КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
БУДІВНИЦТВА І АРХІТЕКТУРИ

Факультет автоматизації інформаційних технологій

Кафедра інформаційних технологій

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО КВАЛІФІКАЦІЙНОЇ РОБОТИ
НА ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ БАКАЛАВР

на тему:

Створення прототипу 3D ігрового рушія на основі імітаційних моделей твердих об'єктів з підтримкою фізичних властивостей

Івахненко Павло Володимирович

Київ 2025 р.

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
БУДІВНИЦТВА І АРХІТЕКТУРИ

автоматизації і інформаційних технологій

(факультет)

інформаційних технологій

(кафедра)

ЗАТВЕРДЖУЮ
Завідувачка кафедри ІТ
д.т.н., доцент Гончаренко Т.А.

„___” _____2025 року

КВАЛІФІКАЦІЙНА РОБОТА ЗДОБУВАЧА СТУПЕНЯ ВИЩОЇ ОСВІТИ БАКАЛАВР

на тему: «**Створення прототипу 3D ігрового рушія на основі імітаційних моделей твердих об'єктів з підтримкою фізичних властивостей**»

Я як здобувач вищої освіти КНУБА розумію і підтримую політику закладу з академічної доброчесності. Я не надавав(-ла) і не одержував(-ла) недозволену допомогу під час підготовки цієї роботи. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

Здобувач
Івахненко Павло Володимирович
122 «Комп’ютерні науки»
(спеціальність)
Інформаційні управляючі системи і технології
(освітня програма)
Групи КН-21
Керівник Рябчун Ю.В.
(прізвище та ініціали)
Доктор філософії
(вчене звання, науковий ступінь)
Рецензент к.т.н., доц. Доля О.В.
(Прізвище та ініціали)

Ідентичність підтверджую

Київ, 2025 р.

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БУДІВНИЦТВА І АРХІТЕКТУРИ

Факультет:	Автоматизації інформаційних технологій
Випускова кафедра:	Інформаційних технологій
Освітній ступінь:	Бакалавр
Спеціальність:	Комп’ютерні науки
Освітня програма:	Інформаційні управляючі системи та технології

ЗАТВЕРДЖУЮ

Завідувач кафедри ІТ

Тетяна ГОНЧАРЕНКО

_____

„___”_____2025 року

З А В Д А Н Н Я

ДО ВИКОНАННЯ КВАЛІФІКАЦІЙНОЇ ВИПУСКНОЇ РОБОТИ НА

ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ БАКАЛАВР

	Івахненко Павлу Володимировичу

1. Тема роботи - **Створення прототипу 3D ігрового рушія на основі імітаційних моделей твердих об'єктів з підтримкою фізичних властивостей**

затверджена наказом ректора КНУБА № 235/23/25 від «14» лютого 2025 року

2. Керівник роботи	Рябчун Юлія Володимирівна, PhD

3. Строк подання Здобувачем роботи до захисту травень 2025 р.

4. Зміст пояснювальної записки за розділами:

Р.1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

Р.2 ПРОЕКТ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Р.3 РЕЗУЛЬТАТИ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Р.4 АНАЛІЗ ТА ПЕРСПЕКТИВИ РОЗВИТКУ

5. Графічний матеріал за розділами:

Робота викладена на 95 аркушах, містить 3 додатки, 4 таблиці, 21 рисунок, список використаної літератури із 29 найменувань.

6. Календарний план виконання роботи:

Види робіт та їх зміст	Дата виконання
Розділ 1	01.02.2025
Розділ 2	06.04.2025
Розділ 3	16.05.2025
Розділ 4	16.05.2025
Остаточне оформлення роботи	25.05.2025
Направлення роботи для перевірки на плагіат	25.05.2025
Попередній захист роботи на випусковій кафедрі	26.05.2025
Направлення роботи на рецензування	26.05.2025

7. Консультанти розділів атестаційної випускної роботи

Розділ	Прізвище, ініціали та посада консультанта	Перевірив	
		дата	підпис
Розділ 1	Рябчун Ю.В., доц.каф.ІТ	01.02.2025	
Розділ 2	Рябчун Ю.В., доц.каф.ІТ	06.04.2025	
Розділ 3	Рябчун Ю.В., доц.каф.ІТ	16.05.2025	
Розділ 4	Рябчун Ю.В., доц.каф.ІТ	16.05.205	

8. Дата видачі завдання листопад 2025 р.

Зав. кафедри			Гончаренко Т.А.
	(підпис)		(прізвище та ініціали)
Керівники			Рябчун Ю.В.
	(підпис)		(прізвище та ініціали)
Здобувач			Івахненко П.В.
	(підпис)		(прізвище та ініціали)

АНОТАЦІЯ

Івахненко П.В. Створення прототипу 3D ігрового рушія на основі імітаційних моделей твердих об'єктів з підтримкою фізичних властивостей.

Атестаційна випускна робота бакалавра за спеціальністю 122 «Комп’ютерні науки», освітня програма «Інформаційні управляючі системи та технології». – Київський національний університет будівництва та архітектури. – Київ, 2025.

Робота присвячена розробці прототипу 3D-ігрового рушія, що дозволяє виконувати моделювання твердих об'єктів у реальному часі з урахуванням базових фізичних властивостей — гравітації, інерції, тертя, зіткнень та пружності. Рушій реалізовано з використанням мови програмування C++ та графічного API OpenGL, що забезпечує ефективну візуалізацію сцени з одночасною фізичною симуляцією. У процесі розробки спроектовано архітектуру рушія, реалізовано модулі обробки подій, рендерингу, симуляції фізичних взаємодій, а також базовий графічний інтерфейс користувача.

Робота викладена на 95 аркушах, містить 3 додатки, 4 таблиці, 21 рисунок, список використаної літератури із 29 найменувань.

Ключові слова: 3D-рушій, OpenGL, C++, фізична симуляція, тверді тіла, колізії, рендеринг, чисельні методи

SUMMARY

Ivakhnenko P.V. Development of a prototype 3D game engine based on rigid body simulation with physical property support.

Bachelor's thesis in the specialty: 122 "Computer Science", educational program "Information Management Systems and Technologies". – Kyiv National University of Construction and Architecture. – Kyiv, 2025.

This work is devoted to the development of a prototype 3D game engine that allows real-time modeling of solid objects with consideration of basic physical properties such as gravity, inertia, friction, collisions, and elasticity. The engine is implemented using the C++ programming language and the OpenGL graphics API, which provides efficient scene visualization with simultaneous physical simulation. In the course of development, the engine architecture was designed, event processing, rendering, physical interaction simulation modules were implemented, as well as the basic graphical user interface.

The paper is presented on 95 pages, contains 3 appendices, 4 tables, 21 figures, and a list of 29 references.

Keywords: 3D engine, OpenGL, C++, physical simulation, solids, collisions, rendering, numerical methods

ЗМІСТ

ВСТУП

Удосконалення 3D-технологій сьогодні виходить за межі чисто розважальної сфери та знаходить широке застосування в інженерних симуляціях, архітектурній візуалізації, віртуальному тестуванні й наукових дослідженнях. 3D-рушії забезпечують єдине середовище для обробки геометрії, рендерингу, симуляції фізичних процесів та управління анімацією, що значно пришвидшує розробку і водночас підвищує точність результатів. Завдяки цьому вони дедалі більше знаходять застосування як для розробки комп’ютерних ігор, так і для створення прототипів у промисловості та науково-дослідних проектах..

Актуальність теми. У сучасному науково-технічному середовищі 3D-рушії виступають універсальним середовищем для обробки геометрії, рендерингу, моделювання фізичних процесів та управління анімацією, що дозволяє вирішувати завдання не лише індустрії розважальних ігор, але й численних професійних галузей. Завдяки розвитку високопродуктивних графічних API, таких як Vulkan і DirectX 12, а також впровадженню хмарних обчислень, значно зросла роль цих платформ у точному моделюванні інженерних систем, архітектурній візуалізації, медичних реконструкціях та кіновиробництві. Зокрема, інженерні симуляції в програмах на кшталт Siemens Simcenter 3D та ANSYS Discovery Live застосовують рушійні компоненти для аналізу теплового обміну, напружень і течій, тоді як CAD-моделі, імпортовані в середовища на базі Unreal Engine із використанням Datasmith, дозволяють створювати високодеталізовані віртуальні тури й презентації архітектурних проектів. У сфері автомобільної індустрії NVIDIA Drive Sim та Unity Simulation забезпечують тестування алгоритмів автономного водіння в умовах, максимально наближених до реальних, тоді як у медицині платформи типу OsiriX і 3D Slicer інтегрують 3D-рушії для реконструкції діагностичних зображень та симуляції хірургічних процедур. Одночасно в кіновиробництві й рекламних проектах Houdini та Cinema 4D, а також секвенсер Unreal Engine, широко використовуються для створення візуальних ефектів і інтерактивних анімаційних сцен із високим ступенем фотореалізму.

Підґрунтям для цих застосувань є провідні комерційні рушії, серед яких Unity вирізняється модульною архітектурою та кросплатформеністю, Unreal Engine забезпечує фотореалістичний рендеринг із апаратною підтримкою рейтрейсингу, а CryEngine славиться можливостями побудови широченних відкритих світів із реалістичною фізикою й освітленням.[1] Поряд із цим NVIDIA Omniverse представляє середовище спільної роботи з інтеграцією CAD-даних і фізичного рушія PhysX, що відкриває нові горизонти для мультидисциплінарних проектів. Хоча деякі продукти, такі як Autodesk Stingray, втратили підтримку, їхні концептуальні рішення стали каталізатором розвитку сучасних інструментів для інтеграції CAD-рушіїв у VR-сценарії.

Одним з ключових компонентів будь-якого 3D-рушія є модуль фізики, який реалізує закони механіки твердих тіл, рідин і газів у режимі реального часу. Серед найпоширеніших рішень для симуляції динаміки віртуального середовища варто згадати оптимізований під GPU NVIDIA PhysX, який широко застосовується як в іграх, так і в інженерних візуалізаціях, бібліотеку Havok, котра традиційно використовується в AAA-проектах, та відкриту Bullet Physics, що легко інтегрується в наукові дослідження. Для спеціалізованих завдань, таких як моделювання динаміки багатокомпонентних систем або поведінки транспортних засобів, застосовують Chrono::Engine, здатну враховувати складні властивості матеріалів — від пружності й в’язкості до коефіцієнтів тертя та демпфування.

Метою даної роботи є створення прототипу 3D рушія, який дозволить симулювати поведінку твердих тіл у віртуальному середовищі з урахуванням фізичних законів. Такий рушій має забезпечувати коректну симуляцію динаміки твердих тіл із урахуванням законів Ньютона, гравітації, інерції та тертя. Відтак, реалізація описаних підходів дасть змогу поглибити теоретичні знання з реального часу та створити прототип універсальної платформи.

Для досягнення поставленої мети необхідно вирішити такі **завдання дослідження:**

1. Аналіз існуючих 3D-рушіїв та фізичних моделей.
2. Проектування архітектури рушія.

3. Розробка алгоритмів фізичної взаємодії.

4. Розробка архітектури візуалізації зображення.

5. Створення прототипу рушія.

Об’єкт дослідження – процес створення 3D-ігрового рушія з підтримкою фізичних симуляцій.

Предмет дослідження – алгоритми обробки фізичних взаємодій у реальному часі, методи виявлення та обробки зіткнень, підходи до моделювання твердих тіл у 3D-середовищі.

Для досягнення поставлених цілей у роботі застосовуються такі **методи**:

- методи комп’ютерного моделювання – для створення 3D-сцени та симуляції фізичних процесів.

- чисельні методи – для розрахунку руху твердих тіл (інтеграція рівнянь руху).

- алгоритми визначення колізій – AABB (Axis-Aligned Bounding Box), Sphere Collision.

- методи оптимізації продуктивності – спрощені моделі фізичних розрахунків.

Наукова новизна і практична значущість роботи. Дана робота допоможе покращити навички в створенні 3D симуляцій, а також в майбутньому може стати основою для більш нішевого продукту. Також, результати роботи можуть бути використані для створення lightweight-рушіїв для інді-ігор або симуляторів.

Отримані алгоритми можуть застосовуватися у навчальних та дослідницьких цілях. Розроблений прототип можна інтегрувати в існуючі проекти, де потрібно моделювати фізику з можливістю тонкого налаштування параметрів симуляції.

Розділ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

Розвиток комп’ютерної графіки та ігрових технологій з кожним роком набирає обертів, породжуючи все складніші візуальні та фізичні системи віртуальних середовищ. Сучасні гравці очікують не лише високоякісного візуального оформлення, але й реалістичної взаємодії об’єктів у просторі гри. Одним із ключових компонентів, що забезпечують таку взаємодію, є ігровий рушій — програмна система, яка об’єднує фізичні моделі, графіку, звук та інші елементи в єдину функціональну платформу.

Особливу роль у цьому контексті відіграє моделювання фізичних властивостей твердих тіл — таких як маса, інерція, тертя, гравітація та зіткнення. Врахування цих факторів у симуляції дозволяє створювати реалістичні та динамічні сцени, що підвищують рівень занурення користувача у віртуальний світ. Проте побудова ігрового рушія, який підтримує всі ці властивості, потребує комплексного підходу до проектування, розробки та тестування програмного забезпечення.

1.1. Основи 3D-графіки та фізичного моделювання

1.1.1. Огляд розвитку 3D графіки

Тривимірна графіка є однією з найважливіших складових сучасних комп'ютерних технологій, що безперервно розвивається. Від перших експериментальних моделей, які складалися з простих полігональних фігур, до сучасних фотореалістичних зображень, що практично не відрізняються від реальних об'єктів, ця галузь пройшла довгий шлях еволюції. Значний прорив у її розвитку став можливим завдяки збільшенню потужності комп’ютерів, появі передових алгоритмів рендерингу та використанню новітніх методів оптимізації зображення [4].

1.1.2. Основні напрямки розвитку та використання 3D графіки

Розвиток 3D графіки сьогодні охоплює кілька ключових напрямків. Одним із головних є підвищення рівня реалістичності візуалізації [4]. Використання

передових технологій трасування променів та нових алгоритмів освітлення дозволяють створювати зображення, що наближені до реальних фотографій. Не менш важливою є оптимізація обчислювальних процесів [4], що дає змогу розробникам ефективніше використовувати ресурси графічних процесорів для досягнення тієї ж якості та одночасного покращення продуктивності..

Також одним з аспектів розвитку графіки є візуалізація фізичних симуляцій: твердих тіл, м’яких тканин, рідин та газів [6]. Це відкриває багато можливостей, як в художньому плані, дозволяє створювати сцени з більшою достовірністю та реалістичною поведінкою об’єктів, так і в дослідницькому, наприклад, дозволяючи створювати детальні віртуальні моделі будівель та інтер'єрів ще до їх фізичного втілення, конструювання складних механізмів та перевірки технічних рішень без необхідності створення дорогих фізичних прототипів.

1.2. Постановка та аналіз проблеми

Завдання конкретно цієї роботи - дослідження принципів роботи 3D рушіїв, їх створення та симуляції фізики твердих об’єктів всередині них. Для цього необхідно розглянути особливості сучасних технологій, визначити ключові підходи до моделювання та симуляції фізичних процесів у віртуальному середовищі.

Аналіз предметної області є важливим етапом дослідження, оскільки він дозволяє визначити поточний стан технологій, виокремити найбільш перспективні методи та зрозуміти основні виклики, з якими стикаються розробники 3D-рушіїв. Це дослідження дає змогу сформувати вимоги до програмного забезпечення, що буде розроблено в рамках цієї роботи, та забезпечити його відповідність сучасним тенденціям і потребам користувачів.

Крім того, детальний аналіз предметної області дозволяє виявити потенційні проблеми, які можуть виникати під час розробки рушія, та розробити методи їх подолання. Оцінка вже існуючих рішень і їх порівняння дозволить виявити сильні

та слабкі сторони сучасних 3D-рушіїв, що, своєю чергою, сприятиме вибору оптимального підходу для реалізації власного рушія.

На основі проведеного аналізу буде сформульовано основні завдання для подальшої розробки, що включатимуть реалізацію базового рендерингу, інтеграцію фізичних симуляцій твердих тіл та оптимізацію продуктивності. Це дозволить досягти результатів швидше і ефективніше, створивши план з дослідження та розуміння принципів роботи 3D-графіки.

1.3 Огляд сучасних ігрових рушіїв

Ігровий рушій - це готова архітектура, яку розробники використовують для запуску своїх ігор [1]. Середньостатистичний ігровий рушій надає розробникам можливість додавати такі речі, як керування фізикою, рендеринг, необхідні скрипти виявлення зіткнень, штучний інтелект тощо, без необхідності їхнього програмування.

Ігровий рушій – це програмне забезпечення, яке забезпечує базову функціональність для створення комп'ютерних ігор. Він включає в себе модулі для:

- Відображення графіки (рендеринг 2D/3D сцен).
- Обробки фізичних взаємодій між об'єктами.
- Управління анімацією та поведінкою персонажів.
- Роботи зі звуком і введенням користувача.
- Оптимізації продуктивності для різних платформ.

Сьогодні існує значна кількість різноманітних ігрових рушіїв для розробки комп'ютерних ігор та систем моделювання на різних платформах. Всі вони відрізняються за платформами, які підтримують, мовами програмування, графічними інтерфейсами, специфічними можливостями, функціональністю та ціною.

На сучасному ринку існує кілька популярних 3D рушіїв, кожен з яких має свої особливості, переваги та недоліки. В цьому розділі розглянемо їх та опишемо

головні особливості, щоб потім сформувати на основі цього вимоги до нашої роботи.

Unity [14] є одним із найпопулярніших рушіїв, який широко використовується, в першу чергу, для створення відеоігор, VR/AR-додатків та інтерактивних симуляцій (рис.1.1, 1.2).[1] Його головна перевага — відносна простота освоєння та велика спільнота користувачів, що дозволяє легко знаходити підтримку та навчальні матеріали. Однак, він має обмеження у продуктивності при роботі з великими сценами, що може стати проблемою для проектів із високими графічними вимогами. Unity має гнучку систему компонентного програмування та використовує C# як основну мову розробки. Вбудовані інструменти, зокрема Unity Asset Store, дозволяють швидко інтегрувати ресурси в проекти. Звісно, так як це в першу чергу ігровий рушій, він дозволяє відмальовувати графіку (не лише три-, а я двовимірну) та має доволі широкий набір інструментів для фізичних симуляцій.

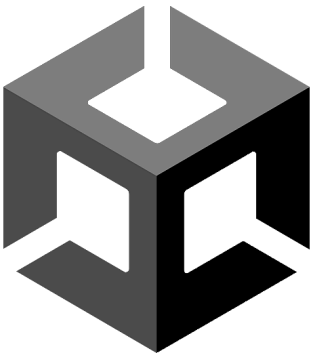

Рисунок 1.1. Логотип Unity

Рисунок 1.2. Редактор Unity

Переваги:

- Простота освоєння.
- Широка підтримка платформ та активна спільнота, тобто просте навчання.
- Вбудований магазин, що значно полегшує створення.[11]
- Наявність широкого інструментарію для створення будь-якого контенту.

Недоліки:

- Менш гнучкий у порівнянні з рушіями з відкритим кодом.
- Високі вимоги до продуктивності при використанні складних сцен.

Unreal Engine [15], як і Unity, є потужним інструментом для розробки відеоігор та візуалізацій (рис.1.3, 1.4), але з більшою орієнтованістю на фотореалістичну графіку. Він також дозволяє прораховувати зображення в реальному часі та має потужні можливості трасування променів. В Unreal Engine розробка ведеться з допомогою C++, хоча існує і візуальне програмування за допомогою Blueprints. Рушій використовується у великих комерційних проектах та іноді кіноіндустрії завдяки широким можливостям анімації. Так само як і Unity має підтримку фізичних симуляцій, та можливість завантажувати вже готові, створені користувачами, компоненти[1][4].

Рисунок 1.3. Логотип Unreal Engine

Рисунок 1.4. Редактор Unreal Engine

Переваги:

- Висока якість графіки та підтримка трасування променів.
- Обмежений доступ до вихідного коду, що дає змогу кастомізувати рушій.
- Потужна система анімації та фізичних симуляцій.

Недоліки:

- Високий поріг входу для початківців.
- Вимогливість до ресурсів, що ще більше ускладнює використання.

Blender [16] вже не можна назвати ігровим рушієм, це 3D пакет для моделювання, анімації, рендерингу та редагування відео (рис. 1.5, 1.6). Ця програма так само дозволяє відмальовувати 3D графіку та має доволі просунутий інструментарій для фізичних симуляцій, за допомогою якого можна симулювати тверді та м’які об’єкти, тканини. Також має доволі широкий інструментарій для

симуляції частинок, що, в свою чергу, можна використовувати для симуляції рідин або газів. Коли мова заходить про саме візуалізацію, Blender може запропонувати цілих 2 рушія для візуалізації картинки (3, якщо враховувати Workbench, що використовується лише для попереднього перегляду): Eevee - схожий на ігрові рушії за принципом роботи та орієнтований на швидкодію та Cycles - орієнтований на реалістичність картинки, але прожерливий до ресурсів та повільний. Кожен з цих методів рендеру має свої переваги і недоліки, що розширює потенційні можливості програми.

Рисунок 1.5. Логотип Blender

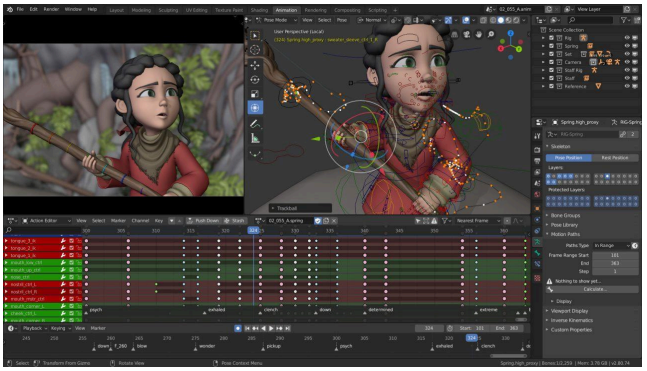

Рисунок 1.6. Редактор Blender

Переваги:

- Повністю безкоштовний і має відкритий вихідний код.
- Потужний набір інструментів для 3D-моделювання, анімації, рендеру та фізичних симуляцій.
- Має два рушія для рендеру, отже може закрити більше потреб.

Недоліки:

- Не розрахований на створення інтерактивних симуляцій, сконцентрований створенні заздалегідь визначених сцен.

Cinema 4D - один з конкурентів Blender, професійне програмне забезпечення для 3D-моделювання, анімації та рендерингу, створене компанією Maxon (рис. 1.7, 1.8). Його головна перевага — простий та інтуїтивний інтерфейс, що дозволяє швидко освоїти програму навіть початківцям. Cinema 4D часто

використовується в анімації, рекламній індустрії та архітектурній візуалізації. Вбудовані модулі, такі як MoGraph, дозволяють створювати складні анімаційні ефекти з мінімальними зусиллями.

Рисунок 1.7. Логотип Cinema4D

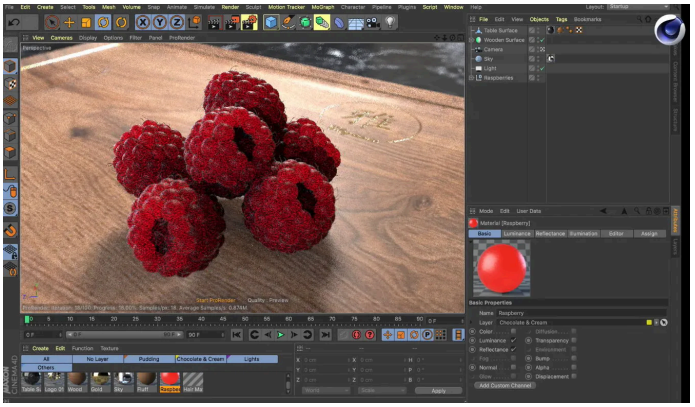

Рисунок 1.8. Редактор Cinema4D

Переваги:

- Простий у використанні та має інтуїтивний інтерфейс.
- Потужні інструменти для анімації та рендерингу.
- Велика кількість вбудованих модулів для створення анімаційних ефектів.

Недоліки:

- Висока вартість ліцензії.
- Менша підтримка реального часу у порівнянні з ігровими рушіями.

Autodesk CFD - це програмне забезпечення для обчислювальної гідродинаміки, яке використовується для моделювання потоків рідин і газів, а також аналізу теплопередачі (рис. 1.9, 1.10). Це потужний інструмент для інженерів та дослідників, які працюють із симуляціями складних фізичних процесів. Autodesk CFD дозволяє проводити числовий аналіз та оптимізацію конструкцій перед їх фізичним втіленням, що значно скорочує витрати на розробку та тестування.

Переваги:

- Потужний інструмент для аналізу потоків та теплопередачі.

- Можливість інтеграції з іншими продуктами Autodesk, такими як Inventor і Fusion 360, що допомагає в інженерній справі.
- Підтримка паралельних обчислень для прискорення симуляцій.

ним втіленням, що значно скорочує витрати на розробку та тестування.

Рисунок 1.9. Логотип CFD

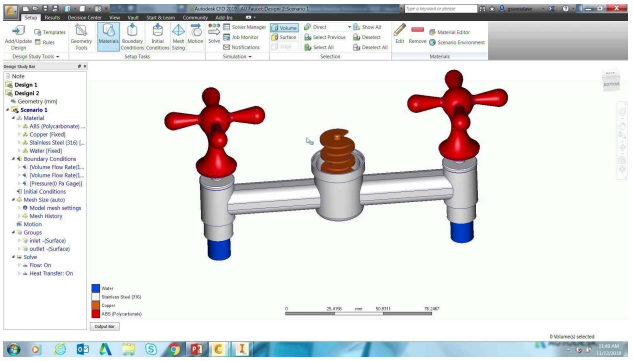

Рисунок 1.10. Редактор CFD

Недоліки:

- Необхідність потужного обладнання для більш точних симуляцій.
- Висока вартість ліцензії та короткий пробний період.

1.4. Проблематика існуючих комерційних 3D-рушіїв

Попри широкий вибір готових 3D-рушіїв, їх використання пов’язане з певними обмеженнями. Одним із головних недоліків є закритість вихідного коду, що унеможливлює глибоку модифікацію рушія та його адаптацію під специфічні потреби користувача. Крім того, комерційні рушії часто супроводжуються жорсткими ліцензійними умовами, які обмежують їх використання в комерційних проектах без відповідних фінансових зобов’язань.

Іншою проблемою є складність освоєння. Попри зручні інтерфейси та потужні можливості, новачкам може бути важко розібратися з усіма аспектами роботи цих рушіїв. Вони містять велику кількість функцій, що, з одного боку, відкриває широкі можливості, але, з іншого, вимагає значних витрат часу на навчання.

Хоч ця робота і не націлена на вирішення описаних вище проблем, варто залишити згадку про них, залишивши можливість розвитку початкової ідеї до чогось більшого.

Створення власного 3D рушія є надзвичайно корисним досвідом, що дозволяє глибше зрозуміти принципи роботи графічних систем. Це сприяє розвитку навичок оптимізації алгоритмів рендерингу, моделювання фізичних процесів та використання графічних API, таких як OpenGL. Це дає змогу отримати цінні знання та навички, які можуть бути використані в будь-якій галузі суміжній з 3D графікою.

1.5. Фізичні моделі у 3D-іграх: види, підходи, алгоритми

Фізичне моделювання у 3D-іграх є критичним елементом, що забезпечує правдоподібну симуляцію поведінки об'єктів, їхню взаємодію та відповідність базовим законам фізики у віртуальному середовищі. Ці моделі дають змогу підвищити ступінь занурення користувача, а також розширити функціональність систем штучного інтелекту, геймплейних механік, візуальної та наукової достовірності. Основні типи фізичних симуляцій у 3D-рушіях включають:

- *Обробку зіткнень* [2] – визначення фактів та місць контакту між тілами, із подальшим розрахунком сили реакції. В просунутих системах для цього використовують алгоритми широкофазного та вузькофазного тестування, зокрема Bounding Volume Hierarchies (BVH), Sweep and Prune, та алгоритм Гілберта – Джонсона – Керті (GJK). З простіших же алгоритмів застосовуються Axis-Aligned Bounding Box (AABB) або ще простіший Sphere Collision.[6][7]
- *Динаміку твердих тіл* [3][6] – чисельне інтегрування рівнянь руху згідно з другим законом Ньютона. Найпоширеніші інтегратори – метод Ейлера, симплектичний метод Ейлера, та інтегратор Рунге – Кутти другого або четвертого порядку.

- *Гравітацію, інерцію та тертя*[3][9] – реалізація сталих сил і моментів, що діють на об’єкти. У симуляції часто застосовуються моделі кулонівського сухого тертя, а також турбулентне або ламінарне повітряне гальмування.
- *Еластичні деформації* – моделювання деформаційних тіл, таких як тканини, гумові структури або біологічні органи. Основними підходами є масово-пружинні системи, метод кінцевих елементів (FEM), позиційно-орієнтована динаміка (PBD) та метод матеріальних точок (MPM).
- *Симуляцію рідин і газів* – обрахунок руху флюїдів за допомогою рівняння Нав'є – Стокса у спрощеному вигляді. Застосовуються методи частинок, зокрема Smoothed Particle Hydrodynamics (SPH), та решітчасті методи на основі об'ємних сіток, як-от метод маркерів та комірок (MAC) або Lattice Boltzmann Method (LBM).

При створенні ігрових рушіїв можуть виникати такі труднощі:

1. *Продуктивність* – складні фізичні розрахунки можуть уповільнювати роботу гри.
2. *Точність симуляції* – спрощення моделей інколи призводить до некоректної поведінки об'єктів (проходження через стіни, некоректний імпульс).
3. *Колізійна обробка* – виявлення зіткнень між об'єктами різної складності потребує балансування між точністю та швидкістю.
4. *Стабільність* – розрахунок фізики в реальному часі може спричиняти "вибухи" фізичних моделей через помилки у числових методах.

Серед підходів до реалізації фізичних моделей у 3D-рушіях слід виокремити аналітичні, евристичні та чисельно-інтеграційні. Аналітичні моделі забезпечують високу точність, але рідко застосовуються в реальному часі через обмеження продуктивності. Евристичні підходи, навпаки, знижують точність заради швидкодії, дозволяючи уникнути чисельних нестабільностей. Чисельні методи, включаючи інтеграцію за Ейлером чи Рунге–Куттою, створюють баланс між точністю й ефективністю, проте потребують ретельної стабілізації симуляції.

До основних труднощів при побудові фізичних систем у реальному часі належать обмеження продуктивності процесора або графічного процесора,

необхідність розпаралелювання обчислень, забезпечення числової стабільності та відсутність затримок між виявленням події і її графічною репрезентацією. Розробники змушені балансувати між візуальною достовірністю, обчислювальною складністю й потребами геймдизайну, що часто зумовлює використання гібридних алгоритмів або заміну фізично коректних симуляцій на емпіричні моделі.

1.6. Вимоги та особливості створення системи

Як ми можемо побачити з поданого вище переліку програмного забезпечення, яке вже існує на ринку, всі вони підтримують дві важливі в нашому контексті функції: візуалізація та рендер 3D сцени на двовимірному екрані та симуляцію фізичних властивостей різного ступеня складності. І хоча покликання цих програм може дуже відрізнятись - від ігрових рушіїв і універсальних 3D пакетів до професійних інженерних застосунків, що виконують конкретну профільну задачу, базово їх функціонал можна звести до двох систем - вище описаних фізичної симуляції та виводу зображення на екран.

На основі цих знань складемо технічні вимоги до проекту.

1.6.1. Фізична симуляція

Фізична симуляція є одним із ключових аспектів цієї роботи. Її основна мета – створення реалістичної взаємодії об'єктів у сцені відповідно до законів фізики. Для цього буде впроваджено алгоритми, які забезпечать точний розрахунок колізій, руху тіл у просторі, гравітаційних сил, сил тертя та пружності.

Ключові потреби при впровадженні фізичної симуляції:

- Достатня реалістичність взаємодії між об'єктами.
- Можливість роботи (переміщення/модифікація фізичних властивостей) з інтерактивними елементами сцени.
- Надати користувачам можливість інтерактивної взаємодії зі сценою.

1.6.2. Рендер зображення

Можливість візуалізації сцени у достатній якості для комфортного її сприйняття користувачем є фундаментальною вимогою для будь-якої 3D-системи [4]. Візуалізація має забезпечувати коректне відображення геометрії, текстур,

світла і тіней, що не будуть надто вимогливими до ресурсів комп’ютера і в той же час достатньо приємними оку.

Основні вимоги до рендерингу:

- Збалансоване використання ресурсів для забезпечення плавної роботи системи.
- Можливість створення наповнених сцен без значних втрат продуктивності.

1.6.3. Зрозумілий інтерфейс

Хоч в цьому проекті він і буде доволі мінімалістичним, користувацький інтерфейс має бути інтуїтивно зрозумілим та адаптивним до потреб розробників і користувачів. Основна мета полягає в наданні можливості швидко розібратися у функціоналі програми та отримати доступ до необхідних налаштувань без зайвих складнощів.

1.6.4. Зрозуміле та зручне керування

Керування у 3D-рушіїмає бути гнучким, зручним та відповідати стандартним практикам у галузі. Це включає систему гарячих клавіш, можливість змінювати параметри в реальному часі, а також використання звичних схем управління камерою та об'єктами у просторі.

Основні вимоги до керування в програмі:

- Використання усталених в індустрії гарячих клавіш для легкої адаптації.
- Можливість зміни деяких параметрів (як-от швидкість переміщення камери) користувачем.
- Мінімалістичність та простота в розумінні.

1.6.5. Достатня швидкодія

Швидкодія рушія є критично важливим фактором, що впливає на його придатність для практичного використання. Висока продуктивність дозволяє працювати з великими сценами, складними фізичними симуляціями та ефективно використовувати доступні обчислювальні ресурси.

Висока продуктивність системи дозволить:

- Працювати з великими сценами без значних втрат продуктивності.

- Мінімізувати затримки у симуляції фізичних процесів.
- Оптимізувати використання апаратних ресурсів для підвищення ефективності роботи.

1.7 Постановка задачі

Провевши деякий предметний аналіз теми, опишемо вхідні та вихідні дані а також функції які має виконувати система:

Вхідні дані

- Положення мишки в вікні програми.
- Натиснуті клавіші на клавіатурі.
- Натиснуті клавіші на мишці.
- Заздалегідь встановлені попередні налаштування.

Вихідні дані

- Зображення зі зрозумілим відображенням стану об’єктів на сцені.
- Статистика роботи програми (кадрів на секунду тощо).
- Необхідні дані відладки.

Функції системи

- Розрахунок точного положення об’єктів на сцені в реальному часі.
- Можливість зміни стану об’єктів під час роботи програми.
- Зміна фізичних параметрів об’єктів під час симуляції
- Можливість симуляції будь-якої кількості об’єктів
- Можливість зміни налаштувань програми.

Розділ 2. ПРОЕКТУВАННЯ ПРОТОТИПУ 3D ІГРОВОГО РУШІЯ

Проектування є ключовим етапом у розробці будь-якого програмного забезпечення, особливо коли йдеться про системи, що поєднують в собі графічну, фізичну та логічну складові, як у випадку з 3D ігровим рушієм. Саме на цьому етапі формується загальна архітектура системи, визначаються її модулі, способи взаємодії між ними, алгоритми обробки фізичних подій та механізми візуалізації.

У контексті створення прототипу 3D ігрового рушія надзвичайно важливим є точне формулювання вимог до системи, вибір оптимальних технологій та інструментів, а також побудова структурних і поведінкових моделей, які забезпечать гнучкість, масштабованість і продуктивність майбутнього продукту. Особливу увагу варто приділити моделюванню фізичних властивостей твердих тіл, які мають бути адекватно відображені у віртуальному середовищі.

Цей розділ присвячено опису процесу проектування програмної архітектури рушія, модулів обробки фізичних взаємодій, структури даних для представлення 3D-об'єктів, а також взаємозв’язків між графічним інтерфейсом та фізичним ядром системи. Результати проектування ляжуть в основу реалізації функціонального прототипу, що дозволить оцінити ефективність обраних підходів та рішень.

2.1 Аналіз вимог до рушія

2.1.1. Головні функціональні можливості

Основною метою створення рушія є забезпечення інтегрованої системи, що об'єднує базову візуалізацію 3D-сцени та симуляцію фізичних процесів. Це означає, що рушій має підтримувати два ключові напрямки роботи: рендеринг, який відповідає за відображення сцени на двовимірному екрані, та фізичну симуляцію, яка забезпечує реалістичну взаємодію об'єктів за законами фізики.

Візуалізація та рендеринг. Рендеринг – це не просто процес виводу зображення, а складний механізм, який має враховувати геометрію, текстури, освітлення, тіні та інші візуальні ефекти. Існує багато різних режимів рендерингу, серед них:

- ***Базова растеризація*** [17]. Це найпоширеніший метод рендерингу, який використовується у більшості ігрових рушіїв. При растеризації тривимірна сцена перетворюється у двовимірне зображення за допомогою проекції. Цей процес включає обчислення положення вершин, створення полігонів (трикутників) і перетворення їх на пікселі, які відображаються на екрані. Метод відомий своєю високою швидкістю та ефективністю, що робить його оптимальним для роботи в реальному часі та випадках коли пріоритетною є швидкодія.
- ***Трасування променів (Ray Tracing)*** [17]. При трасуванні променів симулюється поведінка світла. Кожен піксель зображення визначається шляхом відправлення променя з камери, який взаємодіє з об’єктами сцени. Це дозволяє точно відтворити ефекти відбиття, заломлення та тіні, забезпечуючи реалістичність зображення. Основним недоліком цього методу є висока обчислювальна складність, що може призводити до повільної роботи, особливо при симуляції в режимі реального часу.
- **Гібридні методи рендерингу** [17]. Гібридні методи поєднують переваги базової растеризації та трасування променів. Наприклад, сцена може бути попередньо відрендерена за допомогою растеризації, а ключові елементи, такі як тіні чи відбиття, доповнюються трасуванням променів. Такий підхід дозволяє досягти високої якості зображення з меншою обчислювальною вартістю, ніж повне трасування променів.

У даній роботі використовуватиметься базова растеризація, оскільки вона є найбільш доступною та простішою у реалізації, особливо з використанням OpenGL, графічним API, що буде використано в цій роботі. Растеризація добре підтримується цим графічним API і дозволяє швидко та ефективно відображати 3D-сцени у режимі реального часу, що є ключовим для розробки інтерактивного прототипу. Крім того, вона є достатньою для досягнення високої якості зображення у рамках навчального проекту, дозволяючи зосередитися на інших аспектах розробки рушія.

Фізична симуляція[6]. В комп’ютерних іграх реалістична симуляція фізики

твердих тіл є ключовим елементом, що визначає глибину занурення користувача у віртуальний світ. Завдяки фізичним симуляціям об'єкти в іграх поводяться природно: вони реагують на зіткнення, змінюють напрямок руху, опиняються під впливом гравітації, демонструють ефект тертя та пружності, що робить взаємодію максимально наближеною до реальності.

У професійних додатках, таких як CAD/CAM-системи або інженерні симулятори, детальна фізична симуляція дозволяє проводити численні аналізи до початку виробництва. Наприклад, розрахунок стійкості конструкції під впливом різних навантажень, тестування поведінки механізмів при зіткненнях або визначення ефективності систем охолодження. Це сприяє виявленню потенційних проблем на ранніх етапах розробки, що зменшує ризики та витрати, пов’язані з фізичними прототипами.

Розглянемо основні аспекти фізичної симуляції:

- **Колізії**[6]. Симуляція колізій дозволяє виявляти та розраховувати взаємодію між об'єктами, коли вони стикаються. У іграх це створює відчуття реалістичного контакту між об'єктами, а в професійних додатках – забезпечує точне моделювання зіткнень, що критично для перевірки безпеки конструкцій та механізмів.
- **Динаміка руху**[6]. Моделювання динаміки руху охоплює розрахунок швидкості, прискорення та траєкторій руху об'єктів. Цей аспект дозволяє створити реалістичну поведінку під час переміщення – наприклад, падіння, стрибки або відскоки. У професійних програмах такі симуляції використовуються для аналізу механічних систем та прогнозування поведінки конструкцій у різних умовах експлуатації.
- **Тертя**[6]. Ефект тертя моделює опір руху між поверхнями, що контактують. В іграх це може виявлятись як вплив на поведінку транспортних засобів або рух об'єктів, а в професійних симуляціях є критичним для аналізу ефективності систем охолодження, рухомості механізмів, а також для прогнозування зношення деталей.
- **Пружність**[6]. Пружність відповідає за здатність матеріалів повертатися до

первинної форми після деформації. У віртуальних середовищах це дозволяє моделювати, як об'єкти відбиваються або змінюють форму при зіткненні. В професійних додатках симуляція пружності використовується для аналізу поведінки матеріалів під навантаженням, що є одним з найважливіших параметрів при проектуванні будівель або машин.

- **Інші сили.** До інших сил, що можуть бути змодельовані, належать, наприклад, аеродинамічні ефекти або сили, що виникають при електромагнітних взаємодіях[6]. Хоча в іграх ці аспекти часто спрощуються, в професійних симуляціях вони можуть бути критично важливими для точності прогнозів і розрахунків.

Таким чином, фізичні симуляції у 3D-рушіях дозволяють не лише створювати захоплюючі та реалістичні ігрові світи, але й забезпечувати високоточну симуляцію для професійних застосувань, що вимагають точних розрахунків та аналізу фізичних процесів.

Інтерактивність та керування. Рушій повинен бути спроектований таким чином, щоб дозволити інтерактивну взаємодію з об'єктами у сцені. Це включає можливість зміни положення, фізичних властивостей, а також реагування на дії користувача (натискання клавіш, рух миші тощо). Важливо, щоб інтерфейс керування був зручним та інтуїтивно зрозумілим, що сприятиме швидкому опануванню системи кінцевими користувачами.

Щоб краще зрозуміти внутрішню структуру рушія та логіку його функціонування, доцільно звернутись до узагальненої блок-схеми функціональних модулів (рис.2.1). Вона ілюструє основні компоненти програмної системи та їхні взаємозв’язки. На схемі відображено, як між собою взаємодіють модулі візуалізації, фізичної симуляції, обробки введення, управління сценою та обробки логіки гри. Особлива увага приділена модульності архітектури, що забезпечує гнучкість при реалізації функціоналу та розширюваність системи в майбутньому. Таке розділення дозволяє чітко відокремити відповідальність кожного компонента та зменшити складність їхнього тестування, налагодження і підтримки.

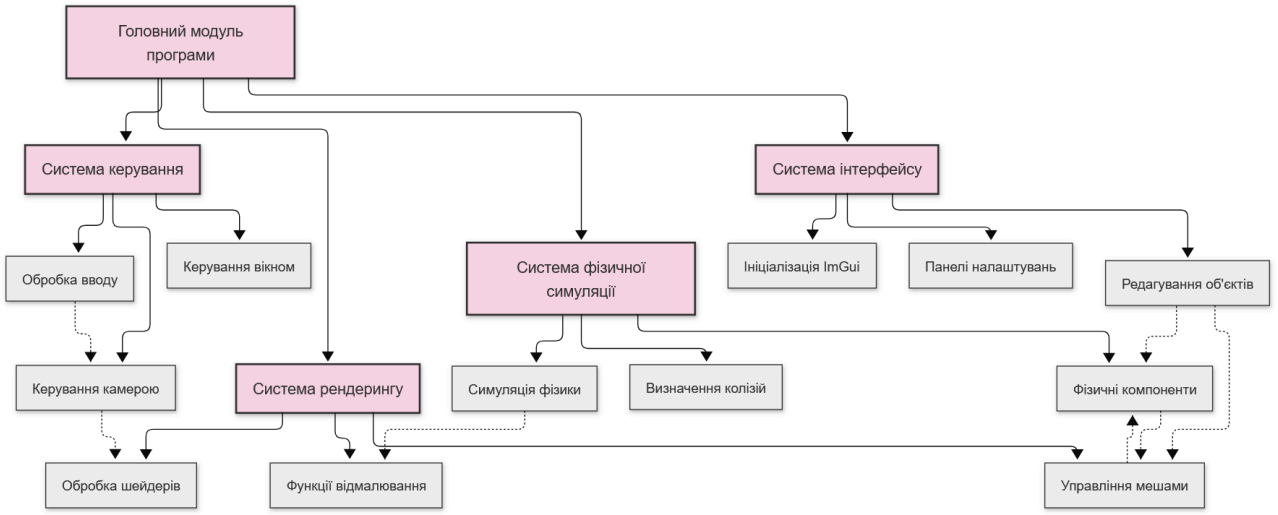

Рисунок 2.1. Блок-схема функціональних модулів програми

Проте взаємодія між модулями вимагає також чіткого визначення потоків даних, що використовуються системою під час роботи. Це особливо важливо для забезпечення коректного і синхронного оновлення стану об'єктів у сцені: фізичних змін, візуальних відображень і реакцій на вхідні дії користувача.

Діаграма потоків даних (рис.2.2) відображає основні етапи обробки інформації у програмі – від моменту надходження команди від користувача до фінального рендерингу сцени. Така структурна візуалізація дозволяє простежити, як відбувається оновлення стану сцени в реальному часі, уникаючи конфліктів між фізичною та графічною частинами рушія.

2.1.2. Мінімальні вимоги до продуктивності

Для того, щоб рушій був придатним для використання у реальному часі, він повинен задовольняти мінімальні вимоги до продуктивності. Продуктивність визначається здатністю системи обробляти велику кількість обчислень, що пов'язані з рендерингом складних сцен та симуляцією фізичних процесів без значних затримок.

Обчислювальна ефективність. Мінімальні вимоги до продуктивності включають можливість обробки достатньо великої кількості полігонів, текстур та світлових ефектів без різкого падіння частоти кадрів (FPS). Це означає, що рушій

повинен бути оптимізований як з точки зору використання пам'яті, так і з точки зору обчислювальної потужності. Для підвищення ефективності роботи можуть бути використані кілька методів оптимізації, таких як:

- **Рівні деталізації (LOD):** Автоматичне зниження кількості полігонів об'єкта залежно від відстані до камери, що дозволяє зменшити навантаження на графічний процесор без помітної втрати якості зображення.

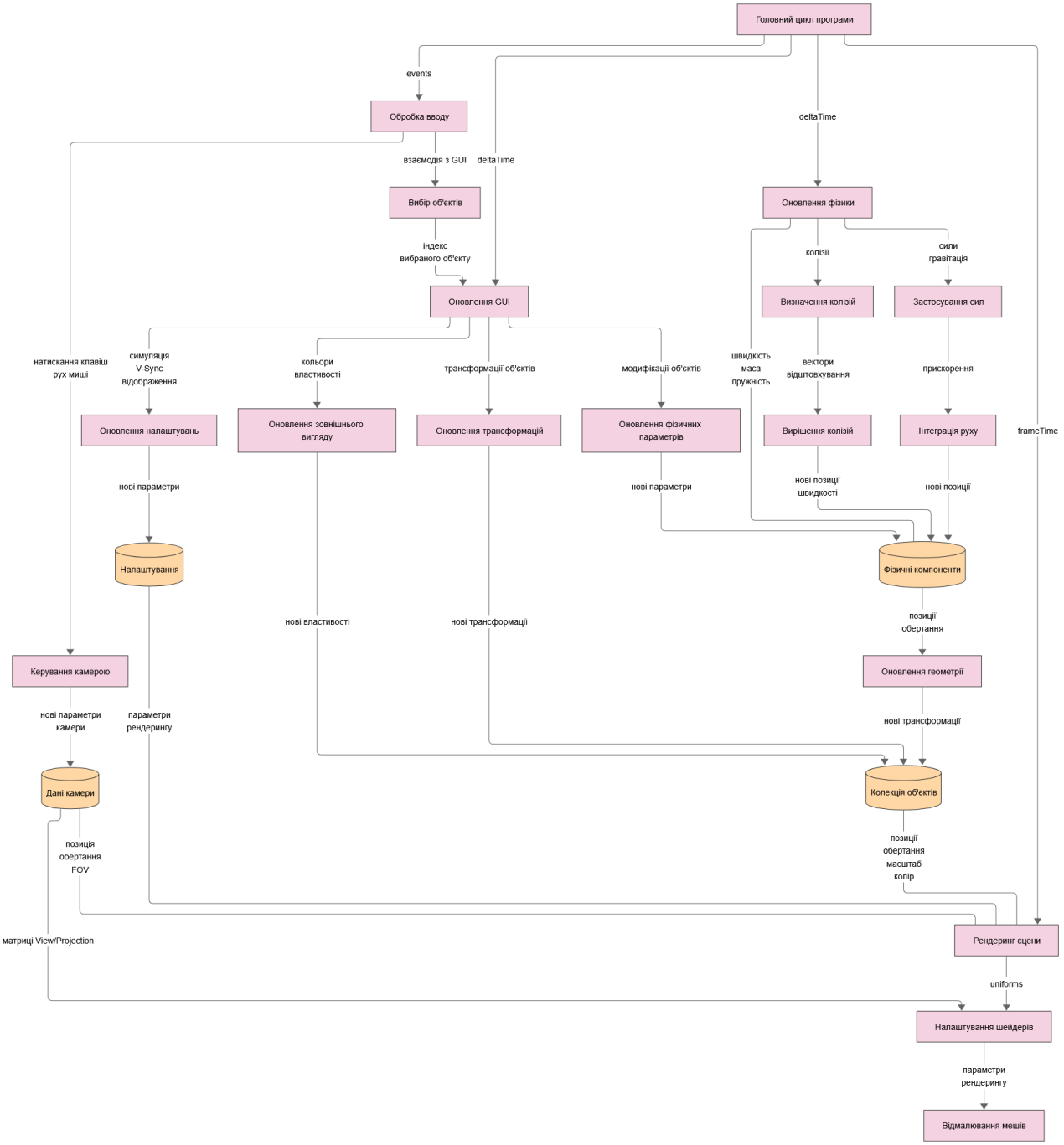

Рисунок 2.2. Діаграма потоку даних

- **GPU instancing:** Техніка, яка дозволяє одночасно рендерити велику кількість однакових об'єктів, використовуючи один і той самий набір геометричних даних, що значно знижує витрати ресурсів.
- **Зменшення кількості викликів рендерингу:** Оптимізація викликів рендерингу шляхом об'єднання об'єктів з подібними властивостями для зниження кількості окремих запитів до GPU, що допомагає покращити продуктивність системи.

Ці методи можна буде впровадити у разі виникнення проблем зі швидкодією системи, що дозволить адаптувати якість зображення до можливостей апаратного забезпечення.

Час фізичної симуляції. Для симуляції фізичних процесів важливо, щоб розрахунки виконувалися у режимі реального часу. Продуктивність повинна бути такою, щоб забезпечувати швидке обчислення руху тіл, колізій та інших фізичних взаємодій, що дозволить уникнути затримок та "розривів" симуляції, які можуть негативно вплинути на користувацький досвід.

Оптимізація ресурсів. Система повинна ефективно використовувати апаратні ресурси, зокрема багатоядерні процесори та сучасні графічні процесори. Це дозволить забезпечити стабільну роботу навіть при високих навантаженнях. Мінімальні вимоги можуть включати критерії, як-от мінімальну кількість оперативної пам’яті, потужність GPU, швидкість процесора та інші параметри, що визначають продуктивність рушія.

2.2 Вибір технологій для реалізації

2.2.1. Вибір мови програмування

Для розробки 3D-рушія можна використати безліч мов програмування, кожна з яких має свої переваги та недоліки. Серед найпопулярніших мов для створення графічних застосунків можна зазначити: **C++**, **Python**, **C#** та **Java**. Кожна з цих мов має певні характеристики, які впливають на продуктивність, гнучкість коду та складність розробки.

Python — одна з найпростіших у вивченні мов програмування, яка широко використовується в наукових розрахунках та машинному навчанні. Завдяки великій кількості бібліотек для роботи з графікою (наприклад, Pygame або Panda3D та навіть PyOpenGL), Python може бути використаний для створення 3D-додатків. Проте його продуктивність значно поступається C++ через інтерпретовану природу мови.

C# активно використовується у 3D-розробці, в першу чергу завдяки його інтеграції з рушієм Unity. Він має високий рівень абстракції, що робить процес розробки швидшим та зручнішим. Проте C# менш ефективний при роботі з низькорівневими графічними операціями у порівнянні з C++, тому не дуже підходить для роботи з OpenGL.

Java також використовується у графічних застосунках, зокрема для розробки ігор на базі бібліотек LWJGL. Однак, через використання віртуальної машини (JVM), має доволі значні обмеження у продуктивності.

Для реалізації 3D-рушія в даній роботі буде використано **C++**. Головна причина такого вибору — висока продуктивність та ефективне управління пам'яттю. На відміну від інтерпретованих мов, таких як, наприклад, Python, C++ компілюється безпосередньо у машинний код, що забезпечує високу швидкість виконання. Крім того, C++ дає змогу гнучко працювати з графічним API OpenGL, що робить його ідеальним вибором для розробки графічних застосунків.

Переваги використання C++:

- Висока швидкість виконання коду.
- Оптимальне управління пам’яттю.
- Велика кількість бібліотек та API для роботи з 3D-графікою.
- Широке застосування у розробці ігрових рушіїв, тобто простота у пошуку інформації.

Нижче наведено порівняльну таблицю 2.1 мов програмування, що показує переваги та недоліки цих мов:

Таблиця 2.1. Порівняння мов програмування

Критерій	**C++**	**Python**	**C#**	**Java**
Швидкодія	Висока. Компілюється в машинний код	Низька через інтерпретовану природу	Середня, частково через використання CLI	Середня через через використання JVM
Складність синтаксису	Вище середнього. Може бути громіздкою.	Низька. Спрощений синтаксис для швидкого написання коду.	Середня. Лаконічний, проста робота з ООП, але має деякі нюанси.	Середня. Дуже схожий на C#, але громіздкіший при роботі з графікою.
Підтримувані графічні API	Будь-які. OpenGL, Vulkan, DirectX.	Обмежена кількість. PyOpenGL, Panda3D, Pygame.	Переважно лише з OpenGL, але переважно через Unity.	Обмежена кількість, лише через обгортки OpenGL.
Інтеграція з OpenGL	Пряма і повноцінна.	Через обгортки, повільніша та з витратами ресурсів.	Обмежена, потрібні сторонні бібліотеки та обгортки.	Обмежена, лише через LWJGL або JOGL.
Швидкість розробки	Низька. Багато часу йде на реалізацію та налогодження.	Висока. Швидке прототипування та тестування.	Висока, але з використанням Unity.	Середня, але необхідна додаткова конфігурація.

2.2.2. Вибір графічного API

Графічний API є основним інструментом для взаємодії програмного забезпечення з графічним процесором (GPU). Найпопулярніші API для рендерингу 3D-графіки – **OpenGL**, **Vulkan** та **DirectX**.

DirectX [18] — це пропрієтарний API від Microsoft, який використовується в операційній системі Windows. Він забезпечує високу продуктивність та глибоку інтеграцію з Windows, проте не підтримується на інших платформах та є доволі складним в розумінні для початківців.

Vulkan [19] — сучасний низькорівневий API, що забезпечує високий рівень продуктивності та контроль над апаратним забезпеченням. Він є гнучкішим та швидшим за OpenGL, проте його освоєння потребує значно більше зусиль.

Через вищеописані недоліки альтернативних варіантів, в цьому проекті буде використано **OpenGL** (рис.2.3). Окрім його простоти до його переваг можна віднести кросплатформеність — OpenGL підтримується на Windows, macOS та Linux (а отже і більшості мобільних пристроїв). Також через свій вік, простоту та розповсюдженість в вільному доступі накопичилась велика кількість навчальних матеріалів та є потужна підтримка спільноти, що дуже допоможе в розробці.

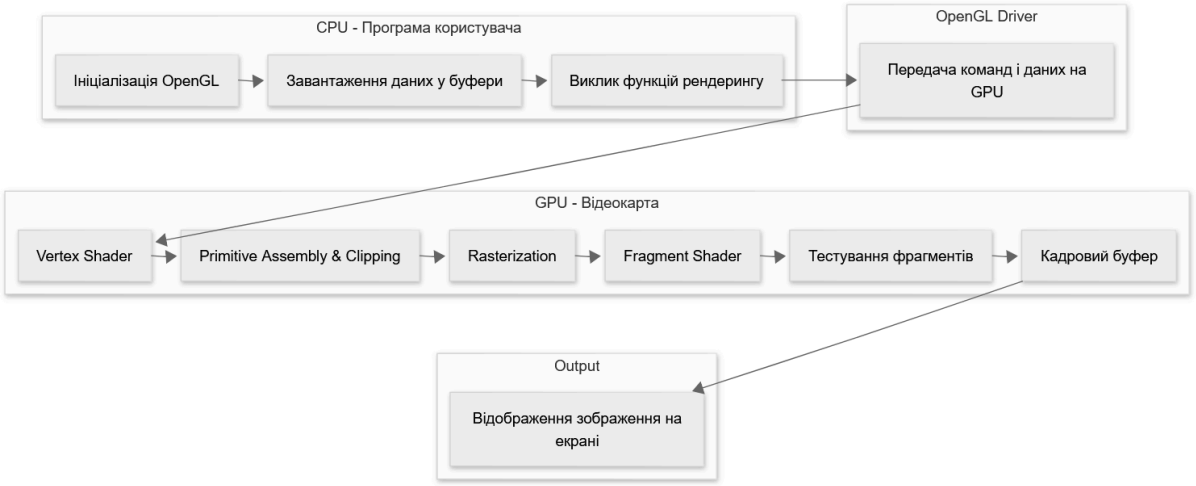

Рисунок 2.3. Схема роботи OpenGL у взаємодії з GPU

2.2.3. Фізичні рушії

Фізичний рушій відповідає за симуляцію фізичних процесів у 3D-сцені, дозволяючи моделювати взаємодію об’єктів, гравітацію, тертя, колізії та інші сили. Від вибору рушія залежить точність розрахунків, швидкодія системи та можливість використання апаратного прискорення. Найвідоміші фізичні рушії, які широко застосовуються в розробці ігор та професійних симуляціях, включають Havok, PhysX та Volt.

Havok [20] — один із найпопулярніших фізичних рушіїв, що використовується в індустрії відеоігор. Його головними особливостями є висока продуктивність та можливість точного моделювання складних фізичних взаємодій, включаючи твердотілі динамічні об’єкти, м’які тканини, рідини та системи уламків. Havok добре оптимізований для роботи у реальному часі та активно використовується в AAA-іграх.

PhysX [21] — фізичний рушій, розроблений компанією NVIDIA. Його основна перевага — підтримка апаратного прискорення на відеокартах NVIDIA, що дозволяє значно покращити продуктивність при обробці складних фізичних взаємодій. PhysX використовується у відеоіграх для симуляції реалістичних ефектів, таких як розліт уламків, моделювання рідин, тканин і частинок.

Volt [22] — сучасний фізичний рушій, який використовується для роботи з масштабними симуляціями. Основна перевага Volt — можливість ефективно обробляти величезну кількість об'єктів у сцені та підтримка сучасних багатопотокових процесорів для розподілених обчислень. Це робить його придатним для наукових симуляцій, архітектурного моделювання та великомасштабних ігрових світів. Цей рушій зараз використовується в інженерних симуляціях – для розрахунку навантажень і руйнування конструкцій та починає інтегруватись в **і**грові рушії нового покоління – задля симуляції великих руйнувань та реалістичної фізики.

Перераховані фізичні рушії мають різні переваги та застосування, але у цьому проекті буде реалізовано власний фізичний рушій (рис.2.4). Це дозволить глибше зрозуміти принципи розрахунку фізичних взаємодій та оптимізації обчислень, а також надати можливість експериментувати з власними алгоритмами для моделювання фізики твердих тіл.

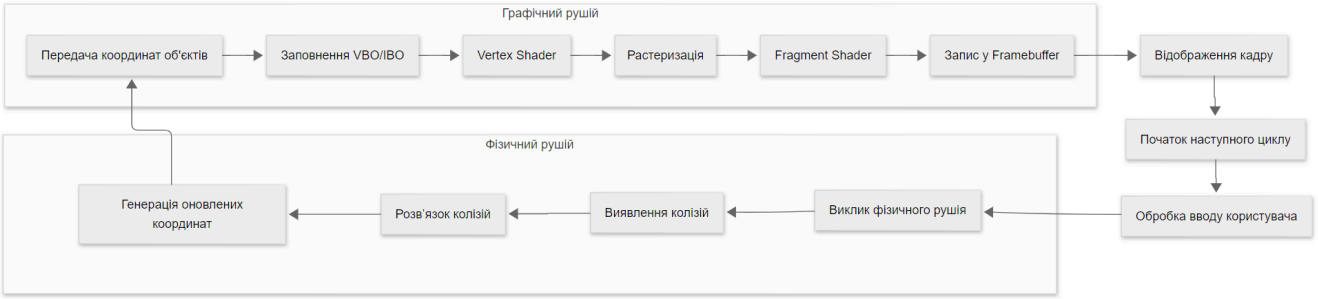

Рисунок 2.4. Схема роботи фізичного рушія та його взаємодії з графічним рушієм

2.2.4. Інші бібліотеки

Крім графічного API, для створення проекту будуть використані додаткові бібліотеки, які значно спрощують роботу з OpenGL та дозволяють розробникам

зосередитися на основних аспектах створення 3D-рушія.

GLFW (Graphics Library Framework) [23] — це одна з найпопулярніших бібліотек для управління вікнами, контекстом OpenGL та обробки подій введення (клавіатура, миша, джойстики). Вона забезпечує простий та ефективний спосіб створення кросплатформеного застосунку з підтримкою OpenGL без необхідності глибокого занурення у специфіку взаємодії з операційною системою.

Основні можливості GLFW:

- Управління вікнами та контекстом OpenGL.
- Обробка введення від клавіатури, миші та ігрових контролерів.
- Підтримка багатоплатформеності (Windows, Linux, macOS).

GLM (OpenGL Mathematics) [24] — це математична бібліотека, спеціально розроблена для роботи з OpenGL. Вона надає можливості для ефективних векторних та матричних операцій, що є критично важливими для реалізації трансформацій, роботи з камерами та інших обчислень у 3D-графіці.

Основні можливості GLM:

- Векторна та матрична алгебра (2D, 3D, 4D).
- Підтримка кватерніонів для роботи з обертанням.
- Спрощення роботи з трансформаціями (переміщення, масштабування, поворот).

ImGui (Immediate Mode GUI) [25] — це бібліотека для створення графічного інтерфейсу користувача, яка широко використовується у 3D-рушіях та інструментах розробників. Її основною перевагою є швидке та просте створення UI-елементів, які оновлюються без необхідності складного управління станами.

Основні можливості ImGui:

- Проста інтеграція з OpenGL.
- Можливість швидкого створення меню, панелей налаштувань та відладочних інструментів.
- Легкість у використанні без потреби створення складних систем UI.

GLEW (OpenGL Extension Wrangler Library) [26] — це бібліотека, яка спрощує доступ до методів OpenGL. Вона дозволяє програмам автоматично

отримувати підтримку нових функцій OpenGL та автоматично перетворює записані машинним кодом функції в відеокарті на зрозумілі користувачу та C++ методи без необхідності вручну перевіряти наявність розширень чи вказувати їх адреси у відеокарті користувача.

Основні можливості GLEW:

- Динамічне завантаження функцій OpenGL.
- Спрощення роботи з розширеними можливостями API.
- Забезпечення підтримки різних версій OpenGL без змін у коді.

Використання цих бібліотек є майже обов'язковим при роботі з сучасним OpenGL, адже вони значно спрощують розробку, зменшують складність реалізації рендерингу та дозволяють зосередитися на створенні функціоналу самого рушія. GLFW забезпечить керування вікном і введенням, GLM — необхідні математичні обчислення, GLEW — доступ до можливостей OpenGL, а ImGui допоможе створити зручний графічний інтерфейс для відладки та управління налаштуваннями рушія[13].

2.3 Архітектура рушія: основні модулі

Архітектура рушія формується як система взаємопов'язаних модулів, кожен з яких відповідає за окрему групу функцій. Правильне розподілення відповідальностей при створенні програми дозволить не лише спростити розробку, але й забезпечити масштабованість та гнучкість системи. Далі буде описано перелік основних модулів, які буде інтегровано в загальну архітектуру рушія.

2.3.1. Модуль ядра додатка

Модуль ядра відповідає за загальну ініціалізацію, основний цикл та фіналізацію роботи рушія. Він налаштовує вікно за допомогою GLFW, створює контекст OpenGL за допомогою GLEW, а також виконує конфігурацію модуля рендера та фізики. У цьому модулі відбувається опрацювання подій вводу та виклик оновлень всіх підсистем кожного кадру. Він також формує шейдери за

вказаними шляхами і встановлює початкові значення змінних середовища.

Основні завдання модуля:

- **Ініціалізація системи:** Налаштування GLFW (встановлення версії OpenGL, що використовується, параметрів вікна), ініціалізація GLEW та ImGui для подальшого відмалювання графічного інтерфейсу. Загалом забезпечує всі інші модулі та програму цілком коректним середовищем для роботи.
- **Цикл рендерингу та оновлення:** Саме в цьому модулі відбувається оновлення часу, обробка вводу, симуляція фізики, виклик рендера та інтерфейсу, а також перемальовування вікна. Він також відповідає за синхронізацію усіх підсистем та забезпечує плавний рендеринг кадрів.
- **Завершення роботи:** Також цей модуль відповідає за завершення роботи програми. Після виходу з основного циклу він викликає функції очистки і коректно звільняє ресурси рушія перед виходом.

2.3.2. Модуль рендерингу

Цей модуль відповідає за відображення тривимірних об'єктів на екрані через OpenGL. Складається із класів *Renderer, Shader, VertexArray, IndexBuffer* та *VertexBuffer*, які інкапсулюють роботу з буферами, шейдерами та викликом *glDrawElements,* функції, що відповідає за виклик відмалювання об’єкта на екрані. Також модуль рендерингу очищає екран (викликаючи *glClear*), задає режими роботи OpenGL (*glEnable()*, *glBlendFunc()*) і управляє станом шейдера. Завдяки йому рушій може малювати різноманітні примітиви з підтримкою кольору, освітлення та з правильною послідовністю завдяки тесту глибини.

Основні завдання модуля:

- **Управління шейдерами** Клас *Shader* завантажує GLSL-код шейдерів з файлів, компілює vertex- і fragment- шейдери, поєднує їх в одну програму та зберігає локації uniform-змінних для подальшого використання. Ці дії забезпечують гнучке налаштування освітлення, кольору та трансформацій матриць моделі і камери.
- **Рендеринг геометрії** Метод Renderer::Draw приймає Mesh або комбінацію

VertexArray/IndexBuffer, робить їх активними елементами і викликає *glDrawElements*. Основне завдання — цього методу передати на GPU вершини та індекси, щоб відобразити поверхні у відповідності до їх налаштувань, серед яких є режим відображення: повне замалювання полігонів або відмальовка лише ліній, що з'єднують точкиє.

- **Управління станом OpenGL** Налаштування тесту глибини, полігонального режиму та інших параметрів для коректного відображення об'єктів у просторі та прозорості.

2.3.3. Фізичний модуль

Забезпечує симуляцію руху тіл, гравітації та взаємодії через зіткнення. Клас *PhysicsSystem* містить список класів *PhysicsComponent*, основне його завдання – послідовно застосовувати сили, що діють на об'єкт. Серед цих сил: гравітація, інтеграція руху, виявлення та розв'язання колізій для всіх компонентів. Загалом цей модуль відповідає за те, щоб об'єкти наявні на сцені поводилися реалістично у рамках простих фізичних законів та надає можливість змінювати налаштування фізичної симуляції.

Основні завдання модуля:

- **Інтеграція руху (Update)** Для кожного об'єкта на сцені застосовується метод Ейлера: швидкість змінюється через прискорення, а позиція — через швидкість. Це дає змогу об'єктам падати, ковзати і прискорюватися під дією сил.

- **Виявлення та розв'язання колізій** За допомогою методів *Collision::TestSphereSphere*, *TestAABBAABB* та *TestSphereAABB* система перевіряє перетини колайдерів і розв'язує зіткнення імпульсним методом, після цього коригуючи позції. Це гарантує, що об'єкти відштовхуються один від одного з урахуванням коефіцієнта відскоку.

- **Керування кінематичними об'єктами** Для об'єктів, які зараз переміщаються за бажанням користувача рух обробляється особливим чином. Це необхідно для того, щоб вони могли правильно взаємодіяти з динамічними

об’єктами, впливаючи на них та коректно змінюючи їх положення.

2.3.4. Модуль GUI та вводу

Для створення інтерфейсу користувача в цьому проекті використовується бібліотека ImGUI. Інтеграція ImGui забезпечує користувацький інтерфейс для налаштування кольору фону, фізики, параметрів камери та об’єктів у сцені. Функції для роботи з цією бібліотекою винесені в окремі методи *initGUI* та *drawInterface.* які будують вікна налаштувань, слайдери й кнопки, а також підключають архітектуру зворотних викликів для обробки подій. Окремо обробляється введення клавіш в методі *processWindowInput* для показу/приховування меню й закриття вікна[12].

Основні завдання модуля:

- **Побудова інтерфейсних вікон** Виклик функцій ImGui, що створюють динамічні панелі управління, для зміни параметрів рушія в реальному часі.
- **Обробка подій вводу** За допомогою функцій зі зворотнім викликом GLFW (наприклад, *glfwSetFramebufferSizeCallback*) та опитуванням *glfwGetKey* модуль реагує на зміну розміру вікна, натискання клавіш Home/Escape для виходу з програми або тимчасове вимкнення головного меню та керування камерою.
- **Синхронізація зі сценою** Відстежуючи зміни UI, модуль транслює нові значення параметрів (гравітація, кут огляду, швидкість миші) до відповідних підсистем руху, рендерингу та фізики.

2.4 Алгоритми імітації фізичних властивостей

В цьому підрозділі будуть розглянуті математичні підходи та алгоритми, що використовуються для моделювання фізичних властивостей об’єктів у 3D-рушії. Також розглянемо основні аспекти симуляції фізики, які стосуються фізичної симуляції, зокрема гравітацію, тертя, виявлення та розв'язання колізій, розрахунок динамічних характеристик тіл та імпульсну взаємодію.

2.4.1. Гравітація, тертя, колізії

Реалістична імітація фізичних явищ у 3D-рушії вимагає математично точного моделювання фундаментальних фізичних сил та взаємодій. Розглянемо основні компоненти фізичної симуляції.

Гравітація. У розробленій системі гравітація реалізована як постійна сила, що діє на об'єкти з масою. Математично це можна описати наступною формулою:

$$F_g = m * g$$

де F_g – сила гравітації, m – маса об’єкта, g – вектор прискорення вільного падіння (в нашому випадку напрямлений вниз по осі Y).

У коді програми гравітаційна сила застосовується до всіх фізичних компонентів за допомогою метода ApplyGravity() системи фізики:

```
void ApplyGravity() {
  for (auto& component : m_PhysicsComponents) {
    if (!component->IsStatic()) {
            component->ApplyForce(m_Gravity * component->GetMass());
    }
  }
}
```

Тертя. Моделювання тертя є критично важливим для реалістичної симуляції ковзання об’єктів. У представленій в цій роботі системі тертя реалізовано через коефіцієнт тертя (змінна *m_Friction*), який визначає опір ковзанню об'єкта. Коефіцієнт тертя обмежений діапазоном [0, 1], де 0 відповідає абсолютно гладкій поверхні, а 1 — максимально шорсткій:

```
void SetFriction(float friction) {
  m_Friction = glm::clamp(friction, 0.0f, 1.0f);
}
```

Для застосування сили тертя необхідно обчислити тангенціальну складову відносної швидкості об'єктів у точці контакту. Сила тертя протидіє цій складовій, зменшуючи ковзання:

$$F_t = - \mu * |F_n| * \frac{v_t}{|v_t|}$$

де F_t – сила тертя, μ – коефіцієнт тертя, F_n – нормальна сила, v_t – тангенціальна складова швидкості.

Колізії. Виявлення зіткнень є одним із найскладніших аспектів фізичної симуляції.

Розроблена система підтримує кілька типів колізій:

- Сфера-сфера. Для колізій між сферами використовується простий алгоритм перевірки відстані між їх центрами.
- Сфера-AABB. Для AABB-колізій використовується метод мінімальних проекцій, який визначає вісь з найменшим перекриттям.
- AABB-AABB. Гібридні колізії сфери з AABB вимагають знаходження найближчої точки на AABB до центру сфери.

У разі виявлення зіткнення, система обчислює:

1. Нормаль колізії (*collisionNormal*) — напрям, у якому об'єкти мають розійтися
2. Глибину проникнення (*penetrationDepth*) — відстань, на яку об'єкти перетнулися
3. Точку зіткнення (*collisionPoint*) — координати, де відбулася колізія

2.4.2. Реалізація твердотільної фізики

Твердотільна фізика (rigid body dynamics) лежить в основі моделювання руху тіл у тривимірному просторі. Розглянемо основні принципи, реалізовані в системі.

Представлення тіла. Кожне тверде тіло в системі представлено компонентом *PhysicsComponent*, який інкапсулює фізичні властивості та стан об'єкта:

```
class PhysicsComponent {
// ...
private:
    glm::vec3 m_Position;
    glm::vec3 m_Velocity;
    glm::vec3 m_Acceleration;
```

```
    float m_Mass;
    float m_InverseMass;
    float m_Restitution;
    float m_Friction;
    ColliderType m_ColliderType;
    BoundingSphere m_SphereCollider;
    AABB m_AABBCollider;
};
```

Це представлення дозволяє ефективно моделювати поведінку об'єктів з різною масою, коефіцієнтами пружності та тертя.

Інтеграція рівнянь руху. Фізичні формули, особливо для розрахунку руху тіл, часто тісно пов’язані з інтегруванням, тому розглянемо декілька найпопулярніших способів чисельної інтеграції та оберемо один з них.

Методи чисельної інтеграції:

- **Метод Ейлера** [27]**:** Це найпростіший метод чисельної інтеграції, який оновлює положення і швидкість об'єктів лінійно, використовуючи поточні значення прискорення. Він простий у реалізації, але може бути неточним і нестабільним при великих часових кроках.
- **Симплектичний інтегратор** [27]**:** Цей метод розроблено для збереження енергетичних та фазових властивостей системи, що є критично важливим при симуляції руху у фізичних системах. Він забезпечує кращу стабільність та збереження енергії порівняно з методом Ейлера, особливо при довготривалих симуляціях.
- **Метод Verlet** [27]**:** Використовується переважно в молекулярній динаміці та інших симуляціях, де важлива стабільність інтеграції. Метод Verlet забезпечує хорошу консервацію енергії і є більш точним, ніж метод Ейлера, завдяки використанню як поточних, так і попередніх значень положення об'єкта для розрахунку нових координат.

Для апроксимації руху твердих тіл створена система використовує метод інтеграції Ейлера через свою простоту та швидкість імплементації., Він обчислює

зміни положення та швидкості на кожному кроці симуляції:

```
void Update(float deltaTime) {
  // Інтеграція Ейлера
     m_Velocity += m_Acceleration * deltaTime;
     m_Position += m_Velocity * deltaTime;
  // Скидання прискорення для наступного кадру
     m_Acceleration = glm::vec3(0.0f);
  // Оновлення колайдерів з новим положенням
     UpdateColliders();
}
```

Хоча метод Ейлера є найпростішим, він забезпечує достатню точність для багатьох застосувань при малих часових кроках. Також для підвищення точності та стабільності в деяких сценаріях, окрім вищеописаних методів, можна було б застосувати більш складні методи, такі як напівнеявний метод Ейлера або метод Рунге-Кутта.

Сили та імпульси. Система підтримує застосування сил до об'єктів, які змінюють їхнє прискорення:

```
void ApplyForce(const glm::vec3& force) {
     m_Acceleration += force * m_InverseMass;
}
```

Використання оберненої маси (*m_InverseMass*) замість прямого ділення на масу дозволяє обробляти статичні об'єкти не створюючи окремий випадок фізичного об’єкта, просто встановлюючи *m_InverseMass = 0*, що по суті надає йому “нескінченну масу”.

Кінематичні об'єкти. Система підтримує кінематичні тіла — об'єкти, положення яких контролюється безпосередньо (наприклад, користувачем), але які можуть впливати на інші динамічні об'єкти. Далі наведено частину коду, що відповідає за динамічну зміну між кінематичним та статичним режимами об’єкта в сцені:

```
void SetKinematic(bool isKinematic) {
     m_IsKinematic = isKinematic;
```

```
// Збереження оригінальної маси при переході в кінематичний режим
    if (isKinematic && !m_WasKinematic) {
        m_OriginalMass = m_Physics->GetMass();
        m_Physics->SetMass(0.0f);
        // Робимо об'єкт статичним (нескінченна маса)
    }
// Відновлення оригінальної маси при поверненні до динамічного режиму
    else if (!isKinematic && m_WasKinematic) {
        m_Physics->SetMass(m_OriginalMass);
    }
    m_WasKinematic = isKinematic;
}
```

2.4.3. Система обробки зіткнень

Після виявлення колізій необхідно відповідно змінити стан об'єктів. Система розв'язання зіткнень має два основні компоненти: обробку імпульсів та корекцію позицій.

Алгоритми виявлення колізій. Одні з найпростіших методів для первинного виявлення колізій включають bounding boxes (AABB, OBB) або bounding spheres. Більш точні алгоритми, такі як Separating Axis Theorem (SAT), дозволяють визначити, чи дійсно два об'єкта перетинаються. Далі наведено список з коротким описом найрозповсюдженіших методів виявлення колізій та їх розв’язання:

Виявлення зіткнень:

- **Bounding Boxes (AABB та OBB)** [28]**:** AABB (Axis-Aligned Bounding Box) можна уявити як прямокутну коробку, орієнтовану по осях координат, яка охоплює об'єкт. Перевірка перетину AABB є досить швидкою, але не завжди точною для об'єктів, що обертаються. OBB (Oriented Bounding Box) – це коробка, яка може бути обернена під будь-яким кутом, що дозволяє точніше охоплювати складні геометрії, проте обчислювально важча, ніж AABB.
- **Bounding Spheres** [28]**:** Bounding sphere схожа на сферу, що охоплює

об'єкт. Вона проста для розрахунків, оскільки перевірка перетину двох сфер зводиться до порівняння відстані між їх центрами із сумою їх радіусів, але може бути менш точною для об'єктів нерівномірної форми.

- **Separating Axis Theorem (SAT)** [28]**:** SAT є більш точним методом визначення колізій, який ґрунтується на пошуку осей, по яких проекції двох об'єктів не перетинаються. Якщо існує хоча б одна така вісь, то об'єкти не стикаються; якщо ні – відбувається колізія. Цей метод дозволяє отримати детальнішу інформацію про точку та напрямок зіткнення, що корисно для подальшого розрахунку фізичних реакцій.

Після успішного виявлення зіткнення його необхідно усунути для подальшої коректної роботи симуляції, для цього існує декілька підходів до розв’язання колізій.

Розв’язання зіткнень:

- **Імпульсний підхід** [29]. У цьому методі розв’язання колізій для кожного зіткнення розраховується імпульс, який змінює швидкість об'єктів. Розрахунок проводиться з урахуванням мас об'єктів, їхньої початкової швидкості та коефіцієнта відскоку, що визначає, скільки енергії зберігається після удару. Цей підхід дозволяє коригувати вектор швидкості об'єктів таким чином, щоб відновити фізичну коректність руху після зіткнення.

- **Пенальті-метод** [29]. Пенальті-метод базується на введенні віртуальної сили, яка «штовхає» об'єкти один від одного, коли вони проникають у межі своїх граничних об'ємів. Сила розраховується пропорційно ступеню проникнення і коригує положення об'єктів, щоб повернути їх у допустимий стан. При цьому також враховуються коефіцієнти, які визначають, як швидко відбувається відшкодування деформації.

Обидва методи використовуються для забезпечення реалістичної взаємодії між об'єктами після виявлення колізії, кожен з них має свої переваги: імпульсний підхід забезпечує точне коригування швидкості, а пенальті-метод дозволяє ефективно усунути проникнення об'єктів, відновлюючи цілісність сцени.

Імпульсна відповідь. Для моделювання пружних і непружних зіткнень наш

фізичний рушій використовує імпульсну відповідь, яка моментально змінює швидкості об'єктів:

```
void ResolveCollision(PhysicsComponent& componentA,
 PhysicsComponent& componentB, const CollisionInfo& collisionInfo) {
// Обчислення відносної швидкості
     glm::vec3 relVelocity = componentB.GetVelocity() -
                                componentA.GetVelocity();
     float velAlongNormal = glm::dot(relVelocity,
                                collisionInfo.collisionNormal);
// Не застосовуємо імпульс, якщо об'єкти віддаляються
     if (velAlongNormal > 0) return;
// Обчислення коефіцієнта реституції (пружності)
     float e = std::min(componentA.GetRestitution(),
                                componentB.GetRestitution());
// Обчислення імпульсного скаляра
     float j = -(1 + e) * velAlongNormal;
     j /= componentA.GetInverseMass() +
                                componentB.GetInverseMass();
// Застосування імпульсу
     glm::vec3 impulse = j * collisionInfo.collisionNormal;
     componentA.SetVelocity(componentA.GetVelocity() -
                          componentA.GetInverseMass() * impulse);
     componentB.SetVelocity(componentB.GetVelocity() +
                          componentB.GetInverseMass() * impulse);
// ...
}
```

Імпульс обчислюється відповідно до закону збереження імпульсу та коефіцієнта реституції, який визначає еластичність зіткнення. Коефіцієнт $e = 1$ відповідає абсолютно пружному зіткненню, а $e = 0$ — абсолютно непружному.

Корекція позицій. Для запобігання проникненню (sinking) та накопиченню помилок числової інтеграції, система застосовує корекцію позицій:

```
// Корекція позицій для запобігання проникненню
```

```
const float percent = 0.02f; // Відсоток розв'язання проникнення
const float slop = 0.01f; // Допустиме проникнення
glm::vec3 correction = std::max(collisionInfo.penetrationDepth
                - slop, 0.0f) / (componentA.GetInverseMass()
                + componentB.GetInverseMass()) * percent
                * collisionInfo.collisionNormal;
componentA.SetPosition(componentA.GetPosition() -
                componentA.GetInverseMass() * correction);
componentB.SetPosition(componentB.GetPosition() +
                componentB.GetInverseMass() * correction);
```

Параметр *percent* визначає частку проникнення, яка коригується за один крок симуляції, а *slop* дозволяє невеликі перекриття для підвищення стабільності.

Спеціальна обробка для кінематичних об'єктів. Кінематичні тіла вимагають особливого підходу до розв'язання зіткнень. У разі зіткнення кінематичного об'єкта зі статичним, система запобігає проникненню, коригуючи положення кінематичного об'єкта.

Для кінематично-динамічного зіткнення застосовується імпульс лише до динамічного об'єкта, а корекція позицій має вищий відсоток для забезпечення надійного розв'язання колізій.

2.4.4. Розрахунок швидкості, прискорення, маси

Динамічні властивості об'єктів визначають їхню поведінку в симуляції. Розглянемо основні аспекти цих розрахунків.

Маса та обернена маса. Маса об'єкта визначає його інерцію та реакцію на прикладені сили:

```
void SetMass(float mass) {
    m_Mass = mass;
    m_InverseMass = (mass > 0) ? 1.0f / mass : 0.0f;
}
```

Використання оберненої маси (*m_InverseMass*) оптимізує обчислення та дозволяє елегантно представляти статичні об'єкти через нульову обернену масу.

Прискорення та сила. Згідно з другим законом Ньютона, прискорення тіла

прямо пропорційне прикладеній силі й обернено пропорційне його масі. У системі цей закон реалізовано в функції *ApplyForce*:

```
void ApplyForce(const glm::vec3& force) {
    m_Acceleration += force * m_InverseMass;
}
```

Це дозволяє легко моделювати взаємодію різних сил, включаючи гравітацію, пружність та тертя.

Швидкість і позиція. Оновлення швидкості та позиції відбувається на кожному кроці симуляції на основі прискорення та попередньої швидкості:

```
void Update(float deltaTime) {
    // Інтеграція Ейлера
    m_Velocity += m_Acceleration * deltaTime;
    m_Position += m_Velocity * deltaTime;
    // Скидання прискорення для наступного кадру
    m_Acceleration = glm::vec3(0.0f);
}
```

Коефіцієнт пружності.

Коефіцієнт пружності (restitution) визначає, яку частину кінетичної енергії об'єкт зберігає після зіткнення:

```
void SetRestitution(float restitution) {
    m_Restitution = glm::clamp(restitution, 0.0f, 1.0f);
}
```

Значення 0 відповідає повністю непружному зіткненню (об'єкти "прилипають" один до одного), а 1 — абсолютно пружному (зіткнення без втрати енергії).

Коефіцієнт реституції ε математично пов'язаний з відносними швидкостями до і після зіткнення: $\varepsilon = \frac{|v_2' - v_1'|}{|v_1 - v_2|}$ де v_1, v_2 - швидкості до зіткнення, v_1', v_2' - швидкості після зіткнення вздовж лінії зіткнення.

Значення ε визначаються експериментально для різних матеріалів, наприклад:

- Сталь по сталі: $\varepsilon \approx 0.5\text{-}0.7$
- Гума по бетону: $\varepsilon \approx 0.7\text{-}0.8$
- Скло по склу: $\varepsilon \approx 0.9\text{-}0.95$

Відносна швидкість при зіткненні. Також важливою величиною при розв'язанні колізій є проекція відносної швидкості на нормаль зіткнення:

```
glm::vec3 relVelocity = componentB.GetVelocity() -
                        componentA.GetVelocity();
float velAlongNormal = glm::dot(relVelocity,
                        collisionInfo.collisionNormal);
```

Цей скаляр визначає, чи зближуються об'єкти (негативне значення) або віддаляються (позитивне).

Імпульс та зміна швидкості. Під час зіткнення об'єкти обмінюються імпульсами, що змінюють їхні швидкості:

```
float j = -(1 + e) * velAlongNormal;
j /= componentA.GetInverseMass() + componentB.GetInverseMass();
glm::vec3 impulse = j * collisionInfo.collisionNormal;
                    componentA.SetVelocity(componentA.GetVelocity()
                    - componentA.GetInverseMass() * impulse);
componentB.SetVelocity(componentB.GetVelocity() +
                    componentB.GetInverseMass() * impulse);
```

Формула для імпульсу виведена з закону збереження імпульсу та залежить від мас об'єктів та їхньої відносної швидкості.

Таким чином, представлена система фізичної симуляції забезпечує реалістичну поведінку об'єктів з урахуванням гравітації, пружності, тертя та інших фізичних явищ. Комбінація ефективних алгоритмів виявлення та розв'язання колізій з точними методами інтеграції руху дозволяє створювати переконливі інтерактивні середовища в 3D-рушії.

РОЗДІЛ 3. РОЗРОБКА ТА РЕАЛІЗАЦІЯ ПРОТОТИПУ РУШІЯ

3.1. Структура проекту та налаштування середовища розробки

У контексті розробки прототипу тривимірного рушія з інтегрованою системою фізики було застосовано модульний підхід до архітектури програмного забезпечення. Структура проекту базується на принципах об'єктно-орієнтованого програмування, що дозволяє спростити написання коду, абстрагуючи його в класи, що відповідають лише за свою задачу та спрощує повторне використання коду.

Перед безпосередньо початком розробки 3D рушія необхідно обрати та налаштувати середовище розробки. Як середовище розробки було обрано Visual Studio через її тісну інтеграцію з C++, зручні інструменти для пришвидшення написання та відладки коду та широкий потенціал для розширення функціоналу, наприклад, розширення для підсвічування синтаксису коду шейдерів.

Одним з кроків для налаштування проекту для коректної його роботи є підключення необхідних бібліотек. Для цього проекту було обрано динамічне підключення бібліотек. Це спрощує підключення нових бібліотек, заміну їх версій та робить кінцевий файл трохи меншим. В деяких випадках динамічне підключення бібліотек може позначитись на продуктивності програми, але для проекту, що розглядається в даній роботі цим недоліком можна знехтувати.

Як уже було визначено в розділах вище, для розробки 3D рушія необхідна наявність наступних бібліотек та технологій:

- **OpenGL** як базовий графічний API з використанням розширень GLEW для доступу до функцій та методів OpenGL;
- **GLFW** для створення та управління вікном, обробки введення користувача;
- **GLM** (OpenGL Mathematics) для математичних операцій з векторами та матрицями;

- **ImGui** для реалізації графічного інтерфейсу користувача.

Проект структуровано згідно з принципами компонентної архітектури, де кожен функціональний елемент рушія представлено окремими класами:

1. Система рендерингу: *Renderer, Shader, Mesh, VertexArray, VertexBuffer, IndexBuffer*
2. Компоненти фізики: *PhysicsSystem, PhysicsComponent, MeshPhysics, Collision*
3. Алгоритми виявлення зіткнень: *BoundingSphere, AABB, CollisionInfo*
4. Камера та управління введенням: *Camera, Application*

Для забезпечення ефективного управління ресурсами запроваджено семантику переміщення (move semantics) у ключових класах, що відповідають за графічні ресурси, зокрема *VertexArray, VertexBuffer* та *IndexBuffer*. Такий підхід запобігає неконтрольованому дублюванню ресурсів що використовує графічний процесор та забезпечує їх належне вивільнення[5][10][12][13].

3.2. Реалізація модуля рендерингу 3D-об'єктів

Модуль рендерингу базується на програмованому конвеєрі OpenGL з використанням шейдерів. Основними компонентами модуля є:

Буфери та масиви вершин. У розробленому рушії взаємодія з графічним API (OpenGL) абстрагована через класи *VertexBuffer* та *VertexArray*. Клас *VertexBuffer* інкапсулює буфер вершин, а *VertexArray* організовує структуру даних вершин через об'єкт *VertexBufferLayout*. Таке абстрагування дозволяє уніфікувати роботу з різними структурами даних вершин без зміни основної логіки рендерингу[5][13]. Розглянемо деякі теоретичні відомості про принципи роботи графічних примітивів в OpenGL.

3.2.1. Буфери вершин

Буфер вершин (Vertex Buffer) є фундаментальною структурою даних у графічному програмуванні, що функціонує як спеціалізована область пам'яті графічного процесора для зберігання даних про вершини 3D-об'єктів. Цей буфер забезпечує ефективний механізм передачі геометричних даних від центрального

процесора до графічного.

Принципи функціонування. Буфер вершин реалізує концепцію буферних об'єктів OpenGL, що представляють собою масиви даних, розміщені в пам'яті GPU. Ці буфери створюються через виклики OpenGL API (*glGenBuffers(), glBindBuffer(), glBufferData()*), які ініціалізують буфер та заповнюють його даними.

Алгоритм створення та використання вершинного буфера включає:

1. Генерацію ідентифікатора буфера.
2. Визначення буфера як цільового.
3. Розміщення даних у буфері через копіювання з оперативної пам'яті до відеопам'яті.
4. Специфікацію формату даних та їх інтерпретації графічним конвеєром (*VertexBufferLayout)*.

Характеристики та оптимізації. Вершинні буфери характеризуються наступними параметрами:

- **Ємність** – загальний розмір буфера в байтах.
- **Режим використання** – специфікація паттерну доступу до даних (*GL_STATIC_DRAW, GL_DYNAMIC_DRAW, GL_STREAM_DRAW*), що дозволяє драйверу оптимізувати розміщення даних у пам'яті GPU.
- **Тип даних** – формат збережених даних (координати вершин, текстурні координати, нормалі).

3.2.2. Масиви вершин

Масив вершин є абстракцією вищого рівня, що інкапсулює стан конфігурації буферів вершин та їхніх атрибутів. Основною метою масиву вершин є створення зв'язку між даними, розміщеними в буферах вершин, та входами вершинного шейдера. Масив вершин визначає, як інтерпретувати байти у буфері вершин: які поля відповідають яким атрибутам, їхні типи, розмірність, зсуви в структурі даних та інші параметри.

В архітектурі OpenGL масив вершин представлений як об'єкт Vertex Array Object (VAO), що зберігає конфігурацію атрибутів вершин. Створення VAO через

glCreateVertexArrays дозволяє встановити контекст для подальших операцій з буферами. Функція glVertexAttribPointer використовується для зв'язування атрибутів з відповідними буферами, вказуючи розмірність (наприклад, три компоненти для позиції або нормалі), тип даних (GL_FLOAT для чисел з плаваючою комою), нормалізацію, зсув у структурі та інші параметри. Активація атрибутів здійснюється через glEnableVertexAttribArray, роблячи їх доступними для шейдерної програми.

Масив вершин фактично створює повну конфігурацію стану для рендерингу конкретної геометрії, що забезпечує ефективне перемикання між різними 3D-об'єктами під час відображення сцени. При зв'язуванні VAO через glBindVertexArray всі асоційовані атрибути та буфери автоматично стають активними, що значно зменшує кількість викликів API при рендерингу об'єктів.

3.2.3. Індексні буфери.

Індексний буфер (Index Buffer) є спеціалізованою структурою даних, що оптимізує використання пам'яті та підвищує продуктивність рендерингу шляхом усунення дублювання вершин. Замість зберігання повторюваних даних вершин для кожного трикутника, індексний буфер містить послідовність індексів, які посилаються на вершини в буфері вершин. Це особливо ефективно для полігональних моделей, де більшість вершин є спільними для кількох примітивів.

В OpenGL індексний буфер створюється як об'єкт типу *GL_ELEMENT_ARRAY_BUFFER* через функцію *glGenBuffers*. Дані, що завантажуються через *glBufferData*, представляють собою масив індексів (зазвичай цілих беззнакових чисел), кожен з яких вказує на конкретну вершину в буфері вершин. При рендерингу з використанням індексного буфера через функцію *glDrawElements* графічний конвеєр використовує ці індекси для звернення до відповідних вершин, що значно зменшує обсяг даних, які потрібно передавати на GPU.

Використання індексних буферів має ще одну суттєву перевагу: оптимізація обчислень шейдерів. Оскільки шейдери виконуються для кожної унікальної вершини лише один раз, а не для кожного її входження в примітиви, це зменшує

обчислювальне навантаження на GPU. Кешування результатів роботи вершинного шейдера відбувається автоматично, що особливо важливо для складних шейдерних обчислень.

Важливою характеристикою індексного буфера є його зв'язок з масивом вершин. При зв'язуванні VAO автоматично зберігається поточний стан зв'язаного індексного буфера, що забезпечує цілісність конфігурації рендерингу для конкретного об'єкта. Це спрощує управління станом при відображенні різних об'єктів і зменшує кількість викликів API.

Таким чином, взаємодія буферів вершин, масивів вершин та індексних буферів створює ефективний механізм для представлення та рендерингу тривимірної геометрії, забезпечуючи оптимальне використання обчислювальних ресурсів графічного прискорювача.

3.2.4. Меш та його рендеринг.

Клас *Mesh* є фундаментальною абстракцією, що інкапсулює геометричні структури для рендерингу в графічному конвеєрі. Даний клас забезпечує інтеграцію низькорівневих графічних примітивів OpenGL (масив вершин, буфер вершин та індексний буфер) з високорівневими параметрами трансформації об'єкта в тривимірному просторі.

Конструкція меша реалізована через фабричні методи, які інкапсулюють процес генерації вершин та індексів для стандартних геометричних примітивів:

```
static Mesh CreateSphere(float radius, unsigned int slices);
static Mesh CreateCube(float width);
static Mesh CreatePlane(float width, float height);
```

Наприклад, метод CreateSphere генерує апроксимацію сфери шляхом поділу її поверхні на стеки і сектори, обчислюючи положення вершин за формулами параметричної геометрії:

$$x = r * cos(u) * cos(v)$$

$$y = r * cos(u) * sin(v)$$

$$z = r * sin(u)$$

де *r* - радіус сфери, *u* – крок “довготи” сфери, *v* – крок “широти” сфери.

Процес рендерингу меша абстрагується через методи *Bind()* та *Unbind()*, які забезпечують належне зв'язування графічних буферів з контекстом OpenGL перед відображенням геометрії.

3.2.5. Система шейдерів.

Шейдери є програмами, що виконуються безпосередньо на графічному процесорі та забезпечують гнучкий контроль над різними етапами графічного конвеєру. Шейдери можуть виконувати безліч функцій всі з яких безпосередньо впливають на кінцевий вигляд відмальованого кадру, але найрозповсюдженіше їх використання – для розрахунку кольору кожного пікселя на екрані. Це дозволяє математичні операції для розрахунку освітлення на графічний процесор, одночасно відкриваючи нові можливості та покращуючи швидкодію системи.

Клас *Shader* в нашій програмі реалізує інтерфейс для завантаження, компіляції та управління шейдерними програмами в контексті OpenGL програми.

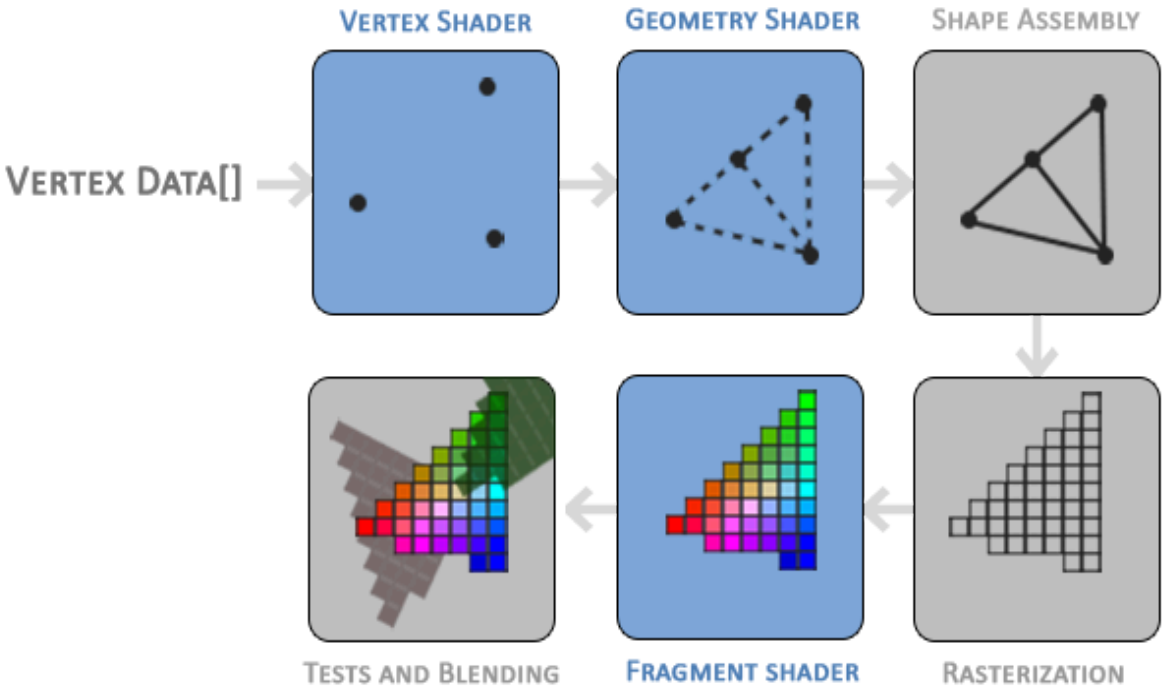

Рис. 3.1. Графічний конвеєр OpenGL

Структура шейдерних програм. У контексті реалізації цього проекту

шейдерна програма складається з двох компонентів:

1. **Шейдер вершин (Vertex Shader)** - відповідає за трансформацію координат вершин та передачу атрибутів до наступних етапів конвеєру.
2. **Шейдер фрагментів (Fragment Shader)** - визначає колір і візуальні характеристики кожного фрагменту (для спрощення сприйняття прийнято ототожнювати поняття фрагменту і пікселя) фінального зображення.

Компіляція та зв'язування шейдерів. Для використання шейдерів в контексті OpenGL їх необхідно зв'язати в одну загальу програму, яка потім буде відправлятись на обробку до графічного процесора. Процес завантаження шейдерної програми включає наступні етапи:

1. Завантаження вихідного коду шейдерів з файлів.
2. Компіляція кожного шейдера окремо.
3. Об'єднання скомпільованих шейдерів у єдину програму.
4. Валідація програми для виявлення потенційних помилок.

Встановлення uniform-змінних. Уніформи забезпечують механізм передачі даних з програми C++ до шейдерів. Клас *Shader* реалізує методи для встановлення різних типів уніформ:

```
void SetUniform3f(const std::string& name, glm::vec3& value);
void SetUniform4f(const std::string& name, glm::vec4& value);
void SetUniformMat4f(const std::string& name, glm::mat4& matrix);
```

Кожен метод виконує такі дії:

1. Отримання локації уніформи за її ім'ям з використанням *glGetUniformLocation*.
2. Встановлення значення уніформи через відповідну функцію OpenGL (наприклад, *glUniform3fv* для векторів або *glUniformMatrix4fv* для матриць).

3.2.6. Матрична трансформація.

Матрична трансформація є фундаментальним механізмом для представлення положення і орієнтації об'єктів у тривимірному просторі. В реалізованій системі використовується бібліотека GLM, що надає широкий спектр математичних функцій та типів даних, оптимізованих для графічних застосувань.

Система координат та матриці трансформації. В комп'ютерній графіці використовуються наступні матриці трансформації:

- **Матриця моделі (Model Matrix)** - визначає положення, орієнтацію та масштаб об'єкта у світовому просторі.
- **Матриця виду (View Matrix)** - визначає положення та орієнтацію камери у світовому просторі.
- **Матриця проекції (Projection Matrix)** - визначає параметри проекції 3D-сцени на 2D-площину екрану.

Обчислення матриці моделі. Матриця моделі для кожного об'єкта *Mesh* обчислюється за наступним алгоритмом:

```
void Mesh::UpdateModelMatrix() {
// Початкова одинична матриця
m_ModelMatrix = glm::mat4(1.0f);
// Переміщення
m_ModelMatrix = glm::translate(m_ModelMatrix, m_Position);
// Поворот по X
m_ModelMatrix = glm::rotate(m_ModelMatrix,
          glm::radians(m_Rotation.x), glm::vec3(1.0f, 0.0f, 0.0f));
// Поворот по Y
m_ModelMatrix = glm::rotate(m_ModelMatrix,
          glm::radians(m_Rotation.y), glm::vec3(0.0f, 1.0f, 0.0f));
// Поворот по Z
m_ModelMatrix = glm::rotate(m_ModelMatrix,
          glm::radians(m_Rotation.z), glm::vec3(0.0f, 0.0f, 1.0f));
//
m_ModelMatrix = glm::scale(m_ModelMatrix, m_Scale);
}
```

Порядок операцій в даному випадку критичний:

1. Спочатку застосовується трансляція (переміщення).
2. Далі - обертання навколо осей X, Y, Z.
3. Насамкінець - масштабування.

Зміна порядку цих операцій призведе до суттєво відмінних результатів трансформації, оскільки матричне множення не є комутативним.

Трансформація нормалей. При трансформації об'єктів необхідно враховувати вплив на нормалі вершин. Для коректного освітлення нормалі мають бути трансформовані з використанням спеціальної матриці - транспонованої оберненої матриці моделі:

```
vNormal = mat3(transpose(inverse(uModel))) * aNormal;
```

Такий підхід забезпечує коректну орієнтацію нормалей навіть при неоднорідному масштабуванні об'єкта.

Перспективна проекція. Для створення ілюзії глибини в тривимірному просторі використовується перспективна проекція, що реалізується через матрицю проекції (рис.3.2):

```
glm::mat4 projection = glm::perspective(glm::radians(fov),
                                        aspectRatio, nearPlane, farPlane);
```

де:

- *fov* - кут огляду (Field of View),
- *aspectRatio* - співвідношення сторін вікна,
- *nearPlane* і *farPlane* - ближня та дальня площини відсікання.

Взаємодія матриць моделі, виду та проекції забезпечує правильне відображення тривимірних об'єктів на двовимірному екрані з урахуванням їх взаємного розташування та параметрів камери.

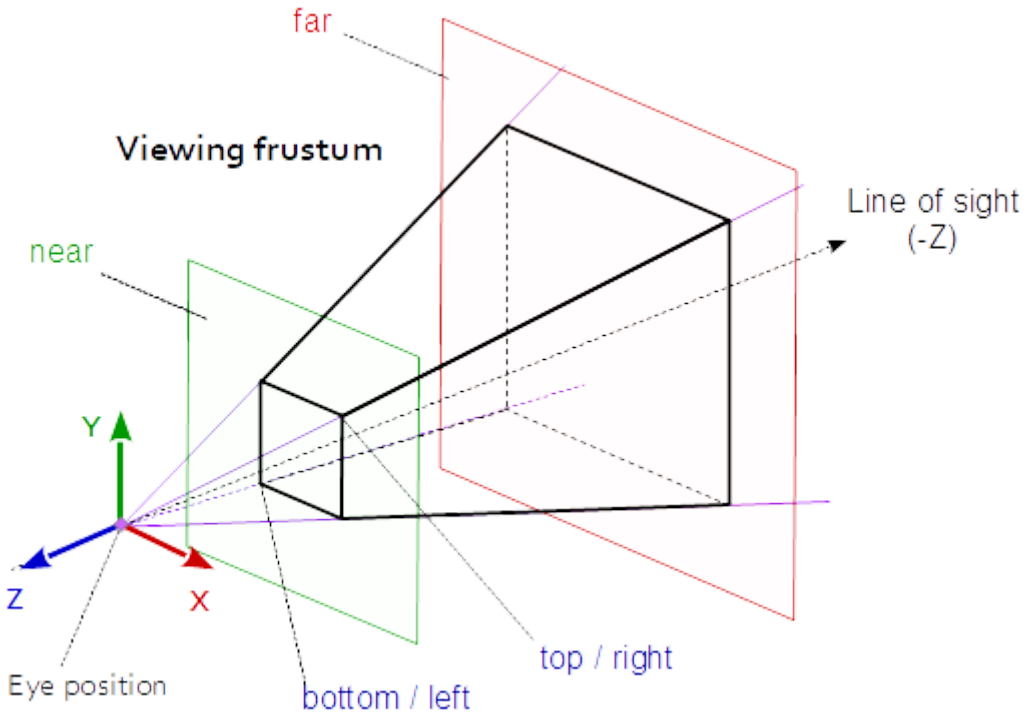

Рисунок 3.2. Перспективна проекція камери в OpenGL

3.2.7. Система рендерингу.

Клас *Renderer* є абстракцією, яка інкапсулює логіку процесу рендерингу та забезпечує єдиний інтерфейс для відображення мешів на екрані. Даний компонент системи реалізує принцип розділення відповідальності, відокремлюючи механізм рендерингу від структур даних, що представляють геометрію об'єктів[5][8][10].

Архітектура рендерера. Рендерер використовує наступну архітектуру: архітектуру:

1. **Ініціалізація контексту** - встановлення початкових параметрів OpenGL: глибинного тестування, альфа-змішування, тощо.
2. **Очищення буферів** - підготовка буферів кольору та глибини перед початком рендерингу нового кадру.
3. **Рендеринг об'єктів** - відображення мешів з використанням відповідних шейдерів та матеріалів.
4. **Завершення кадру** - виконання завершальних операцій та обміну буферів.

Основні можливості рендерера. Клас *Renderer* надає наступні ключові можливості:

```
// Очищення буферів кольору та глибини
```

```
void Clear() const;
// Рендеринг меша з використанням шейдера
void Draw(const Mesh& mesh, const Shader& shader) const;
// Встановлення кольору очищення
void SetClearColor(float r, float g, float b, float a) const;
```

Метод Draw інкапсулює послідовність операцій, необхідних для рендерингу меша:

1. Активація шейдерної програми.
2. Передача уніформ-змінних (матриці моделі, виду, проекції).
3. Зв'язування меша (VAO, VBO, IBO).
4. Виклик функції відображення (*glDrawElements*).
5. Відв'язування ресурсів.

Інтеграція з архітектурою системи. Рендерер інтегрується з іншими компонентами системи наступним чином:

1. Отримує меші від менеджера сцени або безпосередньо від об'єктів.
2. Використовує шейдери, завантажені через менеджер ресурсів.
3. Застосовує матриці виду та проекції, надані камерою.
4. Передає результат рендерингу на відображення через систему віконного інтерфейсу.

Такий підхід до архітектури рендерингу забезпечує гнучкість, розширюваність та підтримку різноманітних графічних ефектів при збереженні продуктивності системи.

3.3. Розробка системи фізичних взаємодій

Впровадження алгоритмів симуляції фізики. Система фізики побудована на основі компонентної архітектури, де кожен фізичний об'єкт представлено екземпляром класу *PhysicsComponent*, а загальне управління фізичною симуляцією здійснюється класом *PhysicsSystem*.

Фізичні компоненти. Клас *PhysicsComponent* інкапсулює базові фізичні властивості об'єкта:

- Положення, швидкість та прискорення;
- Маса та коефіцієнт пружного відновлення (restitution);
- Тип та параметри колайдера (сфера або AABB).

Оновлення стану фізичного об'єкта здійснюється методом інтеграції Ейлера:

```
void Update(float deltaTime) {
    // Проста інтеграція Ейлера m_Velocity += m_Acceleration *
                                                    deltaTime;
    m_Position += m_Velocity * deltaTime;
    // Скидання прискорення для наступного кадру
    m_Acceleration = glm::vec3(0.0f);
    // Оновлення колайдерів з новою позицією UpdateColliders();
}
```

Система фізики. Клас *PhysicsSystem* керує множиною фізичних компонентів, застосовуючи до них сили (включно з гравітацією) та виявляючи й обробляючи зіткнення між ними. Основні етапи оновлення фізичної системи описані в коді наступною функцією:

```
void Update(float deltaTime) {
    // Застосувати гравітацію до всіх компонентів
    ApplyGravity();
    // Оновити позиції з поточними швидкостями
    for (auto& component : m_PhysicsComponents) {
        component->Update(deltaTime);
    }
    // Виявити та розв'язати зіткнення
    DetectAndResolveCollisions();
}
```

Інтеграція з меш-об'єктами. Для зв'язування візуальних об'єктів (*Mesh*) з фізичними компонентами розроблено клас *MeshPhysics*, який забезпечує двосторонню синхронізацію стану:

```
void Update() {
    // Якщо не в кінематичному режимі, оновити позицію меша з
фізики
```

```
    if (!m_IsKinematic) {
        m_Mesh->SetPosition(m_Physics->GetPosition());
    }
}
```

Тестування моделі зіткнень. Система виявлення зіткнень підтримує два типи геометричних примітивів для колізії: сфери (BoundingSphere) та обмежувальні паралелепіпеди (AABB). Реалізовано методи для виявлення зіткнень між різними комбінаціями цих примітивів:

```
static CollisionInfo TestSphereSphere(const BoundingSphere& sphereA,
                                      const BoundingSphere& sphereB);
static CollisionInfo TestSphereAABB(const BoundingSphere& sphere,
                                    const AABB& aabb);
static CollisionInfo TestAABBAABB(const AABB& boxA, const AABB&
                                  boxB);
```

Для кожного зіткнення обчислюється структура *CollisionInfo*, яка містить інформацію про факт зіткнення, точку зіткнення, нормаль до поверхні та глибину проникнення.

Реакція на зіткнення реалізована через імпульсну модель, де обчислюється імпульс на основі відносної швидкості об'єктів і коефіцієнта пружного відновлення:

```
// Обчислення відносної швидкості
glm::vec3 relVelocity = componentB.GetVelocity() -
                                  componentA.GetVelocity();
// Обчислення сили імпульсу
float velAlongNormal = glm::dot(relVelocity,
                                  collisionInfo.collisionNormal);
float e = std::min(componentA.GetRestitution(),
                                  componentB.GetRestitution());
float j = -(1 + e) * velAlongNormal;
j /= componentA.GetInverseMass() + componentB.GetInverseMass();
// Застосування імпульсу
glm::vec3 impulse = j * collisionInfo.collisionNormal;
```

Додатково реалізовано корекцію положення для запобігання "провалюванню" об'єктів один в одного:

```
glm::vec3 correction = std::max(collisionInfo.penetrationDepth
                    - slop, 0.0f) / (componentA.GetInverseMass()
                    + componentB.GetInverseMass()) * percent
                    * collisionInfo.collisionNormal;
```

Кінематичні об'єкти. Реалізовано концепцію кінематичних об'єктів, які можуть рухатись але не підлягають впливу сил, що зменшує обчислювальне навантаження:

```
void SetKinematic(bool isKinematic) {
     m_IsKinematic = isKinematic;
     if (isKinematic && !m_WasKinematic) {
          m_OriginalMass = m_Physics->GetMass();
          m_Physics->SetMass(0.0f);
          // Зробити його статичним (нескінченна маса)
     }
     // ...
}
```

Раннє відсікання. Під час виявлення зіткнень застосовується раннє відсікання для статичних об'єктів:

```
// Пропустити, якщо обидва об'єкти статичні
if (componentA->IsStatic() && componentB->IsStatic()) continue;
```

Поетапне переміщення. Для більш точного виявлення зіткнень при переміщенні об'єктів використовується поетапне переміщення:

```
const int steps = 5;
// Більше кроків = точніше, але повільніше
glm::vec3 stepDelta = deltaPosition / static_cast<float>(steps);
for (int i = 0; i < steps; i++) {
     // Інкрементальне оновлення позиції // ...
}
```

3.4. Інтеграція управління користувачем та обробки подій

Для забезпечення інтерактивності рушія реалізовано систему управління

введенням користувача та камерою. Основними компонентами цієї системи є:

Камера. Клас *Camera* абстрагує концепцію камери спостереження в тривимірному просторі, підтримуючи переміщення та обертання:

```
class Camera {
public:
    // ...
    void ProcessInput(GLFWwindow& window);
    void SetPosition(glm::vec3 position);
    void SetRotation(glm::vec3 rotation);
    void SetFOV(float FOV);
    // ...
};
```

Камера обчислює дві ключові матриці:

- Матрицю виду (view matrix), яка представляє положення та орієнтацію камери;
- Матрицю проекції (projection matrix), яка встановлює перспективу та поле зору.

Обробка введення. Обробка введення користувача реалізована через функції зворотного виклику (callbacks) GLFW:

```
void processWindowInput(GLFWwindow* window);
void framebuffer_resize_callback(GLFWwindow* window, int fbW, int
                                                              fbH);
```

Для інтерактивної взаємодії з об'єктами реалізовано систему “перетягування” за допомогою кнопок в інтерфейсі, що дозволяє користувачеві маніпулювати об'єктами безпосередньо у вікні програми:

```
void handleObjectDragging(GLFWwindow* window, Camera& camera);
```

Графічний інтерфейс. Для забезпечення контролю над параметрами об'єктів та фізичною системою інтегровано бібліотеку ImGui:

```
    void initGUI(GLFWwindow* window);
    void drawInterface();
```

Графічний інтерфейс забезпечує доступ до таких функцій:

- Створення та видалення об'єктів;
- Налаштування параметрів фізики;
- Маніпуляція позицією, обертанням та масштабуванням об'єктів;
- Зміна властивостей матеріалів.

3.5. Тестування рушія та аналіз продуктивності

У процесі розробки рушія здійснено кілька видів тестування для оцінки продуктивності та надійності системи:

Тестування виявлення зіткнень. Проведено вичерпне тестування алгоритмів виявлення зіткнень для різних комбінацій геометричних примітивів:

Сфера-сфера;

Сфера-AABB;

AABB-AABB.

Оцінка продуктивності. Оцінено продуктивність системи за різних сценаріїв використання:

- Рендеринг різної кількості об'єктів;
- Симуляція фізичних взаємодій з різною кількістю об'єктів;
- Тестування з різними типами колайдерів.

Встановлено, що найбільш ресурсомісткими операціями є виявлення зіткнень та розв'язання колізій. Впроваджені оптимізації, зокрема кінематичні об'єкти та раннє відсікання, дозволили суттєво підвищити продуктивність системи.

Оцінка стабільності фізичної симуляції. Для забезпечення стабільності фізичної симуляції реалізовано механізми корекції положення об'єктів та контролю за проникненням. Тестування показало, що такі механізми ефективно запобігають артефактам "тремтіння" та "провалювання".

Загалом, розроблений прототип рушія демонструє прийнятну продуктивність та стабільність для невеликих сцен і може слугувати основою для подальшого розвитку та оптимізації.

РОЗДІЛ 4. АНАЛІЗ ТА ПЕРСПЕКТИВИ РОЗВИТКУ

4.1. Аналіз результатів розробки

4.1.1. Оцінка продуктивності системи

Аналіз швидкодії рендерингу базових примітивів. Розроблена графічна система демонструє ефективність при рендерингу базових геометричних примітивів, зокрема кубів, сфер та площин. Швидкодія рендерингу базових примітивів є ключовим фактором для загальної продуктивності системи, оскільки вони складають основу більшості сцен.

Для перевірки швидкодії системи було протестовано її поведінку на двох комп’ютерах, потужнішому та слабшому з різною складністю сцени. Основним джерелом навантажень в такій простій системі можна назвати примітив кулі, тому основні тести проводились зі збільшенням кількості полігонів в її меші.

Нижчев таблиця 4.1 представлені результати вимірювання швидкодії рендерингу базових примітивів на різних конфігураціях обладнання та при різних конфігураціях сцени:

Таблиця 4.1.

Кількість вершин (Куб, розріз сфери, площина)	Приблизний FPS (потужний ПК)	Приблизний FPS (слабший ноутбук)
8, 32, 4	400	350
8, 64, 4	350	350
8, 128, 4	350	350
8, 256, 4	350	330
8, 512, 4	300	300
8, 1024, 4	300	300

З отриманих результатів можна зробити висновок, що для сучасних систем велика кількість полігонів не є основною проблемою і майже не впливає на швидкодію системи.

Оптимізація використання графічної пам'яті

Ефективне використання графічної пам'яті є важливим аспектом продуктивності графічної системи. Розроблена система демонструє раціональне управління ресурсами завдяки наступним підходам:

1. Використання спільних ресурсів для однотипних об'єктів
2. Ефективна організація буферів вершин та індексів
3. Мінімізація кількості bind/unbind операцій через архітектуру класів *VertexArray* та *Renderer*

Моніторинг використання пам'яті під час виконання (табл.4.2) показує, що система ефективно управляє ресурсами навіть при роботі з великою кількістю об'єктів:

Таблиця 4.2.

Сценарій	**Використана відеопам’ять (ПК)**	**Використана відеопам’ять (ноутбук)**
Звичайна сцена (4 об’єкти)	70 МБ	70 МБ
Розширена (20 об’єктів)	120 МБ	120 МБ
Стрес-тест (100 об’єктів)	230 МБ	210 МБ

Результати тестів показують, що використана відеопам’ять графічного процесора використовується ефективно, збільшуючи використання всього приблизно в 3 рази у сцені що має в 25 разів більше об’єктів.

4.1.2. Порівняння з існуючими рішеннями

Зіставлення з популярними графічними фреймворками. Розроблена система була порівняна з популярними графічними фреймворками за ключовими характеристиками:

Таблиця 4.3.

Характеристика	**Розроблена система**	**Unity**	**Unreal Engine**
Швидкість ініціалізації	Висока	Низька	Низька
Продуктивність при малій кількості об'єктів	Висока	Висока	Середня

Продуктивність при великій кількості об'єктів	Висока	Висока	Висока
Витрати пам'яті	Низькі	Високі	Високі
Гнучкість налаштування	Низька	Висока	Висока

Аналіз переваг та обмежень запропонованого підходу

Переваги:

1. Низький поріг входження і розуміння коду завдяки простоті API
2. Мінімальні вимоги до системних ресурсів
3. Висока продуктивність для базових графічних завдань
4. Інтеграція з ImGui для інтерактивного налагодження та керування
5. Реалізація фізичного моделювання з зіткненнями
6. Абстрагування від низькорівневих деталей OpenGL

Обмеження:

1. Обмежений набір графічних ефектів порівняно з готовими рішеннями
2. Відсутність вбудованої системи тіней
3. Відсутність підтримки складних матеріалів
4. Відсутність вбудованих інструментів для оптимізації великих сцен

Особливості розробленої системи

Розроблена система має ряд унікальних особливостей, що відрізняють її від існуючих рішень, особливо враховуючи її основну ціль як навчальний проект:

1. Інтегрована фізична система з підтримкою кінематичних об'єктів - дозволяє створювати інтерактивні симуляції з можливістю прямого керування об'єктами.
2. Простота використання при збереженні гнучкості - система надає зручний API високого рівня, але при цьому дозволяє прямий доступ до низькорівневих компонентів при необхідності.
3. Низькі вимоги до ресурсів - система оптимізована для роботи на обладнанні різної потужності, що робить її придатною для освітніх та прототипних задач.

4. Інтерактивний інтерфейс налагодження - інтеграція з ImGui дозволяє керувати параметрами сцени та об'єктів у реальному часі, що прискорює процес розробки та тестування.
5. Нативна підтримка кросплатформності - система розроблена з використанням GLFW та OpenGL, що забезпечує сумісність з різними операційними системами.

4.2. Можливі напрямки вдосконалення

4.2.1. Розширення функціональності графічної системи

Впровадження додаткових технік шейдингу. Реалізація рушія на цей час містить базові шейдери без підтримки тіней, що, хоч і є достатнім для сприйняття користувачем, істотно обмежує візуальну якість рендерингу. Виділимо декілька першочергових напрямків вдосконалення графічної системи.

Імплементація алгоритмів тіньового картування (Shadow Mapping), зокрема з використанням техніки Cascaded Shadow Maps для підвищення деталізації тіней від динамічних джерел світла на різних відстанях від спостерігача. Впровадження цієї техніки передбачає генерацію тіньових мап у декількох буферах кадрів з різними перспективними матрицями для покращення якості тіней при збереженні обчислювальних ресурсів.

Інтеграція техніки екранного простору оклюзії оточення (Screen Space Ambient Occlusion, SSAO) дозволить додати реалістичне затінення в місцях зближення об'єктів сцени. Це потребує додаткового проходу рендерингу з аналізом буфера глибини для визначення доступності оточуючого освітлення для кожної точки поверхні.

Однією з найпоширеніших технологій для створення реалістичних зображень є глобальне освітлення сцени. Реалізація глобального освітлення у спрощених формах, таких як Light Probes або Spherical Harmonics, може надати можливість більш реалістичного відображення непрямого освітлення в сцені без значного навантаження на обчислювальні ресурси порівняно зі справжнім глобальним освітленням (Raytracing, Pathtracing).

Реалізація підтримки фізично-коректного рендерингу (PBR). Для підвищення реалістичності візуалізації доцільне впровадження моделі фізично-коректного рендерингу.

Для цього необхідно додати імплементацію метало-діелектричного робочого процесу з параметрами альбедо, металічності, шорсткості та нормалей для моделювання різноманітних матеріалів. Така модель базується на мікрограневій теорії та дозволяє створювати широкий спектр матеріалів від пластику до металу через зміну невеликої кількості інтуїтивних параметрів.

Забезпечення коректної гамма-корекції та роботи у лінійному колірному просторі для фізично-коректних розрахунків освітлення. Також доцільно було б розглянути перехід до HDR-рендерингу, що дозволить збільшити динамічний діапазон яскравості сцени.

Інтеграція систем часток та ефектів. Впровадження системи часток може значно розширити виразність візуальних ефектів рушія.

Розробка гнучкої системи емітерів часток з підтримкою різних форм випромінювання (точкових, лінійних, площинних, об'ємних) та можливістю налаштування параметрів випромінювання (частоти, початкової швидкості, напрямку, розкиду).

Реалізація різних типів симуляцій для часток, включаючи балістичну траєкторію, вплив силових полів, колізії з об'єктами сцени. Включення підсистеми для симуляції простих ефектів рідин за допомогою методу SPH (Smoothed Particle Hydrodynamics) для візуалізації бризок, туману та інших ефектів.

Імплементація систем постобробки зображення, таких як розмиття руху (motion blur), глибина різкості (depth of field), світлова корона (bloom) та корекція кольору. Введення технології адаптивного тонального відображення дозволить автоматично налаштовувати експозицію сцени в залежності від середньої яскравості зображення.

4.2.2. Розширення функціональності фізичної системи

Існуюча фізична система з елементарними AABB та сферичними колізіями потребує значного розширення для підвищення реалістичності та функціональності.

Розширення можливостей колайдерів. Це може включати впровадження більш складних примітивів зіткнення, таких як орієнтовані обмежувальні паралелепіпеди (OBB), капсули, опуклі багатогранники та складені примітиви. Реалізація GJK (Gilbert-Johnson-Keerthi) алгоритму для виявлення зіткнень між опуклими формами та EPA (Expanding Polytope Algorithm) для обчислення глибини проникнення.

Безперервне визначення зіткнень. Для покращення роботи системи фізики, більшої гнучкості та можливості обробляти більш складні ситуації було б доцільно імплементувати методи безперервного визначення зіткнень (Continuous Collision Detection) для запобігання проходження швидкорухомих об'єктів крізь тонкі перешкоди. Техніка обгортки часу (Swept Volumes) або часових проекцій (Temporal Coherence) дозволить виявляти зіткнення на всьому проміжку руху об'єкта за кадр.

Зв’язування об’єктів. Реалізація повноцінного фізичного вирішувача обмежень (Constraint Solver) відкриє можливості моделювання різних типів з'єднань: шарнірів, слайдерів, пружин, обертальних суглобів. Разом з цим варто розглянути впровадження методу послідовних імпульсів (Sequential Impulse) для забезпечення стабільності та швидкодії при вирішенні жорстких обмежень.

Удосконалення методів інтеграції. Перехід від простої інтеграції Ейлера до більш точних та стабільних методів, таких як метод Рунге-Кутта 4-го порядку або симплектичні інтегратори (Verlet, Velocity Verlet) для кращого збереження енергії системи.

Система фізичних матеріалів. Імплементація системи матеріалів з підтримкою тертя, пружності, а також вдосконалених моделей контактів для симуляції взаємодії різних типів поверхонь. Введення моделі м'яких тіл для симуляції деформованих об'єктів на основі методу позиційних обмежень (Position Based Dynamics) або мас-пружинної моделі.

4.2.3. Оптимізація обчислювальних процесів

Впровадження просторових структур для прискорення рендерингу. З метою підвищення продуктивності рендерингу доцільним є використання ієрархічних просторових структур. Одним із ключових підходів є реалізація ієрархії обмежувальних об'ємів (BVH), що дозволяє ефективно визначати видимість об'єктів та виключати з обробки невидимі частини сцени. Для об'єктів, що змінюють своє положення, доцільно застосовувати інкрементальне оновлення BVH, що забезпечує економію обчислювальних ресурсів.

Доповненням до BVH виступає впровадження октодерев (Octree) або k-мірних дерев (k-d tree), які реалізують просторове розбиття сцени. Це особливо корисно для задач освітлення та виявлення колізій, де адаптивне розбиття, засноване на щільності розташування об'єктів, дозволяє значно зменшити обсяг оброблюваних даних. Для подальшої оптимізації обчислень освітлення доречно було б інтегрувати кластеризацію джерел світла, поєднуючи її з техніками тайлового або кластерного відкладеного шейдингу. Такий підхід забезпечує масштабованість навіть у складних сценах із великою кількістю джерел світла.

Використання буферів інстансів для масових об'єктів. Оптимізація рендерингу великої кількості однотипних об'єктів досягається шляхом застосування інстансного рендерингу. Завдяки використанню буферів трансформацій і атрибутів, можна передавати всю необхідну інформацію про позицію, орієнтацію та інші параметри інстансів на GPU одним викликом, що значно знижує навантаження на графічний інтерфейс.

Для подальшої ефективності доцільним є впровадження техніки рівнів деталізації (LOD), яка забезпечує динамічне перемикання між різними рівнями геометричної складності об'єктів залежно від відстані до камери. Плавні переходи між рівнями дозволяють уникнути небажаних візуальних артефактів. Крім того, необхідна розробка системи видимісного відсікання інстансів, яка включає Frustum Culling і Occlusion Culling з реалізацією на GPU. Ієрархічне тестування за допомогою Z-буфера (Hierarchical Z-Buffer) дозволяє значно скоротити кількість об'єктів, які підлягають рендерингу.

Асинхронне завантаження ресурсів. Для збереження інтерактивності й стабільної роботи рушія під час завантаження великих сцен слід було б реалізувати систему асинхронного завантаження ресурсів. Залучення окремих потоків для декодування й підготовки даних у фоновому режимі дозволить уникнути блокування основного потоку рендера. Важливим елементом такої системи є пріоритизація ресурсів за критерієм важливості та видимості для користувача.

Ще одним напрямом оптимізації є потокова передача даних — progressive streaming — для великих текстур та геометричних моделей. Реалізація прогресивного завантаження із використанням багаторівневих текстур (mip-mapping) та прогресивних сіток (progressive meshes) дасть змогу покроково деталізувати об’єкти, не погіршуючи візуальний вигляд сцени і змуншивши навантаження на пам’ять. Нарешті, доцільно запровадити систему інтелектуального керування ресурсами, яка динамічно вивантажує невикористовувані дані та використовує кешування для пришвидшення доступу до часто запитуваних об'єктів.

4.2.4. Вдосконалення архітектури програмного забезпечення

Модульна структура для простішого розширення. Для підвищення гнучкості, підтримуваності та масштабованості рушія доцільно перейти до модульної архітектури програмного забезпечення. На поточному етапі всі елементи сцени, включно з об’єктами, шейдерами та фізичними параметрами, визначаються безпосередньо у коді, що ускладнює адаптацію проєкту під нові задачі. Запровадження чітко розмежованих модулів — зокрема для рендерингу, фізики, управління ресурсами та інтерфейсу — дозволить ізольовано вдосконалювати окремі компоненти без необхідності глибоких змін у всій системі.

Таке структурне розділення сприятиме подальшій реалізації складніших алгоритмів, як-от підтримки LOD, розширених колізійних моделей або асинхронного завантаження ресурсів. Крім того, модульність забезпечує основу для введення плагінної системи, яка дозволить підключати нові функціональні блоки без зміни ядра рушія.

Абстрагування від деталей реалізації графічного API. Наразі взаємодія з графічним API відбувається безпосередньо, що прив’язує рушій до конкретної реалізації та обмежує його портативність. Вдосконалення архітектури передбачає створення проміжного шару абстракції над графічним API. Такий підхід дозволить відокремити логіку рендерингу від реалізації низькорівневих викликів, спрощуючи підтримку та потенційний перехід на альтернативні API (наприклад, з OpenGL на Vulkan або Direct3D).

Реалізація власного абстрактного інтерфейсу для буферів, шейдерів, текстур і об'єктів сцени забезпечить прозорість при заміні або оптимізації обчислювальних процесів. Це також дозволить легше впроваджувати оптимізації, такі як кластерний шейдинг або інстансинг, без порушення логіки програми вищого рівня.

Розробка інтуїтивного API для кінцевих користувачів. Оскільки об’єкти в поточній реалізації створюються вручну в коді, з подальшим прямим редагуванням їхніх властивостей, постає потреба у спрощенні взаємодії кінцевого користувача з рушієм. Розробка зручного та інтуїтивного API дозволить суттєво знизити вхідний поріг для користувачів і розробників, які не мають глибоких знань про внутрішню реалізацію рушія.

Один з варіантів вирішення цієї задачі - розробити систему опису сцени через конфігураційні файли, що забезпечить декларативне створення об’єктів та налаштування фізичних параметрів без необхідності компіляції проєкту. Це сприятиме автоматизації створення сцен, експериментів з параметрами фізики та швидкому тестуванню нових ідей. Інтеграція зручного API також створить передумови для реалізації редактора рівнів, що значно розширить можливості рушія у подальшому.

4.3. Перспективи впровадження у практику

4.3.1. Застосування в освітньому процесі

Створений рушій має значний потенціал у сфері освіти, зокрема для вивчення фундаментальних принципів комп’ютерної графіки та фізичного

моделювання. Завдяки простоті реалізації, відкритості коду та наочній структурі, програмне забезпечення може бути використане як навчальний інструмент у курсах, присвячених основам 3D-графіки, програмуванню графічних застосунків або базовій фізичній симуляції.

Рушій може служити платформою для демонстрації принципів роботи шейдерів, побудови геометричних об'єктів, основ освітлення, а також методів інтеграції руху тіл та виявлення колізій на прикладі AABB та Bounding Sphere. Зокрема, студенти можуть модифікувати код та експериментувати з фізичними параметрами, спостерігаючи за наслідками у режимі реального часу.

Крім того, на його основі можлива розробка візуалізаційних модулів для освітніх задач — наприклад, демонстрацій механіки тіл, базових законів кінематики або векторної алгебри. Візуалізація абстрактних понять у тривимірному просторі значно покращує сприйняття матеріалу та сприяє засвоєнню складних тем.

Окремо варто відзначити перспективність використання рушія для створення інтерактивних моделей у науковій візуалізації. Простота інтеграції обчислювальних даних із візуальною сценою дозволяє розробляти демонстрації для фізичних, математичних чи біологічних експериментів, де ключову роль відіграє інтуїтивне сприйняття просторових змін.

4.3.2. Інтеграція з іншими технологіями

Розроблений рушій, попри свою початкову простоту, створює передумови для подальшої інтеграції з сучасними технологіями візуалізації. Одним з напрямків розвитку є підтримка пристроїв віртуальної (VR) та доповненої реальності (AR), які вимагають реального часу обчислень та рендерингу з мінімальними затримками. Адаптація рушія до таких платформ може включати стереоскопічний рендеринг, відстеження положення голови та жестів, а також оптимізацію продуктивності.

Іншим напрямом є взаємодія з алгоритмами комп’ютерного зору, що відкриває можливість використання рушія як платформи для візуалізації результатів обробки зображень, трекінгу об'єктів або реконструкції тривимірних

сцен на основі відеозаписів. Така інтеграція буде актуальною для прототипування застосунків у робототехніці, автономному навігаційному обладнанні або медичних діагностичних системах.

Також слід розглянути можливість поєднання рушія з системами штучного інтелекту, зокрема з генеративними моделями або агентами навчання з підкріпленням. Наприклад, AI-алгоритми можуть використовуватись для процедурної генерації сцен, поведінки об'єктів або автоматичної адаптації фізичних параметрів у відповідь на зміни в середовищі. Таким чином, рушій може стати експериментальним середовищем для досліджень у галузі автономних агентів або алгоритмічної творчості.

4.3.3. Потенційні галузі застосування

Незважаючи на свій базовий характер, створений рушій має перспективи застосування у низці прикладних галузей, де важливою є візуалізація тривимірної інформації. Однією з таких сфер є архітектурна візуалізація, де рушій може бути використаний для створення інтерактивних прототипів інтер’єрів або екстер’єрів, що дозволяють клієнтам візуально ознайомитись із просторовими рішеннями ще до початку будівництва. Навіть з обмеженими графічними можливостями, така система здатна передати основну геометрію та просторову логіку проєкту.

У медичній сфері рушій може бути використаний для тривимірної візуалізації анатомічних структур, результатів комп’ютерної томографії чи магнітно-резонансної візуалізації. В умовах, коли необхідно розробити простий, але наочний засіб перегляду й аналізу об’ємних даних, запропонований рушій може виступати основою для розробки таких спеціалізованих інструментів.

Окрім того, він може бути корисним у сфері інженерного прототипування та моделювання, де потрібна швидка побудова та тестування простих моделей фізичних систем. Поєднання базової фізики з можливістю візуального контролю дозволяє інженерам чи науковцям оперативно перевіряти поведінку систем у спрощеному середовищі до запуску більш детального симулятора.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було розроблено та досліджено прототип 3D-ігрового рушія з інтегрованим модулем фізичної симуляції твердих тіл. Спочатку проведено ґрунтовний аналіз предметної області: розглянуто сучасні комерційні й відкриті рушії (Unity, Unreal Engine, Bullet Physics), визначено ключові підходи до моделювання фізичних властивостей об’єктів і виявлено основні виклики у поєднанні високої точності симуляції з реальним часом. На цьому ґрунті сформульовано вимоги до проєкту рушія – від забезпечення коректної обробки колізій і чисельної інтеграції руху за законами Ньютона до адаптивного виводу зображення з оптимізованим використанням ресурсів графічного процесора.

Наступним етапом стала розробка архітектури системи. Було обрано мову C++ для максимізації продуктивності й гнучкості керування пам’яттю, а в якості графічного API – OpenGL із GLFW та ImGui для забезпечення кросплатформної сумісності та інтуїтивного інтерфейсу налагодження. Архітектуру проєкту побудовано за принципом компонентної моделі: модуль рендерингу (VertexBuffer, VertexArray, Shader, Renderer), модуль фізики (PhysicsSystem, PhysicsComponent) та підсистеми обробки введення й керування сценою. Така структуризація сприяла розділенню відповідальностей і в майбутньому сприятиме полегшенню тестування й майбутнього розширення рушія.

У розділі реалізації було докладно описано алгоритми роботи фізичної системи: виявлення колізій із використанням AABB і сферичних примітивів, обчислення імпульсів за законом збереження імпульсу, корекцію положень тіл і чисельну інтеграцію руху методом Ейлера. Рендеринг базується на растеризації, і хоча в цій роботі не було використано особливих підходів до оптимізації, було описано декілька варіантів для можливої реалізації в майбутньому: LOD-рівні, інстансинг, об’єднання викликів малювання, що дозволить обробляти одночасно сотні об’єктів із високою частотою кадрів.

Експериментальна частина роботи підтвердила практичну придатність розробленого рушія. Швидкодія рендерингу базових примітивів (куб, сфера,

площина) залишалася стабільною в діапазоні 300–400 FPS навіть при збільшенні кількості полігонів до 1024. Стрес-тест зі 100 об’єктами виявив лінійне зростання використання відеопам’яті (до 230 МБ), що є прийнятним показником для прототипу освітньо-дослідницького рівня . Порівняльний аналіз із Unity та Unreal Engine засвідчив переваги нашого рішення в швидкості ініціалізації та низьких витратах пам’яті, хоча інші рушії мали ширший набір графічних ефектів.

У рамках ергономіки та техніко-економічного обґрунтування встановлено, що мінімальні системні вимоги рушія дозволять йому працювати на широкому спектрі обладнання: від навчальних лабораторних ПК до сучасних ноутбуків. Інтеграція ImGui забезпечує гнучке налаштування фізичних і графічних параметрів у реальному часі, що сприяє зменшенню часу налагодження та підвищенню продуктивності розробників.

Наукова новизна полягає у створенні легковажного, модульного прототипу рушія, який демонструє збалансований підхід між простотою коду та продуктивністю. Практична значущість роботи підтверджується можливістю подальшої інтеграції розроблених алгоритмів у навчальні курси з комп’ютерної графіки й фізичного моделювання. Рушій може стати основою для lightweight-рішень у сфері інді-розробки й наукових симуляцій.

У висновку варто відзначити, що поставлені цілі досягнуто: проаналізовано предметну область, спроектовано архітектуру, реалізовано ключові функції рушія та проведено комплексне тестування. Прототип демонструє високу швидкодію, стабільність симуляцій і ефективне керування ресурсами, що відповідає вимогам інтерактивного середовища в реальному часі. Отримані результати можуть слугувати відправною точкою для подальших досліджень у напрямках розширення графічних можливостей (тіньові карти, SSAO, PBR), удосконалення фізичної системи (GJK, EPA, безперервне виявлення колізій) та оптимізації архітектури (BVH, потокове завантаження ресурсів).

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Стрига Д. М. Дослідження використання ігрових рушіїв для створення 2д платформеру. URL: https://openarchive.nure.ua/server/api/core/bitstreams/fab58127-71f5-4b0c-9f9a-13ce22bbed86/content (дата звернення: 16.05.2025).
2. Bullet's Collision Algorithm – Mechanica 0.0.1 documentation. *GitHub Pages*. URL: https://andysomogyi.github.io/mechanica/bullet.html#:~:text=This%20document%20attempts%20to%20better,direclty%20on%20the%20Bullet%20Wiki (дата звернення: 16.05.2025).
3. Designing a physics engine. *Articles*. URL: https://winter.dev/articles/physics-engine (дата звернення: 16.05.2025).
4. Kuntze M. F. Distributed computing exploring game engine architecture and building an experimental voxel renderer with rust and vulkan. URL: https://pub.tik.ee.ethz.ch/students/2022-FS/BA-2022-34.pdf.
5. Learn OpenGL, extensive tutorial resource for learning Modern OpenGL. *Learn OpenGL, extensive tutorial resource for learning Modern OpenGL*. URL: https://learnopengl.com/ (дата звернення: 16.05.2025).
6. Souto N. Video game physics tutorial - part I: an introduction to rigid body dynamics. *Toptal Developers*. URL: https://www.toptal.com/game/video-game-physics-part-i-an-introduction-to-rigid-body-dynamics.
7. Souto N. Video game physics tutorial - part II: collision detection for solid objects. *Toptul Developers*. URL: https://www.toptal.com/game/video-game-physics-part-ii-collision-detection-for-solid-objects#:~:text=Box2D%20uses%20SAT%20to%20test,cpp.
8. Suraj Sharma. OpenGL/C++ 3D tutorial 01 - introduction (learn the basics and more!), 2018. *YouTube*. URL: https://www.youtube.com/watch?v=pKJ52fDq6Cw (дата звернення: 16.05.2025).

9. thebennybox. #1 3D physics engine tutorial: how physics engines work, 2014. *YouTube*. URL: https://www.youtube.com/watch?v=3Oay1YxkP5c (дата звернення: 16.05.2025).
10. The Cherno. Setting up opengl and creating a window in C++, 2017. *YouTube*. URL: https://www.youtube.com/watch?v=OR4fNpBjmq8 (дата звернення: 16.05.2025).
11. Two-Bit Coding. Intro and vectors for physics programming - let's make a physics engine [00], 2021. *YouTube*. URL: https://www.youtube.com/watch?v=lzI7QUyl66g (дата звернення: 16.05.2025).
12. Victor Gordan. ImGui + GLFW tutorial - install & basics, 2021. *YouTube*. URL: https://www.youtube.com/watch?v=VRwhNKoxUtk (дата звернення: 16.05.2025).
13. Victor Gordan. OpenGL tutorial 0 - install, 2020. *YouTube*. URL: https://www.youtube.com/watch?v=XpBGwZNyUh0 (дата звернення: 16.05.2025).
14. Wondering what Unity is? Find out who we are, where we've been and where we're going. *Unity*. URL: https://unity.com/our-company (дата звернення: 16.05.2025).
15. Contributors to Wikimedia projects. Unreal Engine - Wikipedia. *Wikipedia, the free encyclopedia*. URL: https://en.wikipedia.org/wiki/Unreal_Engine (дата звернення: 16.05.2025).
16. About – blender.org. *blender.org*. URL: https://www.blender.org/about/ (date of access: 16.05.2025).
17. Rendering Techniques. *Uxcel*. URL: https://app.uxcel.com/courses/3d-design-foundations/d-rendering-techniques-903
18. Учасники проектів Вікімедіа. DirectX – вікіпедія. *Вікіпедія*. URL: https://uk.wikipedia.org/wiki/DirectX (дата звернення: 16.05.2025).
19. Home | Vulkan | Cross platform 3D Graphics. *Home | Vulkan | Cross platform 3D Graphics*. URL: https://www.vulkan.org/ (дата звернення: 16.05.2025).

20. Havok. *Havok*. URL: https://www.havok.com/about-havok/ (дата звернення: 16.05.2025).
21. PhysX SDK. *NVIDIA Developer*. URL: https://developer.nvidia.com/physx-sdk (дата звернення: 16.05.2025).
22. GitHub - misyltoad/vphysics-jolt: volt (vphysics jolt) is a replacement physics module for the source engine. *GitHub*. URL: https://github.com/misyltoad/VPhysics-Jolt (дата звернення: 16.05.2025).
23. GLFW documentation. *GLFW*. URL: https://www.glfw.org/documentation.html (дата звернення: 16.05.2025).
24. GitHub - g-truc/glm: OpenGL Mathematics (GLM). *GitHub*. URL: https://github.com/g-truc/glm (дата звернення: 16.05.2025).
25. GitHub - ocornut/imgui: Dear ImGui: Bloat-free Graphical User interface for C++ with minimal dependencies. *GitHub*. URL: https://github.com/ocornut/imgui (дата звернення: 16.05.2025).
26. GLEW: the opengl extension wrangler library. *GLEW: The OpenGL Extension Wrangler Library*. URL: https://glew.sourceforge.net/ (дата звернення: 16.05.2025).
27. Physics tutorial 2: numerical integration methods. *research.ncl.ac.uk; ; Newcastle University*. URL: https://research.ncl.ac.uk/game/mastersdegree/gametechnologies/previousinformation/physics2numericalintegrationmethods/2017%20Tutorial%202%20-%20Numerical%20Integration%20Methods.pdf (дата звернення: 16.05.2025).
28. 3D collision detection - Game development | MDN. *MDN Web Docs*. URL: https://developer.mozilla.org/en-US/docs/Games/Techniques/3D_collision_detection (дата звернення: 16.05.2025).
29. Hash table collision resolution. *Educative*. URL: https://www.educative.io/answers/hash-table-collision-resolution (дата звернення: 16.05.2025).

Додаток А. Опис програми

А.1 Загальні відомості

Позначення та найменування програми: прототип 3D ігрового рушія на основі імітаційних моделей твердих об'єктів з підтримкою фізичних властивостей.

Програмне забезпечення, необхідне для функціонування програми: комп’ютер з апаратною підтримкою OpenGL версії 3.3 і вище.

Мови програмування, на яких написана програма: програма написана у середовищі програмування Visual Studio на мові C++.

А.2 Функціональне призначення

Програма призначена для вивчення принципів роботи 3D рушіїв, імплементації алгоритмів рендерингу та фізичних рушіїв. А також можливість динамічної взаємодії зі сценою.

А.3 Структура програми з описом функцій складових частин і зв’язки між ними

1. Зміна фізичних властивостей об’єкта:
   Вхідні дані: пружність об’єкта, його маса.
   Вихідні дані: зміна взаємодії об’єкта зі сценою.
2. Прикладання сили до об’єкта:
   Вхідні дані: величина сили, прикладена до конкретної осі.
   Вихідні дані: реакція об’єкта на прикладену силу.
3. Зміна кольору об’єкта:
   Вхідні дані: новий колір об’єкта.
   Вихідні дані: зміна кольору об’єкта на сцені.
4. Зміна кольору заднього фону:
   Вхідні дані: новий колір заднього фону.
   Вихідні дані: зміна кольору заднього фону.
5. Зміна положення об’єкта на сцені:
   Вхідні дані: нові координати об’єкта на сцені.
   Вихідні дані: переміщення об’єкта та реакція інших об’єктів на нього.

6. Зміна налаштувань відображення сцени:

   Вхідні дані: тип відображення об’єктів (заповнений, лінії з’єднання), налаштування параметра вертикальної синхронізації.

   Вихідні дані: зміна вигляду сцени, відключення ліміту кадрів на секунду.

7. Зміна налаштувань симуляції фізики:

   Вхідні дані: параметри гравітації, стан перемикача симуляції фізики.

   Вихідні дані: відповідні зміни в симуляції на сцені.

А.4 Використовувані технічні засоби

Технічним засобом для запуску цієї програми може бути будь-який комп’ютер з операційною системою Windows та апаратною підтримкою OpenGL 3.3 або вище.

А.5 Виклик і завантаження: запуск програми відбувається напряму з компілятора Visual Studio. Для цього заздалегідь необхідно завантажити вихідний код та налаштувати підключення зовнішніх бібліотек якщо в проекті цього ще не зроблено.

Додаток Б. Керівництво користувача

Для запуску програми необхідно завантажити її вихідний код, відкрити його в середовищі Visual Studio та налаштувати підключення зовнішніх бібліотек якщо в проекті цього ще не зроблено.

Запуск програми здійснюється за допомогою функції відладки коду в Visual Studio.

Після запуску відкривається вікно програми з минулим розположенням елементів керування.

Вигляд вікна програми після запуску представлено на рисунку Б.1.

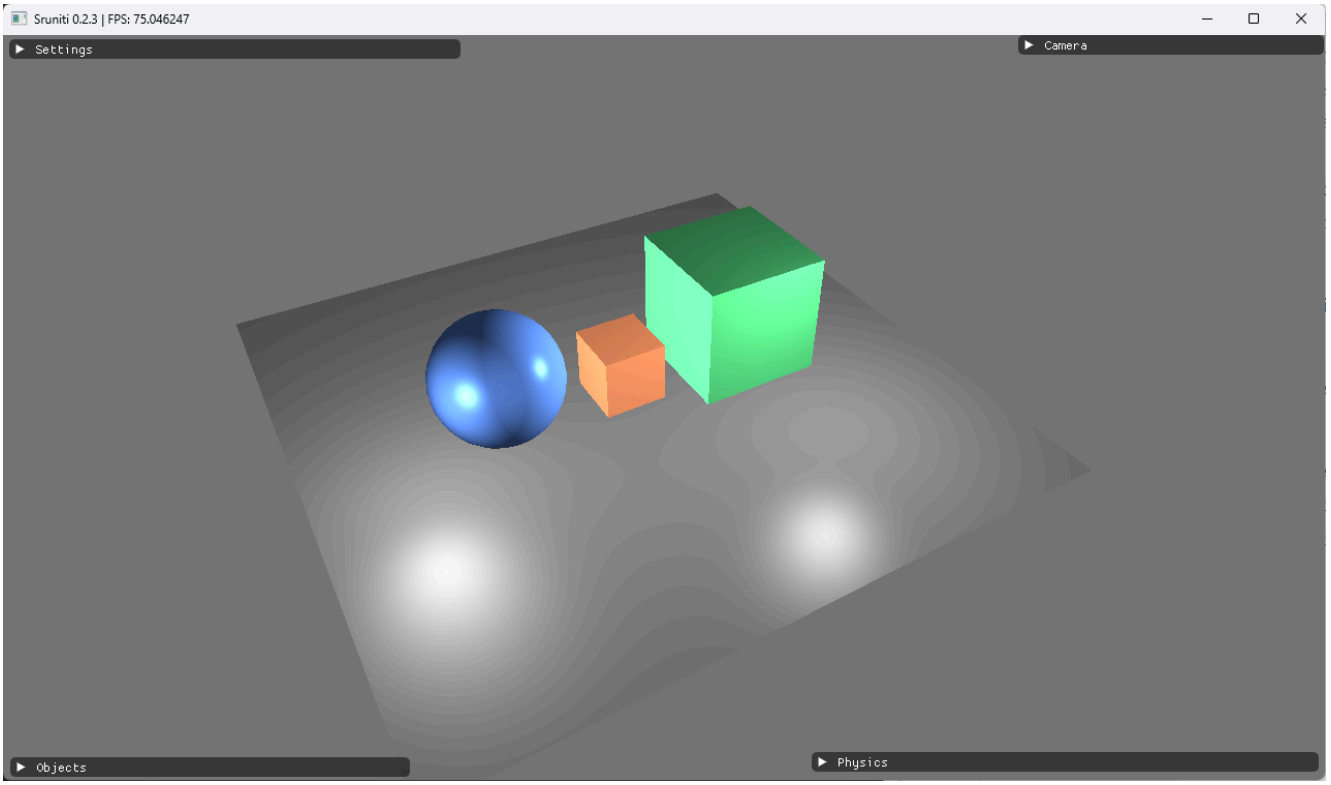

Рис. Б.1. Програма після запуску

Після натискання на вікно “Settings” воно розгортається і з'являються відповідні налаштування. Так само себе поводять інші вікна всередині програми. Результати натиснення на елементи меню вказані на відповідних рисунках.

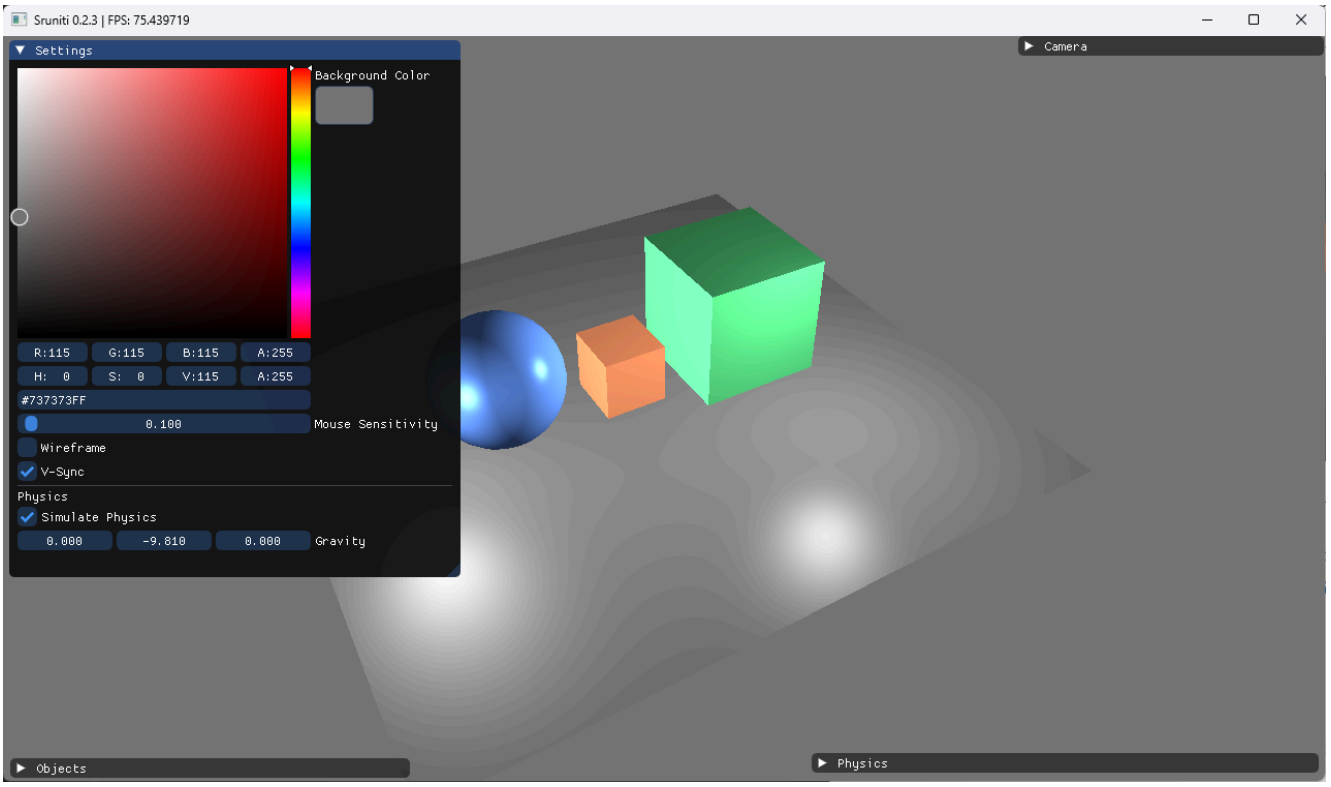

Рис. Б.2. вікно програми після відкриття вкладки “Settings”

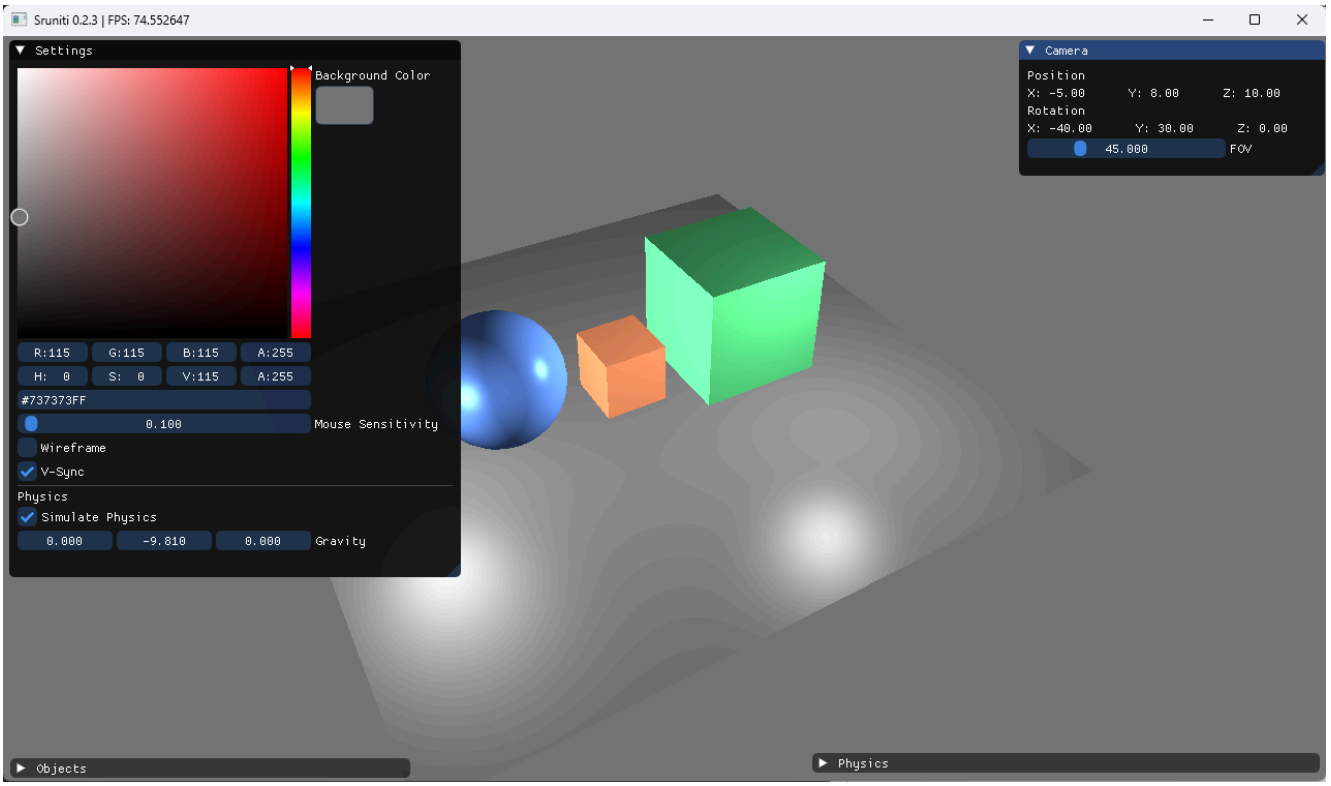

Рис. Б.3. Вікно програми після відкриття вкладки “Camera”

Рис. Б.4. Вікно програми після розкриття і переміщення вкладки “Objects”

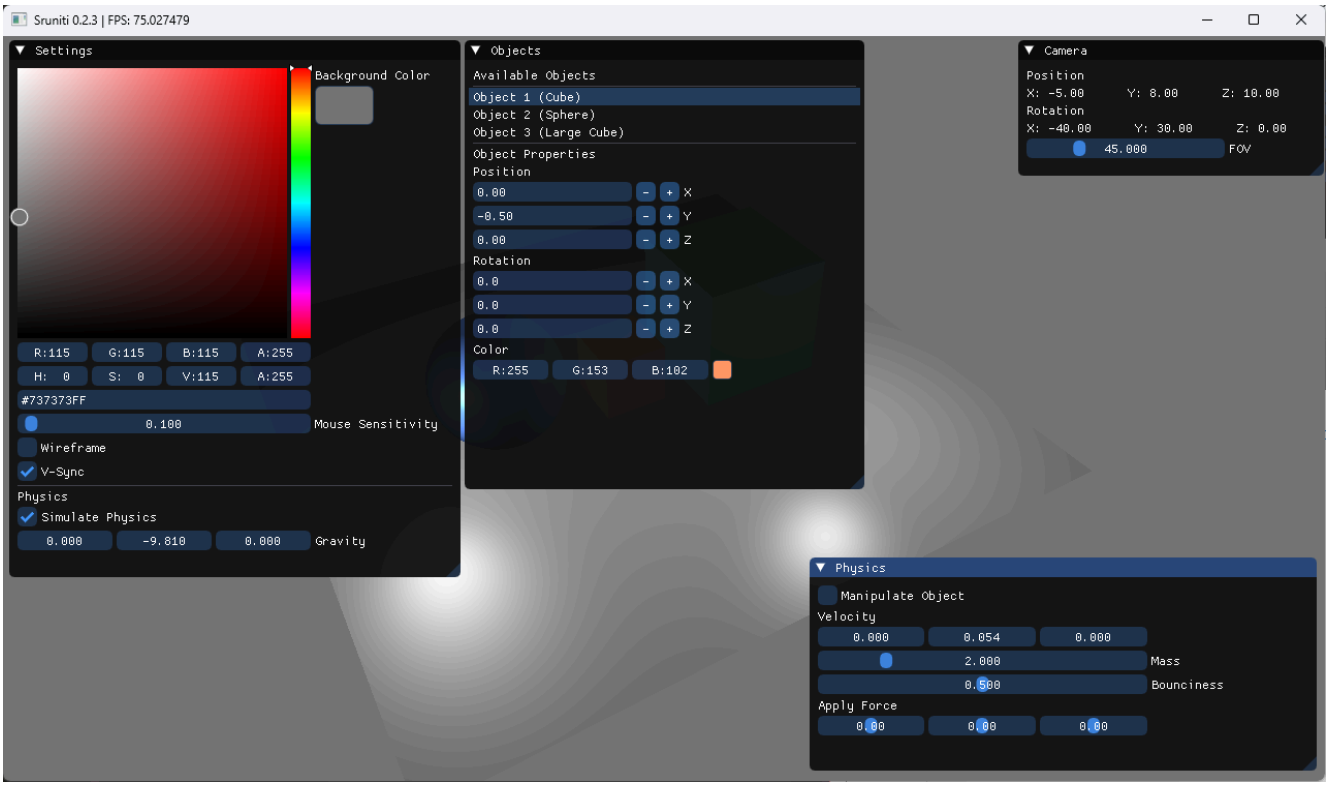

Рис. Б.5. Вікно програми після розкриття вкладки “Physics”

Для прикладення сили до об’єкта необхідно обрати об’єкт та потягнути відповідний слайдер з вектором прикладання сили. Система динамічно реагує на дії над об’єктами, тому результат можна буде одразу побачити.

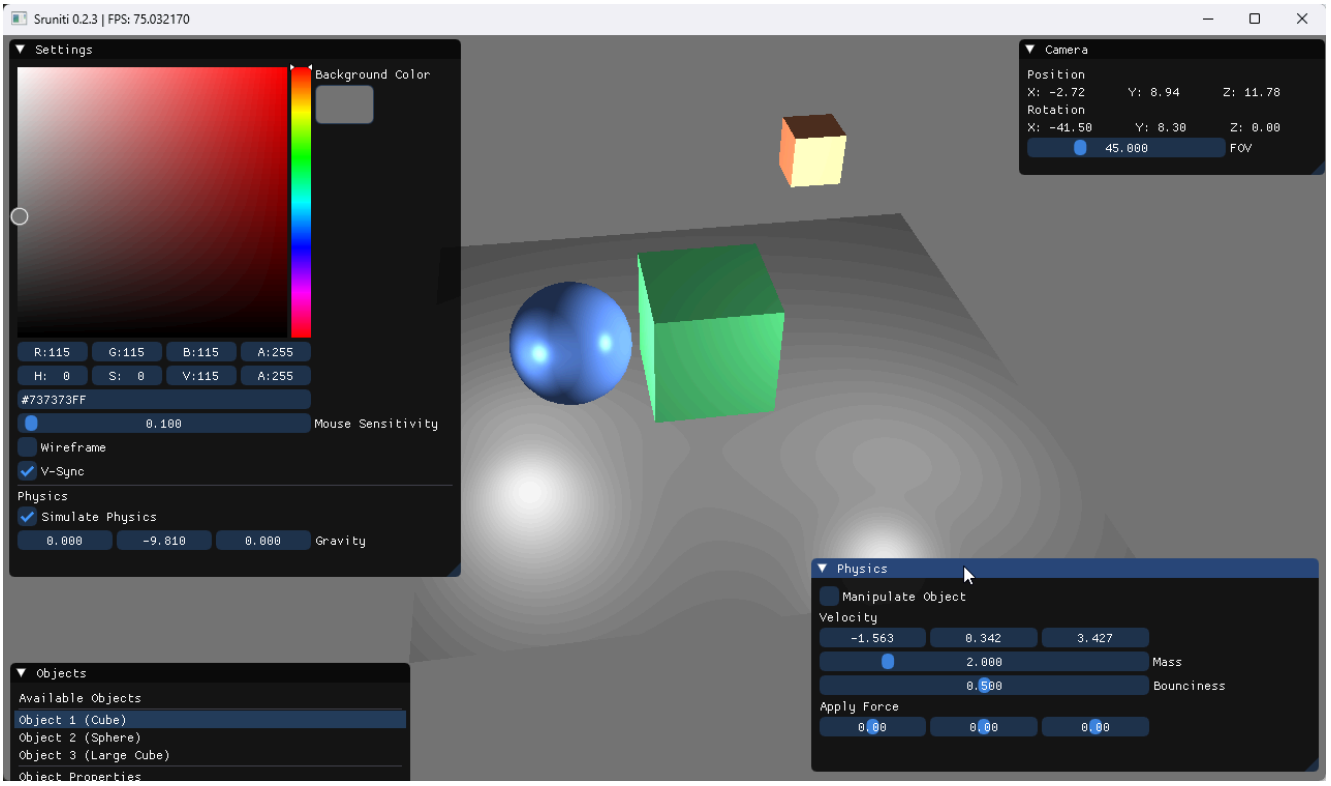

Рис. Б.6. Стан сцени після прикладання деяких сил до маленького куба.

Однією з функцій програми є зміна кольору заднього фону. Для цього необхідно перетягнути точку на кольоровій палітрі в вікні “Settings”. Результат можна буде побачити одразу.

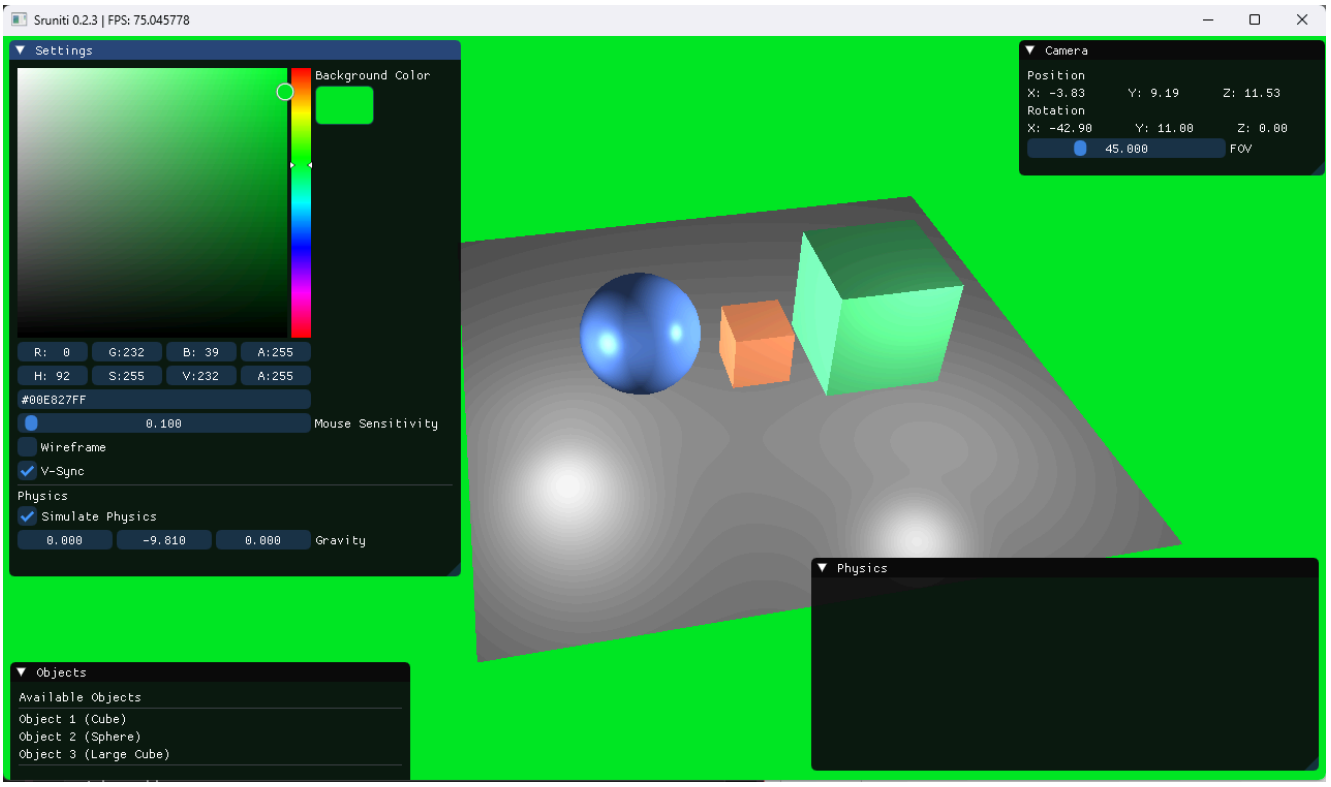

Рис. Б.7. Зміна кольору заднього фону

Також, як можна побачити з рисунка вище, вікно “Physics” може бути пустим. Щоб це виправити необхідно обрати один з доступних об’єктів в вікні “Objects”.

Додаток В. Фрагмент - код програми

Лістинг В.1. - Файл заголовка "AABB.h"

```
#pragma once
#include <iostream>
#include <glm.hpp>

class AABB
{
public:
	AABB(const glm::vec3& min, const glm::vec3& max)
	: m_Min(min), m_Max(max) {

	std::cout << "AABB created with min: " << min.x << " " << min.y << " " << 
min.z << " "
		<< " and max: " << max.x << " " << max.y << " " << max.z << " " << 
std::endl;
	}

	// Create an AABB from center and half-extents
	static AABB FromCenterAndExtents(const glm::vec3& center, const glm::vec3& 
halfExtents) {
	return AABB(center - halfExtents, center + halfExtents);
	}

	// Check if a point is inside the AABB
	bool Contains(const glm::vec3& point) const {
	return (point.x >= m_Min.x && point.x <= m_Max.x) &&
		(point.y >= m_Min.y && point.y <= m_Max.y) &&
		(point.z >= m_Min.z && point.z <= m_Max.z);
	}

	// Check if another AABB intersects with this one
	bool Intersects(const AABB& other) const {
	return (m_Min.x <= other.m_Max.x && m_Max.x >= other.m_Min.x) &&
		(m_Min.y <= other.m_Max.y && m_Max.y >= other.m_Min.y) &&
		(m_Min.z <= other.m_Max.z && m_Max.z >= other.m_Min.z);
	}
```

```
        // Getters
        const glm::vec3& GetMin() const { return m_Min; }
        const glm::vec3& GetMax() const { return m_Max; }
        glm::vec3 GetCenter() const { return (m_Min + m_Max) * 0.5f; }
        glm::vec3 GetExtents() const { return (m_Max - m_Min) * 0.5f; }

private:
        glm::vec3 m_Min;
        glm::vec3 m_Max;
};
```

Лістинг В.2. - Файл імплементації "Application.cpp"

```
#include "Application.h"

// File paths for shaders
const std::string vertexShader = "res/Shaders/vertex_core.glsl";
const std::string fragmentShader = "res/Shaders/fragment_core.glsl";

//Initial window size
const unsigned int WINDOW_WIDTH = 1280, WINDOW_HEIGHT = 720;

//Window title, color and UI settings variables
char windowTitle[] = "Sruniti 0.2.3";
float windowColor[4] = { 0.45, 0.45, 0.45, 1 };

float sliderValue = 12.4;
bool showMenu = true;
bool homeKeyPressed = false; // Tracks whether the Home key was previously pressed
bool showWireframe = false;
bool enableVSync = true;

float mouseSensitivity = 0.1f;

int selectedObjectIndex = -1; // -1 means no object selected
glm::vec3 objectPosition = glm::vec3(0.0f);
glm::vec3 objectRotation = glm::vec3(0.0f);
glm::vec3 objectColor = glm::vec3(1.0f, 1.0f, 1.0f);
bool objectSettingsChanged = false;
```

```
//Camera variables - keeping these as globals for ImGui access
glm::vec3 cameraRotation = { -40.f, 30.f, 0.f };
float cameraFov = 45.f;
glm::vec3 camPosition = { -5.f, 8.f, 10.f };

//Physics variables
PhysicsSystem physicsSystem;
std::vector<std::unique_ptr<MeshPhysics>> meshPhysics;
bool simulatePhysics = true;
float physicsTimeStep = 1.0f / 60.0f;
float accumulatedTime = 0.0f;
glm::vec3 gravity = glm::vec3(0.0f, -9.81f, 0.0f);

std::vector<std::unique_ptr<Mesh>> meshes;
//Main function
int main() {

	//Init GLFW
	if (!glfwInit()) {
	std::cout << "GLFW Failed to initialize!\n";
	glfwTerminate();
	return 1;
	}

	//Set GLFW options
	glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
	glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
	glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
	glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
	glfwWindowHint(GLFW_TRANSPARENT_FRAMEBUFFER, 1);
	glfwWindowHint(GLFW_RESIZABLE, GL_TRUE);

	//Create window
	GLFWwindow* window = glfwCreateWindow(WINDOW_WIDTH, WINDOW_HEIGHT,
windowTitle, NULL, NULL);

	//Check if window was created successfully
	if (!window) {
	std::cout << "Failed to create window :(\n";
	glfwTerminate();
```

```
	return 1;
	}

	//Create framebuffer variables and set resize callback
	int bufferWidth, bufferHeight;
	glfwSetFramebufferSizeCallback(window, framebuffer_resize_callback);


	//Set context for GLFW and make window current
	glfwMakeContextCurrent(window);

	//Init GLEW
	glewExperimental = GL_TRUE;

	//Check if GLEW is initialized
	if (glewInit() != GLEW_OK) {
	std::cout << "GLEW Failed to initialize!\n";
	glfwDestroyWindow(window);
	glfwTerminate();
	return 1;
	}
```

Лістинг В.8. - Файл заголовка "Collision.h"

```
#pragma once
#include "BoundingSphere.h"
#include "AABB.h"
#include <glm.hpp>
#include <memory>

// Forward declarations
class Mesh;

// Collision information structure
struct CollisionInfo {
	bool hasCollision = false;
	glm::vec3 collisionPoint = glm::vec3(0.0f);
	glm::vec3 collisionNormal = glm::vec3(0.0f);
	float penetrationDepth = 0.0f;
```

```
};

class Collision
{
public:
	// Sphere-Sphere collision detection
	static CollisionInfo TestSphereSphere(const BoundingSphere& sphereA, const
BoundingSphere& sphereB);

	// Sphere-AABB collision detection
	static CollisionInfo TestSphereAABB(const BoundingSphere& sphere, const
AABB& aabb);

	// Sphere-Plane collision detection (where plane is defined by a point and
normal)
	static CollisionInfo TestSpherePlane(const BoundingSphere& sphere, const
glm::vec3& planePoint, const glm::vec3& planeNormal);

	// AABB-AABB collision detection
	static CollisionInfo TestAABBAABB(const AABB& boxA, const AABB& boxB);

	// Helper functions for mesh collision detection
	static BoundingSphere CreateBoundingSphere(const Mesh& mesh);
	static AABB CreateAABB(const Mesh& mesh);
};
```

Лістинг В.9. - Файл імплементації "IndexBuffer.cpp"

```
#include "IndexBuffer.h"
#include <iostream>

IndexBuffer::IndexBuffer(IndexBuffer&& other) noexcept
	: m_RendererID(other.m_RendererID), m_Count(other.m_Count)
{
	other.m_RendererID = 0;
}

IndexBuffer::IndexBuffer(const unsigned int* data, unsigned int count)
	: m_Count(count)
{
```

```
        glGenBuffers(1, &m_RendererID);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, m_RendererID);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER, count * sizeof(unsigned int), data,
GL_STATIC_DRAW);

}
```