
5. Провести перевірку розробленого рішення за допомогою тестування, демонстраційних сцен і debug-візуалізації. **3**

АНАЛІЗ АНАЛОГІВ

Характеристика/рушій	Box2D	Chipmunk2D	Matter.js	Enjoyer2D
Повноцінне інтерактивне демонстраційне середовище	+	-	+	+
Debug-візуалізація фізичних параметрів	+	-	+	+
Зміна параметрів тіл у реальному часі	+	-	+	+
Можливість використання як готового професійного рушія	+	+	+	+/-
Орієнтація на навчальне вивчення внутрішньої будови рушія	-	-	-	+
Простота архітектури для покрокового аналізу	-	-	+/-	+
Повний контроль над реалізацією рушія	-	-	-	+

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги

1. Моделювання руху твердих тіл у двовимірному просторі з урахуванням положення, швидкості, сили, маси та гравітації.
2. Підтримка базових форм фізичних тіл (прямокутники та кола), а також статичних і динамічних тіл.
3. Виявлення колізій між тілами та формування контактних даних: нормалі зіткнення, точки контакту й глибини проникнення.
4. Обробка зіткнень імпульсним методом з урахуванням коефіцієнтів пружності та тертя.
5. Використання фіксованого кроку часу для стабільнішої симуляції незалежно від частоти рендерингу.
6. Наявність інтерактивного демонстраційного застосунку для створення, вибору, редагування та видалення фізичних тіл.
7. Можливість зміни параметрів тіл і фізичного світу в реальному часі.
8. Підтримка керування симуляцією: пауза, продовження, скидання сцени та виконання фізичного кроку.
9. Наявність debug-візуалізації.
10. Наявність демонстраційних сцен і можливість збереження та завантаження сцен у форматі JSON.

5

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Нефункціональні вимоги

1. Програмне забезпечення має бути реалізоване мовою C++ із чітким поділом на рівень фізичного рушія, рівень сцен і рівень демонстраційного застосунку.
2. Основні етапи фізичного кроку мають бути логічно відокремлені: інтегрування, виявлення колізій, формування контактів, позиційна корекція та імпульсне вирішення зіткнень.
3. Програмне забезпечення має працювати в режимі реального часу з використанням фіксованого фізичного кроку (1/60).
4. Ключові компоненти фізичного рушія мають бути придатними для модульного тестування;
5. Реалізація має передбачати можливість подальшого розширення, зокрема додавання нових типів колізій, обертання тіл, з'єднань та оптимізацій продуктивності.
6. Програмне забезпечення має бути придатним для збирання та запуску на основних операційних системах: Windows, MacOS, Linux.

6

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

CMake – система збирання проєкту та підключення зовнішніх бібліотек.

SDL3 – створення вікна, обробка подій клавіатури й миші, базова взаємодія з ОС.

SDL_Renderer – 2D-рендеринг тіл, debug-візуалізації, ліній, прямокутників, кіл та допоміжних елементів сцени.

Dear ImGui – побудова інтерактивного інтерфейсу testbed-застосунку: панелі керування, інспектор об'єктів, налаштування сцени.

GLM – робота з векторами та математичними операціями, необхідними для фізичних розрахунків.

Catch2 – автоматичне тестування окремих модулів фізичного рушія.

nlohmann/json – бібліотека для роботи з форматом JSON.

 **Git** – контроль версій, збереження історії змін і керування кодом проєкту.

 **CLion** /  **Microsoft Visual Studio** – середовища розробки для написання, збирання та налагодження C++ коду.

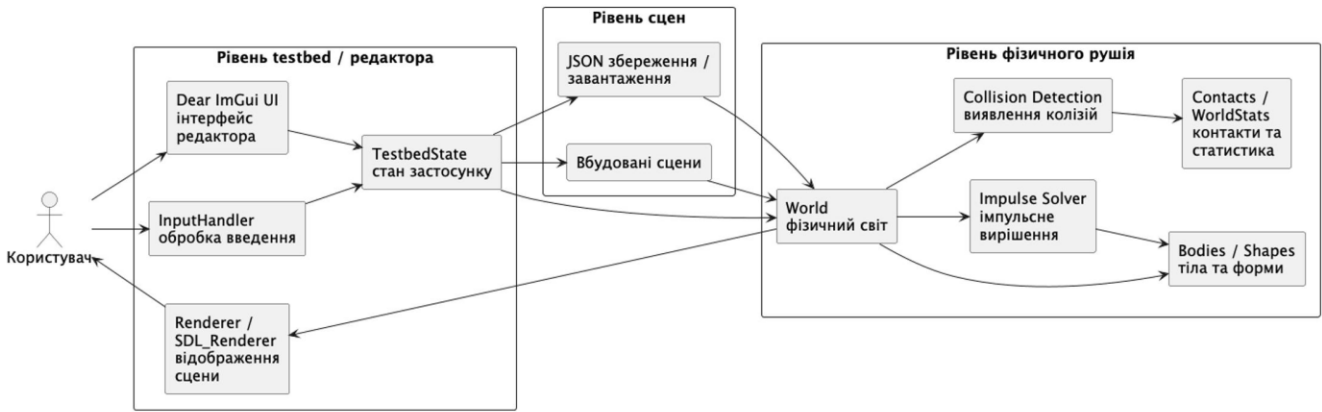
7

Діаграма варіантів використання

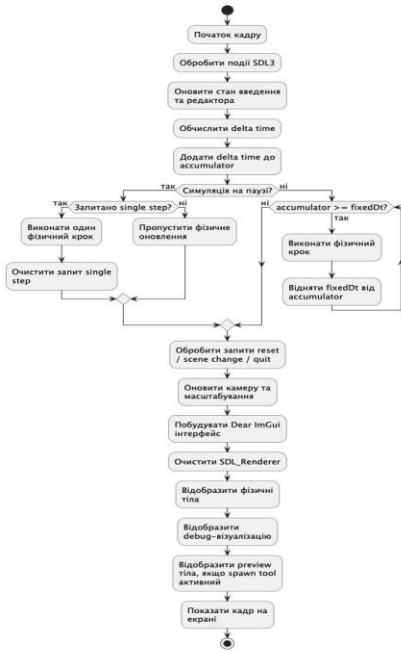


8

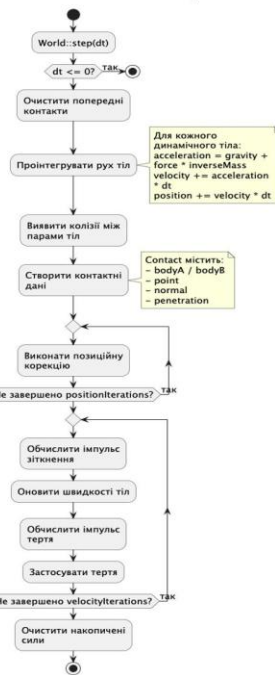
Схема функціонування програми



Діаграма діяльності демонстраційного застосунку

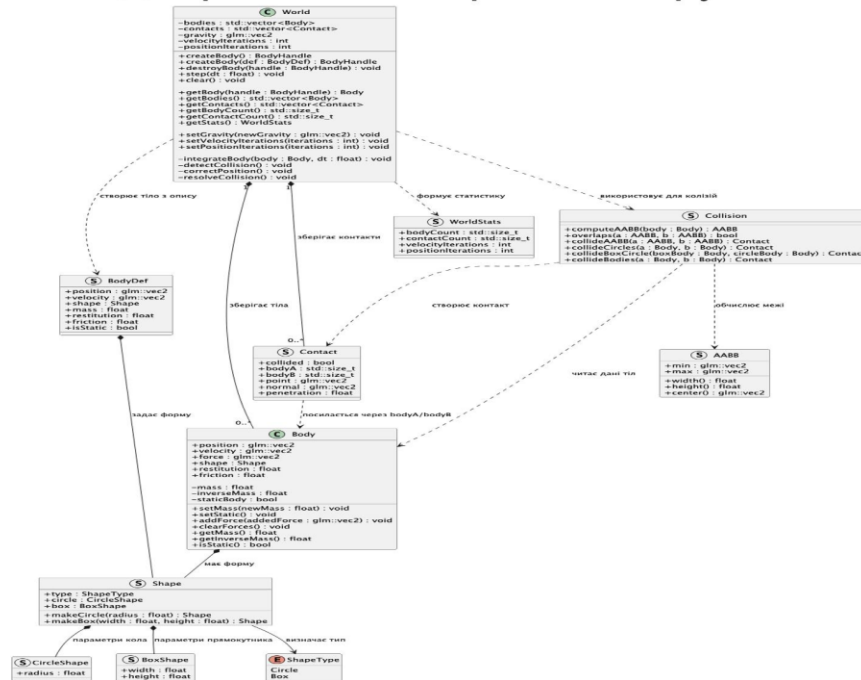


Діаграма діяльності фізичного кроку



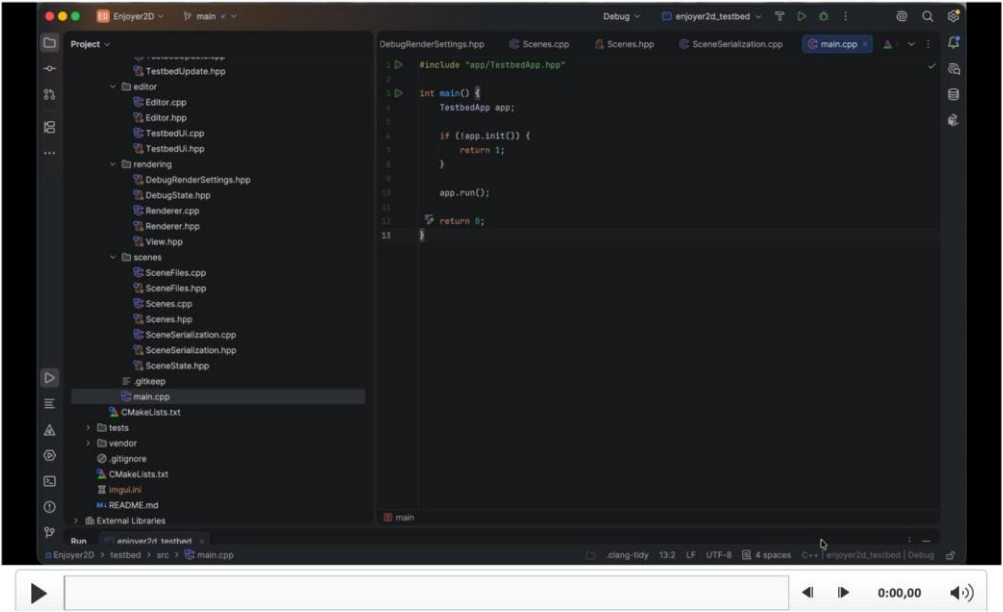
11

Діаграма класів фізичного рушія



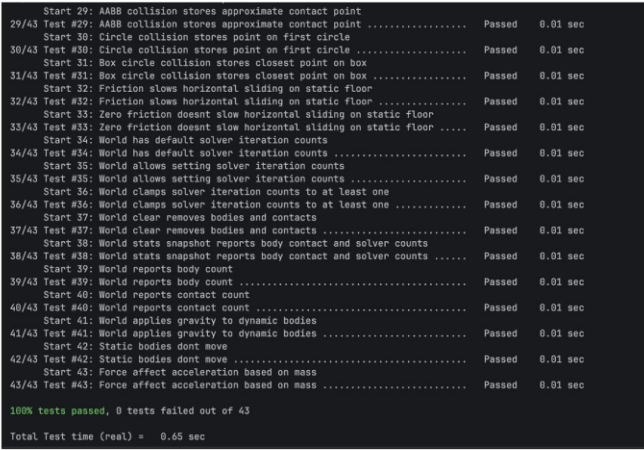
12

ЕКРАННІ ФОРМИ

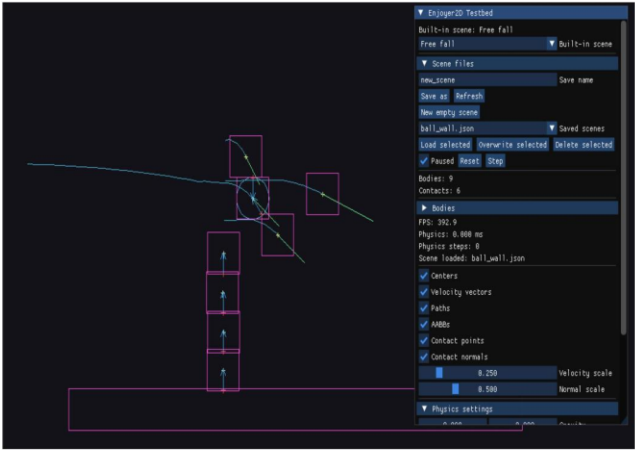


13

Тестування застосунку



Модульне тестування фізичного рушія



Демонстраційне та debug тестування

14

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

Тези доповідей:

1. Єфимов О.Є., Колодюк А.В. Визначення вимог до фізичного рушія. Матеріали II Всеукраїнської науково-технічної конференції «Виклики та рішення в програмній інженерії», 26 листопада 2025 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. – К.: ДУІКТ, 2025, С. 392 – 393.
2. Єфимов О.Є., Колодюк А.В. Розробка та 2D реалізація фізичного рушія на мові програмування C++. Матеріали VII Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 23 квітня 2026 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. – К.: ДУІКТ, 2026, С. 135 – 137.

15

ВИСНОВКИ

1. Проаналізовано теоретичні основи моделювання руху твердих тіл у двовимірному просторі, зокрема інтегрування руху, виявлення колізій, формування контактів і вирішення зіткнень. У результаті визначено базові фізичні процеси, які мають бути реалізовані в рушії: оновлення положення та швидкості тіл, обробка зіткнень між об'єктами, розрахунок контактних даних, а також урахування маси, тертя та пружності.
2. Досліджено існуючі 2D-фізичні рушії, визначено функціональні та нефункціональні вимоги, а також встановлено, що навчальне спрямування Enjoe2D є його основною відмінністю від готових аналогів.
3. Обґрунтовано вибір CMake, GLM, SDL3, SDL_Renderer, Dear ImGui, Catch2 та plohmann/json як засобів для створення рушія і демонстраційного застосунку.
4. Сформовано архітектуру з поділом на фізичний рушій, рівень сцен і демонстраційний застосунок, що дозволяє окремо аналізувати фізичну логіку та візуальну частину.
5. Перевірено розроблене рішення за допомогою тестування, демонстраційних сцен і debug-візуалізації.
6. Реалізовано програмне забезпечення, яке забезпечує наочне дослідження, перевірку та навчальне пояснення поведінки твердих тіл у двовимірному просторі під час дії сил, зіткнень, тертя та пружності.

16

ДОДАТОК Б. ФРАГМЕНТИ РЕАЛІЗАЦІЇ ФІЗИЧНОГО РУШІЯ ENJOYER2D

4.1

```

struct BodyDef {
    glm::vec2 position {0.0f, 0.0f};
    glm::vec2 velocity {0.0f, 0.0f};

    Shape shape {};

    float mass {1.0f};
    float restitution {0.2f};
    float friction {0.5f};

    bool isStatic {false};
};

class Body {
public:
    glm::vec2 position {0.0f, 0.0f};
    glm::vec2 velocity {0.0f, 0.0f};
    glm::vec2 force {0.0f, 0.0f};

    Shape shape {};

    float restitution {0.2f};
    float friction {0.5f};

    void setMass(float newMass);
    void setStatic();
    void addForce(const glm::vec2& addedForce);
    void clearForces();

private:
    float mass {1.0f};
    float inverseMass {1.0f};
    bool staticBody {false};
};

```

4.3

```

void step(float dt) {
    if (dt <= 0) {
        return;
    }

    contacts.clear();

    for (Body& body : bodies) {
        integrateBody(body, dt);
    }

    detectCollision();
}

```

4.2

```

enum class ShapeType {
    Circle,
    Box
};

struct CircleShape {
    float radius {0.5f};
};

struct BoxShape {
    float width {1.0f};
    float height {1.0f};
};

struct Shape {
    ShapeType type {ShapeType::Box};

    CircleShape circle {};
    BoxShape box {};

    static Shape makeCircle(float radius) {
        Shape shape;
        shape.type = ShapeType::Circle;
        shape.circle.radius = radius;
        return shape;
    }

    static Shape makeBox(float width, float height) {
        Shape shape;
        shape.type = ShapeType::Box;
        shape.box.width = width;
        shape.box.height = height;
        return shape;
    }
};

```

4.4

```

void integrateBody(Body& body, float dt)
const{
    if (body.isStatic() ||
    body.getInverseMass() <= 0.0f) {
        body.clearForces();
        return;
    }

    glm::vec2 acceleration = gravity +
    body.force * body.getInverseMass();

    body.velocity += acceleration * dt;
    body.position += body.velocity * dt;
}

```

```

    for (int i = 0; i < positionIterations;
++i) {
        correctPosition();
    }

    for (int i = 0; i < velocityIterations;
++i) {
        resolveCollision();
    }
}

```

4.5

```

[[nodiscard]] inline AABB computeAABB(const
Body& body) {
    const glm::vec2 center = body.position;

    if (body.shape.type == ShapeType::Circle) {
        const float radius =
body.shape.circle.radius;
        return {center - glm::vec2 {radius,
radius},
center + glm::vec2 {radius, radius},};
    }

    const float halfWidth =
body.shape.box.width * 0.5f;
    const float halfHeight =
body.shape.box.height * 0.5f;

    return {center - glm::vec2 {halfWidth,
halfHeight},
center + glm::vec2 {halfWidth,
halfHeight},};
}

[[nodiscard]] inline AABB computeAABB(const
Body& body) {
    const glm::vec2 center = body.position;

    if (body.shape.type == ShapeType::Circle) {
        const float radius =
body.shape.circle.radius;
        return {center - glm::vec2 {radius,
radius},
center + glm::vec2 {radius, radius},};
    }

    const float halfWidth =
body.shape.box.width * 0.5f;
    const float halfHeight =
body.shape.box.height * 0.5f;

    return {center - glm::vec2 {halfWidth,
halfHeight},
center + glm::vec2 {halfWidth,
halfHeight},};
}

```

4.7

```

    body.clearForces();
}

```

4.6

```

struct Contact {
    bool collided {false};

    std::size_t bodyA {0};
    std::size_t bodyB {0};

    glm::vec2 point {0.0f, 0.0f};
    glm::vec2 normal {0.0f, 0.0f};
    float penetration {0.0f};
};

void detectCollision() {
    for (std::size_t i = 0; i <
bodies.size(); ++i) {
        for (std::size_t j = i + 1; j <
bodies.size(); ++j) {
            Contact contact =
collideBodies(bodies[i], bodies[j]);

            if (contact.collided) {
                contact.bodyA = i;
                contact.bodyB = j;
            }
        }
    }
}

```

4.8

```

const glm::vec2 relativeVelocity = b.velocity
- a.velocity;
const float velocityAlongNormal =
glm::dot(relativeVelocity, contact.normal);

if (velocityAlongNormal > 0.0f) {
    continue;
}

const float restitution =
std::min(a.restitution, b.restitution);

const float impulseMagnitude = -(1.0f +
restitution) *
velocityAlongNormal/totalInverseMass;

const glm::vec2 impulse = impulseMagnitude *
contact.normal;

a.velocity -= impulse * a.getInverseMass();
b.velocity += impulse * b.getInverseMass();

```

```

const glm::vec2
relativeVelocityAfterNormalImpulse =
b.velocity - a.velocity;

glm::vec2 tangent =
relativeVelocityAfterNormalImpulse -
glm::dot(relativeVelocityAfterNormalImpulse,
contact.normal) * contact.normal;

if (glm::length(tangent) > 0.0f) {
    tangent = glm::normalize(tangent);

    const float velocityAlongTangent =
glm::dot(relativeVelocityAfterNormalImpulse,
tangent);

    const float frictionImpulseMagnitude = -
velocityAlongTangent / totalInverseMass;

    const float friction =
std::sqrt(a.friction * b.friction);

    const float maxFrictionImpulse =
impulseMagnitude * friction;

    const float
clampedFrictionImpulseMagnitude =
std::clamp(frictionImpulseMagnitude, -
maxFrictionImpulse, maxFrictionImpulse);

    const glm::vec2 frictionImpulse =
clampedFrictionImpulseMagnitude * tangent;

    a.velocity -= frictionImpulse *
a.getInverseMass();
    b.velocity += frictionImpulse *
b.getInverseMass();
}

```

ДОДАТОК В. ФРАГМЕНТИ РЕАЛІЗАЦІЇ ДЕМОНСТРАЦІЙНОГО ЗАСТОСУНКУ

5.2

```
bool TestbedApp::init() {
    if (!SDL_Init(SDL_INIT_VIDEO)) {
        return false;
    }

    if (!SDL_CreateWindowAndRenderer("Enjoyer2D
Testbed", 1280, 720, 0, &window, &renderer)) {
        SDL_Quit();
        return false;
    }

    SDL_SetRenderDrawBlendMode(renderer,
SDL_BLENDMODE_BLEND);

    ImGui::CreateContext();
    ImGui_ImplSDL3_InitForSDLRenderer(window,
renderer);
    ImGui_ImplSDLRenderer3_Init(renderer);

    loadTestbedScene(testbed, SceneId::FreeFall);

    previousCounter = SDL_GetPerformanceCounter();
    frequency =
static_cast<double>(SDL_GetPerformanceFrequency());

    running = true;

    return true;
}

void TestbedApp::run() {
    while (running) {
        handleEvents();
        updateFrame();
        renderFrame();
    }
}

void TestbedApp::handleEvents() {
    SDL_Event event;

    while (SDL_PollEvent(&event)) {
        handleEvent(event, testbed, input, view,
running);
    }
}

void TestbedApp::renderFrame() {
    ImGui_ImplSDLRenderer3_NewFrame();
    ImGui_ImplSDL3_NewFrame();
    ImGui::NewFrame();

    drawTestbedUi(testbed, debugRender);
}
```

5.3

```
glm::vec2 gravity =
testbed.world.getGravity();

if (ImGui::DragFloat2("Gravity",
&gravity.x, 0.1f)) {
    testbed.world.setGravity(gravity);
}

int velocityIterations =
testbed.world.getVelocityIterations();
if (ImGui::SliderInt("Velocity
iterations", &velocityIterations, 1,
30)) {

testbed.world.setVelocityIterations(ve
locityIterations);
}

ImGui::DragFloat("Mass##Spawn",
&testbed.editor.spawn.mass, 0.1f,
0.1f, 100.0f);
ImGui::DragFloat2("Velocity##Spawn",
&testbed.editor.spawn.velocity.x,
0.05f, -20.0f, 20.0f);
ImGui::DragFloat("Friction##Spawn",
&testbed.editor.spawn.friction, 0.01f,
0.0f, 2.0f);
ImGui::DragFloat("Restitution##Spawn",
&testbed.editor.spawn.restitution,
0.01f, 0.0f, 1.0f);
    ImGui::Checkbox("Static##Spawn",
&testbed.editor.spawn.isStatic);

ImGui::DragFloat2("Position",
&body.position.x, 0.05f);
ImGui::DragFloat2("Velocity",
&body.velocity.x, 0.05f);
ImGui::DragFloat2("Force",
&body.force.x, 0.05f);

ImGui::DragFloat("Restitution",
&body.restitution, 0.01f, 0.0f, 1.0f);
ImGui::DragFloat("Friction",
&body.friction, 0.01f, 0.0f, 2.0f);
}
```

```

    SDL_SetRenderDrawColor(renderer, 18, 18, 24,
255);
    SDL_RenderClear(renderer);

    const RenderContext renderContext {view,
debugRender, testbed.debug,
testbed.editor.selectedBodyIndex};

    drawWorld(renderer, testbed.world,
renderContext);

    ImGui::Render();

ImGui_ImplSDLRenderer3_RenderDrawData(ImGui::GetDra
wData(), renderer);

    SDL_RenderPresent(renderer);
}

```

5.4

```

struct DebugRenderSettings {
    bool showCenters = false;
    bool showVelocities = false;
    bool showAabbs = false;
    bool showPath = false;
    bool showContactPoints = false;
    bool showContactNormals = false;

    float velocityScale = 0.25f;
    float normalScale = 0.5f;
};

if (context.debugSettings.showVelocities) {
    SDL_SetRenderDrawColor(renderer, 120, 255,
160, 255);

    for (const Enjoyer2D::Body& body :
world.getBodies()) {
        if (!body.isStatic()) {
            drawBodyVelocity(renderer, body,
context.view,
context.debugSettings.velocityScale);
        }
    }
}

if (context.debugSettings.showAabbs) {
    SDL_SetRenderDrawColor(renderer, 255, 80,
220, 255);

    for (const Enjoyer2D::Body& body :
world.getBodies()) {
        drawAabb(renderer,
Enjoyer2D::computeAABB(body), context.view);
    }
}

if (context.debugSettings.showContactPoints) {
    SDL_SetRenderDrawColor(renderer, 255, 80,
80, 255);
}

```

5.5

```

void loadScene(Enjoyer2D::World& world,
SceneId scene) {
    world.clear();

    switch (scene) {
        case SceneId::FreeFall: {
            world.setGravity({0.0f, -
9.8f});
            world.setVelocityIterations(8);
            world.setPositionIterations(3);
            // створення тіл сцени
            break;
        }

        case SceneId::Bounce: {
            world.setGravity({0.0f, -
9.8f});
            world.setVelocityIterations(12);
            world.setPositionIterations(4);
            // створення тіл сцени
            break;
        }

        // інші сцени
    }
}

void addCircle(Enjoyer2D::World& world,
const glm::vec2& position, float radius,
bool isStatic, float mass, float friction,
float restitution, const glm::vec2&
velocity = {0.0f, 0.0f}) {
    Enjoyer2D::BodyDef body;
    body.position = position;
    body.shape =
Enjoyer2D::Shape::makeCircle(radius);
    body.isStatic = isStatic;
    body.mass = mass;
    body.friction = friction;
    body.restitution = restitution;
}

```

```

    for (const Enjoyer2D::Contact& contact :
world.getContacts()) {
        drawContactPoint(renderer, contact,
context.view);
    }
}

if (context.debugSettings.showContactNormals)
{
    SDL_SetRenderDrawColor(renderer, 80, 180,
255, 255);

    for (const Enjoyer2D::Contact& contact :
world.getContacts()) {
        drawContactNormal(renderer, contact,
context.view,
context.debugSettings.normalScale);
    }
}

```

```

    body.velocity = velocity;
world.createBody(body);
}

```

5.6

```

json bodyToJson(const Enjoyer2D::Body& body) {

```

```

    json bodyJson;

```

```

    bodyJson["position"] = {body.position.x, body.position.y};
    bodyJson["velocity"] = {body.velocity.x, body.velocity.y};
    bodyJson["mass"] = body.getMass();
    bodyJson["static"] = body.isStatic();
    bodyJson["friction"] = body.friction;
    bodyJson["restitution"] = body.restitution;

```

```

    if (body.shape.type == Enjoyer2D::ShapeType::Box) {
        bodyJson["shape"] = {{"type", "box"}, {"width", body.shape.box.width}, {"height",
body.shape.box.height}};
    }

```

```

    if (body.shape.type == Enjoyer2D::ShapeType::Circle) {
        bodyJson["shape"] = {{"type", "circle"}, {"radius", body.shape.circle.radius}};
    }

```

```

    return bodyJson;
}

```

```

bool saveWorldToFile(const Enjoyer2D::World &world, const std::string &path) {
    json sceneJson;

```

```

    const glm::vec2 gravity = world.getGravity();

```

```

    sceneJson["gravity"] = {gravity.x, gravity.y};
    sceneJson["velocityIterations"] = world.getVelocityIterations();
    sceneJson["positionIterations"] = world.getPositionIterations();
    sceneJson["bodies"] = json::array();

```

```

    for (const Enjoyer2D::Body &body : world.getBodies()) {
        sceneJson["bodies"].push_back(bodyToJson(body));
    }

```

```

    const std::filesystem::path outputPath(path);

```

```

    if (outputPath.has_parent_path()) {
        std::filesystem::create_directories(outputPath.parent_path());
    }

    std::ofstream file(outputPath);

    if (!file.is_open()) {
        return false;
    }

    file << sceneJson.dump(4);

    return true;
}

bool loadWorldFromFile(Enjoyer2D::World &world, const std::string &path) {
    std::ifstream file(path);

    if (!file.is_open()) {
        return false;
    }

    json sceneJson;

    file >> sceneJson;

    world.clear();

    world.setGravity(vec2FromJson(sceneJson["gravity"]));
    world.setVelocityIterations(sceneJson["velocityIterations"].get<int>());
    world.setPositionIterations(sceneJson["positionIterations"].get<int>());

    for (const json& bodyJson : sceneJson["bodies"]) {
        world.createBody(bodyDefFromJson(bodyJson));
    }

    return true;
}

```