

Міністерство освіти і науки України
Харківський національний університет імені В.Н. Каразіна
Навчально-науковий інститут комп'ютерних наук та штучного інтелекту

Рекомендовано до захисту:
протокол засідання кафедри
№__від____2025 р.
Завідувач кафедри ІПСІТ

Олешко О.І.
(підпис) (прізвище та ініціали)

Пояснювальна записка

до кваліфікаційної роботи бакалавра

на тему Розробка програмних компонентів для кросплатформного ігрового рушія з елементами фізичної симуляції

Захищено на засіданні ЕК № _____
протокол № ____ від ____ . ____ .2025 р.
Оцінка ____ / _____
Голова ЕК

Фесенко Г.В.
(підпис) (прізвище та ініціали)

Виконав:
студент 4 курсу, групи КС- 42
Спеціальності:

122 – Комп'ютерні науки
(шифр і назва спеціальності підготовки)

Коханчук Юрій Сергійович 
(прізвище, ім'я та по батькові, підпис)

Керівник д.ф.-м.н., доцент Карась І. В.

(ступень, звання, прізвище та ініціали, підпис)

Рецензент к.е.н., доцент Чуб О.І.
(ступень, звання, прізвище та ініціали, підпис)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет імені В.Н.Каразіна

Навчально-науковий інститут комп'ютерних наук та штучного інтелекту

Кафедра інтелектуальних програмних систем і технологій

Рівень вищої освіти (освітньо-кваліфікаційний рівень) бакалавр

Спеціальність 122 – Комп'ютерні науки

ЗАТВЕРДЖУЮ

В.о. завідувача кафедри ІПСіТ

_____ Олег ОЛЕШКО
підпис ім'я, прізвище

“ _____ ” _____ 2025 року

**З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ**

Коханчук Юрій Сергійович
(прізвище, ім'я, по батькові студента)

1. Тема роботи: «Розробка програмних компонентів для кросплатформного ігрового рушія з елементами фізичної симуляції»

керівник роботи к.ф.-м.н., доцент Карась Ірина В'ячеславівна
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від “16” квітня 2025 року № 4101-5/962

2. Строк подання студентом роботи _____ 2 червня 2025 року _____

3. Перелік питань, які потрібно розробити:

Ознайомитися із алгоритмами та методами, які використовуються у комп'ютерній графіці, провести огляд існуючих рішень для симуляції поведінки абсолютно твердих тіл (АТТ). Познайомитися із рендерингом на фізичних основах. Реалізувати ігровий рушій, що підтримує рендеринг на фізичних основах, симуляцію АТТ та інші технології, властиві таким рішенням. Провести тестування створеного додатка та зробити висновки.

4. План роботи

№ з/п	Назви етапів роботи
1	Формулювання мети та повного переліку завдань для виконання кваліфікаційної роботи (КР).
2	Пошук та аналіз інформаційних джерел за темою КР.
3	Аналіз функціональності та особливостей існуючих програмних продуктів, подібних за тематикою.
4	Проектування архітектури програмного рушія для обробки тривимірної графіки в реальному часі.
5	Розробка математичних моделей для моделювання поведінки абсолютно твердих тіл у реальному часі.
6	Реалізація графічного рендерингу з використанням API OpenGL та шейдерних програм (GLSL).
7	Програмна реалізація рушія мовою Java з урахуванням модульності та розширюваності.
8	Розробка демонстраційного прикладу/додатку з використанням розробленого рушія.
9	Аналіз та оцінка ефективності розробленого рішення.
10	Підсумок отриманих результатів та оформлення пакету документів для процедури захисту КР в екзаменаційній комісії ННІ КН та ІІІ.

5. Дата видачі завдання 17.01.2025

Студент


(підпис)

Ю.С. Коханчук
(ініціали, прізвище)

Керівник роботи


(підпис)

І.В. Карась
(ініціали, прізвище)

АНОТАЦІЯ

Звіт з кваліфікаційної роботи бакалавра на тему «Розробка програмних компонентів для кросплатформного ігрового рушія з елементами фізичної симуляції» містить 88 сторінок, 60 рисунків, 38 джерел та 3 додатки.

Мета роботи: Дослідження алгоритмів комп'ютерної графіки та фізики. Розробка повноцінного 3D-рушія для ігор, здатного моделювати поведінку абсолютно твердих тіл.

Об'єкт дослідження: Алгоритми комп'ютерної графіки, методи моделювання фізичних тіл та архітектура програмного забезпечення для рушіїв 3D-ігор у реальному часі. Фізичний рендеринг, відкладений рендеринг.

Методи дослідження: Аналіз, порівняння, вивчення, емпіричне тестування та оцінка ефективності, тестування продуктивності.

Використані програмно-апаратні рішення: Java OpenJDK21, LWJGL 3.3.3, GLSL 460, OpenGL 4.6, JOML 1.10.6, ImGui 1.87.0, ApacheCommons, OpenCV, NumPy.

Результати дослідження: Розроблено рушій 3D-ігор, здатний до ефективного моделювання в контексті фізики твердого тіла. Проведено дослідження методів моделювання та комп'ютерної графіки. Рушій підтримує рендеринг на фізичних основах, з можливістю відкладеного рендерингу, використання карт тіней, карт оточення та користувацького освітлення.

Ключові слова: ФІЗИЧНЕ МОДЕЛЮВАННЯ, ОСВІТЛЕННЯ НА ОСНОВІ ЗОБРАЖЕНЬ, ФІЗИЧНИЙ РЕНДЕРИНГ, ОБРОБКА КОЛІЗІЙ, РОЗРОБКА ІГОР, 3D-ГРАФІКА, ШЕЙДЕРИ OPENGL, ЧИСЕЛЬНІ МЕТОДИ

ABSTRACT

Report on the qualification work at the bachelor's level on the topic “Developing program components of a cross-platform game engine with elements of physical simulation” contains 88 pages, 60 figures, 38 references, 3 appendices

Objective: Research the algorithms of computer graphics and computer physics, develop a full-fledged 3D game engine capable of numerical rigid body simulation.

Object of research: Computer graphics algorithms, methods of modelling physical bodies, and software architecture for real-time 3D game engines. Physical rendering, deferred rendering.

Research methods: Analysis, comparison, study, empirical testing and evaluation of efficiency, performance testing.

Software and hardware solutions used: Java OpenJDK21, LWJGL 3.3.3, GLSL 460, OpenGL 4.6, JOML 1.10.6, ImGui 1.87.0, ApacheCommons, OpenCV, NumPy.

Results: A 3D game engine capable of efficient modelling in the context of rigid body physics is developed. The simulation methods and computer graphics were studied. The engine supports physics-based rendering, featuring deferred rendering, shadow maps, environment maps, and custom lighting.

Keywords: PHYSICAL SIMULATION, IMAGE-BASED LIGHTING, PHYSICALLY-BASED RENDERING, COLLISION PROCESSING, GAME DEVELOPMENT, 3D GRAPHICS, OPENGL SHADERS, NUMERICAL METHODS

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ	6
ВСТУП	7
РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ 3D-ГРАФІКИ.....	9
1.1 Основи 2d та 3d графіки	9
1.2 Часто використовувані структури даних.....	10
1.3 Основи фізичного рендерингу	11
1.4 Джерела світла.....	15
1.5 Огляд існуючих продуктів	16
1.6 Стан розвитку у вітчизняній та закордонній науці	19
РОЗДІЛ 2. ТЕОРЕТИЧНІ ОСНОВИ СИМУЛЯЦІЙ ФІЗИЧНИХ ТІЛ	21
2.1 Загальні відомості про атт.....	21
2.1.1 Властивості атт.....	21
2.1.2 Виведення рівнянь руху	21
2.1.3 Значення інерції	23
2.1.4 Загальні сили	26
2.1.5 Інші сили.....	26
2.2 Обробка колізій	27
2.2.1 Примітиви колізій	27
2.2.2 Апроксимація складних об'єктів.....	29
2.2.3 Обчислення реакції на зіткнення	31
2.3 Симуляція хвиль.....	32
2.4 Огляд існуючих продуктів	33
2.5 Стан розвитку у вітчизняній та закордонній науці	34
РОЗДІЛ 3. РЕАЛІЗАЦІЯ ПОСТАВЛЕНИХ ЗАДАЧ.....	36
3.1 Комп'ютерна графіка.....	36
3.1.1 Налаштування проєкту	36
3.1.2 Використання ресурсів.....	36
3.1.3 Життєвий цикл додатка.....	44
3.1.4 Шейдери glsl.....	45

3.1.5 Впроваджені алгоритми	49
3.2 Симуляція фізики	56
3.3 Проєктування, архітектура та запуск додатка	61
3.3.1 Деплой артефактів	61
3.3.2 Фінальна розробка додатків.....	61
3.3.3 Проєктне планування	66
РОЗДІЛ 4. ТЕСТУВАННЯ ТА АНАЛІЗ	68
4.1 Метрики продуктивності.....	68
4.2 Результати тестів.....	69
4.3 Висновки тестування	74
4.4 Аналіз коду	74
ВИСНОВКИ.....	77
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ	78
ДОДАТОК А.....	82
ДОДАТОК Б	84
ДОДАТОК В.....	85

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

BRDF — Bidirectional reflectance distribution function; Функція, що описує відбите та поглинуте тілом світло.

CPU — Central Processing Unit

GLSL — OpenGL Shading Language

GPU — Graphics Processing Unit

IBL — Image Based Lighting; це вид освітлення, який дозволяє симулювати світлове випромінювання від оточення. Часто використовується у комбінації із HDR-картами.

JVM — Java Virtual Machine

MTL — Material Library

OBB — Oriented Bounding Box

OOD — Object Oriented Design

OOP — Object Oriented Programming

PBR — Physically Based Rendering; Рендеринг заснований на фізичних принципах. Набір парадигм, що роблять його реалістичним

RBD — Rigid Body Dynamics; Динаміка твердих тіл. Розділ фізики, який вивчає причини виникнення руху у АТТ.

VAO — Vertex Array Object; Це об'єкт який безпосередньо здійснює відображення моделі на екрані. Містить у собі певну кількість буферів VBO.

VBO — Vertex Buffer Object; Це абстракція об'єкта, що зберігає дані необхідні для рендерингу. Можуть бути: точки, нормалі, текстурні координати, тощо.

ВСТУП

Актуальність роботи. Запит на 3D-додатки в реальному часі все ще зростає. Існує безліч додатків, які потребують передових рішень у сфері 3D візуалізації та моделювання руху. У цьому контексті розробка графічних та фізичних рушіїв має велику актуальність. Процес розробки включає в себе розробку графіки та фізики, а також інфраструктуру – ввід-вивід, управління, звук та оптимізоване завантаження ресурсів. Такий уніфікований програмний пакет вирішив би проблему надмірно складних програмних продуктів, які стають результатом розробки без використання фреймворків. Використовуючи кросплатформні технології та поєднуючи їх із сучасними графічними стандартами, ми отримаємо потужне рішення, здатне працювати на різних платформах, використовувати різні типи вводу, фізичний рендеринг, ефекти пост-обробки, відображення тіней та інші технології.

Такий проект складається з великої кількості модулів, що сприяє складності реалізації. Вибір правильного технологічного стеку, правильна реалізація та оптимізація рішень ще більше ускладнюють це завдання.

Одночасне використання фізики та графіки є досить поширеним наприклад, у комп'ютерних іграх, військових симуляторах, симуляторах водіння та медичних візуалізаціях. Такі системи завдячують своїм існуванням графіці та симуляції.

Об'єктом дослідження є Алгоритми комп'ютерної графіки, методи імітаційного моделювання та архітектура програмного забезпечення для рушіїв 3D-ігор у реальному часі. Фізичний рендеринг, відкладений рендеринг.

Мета дослідження: Дослідження алгоритмів комп'ютерної графіки та розробка повноцінного 3D-рушія для ігор, здатного моделювати поведінку АТТ.

Практичне значення полягає в тому, що методи та алгоритми комп'ютерної графіки та фізики використовуються щоденно для абсолютно

різних задач симуляції. Вони забезпечують візуалізацію, моделювання поведінки об'єктів та взаємодії з оточенням. Такі технології застосовуються не лише в іграх, а й у віртуальній та доповненій реальності, симуляторах, навчальному ПЗ та медичних розробках. Завдяки їм можливе створення інтерактивних і точних моделей реального світу. Це дозволяє створювати просунуте ПЗ та вирішувати проблеми у багатьох сферах життя.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ 3D-ГРАФІКИ

1.1 Основи 2d та 3d графіки

Щоб розглянути цю тему, нам доведеться вивчити основні елементи будь-якої програми, що використовує 3D-графіку. Після закінчення будь-якого графічного конвеєра отримується фреймбуфер – растрове зображення з наперед визначеними висотою та шириною.

Такий конвеєр зазвичай складається з етапів обробки вершин і фрагментів. Наприклад, у OpenGL (2.1+) використовуються так звані шейдерні програми. На етапі обробки вершин використовуються різні позиції, вектори та нормалі для подальшого рендерингу. Етап фрагментів відповідає за обробку растрованих даних між вершинами.

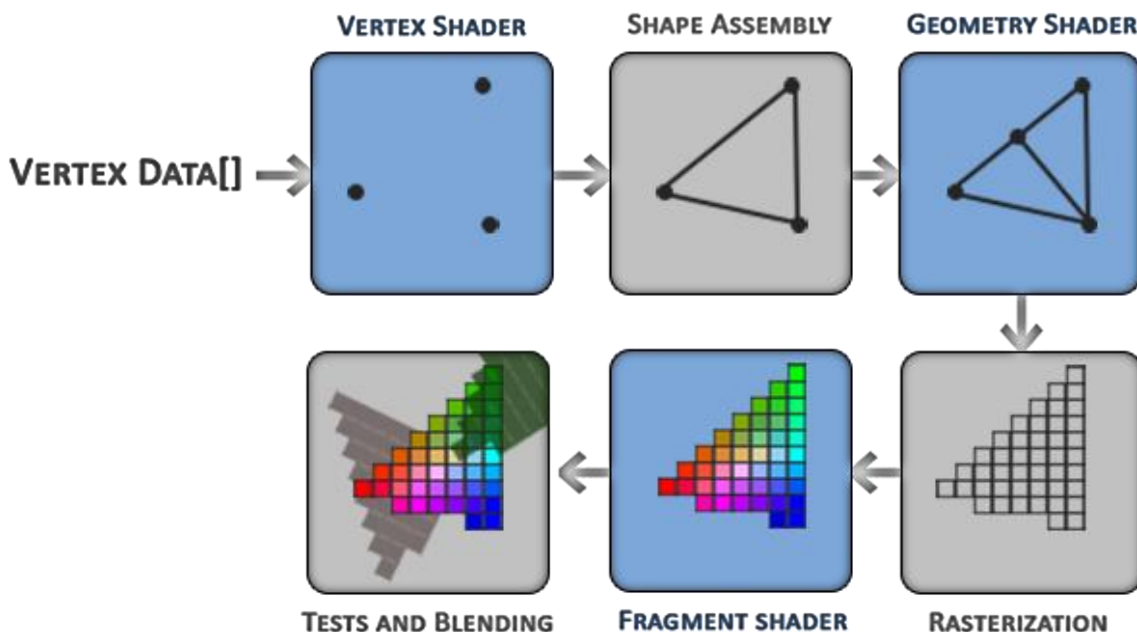


Рисунок 1.1 – Графічний конвеєр OpenGL [31]

Вершинний шейдер часто передбачає множення матриць. Шейдери пишуться мовами DSL, такими як GLSL, HLSL, MSL тощо. Більшість з них нагадують мову програмування C, тому проблем з синтаксисом не повинно виникнути.

Будь-який графічний процесор можна розглядати як спеціалізовану багатопроцесорну систему. З моменту появи уніфікованої моделі шейдерів їхні чіпи мають лише уніфіковані шейдерні ядра. Кожен процесор виконує окрему інструкцію, яка де-факто визначається його номером. Ядро графічного процесора обчислює власну вершину на етапі вершин або піксель на етапі фрагментів. Крім того, існують деякі відмінності, такі як ROP, TMU або VRAM.

1.2 Часто використовувані структури даних

Світ комп'ютерної графіки має свій власний набір традиційних структур даних, які використовуються майже в кожній програмі. Це вектор, матриця та кватерніон.

Вектори – це алгебраїчні об'єкти в просторі R^n , тобто вони складаються з n елементів. Зазвичай, типовий вектор є 2-, 3- або 4-вимірним. 2D-вектори представляють координати текстури для растеризації та вибірки текстури, а 3D-вектори представляють позиції.

Матриці – це об'єкти у просторі R^{n^2} , що передають трансформації. Вони можуть легко виконувати обертання, масштабування, переклад або навіть деформацію, наприклад, скіс.

Кватерніон – це чотиривимірний алгебраїчний об'єкт, який розширює поняття комплексних чисел

$$q = a + bi + cj + dk \quad (1.1)$$

де a – скаляр (дійсне число)

b , c та d – скаляри (дійсні числа), що представляють коефіцієнти уявних одиниць i , j та k

Уявні одиниці i , j та k мають наступні властивості:

$$\bullet \quad i^2 = j^2 = k^2 = -1$$

- $ij = k, jk = i, ki = j$
- $ji = -k, kj = -i, ik = -j$

Множення кватерніонів не є комутативним, тобто порядок множення має значення. Вони в основному використовуються для 3-вимірних обертань. Для обертання навколо довільної осі використовується така формула

$$q = \frac{\cos\theta}{2} + \frac{\sin\theta}{2} \cdot n = \left(\frac{\cos\theta}{2}, \frac{\sin\theta}{2} \cdot n_x, \frac{\sin\theta}{2} \cdot n_y, \frac{\sin\theta}{2} \cdot n_z\right) \quad (1.2)$$

Після того, як було здійснено обертання, кватерніон потрібно нормалізувати.

Не існує єдиного стандарту для структури даних 3D-моделі. Але я вирішив використовувати К-арні дерева. Використання дерев дозволяє «спускати» трансформації, матеріали або будь-які інші властивості, які підпорядковані «головному вузлу» – кореню дерева або вузлу вище. Крім того, воно має швидкі операції пошуку, вставки, видалення, навіть якщо не є повністю збалансованим.

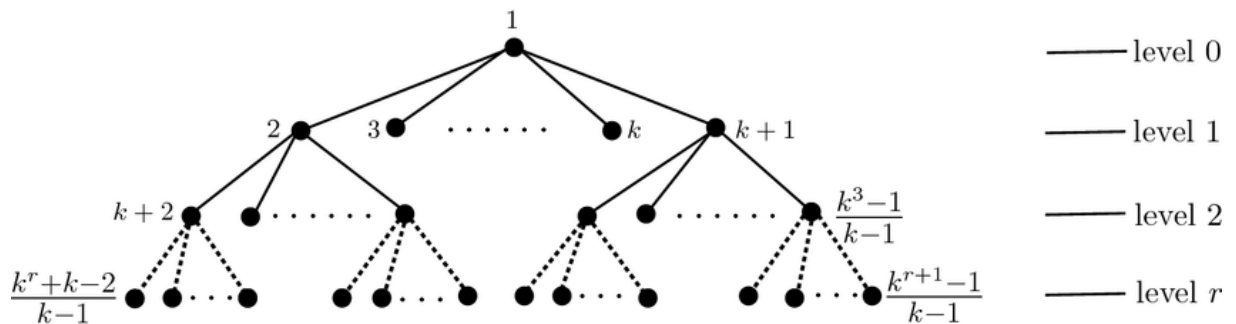


Рисунок 1.2 – Н-арне дерево [26]

1.3 Основи фізичного рендерингу

Фізично заснований рендеринг (PBR) – це підхід до комп'ютерної графіки, який спрямований на рендеринг зображень у спосіб, що моделює світло та поверхні з реалістичними оптичними властивостями. Його часто називають «Освітленням на фізичних засадах» або «Затіненням на фізичних

засадах». Ціль такого підходу – забезпечити фотореалізм. Здійсненні та швидкі наближення функції розподілу двонаправленого відбиття та рівняння рендерингу мають вирішальне значення. Принципи PBR можуть бути реалізовані з використанням шейдерів або трасування променів, чи інших стохастичних підходів [3].

PBR-матеріал – це набір текстур (карт) і значень, таких як шершавість, непрозорість і металічність. Матеріали визначають властивості поверхні 3D-об'єктів і те, як вони взаємодіють зі світлом. Вони часто включають такі карти:

Таблиця 1.1 – Карти відображення у відкладених графічних конвеєрах

Назва карти (map)	Використання
Albedo (Базовий колір)	Визначає базовий колір об'єкта
Карта нормалей	Визначає нормальні вектори для кожного пікселя
Карта жорсткості	Визначає значення нерівності для кожного пікселя.
Карта металічності	Визначає значення металічності для кожного пікселя.
Карта випромінювання	Впливає на те, яка частина об'єкта виділяє світло, може використовуватися для просунутих ефектів.
Карта висоти	Визначає висоту поверхні, що використовується для ефектів паралакса або тесселяції.
Карта яскравості	Подібний до жорсткості, але перевернутий; контролює блиск поверхні.
Карта прозорості	Керує прозорістю або непрозорістю матеріалу.



Рисунок 1.3 — Ілюстраційне зображення матеріала [32]

Основне рівняння рендерингу [16]

$$L_0(p, \vec{\omega}_0) = \int_{\Omega} f_r(p, \vec{\omega}_i, \vec{\omega}_0) L_i(p, \vec{\omega}_i) * (\vec{n} \cdot \vec{\omega}_i) d\vec{\omega}_i$$

де $\vec{n}$ – нормальний вектор;

$\vec{\omega}_0$ – напрямок камери;

$\vec{\omega}_i$ – напрямок світла;

$(\vec{n} \cdot \vec{\omega}_i)$ – Косинусний множник Ламберта, визначає кількість світла, що падає залежно від кута;

$L_0(p, \vec{\omega}_0)$ – Яскравість потоку світла у точці p , що має напрямок ω_0 (камера), Вт/(м² * ср);

$f_r(p, \vec{\omega}_i, \vec{\omega}_0)$ – BRDF, Складає розсіяне та відбите світло;

$L_i(p, \vec{\omega}_i)$ – Яскравість потоку світла, що приходить з напрямку ω_i , Вт/(м² * ср).

Це рівняння обчислює загальну яскравість потоку світла від точки на поверхні шляхом підсумовування внесків з усіх вхідних напрямків. Оскільки

цей метод вимагає трасування променів, він був фактично неможливим для рендеринга в реальному часі в більшості випадків. Тому програмісти придумали техніку для наближення цього рівняння [16].

$$f_r = \Sigma(f_{diffuse}, f_{specular}) \quad (1.4)$$

$$f_{diffuse} = \frac{1}{\pi} \begin{pmatrix} r \\ g \\ b \end{pmatrix}_{albedo} \quad (1.5)$$

$$f_{specular} = \frac{D(\vec{h})F(\vec{w}_0, \vec{h})G(\vec{w}_l, \vec{w}_0)}{4(\vec{n} \cdot \vec{w}_l)(\vec{n} \cdot \vec{w}_0)} \quad (1.6)$$

де $D(\vec{h})$ – Функція розподілу нормальних векторів. Описує те, як модель «мікрограней» розсіює світло по об'єкту.

$F(\vec{w}_0, \vec{h})$ – Множник Френеля. Описує те, яка кількість світла відбивається від об'єкта залежно від серединного вектора та напрямку світла.

$G(\vec{w}_0, \vec{w}_l)$ – Геометричний множник. По суті доповнює функцію нормального розподілу, додаючи мікро тіні чи засвіти на об'єкті

$$\vec{h} = \frac{\vec{w}_l + \vec{w}_0}{\|\vec{w}_l + \vec{w}_0\|} \quad (1.7)$$

Серединний вектор між $\vec{w}_l$ та $\vec{w}_0$, що позначається літерою $\vec{h}$.

Яскравість вихідного потоку світла можна наближено обчислити за наступною формулою:

$$L_0(p, \vec{w}_0) = \sum_{i=0}^n \left[L_i \left(\frac{1}{\pi} \begin{pmatrix} r \\ g \\ b \end{pmatrix}_{albedo} + \frac{D(\vec{h})F(\vec{w}_0, \vec{h})G(\vec{w}_l, \vec{w}_0)}{4 * (\vec{n} \cdot \vec{w}_0)} \right) (\vec{n} \cdot \vec{w}_l) \right] \quad (1.8)$$

Обчислення множника Френеля

$$F(w_i, h) = F_0 + (1 - F_0) * (1 - \cos \theta)^5 \quad (1.9)$$

$$\cos \theta = \|\vec{h} \times \vec{w}_0\| \quad (1.10)$$

Цей множник впливає на обробку світлових віддзеркалень.

Функція Шліка – це функція, яка використовується для опису розподілу на основі нормалей, серединного вектора та жорсткості.

$$D_{GGX}(n, h, \alpha) = \frac{\alpha^2}{\pi((n \cdot h)^2(\alpha^2 - 1) + 1)} \quad (1.11)$$

де α – нормалізоване значення жорсткості, безрозмірна.

Геометрична функція Сміта

$$G_{Smith}(\vec{n}, \vec{w}_0, L, \alpha) = D_{GGX}(\vec{n}, \vec{w}_0, \alpha) \times D_{GGX}(\vec{n}, L, \alpha) \quad (1.12)$$

Це рівняння є ключовим поняттям у багатьох алгоритмах рендерингу, таких як трасування променів, трасування шляхів, які використовуються для створення реалістичних зображень і симуляцій.

1.4 Джерела світла

Існують різні види освітлення, які в різних випадках більше підходять для реалізації.

- Точкове світло (Point Light)
- Спрямоване світло (Directional Light)
- Прожекторне світло (Spotlight Light)

- Площинне світло (Area Light)
- HDRI світло (HDRI Light)
- Розсіяне світло (Ambient Light)

У моїй моделі освітлення використовується спрямоване та точкове світло.

Спрямоване світло – це тип світла, який має колір, інтенсивність і напрямок. На нього не впливає згасання.

Точкове світло – це тип світла, яке має колір, інтенсивність і спрямованість. На нього впливає згасання.

Фізично коректний коефіцієнт згасання обчислюється таким чином:

$$I = \frac{I_0}{d^2} \quad (1.13)$$

де I – Яскравість світлового потоку, Вт · м – 2 · ср – 1;

d – відстань, м.

Але вона має ряд недоліків у нашому випадку. Тому я використовував наступну формулу [30]:

$$I = I_0 * \frac{(1-s^2)^2}{1+Fs^2} \quad (1.14)$$

де $s = d/R$, безрозмірна;

F – Відстань, після якої світло більше не впливає на об'єкти, м.

1.5 Огляд існуючих продуктів

Переводячи розмову про технічні деталі на вищий рівень абстракції, хотілося би сказати про комерційні продукти, що користуються приблизно такими самими рішеннями і мають прибуток зі свого розповсюдження.

Ринок ігрових рушіїв представлений великою кількістю продуктів різних штибів. Їх можна умовно поділити на кілька основних категорій.

Перша категорія – це безкоштовні рушії із відкритим кодом. До таких відносяться Godot Engine, Stride, Ogre3D. Вони дозволяють створювати повноцінні проекти і дають розробникам повний контроль над кодом. Зазвичай ці рушії поступаються комерційним аналогам, бо не мають теселяції, DLSS або трасування променів. Також можливості інтеграції складних систем фізики, анімації або штучного інтелекту можуть бути обмежені або вимагати сторонніх рішень [34].

Друга категорія – це платні рушії або рушії із змішаною ліцензією. Це насамперед такі продукти, як Unreal Engine, Unity3D, CryEngine. Вони пропонують широкий набір інструментів для роботи з фізикою, анімацією, штучним інтелектом та багато іншого [34]. Такі продукти вимагають оформлення підписки, що варто враховувати при виборі засобів та інструментів розробки.

Окрім того, існують спеціалізовані рушії, орієнтовані на створення специфічних продуктів (наприклад, RPG Maker або Ren'Py), а також пропрієтарні рушії, які великі компанії розробляють для власних потреб [34].

Таким чином, вибір рушія залежить від цілей проекту, необхідного рівня технологій та виділеного бюджету.

Таблиця 1.1 – Різні категорії рушіїв [34, 35, 36, 37, 38]

Категорія	Приклади	Ліцензія	Підтримка сучасної графіки
Безкоштовні з відкритим кодом	Godot Engine, Stride, Ogre3D	Безкоштовна, відкритий код	Обмежена або відсутня
Платні або змішані ліцензії	Unreal Engine, Unity3D, CryEngine	Платна або роялті	Повна підтримка

Закінчення таблиці 1.1

Категорія	Приклади	Ліцензія	Підтримка сучасної графіки
Спеціалізовані рушії	RPG Maker, Ren'Py, Phaser	Зазвичай платні/безкоштовні	Немає (обмежено жанром)
Пропрієтарні рушії	RE Engine, RAGE, IW Engine	Приватна, внутрішнє використання	Найвища, кастомна

Таблиця 1.2 – Переваги різних категорій рушіїв [34, 35, 36, 37, 38]

Категорія	Основні переваги	Основні недоліки
Безкоштовні рушії	Вільний доступ до коду, безкоштовність	Менше фіч для AAA-ігор
Платні рушії	Потужний інструментарій, підтримка спільноти	Вартість ліцензії або роялті
Спеціалізовані рушії	Дуже швидка розробка у своїй ніші	Вузька сфера застосування
Пропрієтарні рушії	Максимальна оптимізація під продукт	Недоступні для сторонніх розробників

Пропрієтарні рушії можуть бути створені у явному виді (The Witcher 3, Half-Life 2) або неявному виді (Minecraft, Tanki Online). Також варто додати, що додатковою специфікою є платформа розробки. Це можуть бути мобільні пристрої, ігрові консолі, персональні комп'ютери; Платформа безпосередньо впливає на перелік доступних технологій.

Деякі просунуті рушії намагаються відійти від написання коду зовсім. Інші ж використовуються мови вищого рівня, лишаючи ядро на C++. У випадку використання таких важких програм іноді краще відділити «ядро» проекту на мові C++ архітектурно від рішення. І вже потім підключати його до інших мов програмування. Таке рішення дозволить полегшити управління пам'яттю та спростити написання програмного коду більш високорівневими елементами, як функціональне програмування або просунуте ООП.

1.6 Стан розвитку у вітчизняній та закордонній науці

У вітчизняній науці фактично відсутній розробки такого класу через надзвичайно велику складність розробки та відсутність необхідної бази технічних напрацювань та знань. Наукові з теми 3-вимірної графіки майже відсутні українською, а трофейною мовою – здебільшого перепечатки англomовних статей.

Виходячи із цього, я користувався джерелами французькою, німецькою та англійською мовами.

- David Lettier. 3D Game Shaders for Beginners [1]

Дає початкові знання зі створення шейдерів у 3D-іграх, включаючи основи освітлення, текстурних карт та пост-обробки.

- Khronos Group. GLSL Specification [2]

Офіційна документація по мові шейдерів GLSL. Окрім визначення синтаксису та семантики мови, на сайті є форум для обговорення проблем і можна дуже детально ознайомитися з вирішеннями деяких проблем

- LearnOpenGL. Advanced Lighting Techniques [11], [31]

Пропонує практичні уроки зі складних технік освітлення, зокрема ефект bloom. Також містить вступ до OpenGL на прикладі кольорового трикутника.

- NVIDIA GPU Gems. Efficient Soft-Edged Shadows Using Pixel Shader Branching [18]

Описує техніку реалізації м'яких тіней із використанням рандомізованого семплування в піксельних шейдерах, що підвищує якість графіки без великої втрати продуктивності. Це також зменшує витрати необхідні для зберігання карти тіней.

- Lisyrus. Point Light Attenuation [30]

Стаття присвячена математичній моделі загасання світла від точкових джерел. Включає формули та приклади реалізації. Вона також надає нову, введену автором формулу атенюації, яку він вивів емпіричним шляхом.

- Wikipedia. Physically Based Rendering (PBR) [3]

Узагальнення принципів PBR — методу рендерингу, який враховує фізичні властивості матеріалів і світла для досягнення фотореалізму. Також описуються його недоліки та переваги.

- Disney. Physically Based Shading at Disney [5]

Один із найбільш впливових документів у сфері PBR. Містить математичні моделі BRDF, які використовуються у студії Disney для шейдингу.

- Stack Overflow. SkyBox Unwrapping (Panorama → CubeMap) [19]

Обговорення методів конвертації панорам у кубічні карти (skybox).

- Blockade Labs. AI Skybox Generation [12]

Онлайн-інструмент для генерації skybox-ів за допомогою ШІ — корисний у прототипуванні або як джерело HDR-карт.

- Song Ho. Projection Matrices [29]

Описує методи функціонування проекцій в OpenGL (ортографічний та перспективний), що критично для правильного рендерингу сцени.

- Philipps-Marburg Universität. Graphics and Multimedia Programming Lectures [16]

Академічний курс, присвячений графічному програмуванню, охоплює теорію та практику OpenGL та мультимедійних застосунків.

РОЗДІЛ 2. ТЕОРЕТИЧНІ ОСНОВИ СИМУЛЯЦІЙ ФІЗИЧНИХ ТІЛ

2.1 Загальні відомості про атт

2.1.1 Властивості атт

Найважливішим тут є поняття абсолютно твердого (жорсткого) тіла. Це модель, яка використовується для представлення тіла: з набором точок $\{P_1, P_2 \dots P_{n-1}, P_n\}$ і центром тіла P_0 . Відстань від P_i до P_0 залишається незмінною незалежно від того, які дії відбуваються з тілом. Наприклад, лінійний рух, кутовий рух, зіткнення з іншими тілами тощо.

Як згадувалося раніше, для АТТ визначено два типи руху.

Лінійний рух являє собою простий рух від P_j до P_{j+1} . Він представлений такими величинами: швидкість $\vec{v}$, радіус-вектор $\vec{r}$, прискорення $\vec{a}$. Також іноді додаються і інші похідні від радіус-вектора, як, наприклад, ривок, але у моїй моделі прискорення вважається дискретною величиною.

Кутовий рух набагато складніший. Він представлений кутом ϕ та кутовою швидкістю ω_0 . Для ефективної роботи з цим типом руху можуть бути корисними попередні знання про кватерніони.

2.1.2 Виведення рівнянь руху

Багато задач для симуляції (імітаційне моделювання) можна розглянути як задачу Коші (Initial value problem) – розв’язок диференційного рівняння із відомими початковими константами. У випадку із поступальним рухом це рівняння виглядає наступним чином:

$\vec{v}_0, \vec{a}, \vec{S}_0$ – Відомі значення

$$F = m \frac{d^2 \vec{S}}{dt^2} \longrightarrow \vec{a} = \frac{d^2 \vec{S}}{dt^2} \quad (2.1)$$

$$\frac{d\vec{S}}{dt} = \int \vec{a} dt + \vec{v}_0 \quad (2.2)$$

$$\vec{S} = \int [\int \vec{a} dt + \vec{v}_0] dt + \vec{S}_0 \quad (2.3)$$

Таким чином ми отримаємо значення u , $\dot{u}$ та $\ddot{u}$, залежно від початкових значень. У цьому випадку ми користуємося аналітичним розв'язком, який часто називається Verlet Integration (метод Стермера-Верле) у відповідних джерелах. Може здатися, що це універсальний розв'язок, проте якщо буде складна симуляція із рівняннями в абсолютних змінних, як наприклад, система пружин, то система може неточно відпрацювати. Особливо сильно погіршується ситуація, якщо в нас нестабільний або стабільно великий крок інтегрування. У такому випадку цей метод інтегрування починає більше нагадувати квадратичну апроксимацію. Тому у більш складних випадках використовуються покращені методи інтегрування або фіксований час інтегрування із іншим потоком.

$$\vec{M} = I \frac{d^2 q}{dt^2} \quad (2.4)$$

$$I^{-1} \vec{M} = \frac{d^2 q}{dt^2} = \vec{\beta} \quad (2.5)$$

$$\frac{dq}{dt} = \int \vec{\beta} dt + \vec{w}_0 \quad (2.6)$$

$$q = q_0 * \langle \frac{1}{2} \int [\int \vec{\beta} dt + \vec{w}_0] dt, 1 \rangle \quad (2.7)$$

де $q_1 = \langle \frac{1}{2} \int [\int \vec{\beta} dt + \vec{w}_0] dt, 1 \rangle$ це кватерніон з кутовою швидкістю хузу та одиничною компонентою w . Його потрібно помножити, оскільки операція додавання визначена як множення для кватерніонів.

$$\begin{cases} \overrightarrow{S(t)} = \overrightarrow{S_0} + \overrightarrow{v_0}t + \frac{\vec{a}t^2}{2} \\ \overrightarrow{v(t)} = \overrightarrow{v_0} + \vec{a}t \\ \vec{a} = \frac{\vec{F}}{m} \end{cases} \quad (2.8)$$

$$\begin{cases} q(t) = q_0 * \left(\overrightarrow{\omega_0} * \frac{t}{2} \right) \\ \overrightarrow{\omega(t)} = \overrightarrow{\omega_0} + \vec{\beta}t \\ \vec{\beta} = I^{-1} * \overrightarrow{M_{diff}} \end{cases} \quad (2.9)$$

$$\overrightarrow{M_{diff}} = \vec{M} - (\overrightarrow{\omega_0} \times (I \cdot \overrightarrow{\omega_0})) \quad (2.10)$$

2.1.3 Значення інерції

У моїй програмі тензор інерції апроксимується кубоїдами, що є прийнятним для 3D ігор. Можна використовувати інші примітиви або використовувати обхід безпосередньо тіла. Але я вважаю, що це не має значення для мого випадку.

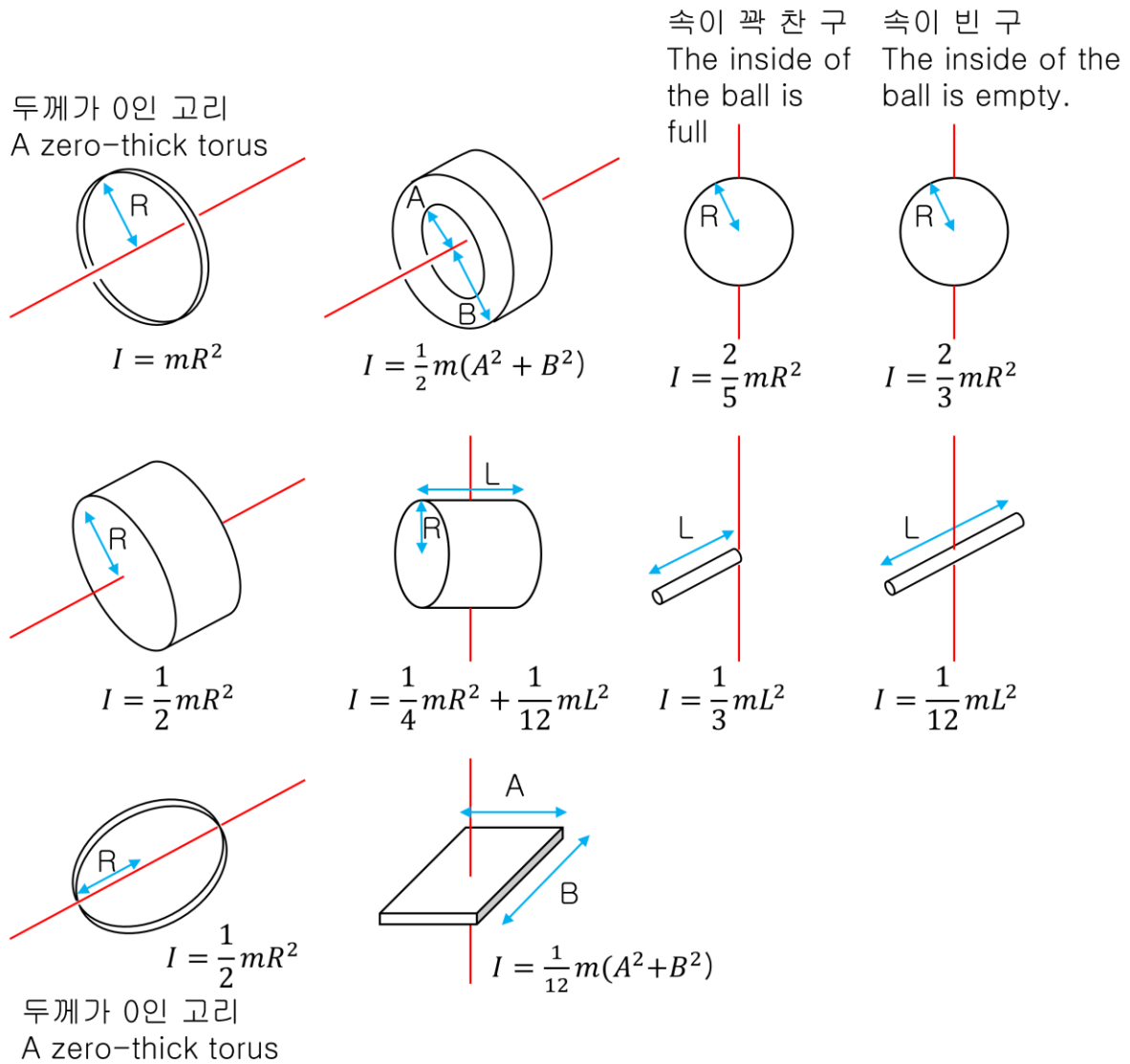


Рисунок 2.1 – Різні тіла та їх інерції [22]

Для того, щоб обчислити тензор інерції, нам потрібна теорема про паралельні осі, також відома як теорема Гюгенса-Штайнера, опорна точка і формула для самого тензора. Точка відліку обчислюється таким чином

$$P_0 = \frac{\sum m \vec{r}}{\sum m} \quad (2.11)$$

де m – маса, кг;

$\vec{r}$ – радіус-вектор.

Теорема Гюйгенса-Штайнера допомагає нам обчислити вектори інерції тіл, які мають зміщення від головної осі.

$$I_s = I_R + md^2 \quad (2.12)$$

де I_s – кінцеве значення інерції, кг·м²;

I_R – інерція об'єкта, кг·м²;

d^2 – квадратична відстань, м².

Тензор інерції обраховується наступним чином:

$$I = \begin{bmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{xy} & I_{yy} & -I_{yz} \\ -I_{xz} & -I_{yz} & I_{zz} \end{bmatrix} \quad (2.13)$$

де I_{xx} , I_{yy} , та I_{zz} – головні моменти інерції вздовж осей координат;

I_{xy} , I_{xz} , та I_{yz} – добутки інерції, які описують розподіл маси відносно відповідних площин.

$$I_{xx} = \sum_{i=0}^n I_i[x] + m_i * (y^2 + z^2) \quad (2.14)$$

$$I_{yy} = \sum_{i=0}^n I_i[y] + m_i * (x^2 + z^2) \quad (2.15)$$

$$I_{zz} = \sum_{i=0}^n I_i[z] + m_i * (x^2 + y^2) \quad (2.16)$$

$$I_{xy} = m_i * x * y \quad (2.17)$$

$$I_{xz} = m_i * x * z \quad (2.18)$$

$$I_{yz} = m_i * z * y \quad (2.19)$$

де x, y, z – це просто зсув за відповідною віссю відносно геометричного центру примітиву, який вираховуємо. В нашому випадку таким є паралелепіпед, м.

2.1.4 Загальні сили

Не викликає здивування той факт, що і сила, і крутний момент представляються 3-вимірними векторами.

Сила – це вплив, який може спричинити зміну швидкості об'єкта, якщо він не врівноважений іншими силами аналогічної природи [24]. Момент сили є обертальним аналогом лінійної сили [25].

Нас також цікавлять результуюча сила та результуючий момент. Це концепція, яка полегшує моделювання так, ніби в певний момент часу прикладається лише одне значення крутного моменту або сили.

$$\vec{F}_{net} = \sum \vec{F}_i \quad (2.20)$$

$$\vec{M}_{net} = \sum \vec{r}_i \times \vec{F}_i \quad (2.21)$$

де $\vec{F}_i$ – Сила, яку було прикладено до тіла, Н;

Цей метод використовує мішаний добуток для обчислення проекцій, оскільки він не залежить від розмірності, на відміну від підходу з функціями $\sin$ або $\cos$.

2.1.5 Інші сили

Сила нормальної реакції (іноді нормальна реакція опори) – сила, що діє на тіло з боку опори і спрямована перпендикулярно («по нормалі», «нормально») до поверхні дотику.

Сила тертя [24]

$$\vec{F}_n = (\vec{F} \cdot \vec{N})\vec{N} \quad (2.22)$$

Сила опору повітря [22]

$$\vec{F}_D = -\frac{1}{2} * \rho * v^2 * S * C_d * \hat{v} \quad (2.23)$$

Підйомна сила [22]

$$\vec{F}_L = \frac{1}{2} * \rho * v^2 * S * C_l * \hat{n} \quad (2.24)$$

де $\hat{n}$ – нормальний вектор;

$\hat{v}$ – орт швидкості;

S – площа поверхні, м²;

v^2 – швидкість в квадраті, (м/с²)²;

ρ – щільність повітря на даній висоті, кг/м³;

C_l, C_d – коефіцієнти підйомної сили та аеродинамічного опору, безрозмірна.

2.2 Обробка колізій

2.2.1 Примітиви колізій

Незважаючи на те, що майже неможливо обчислити точне зіткнення для тіл з багатьма вершинами та гранями, досить легко, як з точки зору обчислень, так і з точки зору написання коду, змодельовати зіткнення для більш простих форм.

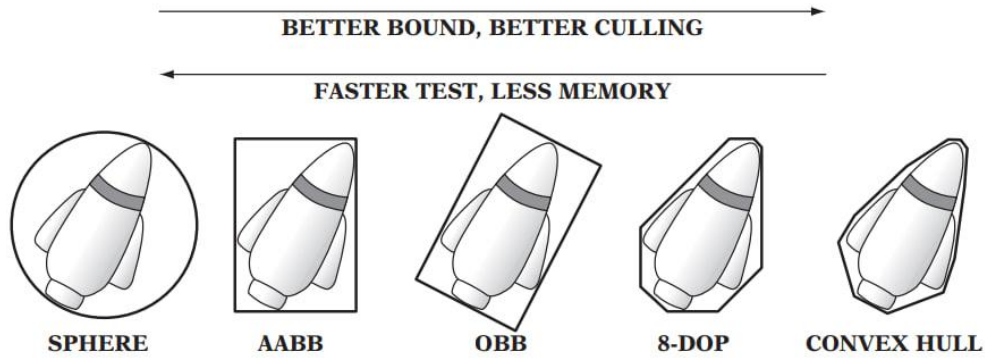


Figure 4.2 Types of bounding volumes: sphere, axis-aligned bounding box (AABB), oriented bounding box (OBB), eight-direction discrete orientation polytope (8-DOP), and convex hull.

Рисунок 2.2 – Типи примітивів колізії [21]

Існує велика різноманітність підходів до апроксимації тіл. До них відносяться ВРТ, ОВВ, Сфери тощо. Я вирішив зупинитися на орієнтованій коробці. Орієнтована коробка (Oriented Bounding Box) є примітивом для визначення зіткнень, яка має розміри, обертання та позицію. Розмір цього примітива становить: $3 * \text{Float} + 4 * \text{Float} + 3 * \text{Float} = 10 \text{ Float} = 40$ байтів.

Що є дуже добрим поєднанням точності та витрат на обчислення і зберігання цього примітива.

Таблиця 2.1 – Порівняння примітивів

Примітив	Складники	Використання пам'яті	Швидкість обчислення
Сфера	Позиція (3 floats) + Радіус (1 float)	16 байтів	Найшвидша(Single distance check)
Орієнтована коробка (OBB)	Розміри (3 floats) + Обертання (4 floats) + Позиція (3 floats)	40 байтів	Середня (OBB-OBB collision requires separating axis theorem)

Опукла оболонка (convex hull)	M Точок ($M \times 3$ floats) + F Індексів ($F \times 3$ integers) + Обертання (4 floats) + Позиція (3 floats)	$12(M + F) + 28$ байтів	Дещо повільна (Requires complex algorithms like GJK, EPA, or SAT)
----------------------------------	---	-------------------------	--

2.2.2 Апроксимація складних об'єктів

Щоб врахувати неузгодженості, які можуть виникнути при використанні окремих ОВВ, я представляю метод апроксимації будь-якої сітки за допомогою фіксованої кількості ОВВ. Він використовує k -середні для кластеризації даних та PCA.

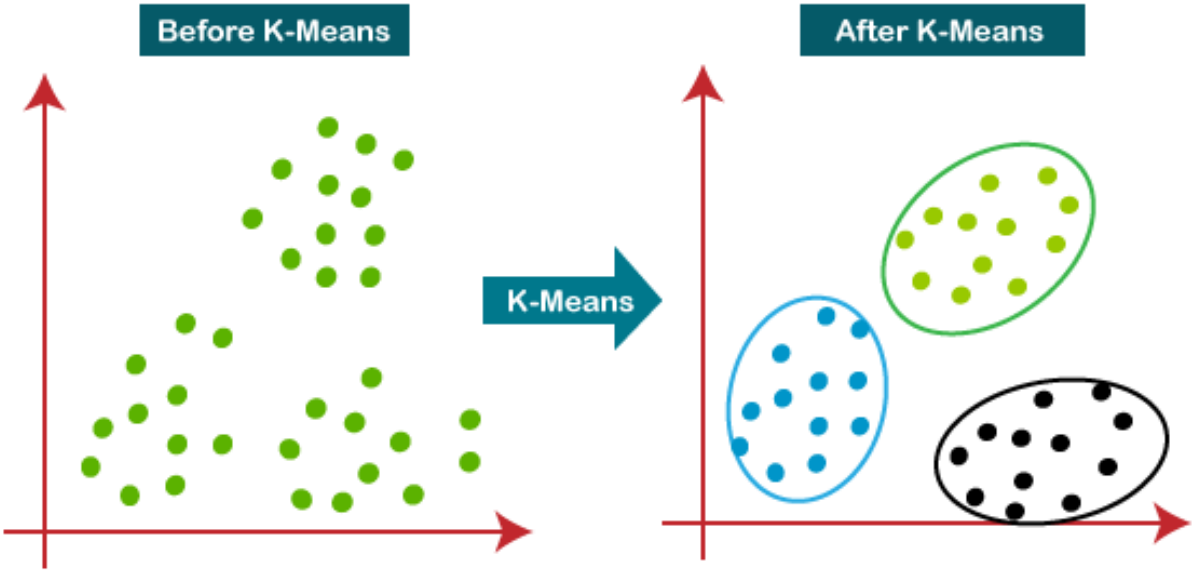


Рисунок 2.3 – Візуалізація К-середніх [26]

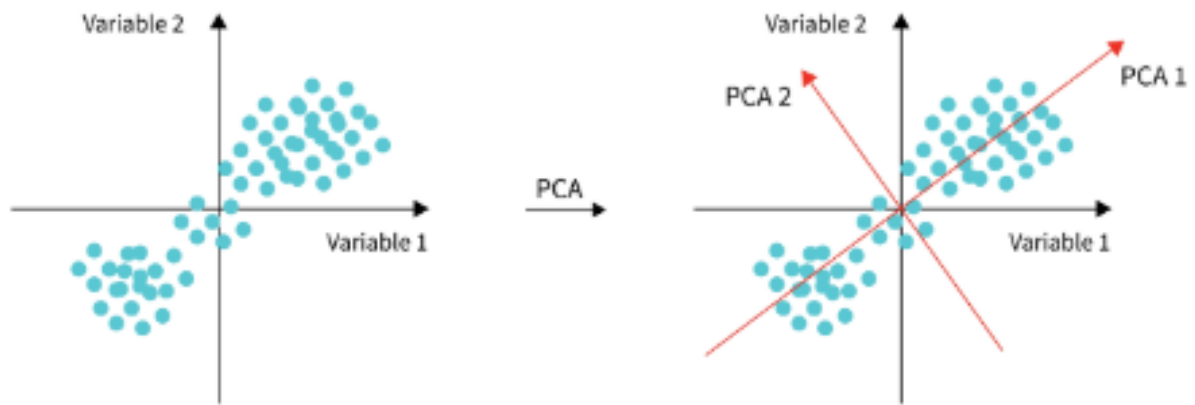


Рисунок 2.4 – Візуалізація PCA [27]

Це не дуже типовий метод для цієї задачі. Кращою ідеєю було б реалізувати двійкове розбиття простору, але мені було цікаво спробувати PCA в цій нетиповій ситуації.



Рисунок 2.5 – PCA застосований до 3д-моделі



Рисунок 2.6 – Апроксимація коробкою

2.2.3 Обчислення реакції на зіткнення

Фізична реакція розраховується за допомогою імпульсного методу

$$v_{rel} = (v_1 - v_2) + (w_1 \times r_1) - (w_2 \times r_2) \quad (2.25)$$

$$J = \frac{-(1+e)(v_{rel} \cdot n)}{\frac{1}{m_1} + \frac{1}{m_2} + n * ((I_1^{-1}((r_1 \times n) \times r_1)) + (I_2^{-1}((r_2 \times n) \times r_2)))} \quad (2.26)$$

В результаті деяких наближень, ми формулу для обчислення зіткнення, що підходить для наших потреб при незначних витратах часу. Це дозволяє виконувати обчислення у реальному часі.

2.3 Симуляція хвиль

Я вирішив використати рівняння Герстнера для моделювання води. Хвилі Герстнера – це математична модель, яка використовується в комп'ютерній графіці та фізичних симуляціях для хвиль на воді. Вони зручні для створення реалістичного руху хвиль у додатках реального часу, таких як ігри, де обчислювальна ефективність має вирішальне значення [7].

$$F(A, p, k, w, t, \varphi, \hat{d}) = \begin{cases} x = x_0 - A \cdot \cos(kx_0 - \omega t + \varphi) \cdot \hat{d}_x \\ y = y_0 + A \cdot \sin(kx_0 - \omega t + \varphi) \\ z = z_0 - A \cdot \cos(kx_0 - \omega t + \varphi) \cdot \hat{d}_z \end{cases} \quad (2.27)$$

де $p = (x_0 \ y_0 \ z_0)$ – початкова позиція хвилі;

A – амплітуда хвилі, м;

$k = 2\pi\lambda$ – хвильове число;

λ – довжина хвилі, м;

$\omega = 2\pi f$ – кутова частота, рад/с;

f – частота хвилі, Гц;

t – час, с;

φ – фазовий зсув, рад;

$\hat{d} = (\hat{d}_x, \hat{d}_z)$ – Вектор розповсюдження хвилі.

Для більшої реалістичності кілька хвиль можна об'єднати.

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix}_{final} = \sum_{i=0}^n F(A, p, k, w, t, \varphi, \hat{d}) \quad [8] \quad (2.28)$$

Де $A, p, k, w, t, \varphi, \hat{d}$ відповідають попередньо визначеним коефіцієнтам і векторам для кожної хвилі.

Хоча це наближення хвиль і враховує характер руху хвиль, воно не задовольняє всі умови і потребує додаткових удосконалень для більш детальної реалізації.

2.4 Огляд існуючих продуктів

Часто схожі бібліотеки використовують для написання коду великі компанії. Знизу можна прочитати короткий огляд продуктів на їх основні властивості.

- **NVIDIA PhysX**

Розробник: NVIDIA

Тип ліцензії: BSD, інтегрується з рушіями (наприклад, Unreal Engine)

Короткий опис: PhysX – один із найпоширеніших фізичних рушіїв, який підтримує АТТ, колізії, симуляцію тканин, рідин, частинок тощо. Головною перевагою є можливість апаратного прискорення за допомогою GPU (CUDA).

Використання: Unreal Engine 4, Unity, власні ігри NVIDIA, інженерні симуляції.

- **Havok Physics**

Розробник: Havok (підрозділ Microsoft)

Тип ліцензії: Пропрієтарна

Короткий опис: Havok – комерційний фізичний рушій, який використовується у багатьох іграх. Підтримує АТТ, м'які тіла, кінематику, зіткнення, тканини, рідини. Має модулі для анімації, поведінки, руйнування об'єктів.

Використання: Ігри серій Assassin's Creed, Elder Scrolls, Halo, Uncharted. Дуже добре підходить для AAA-проектів.

- **Bullet Physics**

Розробник: Erwin Coumans (тепер — Google)

Тип ліцензії: zlib

Короткий опис: Безкоштовний фізичний рушій із відкритим кодом. Підтримує динаміку АТТ, м'які тіла, тканини, колізії, обробку транспортних засобів. Має інтеграцію з Blender, Unity, Panda3D та іншими.

Використання: Blender Game Engine, робототехнічні симулятори (наприклад, PyBullet), ігри з низьким бюджетом або інді-розробка.

- Jolt Physics

Розробник: Джейсон Бек

Тип ліцензії: MIT

Короткий опис: Високопродуктивний сучасний фізичний рушій для жорсткотілої динаміки. Орієнтований на використання SIMD, багатопоточність, інтеграцію в AAA-ігри. Позиціонується як альтернатива PhysX і Havok із відкритим кодом.

Використання: Починає з'являтися в нових проектах. Популярний серед ентузіастів продуктивного геймплею.

- ODE (Open Dynamics Engine)

Розробник: Russell Smith

Тип ліцензії: BSD

Короткий опис: Старіший фізичний рушій із підтримкою жорсткотілої динаміки, корисний у симуляторах і технічних проєктах. Простий у використанні, але менш точний, ніж Bullet чи PhysX.

Використання: Освітні симулятори, ігри з відкритим кодом.

Фізичні рушії значно спрощують створення симуляцій, зменшуючи потребу в ручному моделюванні фізичних взаємодій. Великі компанії обирають PhysX та Havok за надійність і потужність, тоді як Bullet та Jolt — оптимальні для відкритих, кросплатформних рішень і наукових задач.

2.5 Стан розвитку у вітчизняній та закордонній науці

Із розвитком класичних наук у вітчизняній науці дещо краще, але я використовував джерела англійською для того, щоб не втрачати сучасний контекст.

- Jakob Maier. Flight Simulation (Blog) [6]

Містить опис створення моделі польоту для симулятора, включаючи застосування аеродинамічних сил: тяга, опір, підйомна сила.

- Thomas R. Yechout et al. Introduction to Aircraft Flight Mechanics (Книга) [22]

Академічне джерело, що розглядає статичну та динамічну стабільність, а також моделі керування літальними апаратами. Дуже корисне для симуляції авіації.

- Christer Ericson. Real-Time Collision Detection [21]

Розглядає фізику взаємодії тіл, зіткнення, зісковзування, що критично для фізичного рушія.

- Wikipedia. Force, Torque (Статті) [24], [25]

Вступ до базових понять сили та моменту сили, які прямо використовуються в симуляціях динаміки тіл.

- Javalab. Moments of Inertia [23]

Інтерактивне пояснення моменту інерції, що є основоположним при обертанні тіл в фізичних симуляціях.

- Jocelyn Frechot. Realistic Simulation of Ocean Surface Using Wave Spectra [9]

Академічна праця, що використовує спектральні моделі (наприклад, спектр Пієрсона-Московица) для моделювання великомасштабної динаміки океану.

- Max madblade. Vertex shader implementation of Gerstner waves [17]

Практичне застосування хвиль Герстнера, класичного методу опису періодичних хвиль у фізиці океану, адаптованого до вершинного шейдера.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ПОСТАВЛЕНИХ ЗАДАЧ

3.1 Комп'ютерна графіка

3.1.1 Налаштування проєкту

Всім додатком керує Apache Maven (v4.0.0). Це сприяє кросплатформності додатка. Використовуючи `pom.xml`, можна надати специфічні для платформи власні та стандартні JAR-файли або артефакти, розміщені на вашому комп'ютері або в Maven Central. На цьому етапі підключаються всі розширення, бібліотеки фізики та користувацькі класи.

3.1.2 Використання ресурсів

Щоб розкрити цю тему, потрібно описати всі типи ресурсів, які програма буде використовувати і якими вона буде керувати. Кожен рушій 3D-гри повинен використовувати 3D-моделі. Вони представлені в основному файлами Wavefront Obj. Ця специфікація широко підтримується і є звичайним текстовим файлом, тому її легше використовувати, аналізувати, виправляти тощо.

По-друге, він повинен підтримувати завантаження текстур. Tourmaline підтримує 2D і 3D текстури, представлені `GL_TEXTURE_2D` і `GL_TEXTURE_3D` відповідно. Хочу підкреслити, що важливо використовувати різні типи даних для економії пам'яті та забезпечення достатньої деталізації, щоб зробити вивід барвистим і неспотвореним. До них відносяться `GL_RGB8F`, `GL_RGB16F`, `GL_RGB32F` та будь-які інші доступні типи.

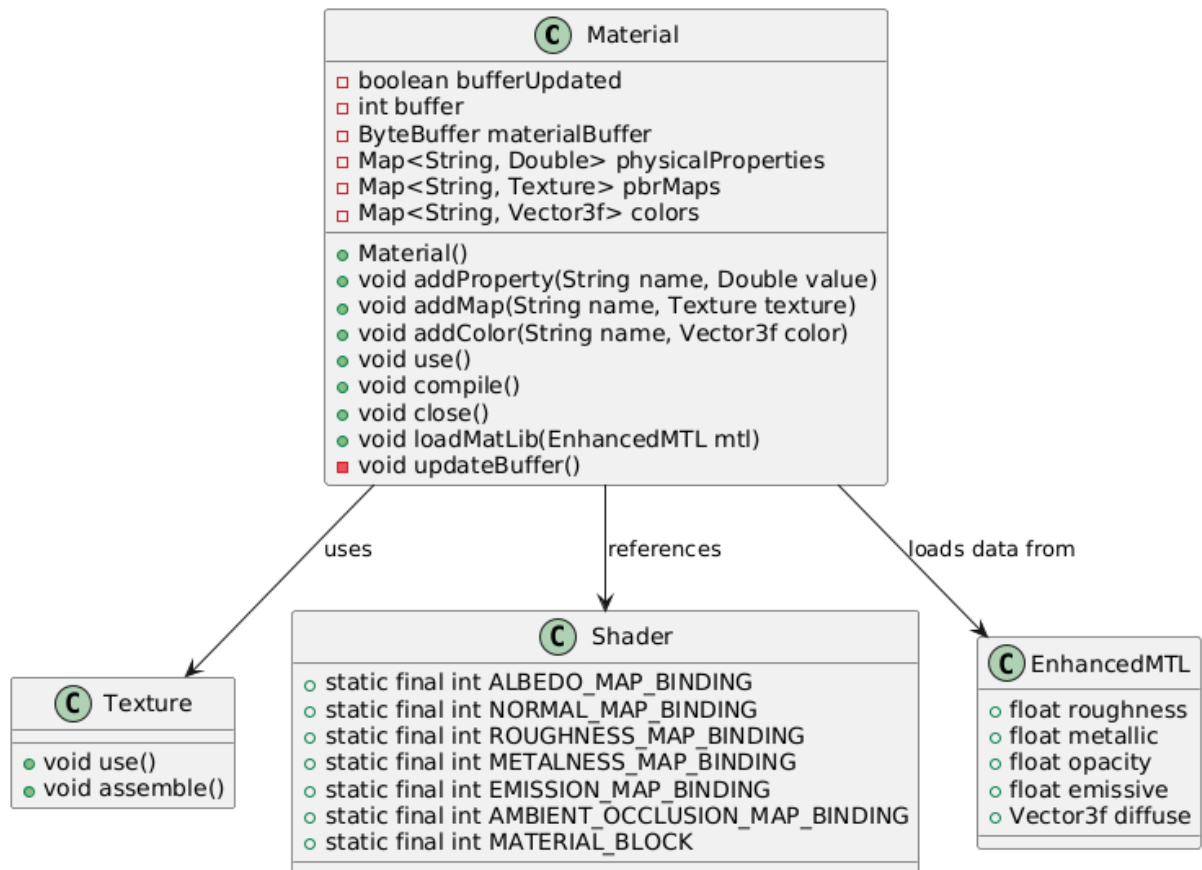


Рисунок 3.1 – UML-діаграма класу матеріала

Щоб забезпечити швидке завантаження ресурсів, необхідно використовувати методи паралельного програмування. У моєму випадку завантаження відбувається одночасно (concurrent) – використовується fixed worker pool, який розділений так, щоб система могла обробляти різні типи даних одночасно. Кожен завантажений об'єкт представлений інтерфейсом, який обробляє файл і змінює стан цього об'єкта. Після завантаження об'єкти **EnhancedLoadables** компілюються або збираються. Через однопоточну архітектуру OpenGL неможливо завершити завантаження об'єкта в неосновному потоці. Це одна з причин, чому Vulkan стрімко набирає популярність. Після цього 3D-модель збирається в n-арне дерево, яке зазвичай являє собою об'єкт з багатьма гілками. Цей тип дозволив би мені використати деякі переваги, такі як передача трансформацій моделі вниз по дереву. Наприклад, потяг рухається, і всі вагони, що слідують за ним, автоматично роблять те ж саме, навіть якщо вони мають окремі матриці.

Такий великий проект потребує багато інфраструктури для нормального функціонування. Один з таких елементів був реалізований на Python. Мій додаток допомагає побудувати HDR-скайбокс, використовуючи сферичну проекцію.



Рисунок 3.2 – Панорама для IBL

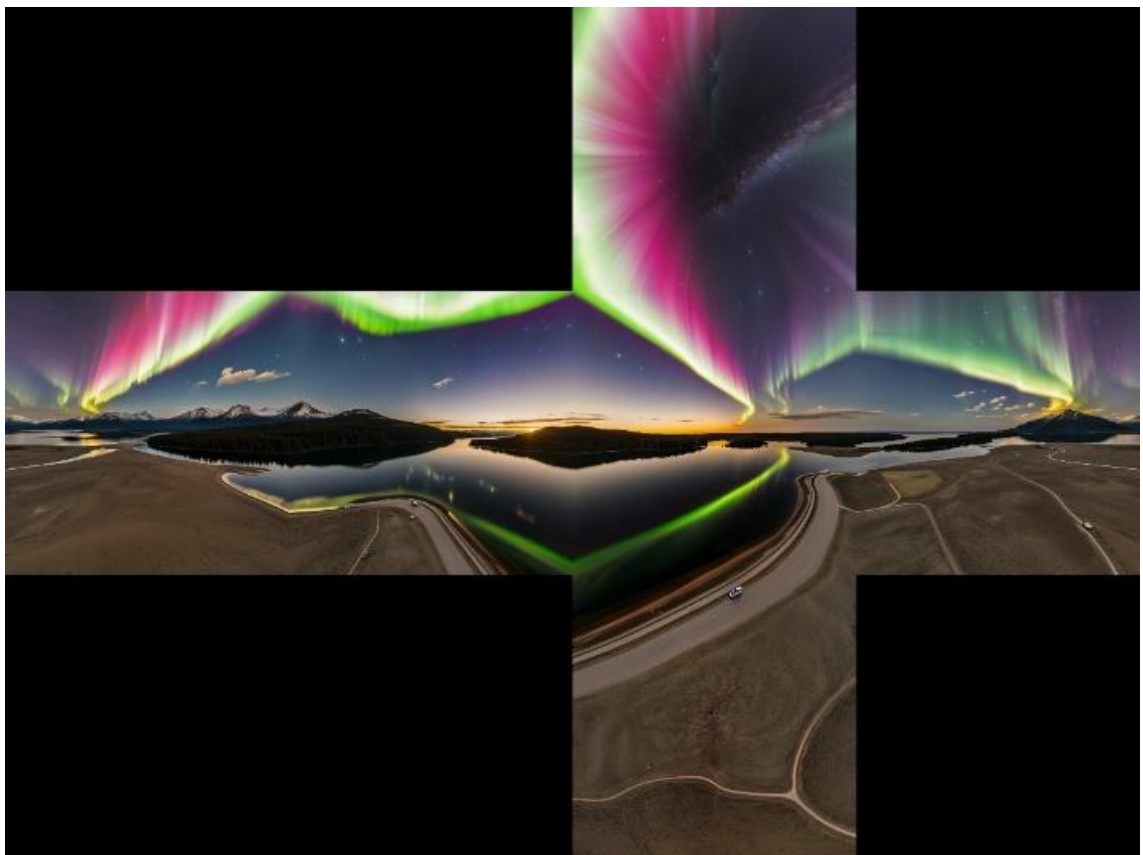


Рисунок 3.3 – Скайбокс

Для створення цих HDR-карт було використано сервіс для генерації ІП-скайбоксів [12]. Після цього вони діляться на окремі поверхні куба для трьох різних карт: skybox, radiance, irradiance. Вони використовуються в обчисленнях далі і виглядають наступним чином:

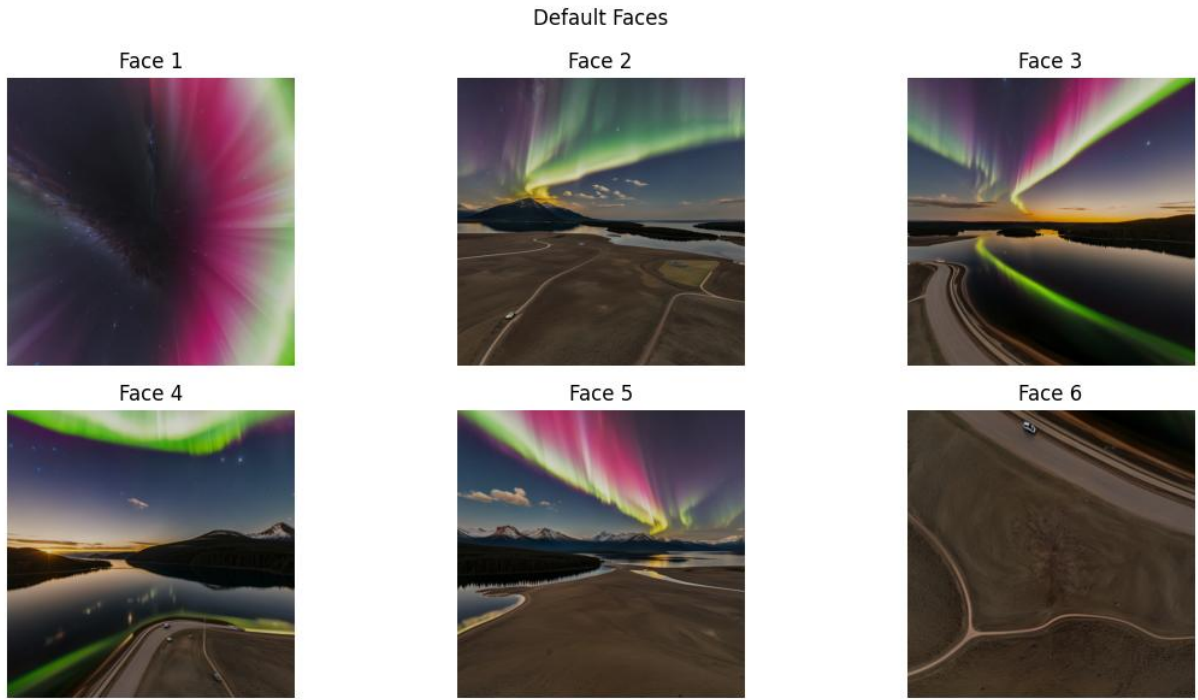


Рисунок 3.4 – Поділені поверхні

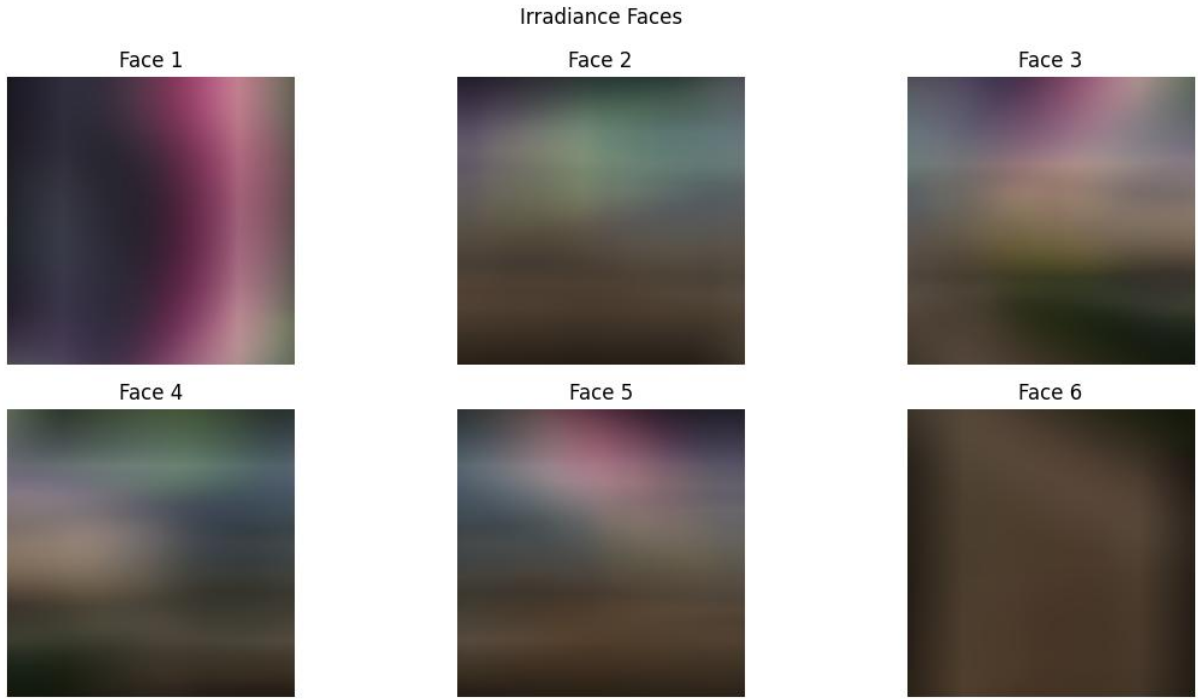


Рисунок 3.5 – Поділені поверхні



Рисунок 3.6 – Поділені поверхні

Рекомендується використовувати клас `AutoLoader`. Він рекурсивно обходить усі директорії і завантажує моделі у вигляді класів `MeshTree` та текстур у матеріалах. Єдина умова – правильні імена файлів.

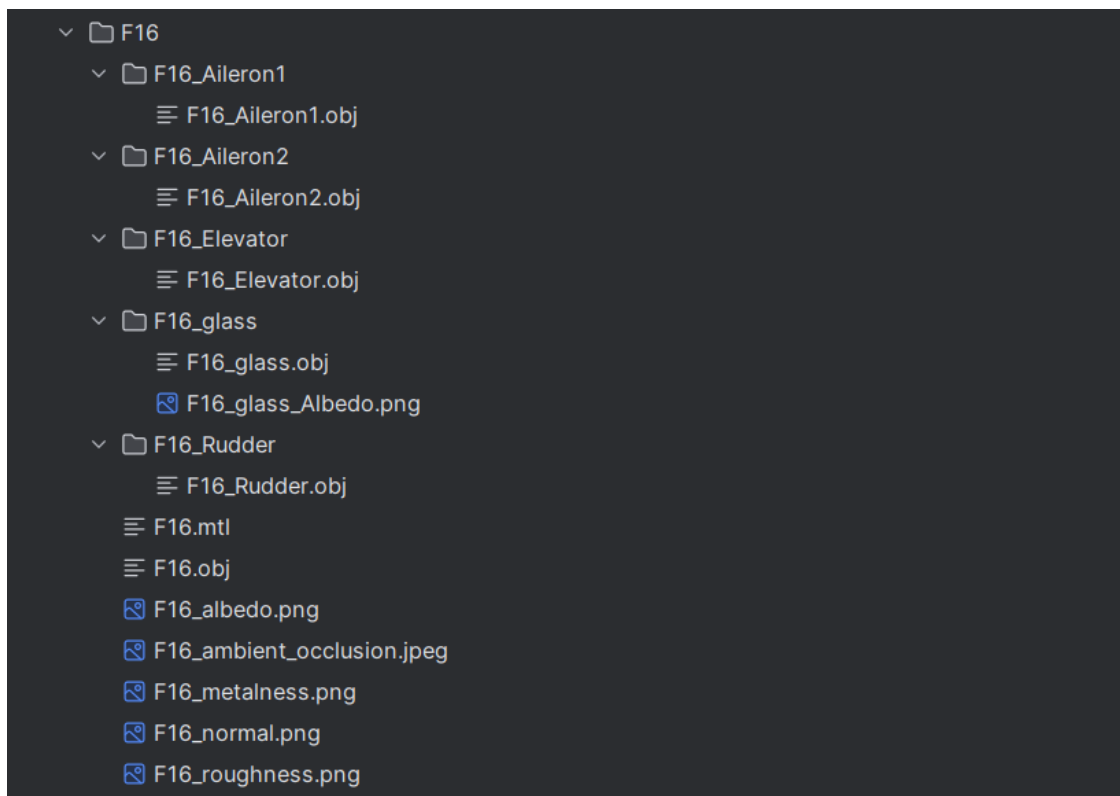


Рисунок 3.7 – Структура файлів моделі

Можна включати підмоделі, оскільки папка обходиться рекурсивно. Вони завантажуються як дочірні елементи моделі, розташованої вище. Клас MeshTree містить функцію, яка дозволяє користувацьке використання – можна зробити додаткову обробку за допомогою функціонального програмування та лямбда-функцій.

```
public void traverse(Consumer<Mesh> action) {
    MeshTree node = this;
    List<TreeNode<Mesh>> list = node.getChildNodes();

    if (node.getNodeValue() != null) {
        action.accept(node.getNodeValue());
        if (list != null) {
            for (TreeNode<Mesh> meshTreeNode : list) {
                ((MeshTree) meshTreeNode).traverse(action);
            }
        }
    }
}
```

Рисунок 3.8 – Метод рекурсивного обходу моделі

Приблизно так це можна використовувати:

```
public void draw() {
    traverse(Mesh::draw);
}
```

Рисунок 3.9 – Функціональне використання методу зверху

Обговорюючи рендеринг, важливо згадати про відкладений рендеринг і порівняти його з прямим рендерингом. PBR-конвеєр рендерингу може використовувати обидва підходи. Однак, використовуючи G-Buffer, конвеєр отримує набагато більше можливостей для пост-обробки. G-Buffer дозволяє використовувати різні значення, такі як положення, базовий колір, нормалі, для різних текстур.

Таблиця 2.1 – Порівняння видів рендерингу

Критерій	Відкладений	Прямий
Споживання пам'яті	Високе (Через використання G-буфера)	Низьке

Закінчення таблиці 2.1

Критерій	Відкладений	Прямий
Масштабованість	Висока (Ефективно працює із динамічним освітленням)	Низька (Сильно падає продуктивність з великою кількістю джерел світла)
Складність реалізації	Висока (багаторівневий конвеєр, складні шейдери)	Низька (простий конвеєр, стандартні шейдери)
Робота із прозорістю	Складна (Через те, що у G-буфері записано лише один шар)	Проста (Змішує кольори без додаткових налаштувань)
Робота згладжування	Складна (Потрібно самому писати)	Проста (Вбудований алгоритм MSAA)
Масштабованість світла	Набагато краща та не залежить від геометрії.	Погіршується із більшою кількістю об'єктів
Складність матеріалів	Складніші розширюється, але працює універсально	Легко розширюється, але може призвести до хаотичного коду
Пост-обробка	Проста	Неможлива без додаткових буферів
Складність шейдерних програм	Висока	Низька
Вузькі місця	Пропускна здатність пам'яті та G-Буфер	Обробка точок та погана масштабованість
Дебагінг	Більш складний	Більш простий

У підсумку я вирішив обрати саме відкладений графічний конвеєр, бо його переваги є більш значущими, аніж недоліки, які він приносить.

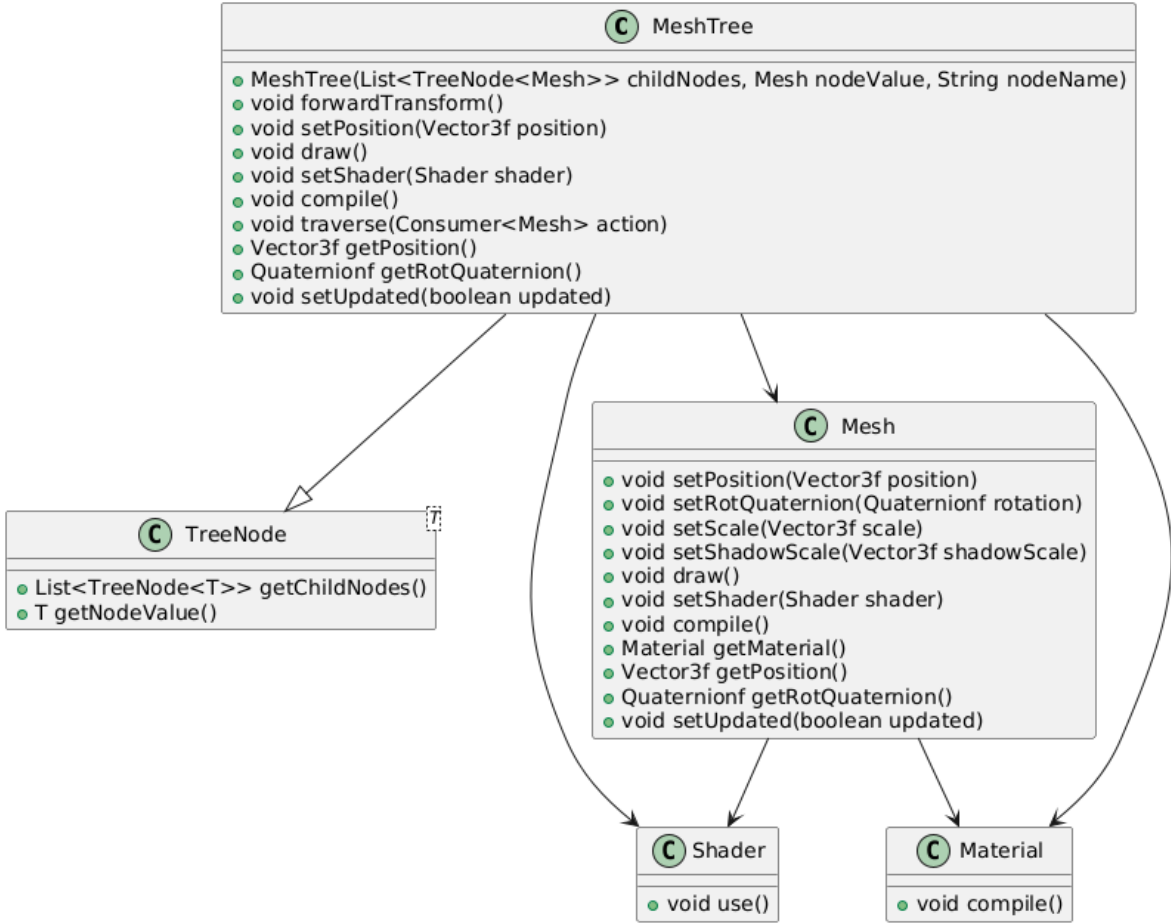


Рисунок 3.10 – UML-діаграма MeshTree

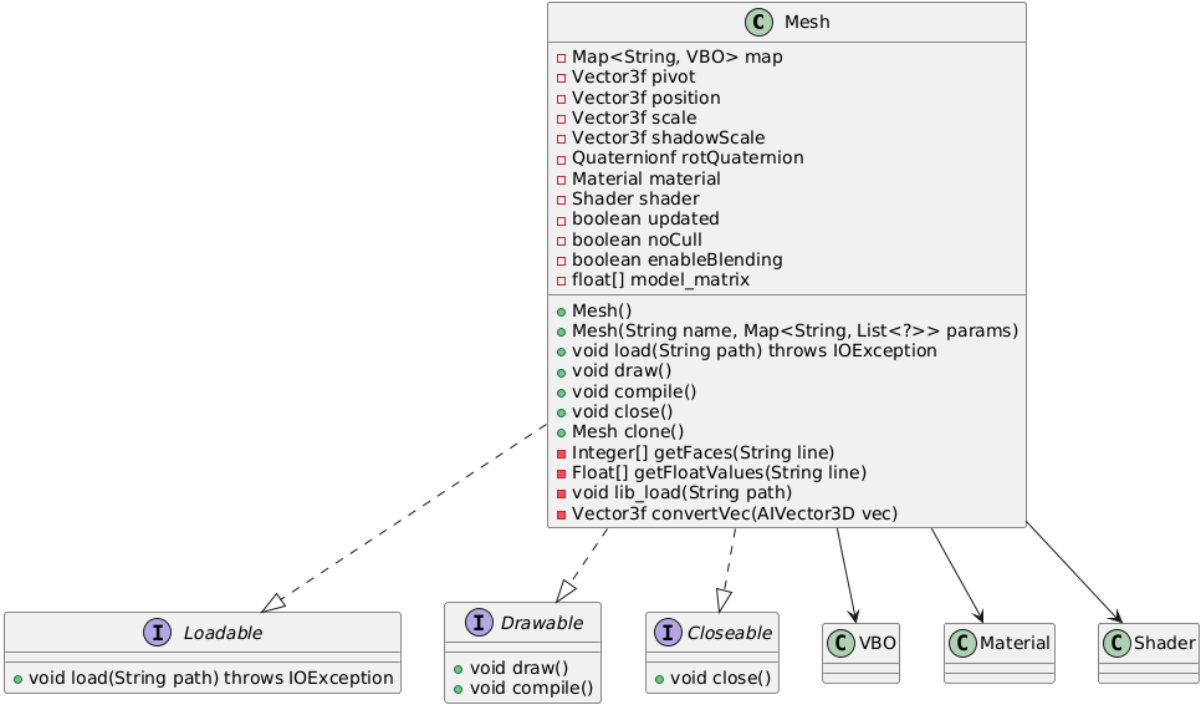


Рисунок 3.11 – UML-діаграма Mesh

3.1.3 Життєвий цикл додатка

Ця блок-схема відображає життєвий цикл мого продукту. Після ініціалізації ми потрапляємо в основний цикл, який постійно відображає 3D моделі, ефекти та елементи інтерфейсу.

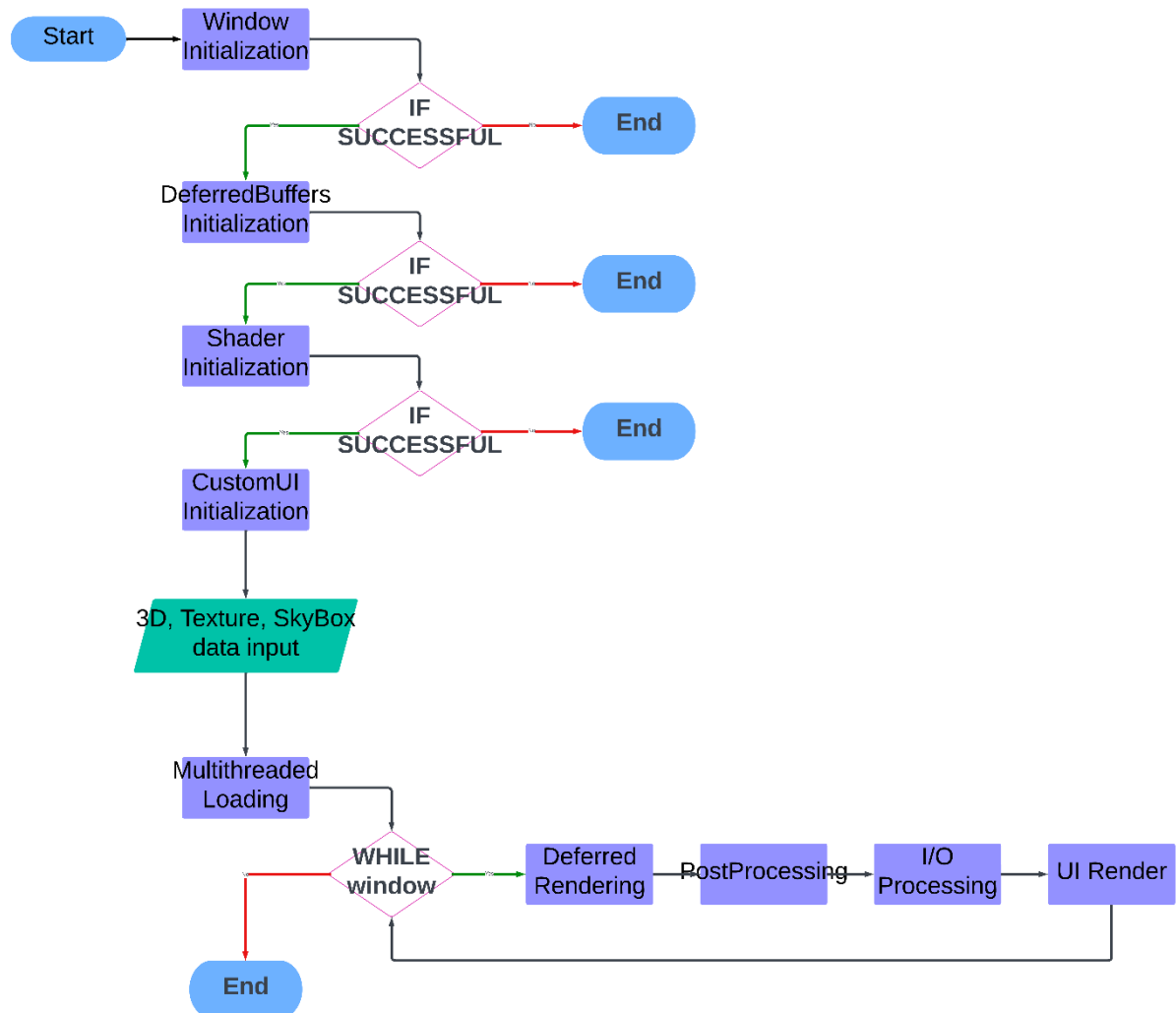


Рисунок 3.11 – Життєвий цикл

Головне вікно позначається анотацією `@OpenGLWindow`, подібно до `SpringBoot`, і розширює `BasicWindow`.

Процес візуалізації неможливий без використання камери. Камера визначає матриці проєкцій, які використовуються у розрахунках GLSL. Існує два типи проєкцій, які я буду використовувати: перспективна та ортографічна [29]. Клас камери забезпечує функціональність, необхідну для побудови цих матриць, зберігаючи положення, обертання та точку фокусу;

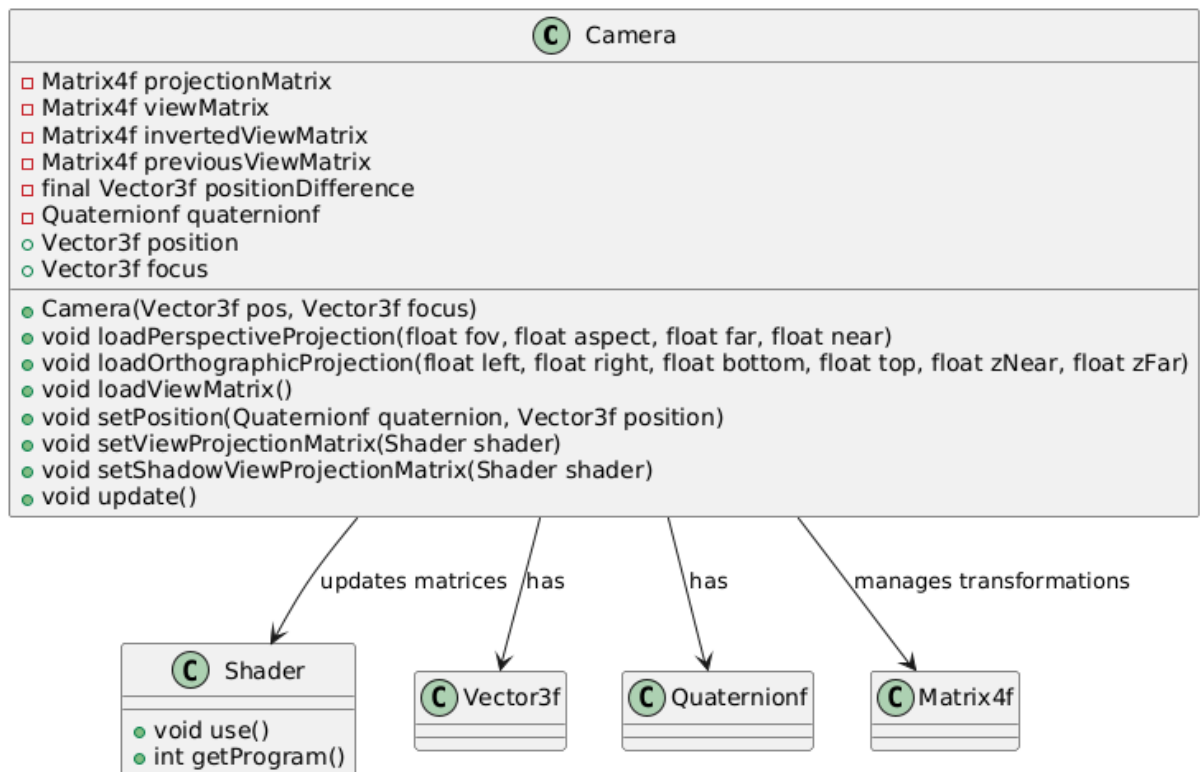


Рисунок 3.12 – UML-діаграма класу Camera

3.1.4 Шейдери glsl

Для цього типу додатків існує безліч шейдерних програм. Зазвичай шейдер складається з 2 файлів (фрагментарний та вершинний). Обов'язковою частиною нашого конвеєра є відкладений шейдер, комбінуючий шейдер та шейдер постобробки. Шейдер відображення тіней призначений для відображення тіней. Для спецефектів, таких як дим, вибухи та дим, можна використовувати шейдер спеціальних ефектів.

Програма відкладеного шейдера обчислює деякі змінні, необхідні для подальшого рендерингу, виконує геометричні операції, обчислює нормальні вектори, якщо потрібно, і передає їх у вигляді карт (текстур) на наступний етап – комбінуючому шейдеру.

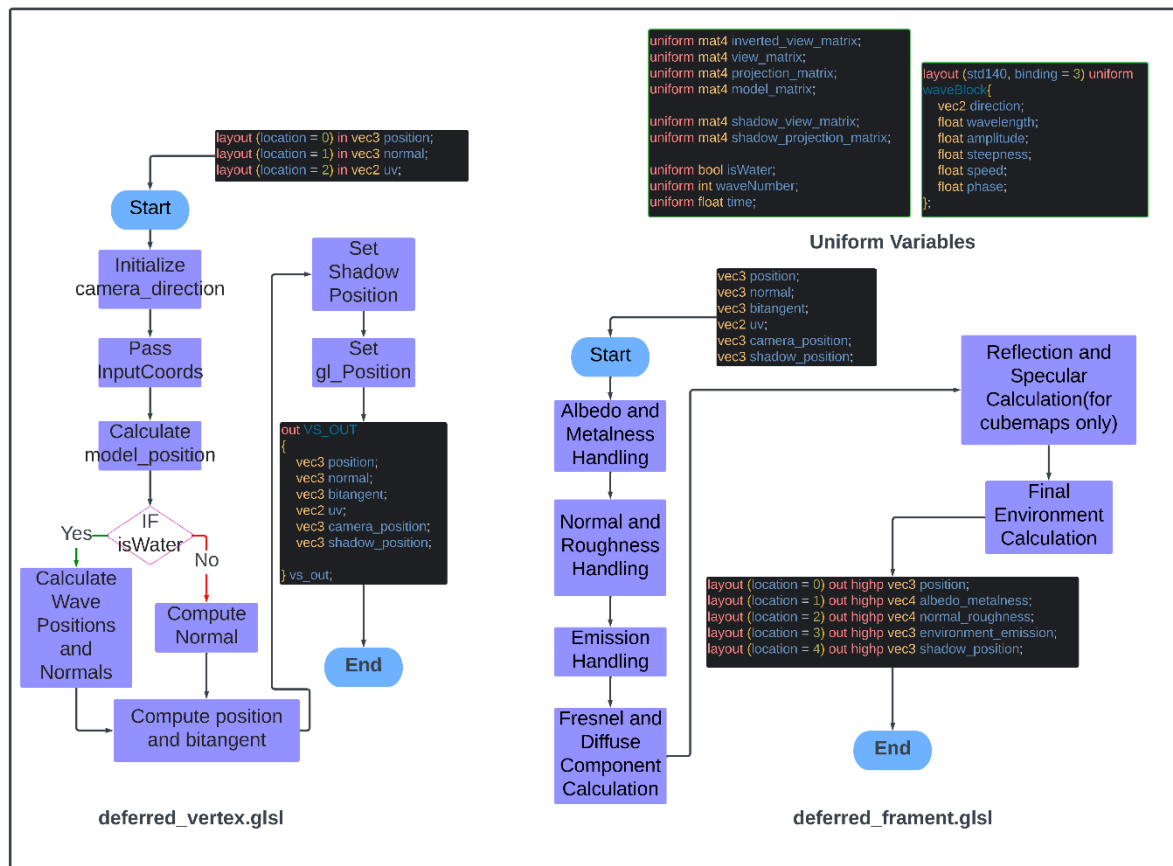


Рисунок 3.13 – Блок-схема відкладеного шейдера

Комбінуючі шейдери обчислюють освітлення та використовують карти тіней для формування освітлення відповідно до PBR. Skybox-шейдери використовуються після відкладених шейдерів і малюють skybox, який використовує HDR-файли, щоб забезпечити чудовий досвід для власників HDR-моніторів і забезпечує точність, яскравість і гнучкість зображення. У кінці розробки я також додав шейдер для прозорих моделей, щоб покращити відображення прозорих об'єктів.

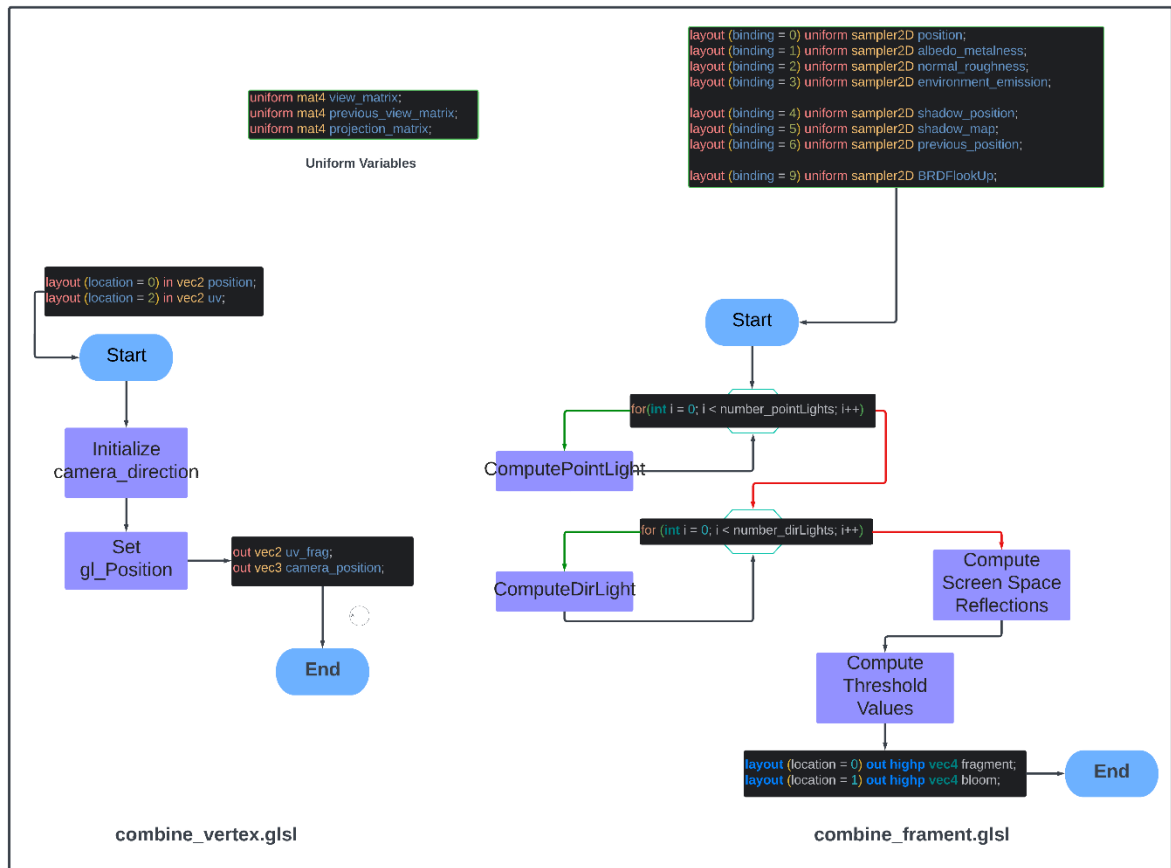


Рисунок 3.14 – Блок-схема об'єднуючого шейдера

Шейдер постобробки використовується для накладання додаткових ефектів. Він працює з текстурами, які відображаються на простому квадраті. Я використовую його для FXAA згладжування і блуму.

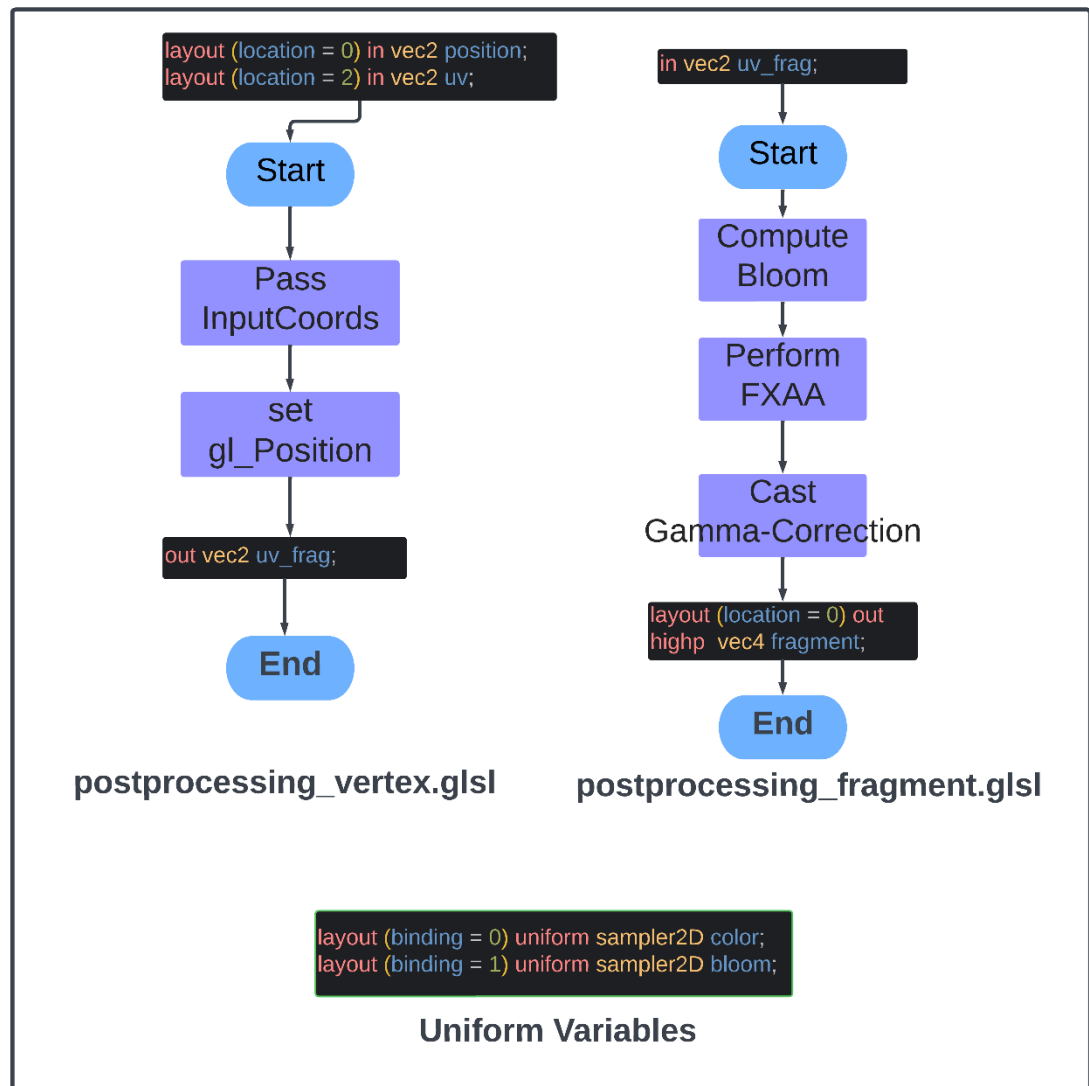


Рисунок 3.15 – Блок-схема шейдера постобробки

Шейдер спецефектів використовується для таких ефектів, як реактивний потік або вибух. Він може бути використаний для будь-якого ефекту на основі 3Д-мешу. Цей шейдер не підходить для ефектів на основі частинок, але користувач може його модифікувати.

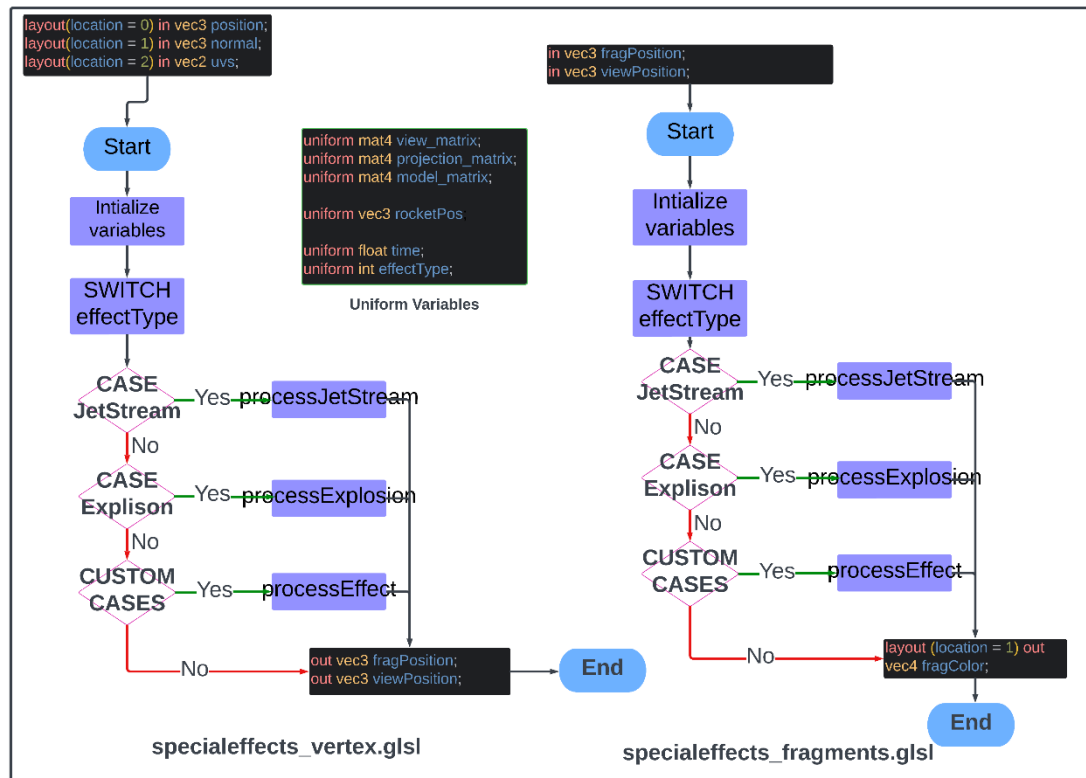


Рисунок 3.16 – Блок-схема шейдерів спецефектів

За замовчуванням GLSL не підтримує директиву `#include` [22]. З цієї причини я реалізував препроцесор, який підтримує заголовні файли на кшталт GCC або Clang.

Кожна програма повинна мати директиву `#version`. Вона визначає версію GLSL, яку ви використовуєте, за якою слідує профіль (`core/compatibility`). У моєму випадку це 460 core. Код шейдерів компонується і компілюється під час виконання програми і може бути відредагований або покращений кінцевим користувачем.

3.1.5 Впроваджені алгоритми

3.1.5.1 Освітлення на базі hdr-карт

Зображення демонструє, як мої шейдерні програми реалізують фізичний рендеринг, успішно використовуючи множник Frensel, розподіл GGX для мікрофасетної моделі, SchlickGGX для геометричного множника, а також BDRF, попередньо обчислена і використана як текстура.



Рисунок 3.17 – Відображення карти оточення на літаку

3.1.5.2 Затінення об'єктів



Рисунок 3.18 – Відображення тіней

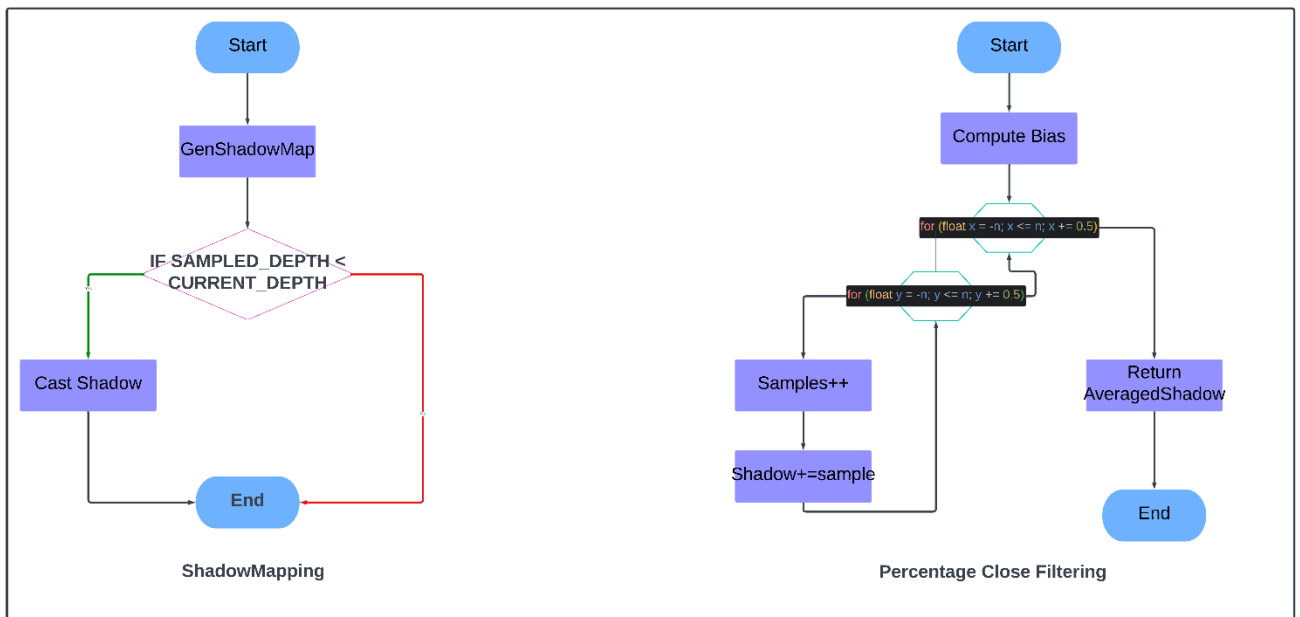


Рисунок 3.19 – Блок-схема алгоритму відображення тіней

Алгоритм відображення тіней полягає у створенні карти тіней у системі координат світлопростору. Вона зберігається в об'єкті фреймбуфера. На етапі комбінування вона виділяється за допомогою фільтрації та зсуву, щоб забезпечити більш чисте відображення тіней. Значення глибини нарешті порівнюється з поточним значенням глибини, переведеним у світловий простір. Результат зменшує яскравість поточного фрагмента і, відповідно, відкидає тінь.

3.1.5.3 Ефект блум (bloom)

Цей ефект просто об'єднує основну карту кольору із обробленою картою: виділяються найбільш яскраві зони та розмиваються.



Рисунок 3.20 – Фільтр Bloom [11]

3.1.5.4 Віддзеркалення у екранному просторі (ssr)

Ця техніка є досить складною і часто вивчається на просунутих курсах комп'ютерної графіки. Вона часто асоціюється з Unity3D або Unreal Engine, що робить її атрибутом дорогих і «дорослих» продуктів і рішень.

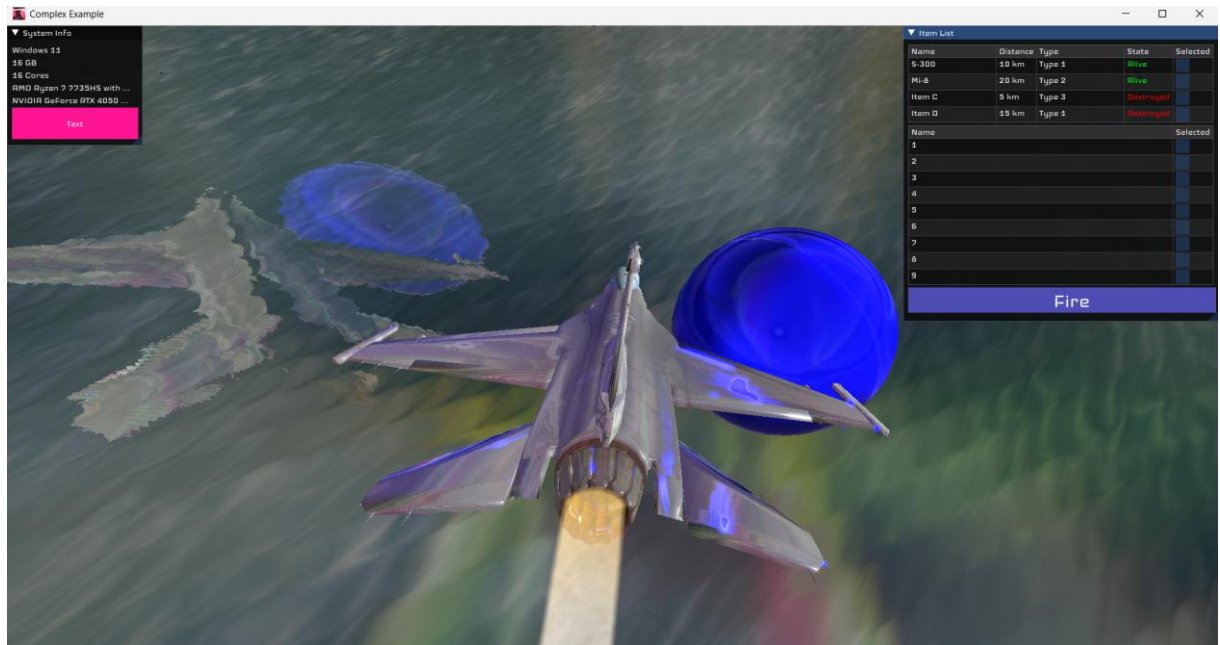


Рисунок 3.21 – Віддзеркалення у екранному просторі

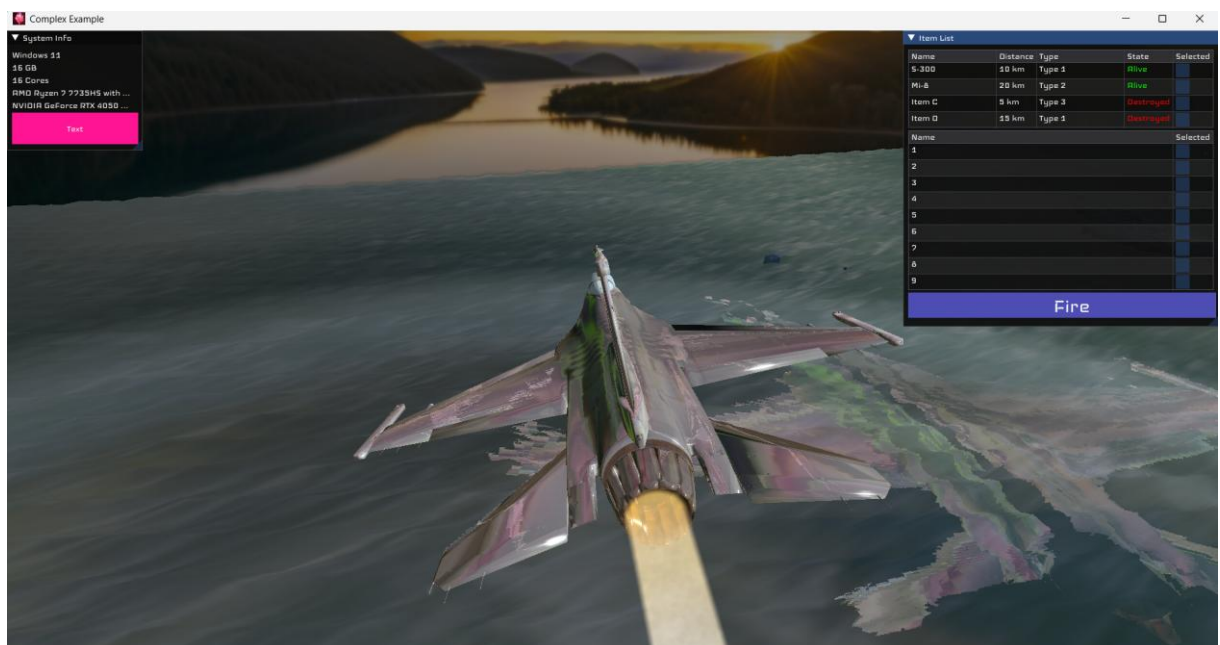


Рисунок 3.22 – Віддзеркалення у екранному просторі

Він використовує оптимізований і наближений алгоритм трасування променів, який називається ray-marching. Проходячи буфер глибини, ми

отримуємо відбиття від металевих або відбивних поверхонь (наприклад, води) [10].

3.1.5.5 Налаштування джерел освітлення

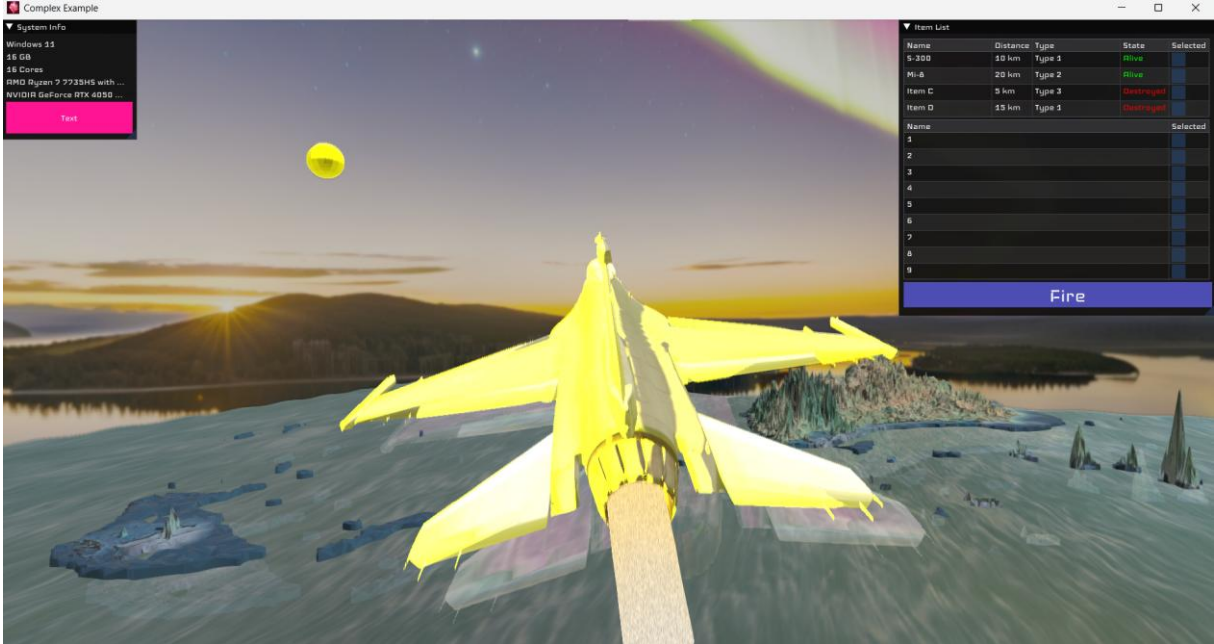


Рисунок 3.23 – Жовте точкове освітлення

3.1.5.6 Типи розмиття



Рисунок 3.24 – Gaussian Blur

3.1.5.7 Рендеринг води

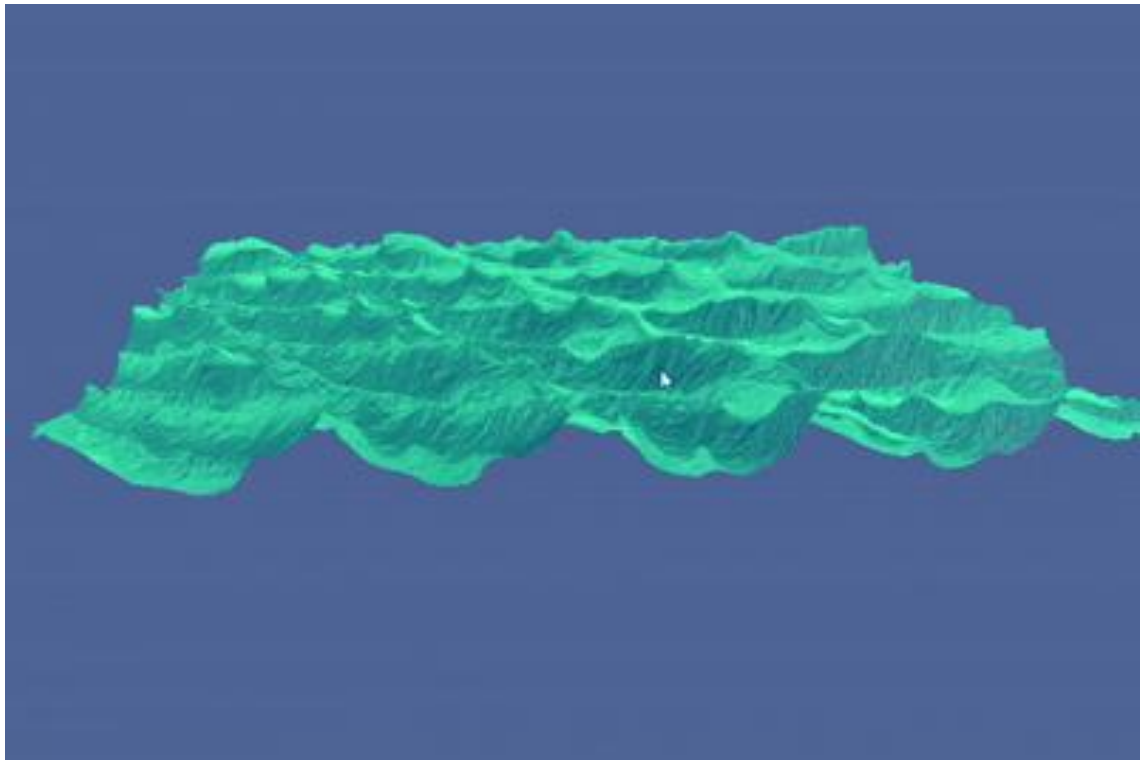


Рисунок 3.25 – Рендеринг води

3.1.5.8 Інші ефекти

Я запровадив різні методи для використання різних кольорових просторів у додатку. Якщо буде потрібно, їх можна використати.

- Greyscale
- YCbCR
- CMYK
- HSV

Також я розробив методи для використання різних градієнтів.

- Лінійний
- Радіальний
- Конічний (Кутовий)

Types of Gradients



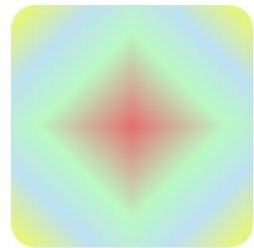
Linear Gradient



Angular Gradient



Radial Gradient



Diamond Gradient



Рисунок 3.26 – Типи градієнтів [13]

Було реалізовано два види спецефектів. Вони були використані в шейдері спецефектів для імітації реактивного потоку та вибуху.

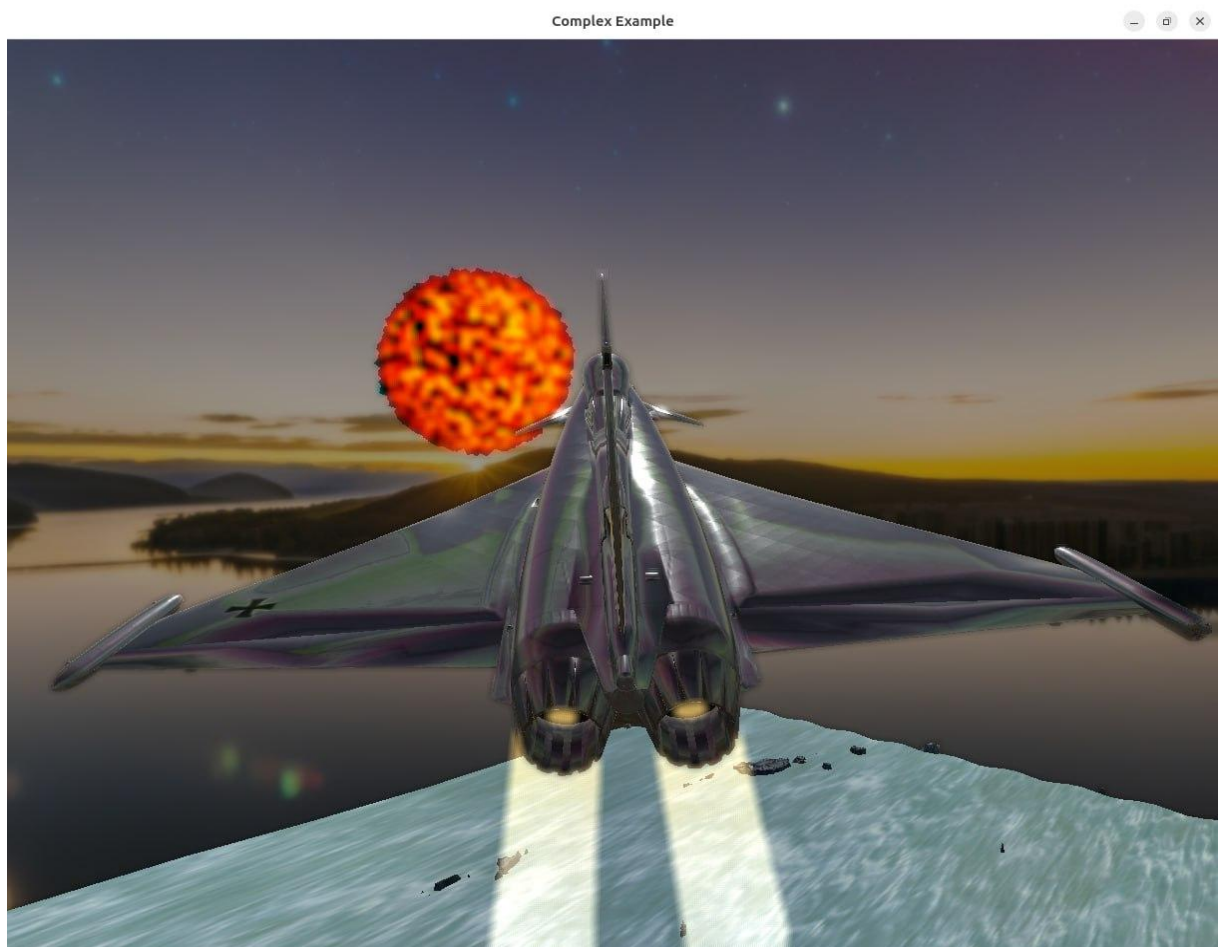


Рисунок 3.27 – Вибух та реактивний струмінь

3.1.6 Обробка вводу та виводу (i/o)

У світі існує величезна кількість пристроїв вводу-виводу для ігор та ПК загалом: сенсорний екран, джойстик (контролер), миша, клавіатура, рульове колесо. Мій рушій підтримує операції вводу та виводу з миші та клавіатури. І дозволяє відносно просто модифікувати його для підтримки різних типів за потреби.

3.2 Симуляція фізики

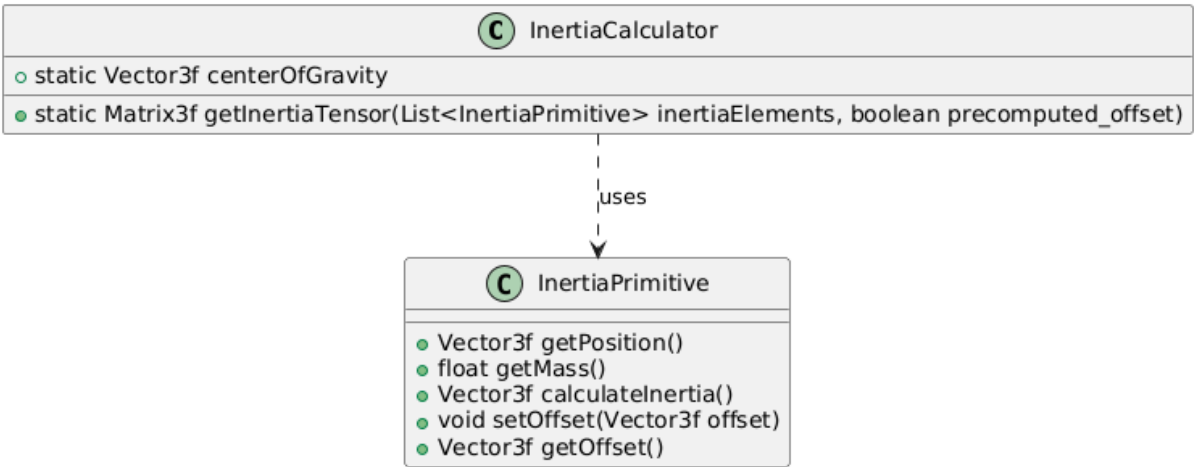


Рисунок 3.28 – UML класу Inertia Calculator

Цей клас дозволяє обчислювати тензор інерції. Це критично важливо для створення стабільної симуляції.

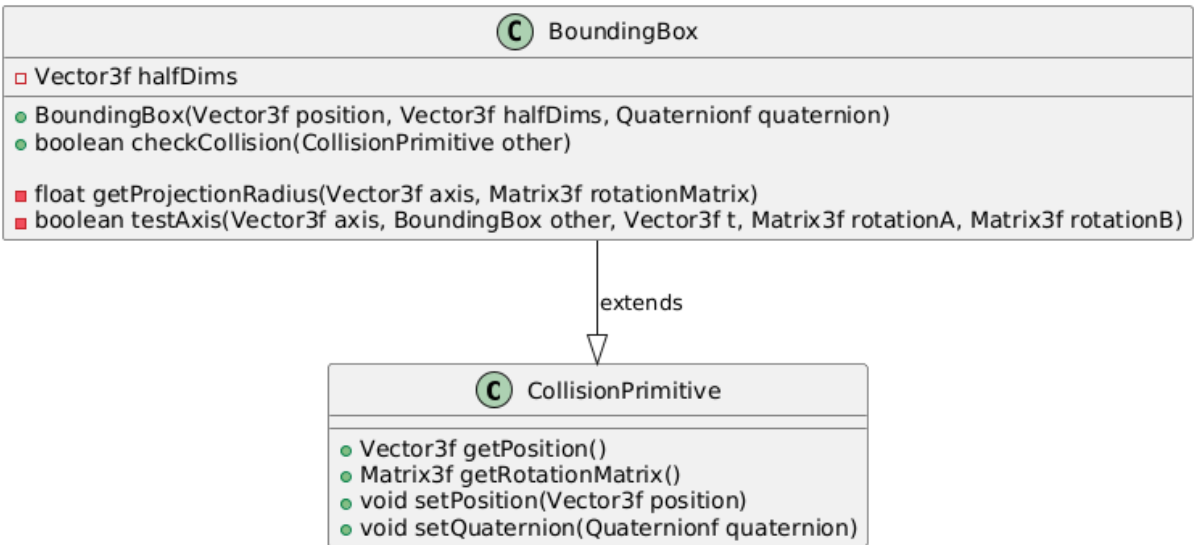


Рисунок 3.29 – UML класу Bounding Box

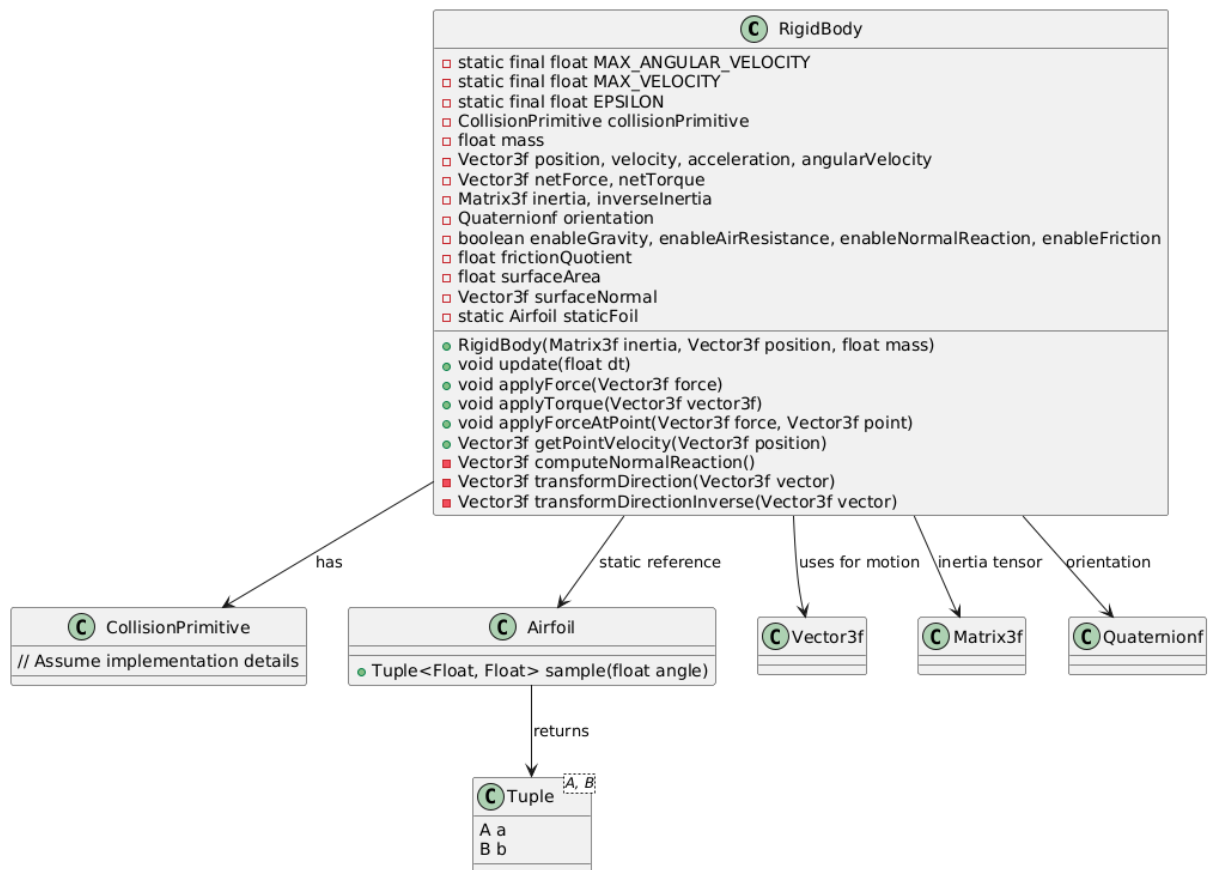


Рисунок 3.30 – UML класу Rigid Body

Примітив колізії описує те, як буде поводитися тіло при зіткненні, клас Airfoil зберігає в собі функцію двох змінних із коефіцієнтом підйомної сили на коефіцієнтом опору повітря. Для забезпечення числової стабільності значення, обмежуються значеннями MAX_VELOCITY і MAX_ANGULAR_VELOCITY.

Чиста сила і крутний момент обчислюються за допомогою функцій addForce, addTorque і addForceAtPoint. Останній метод застосовує і крутний момент, і силу одночасно. Метод void update(float dt) є інтегратором цього класу і використовує метод Стермера-Верле.

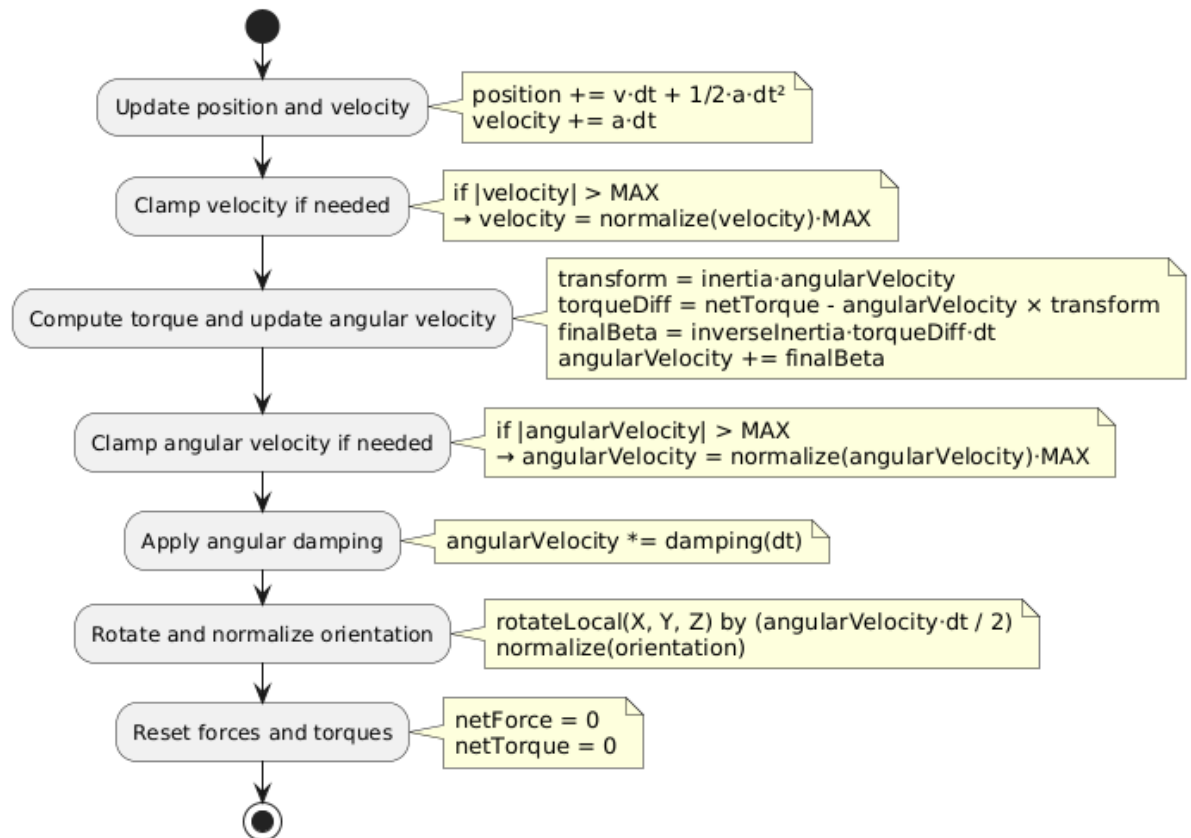


Рисунок 3.31 – Алгоритм неявного інтегрування

У цьому методі було використано інтеграцію через швидкість та прискорення. У більш складних варіантах можна розкласти функцію в ряд Тейлора і використати похідні від прискорення. Оскільки у цьому випадку прискорення, як і сила, – це дискретна величина, то таке поняття як ривок, до нього не можна застосувати.

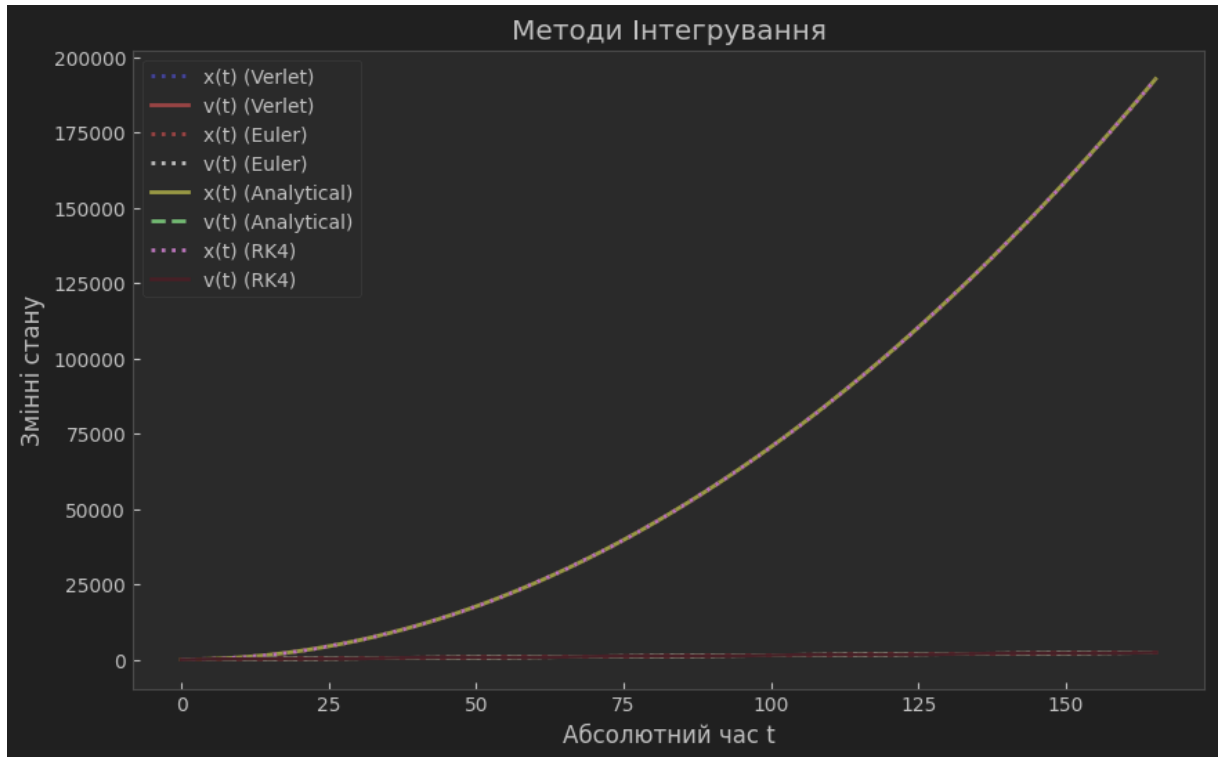


Рисунок 3.32 – Порівняння алгоритмів інтегрування

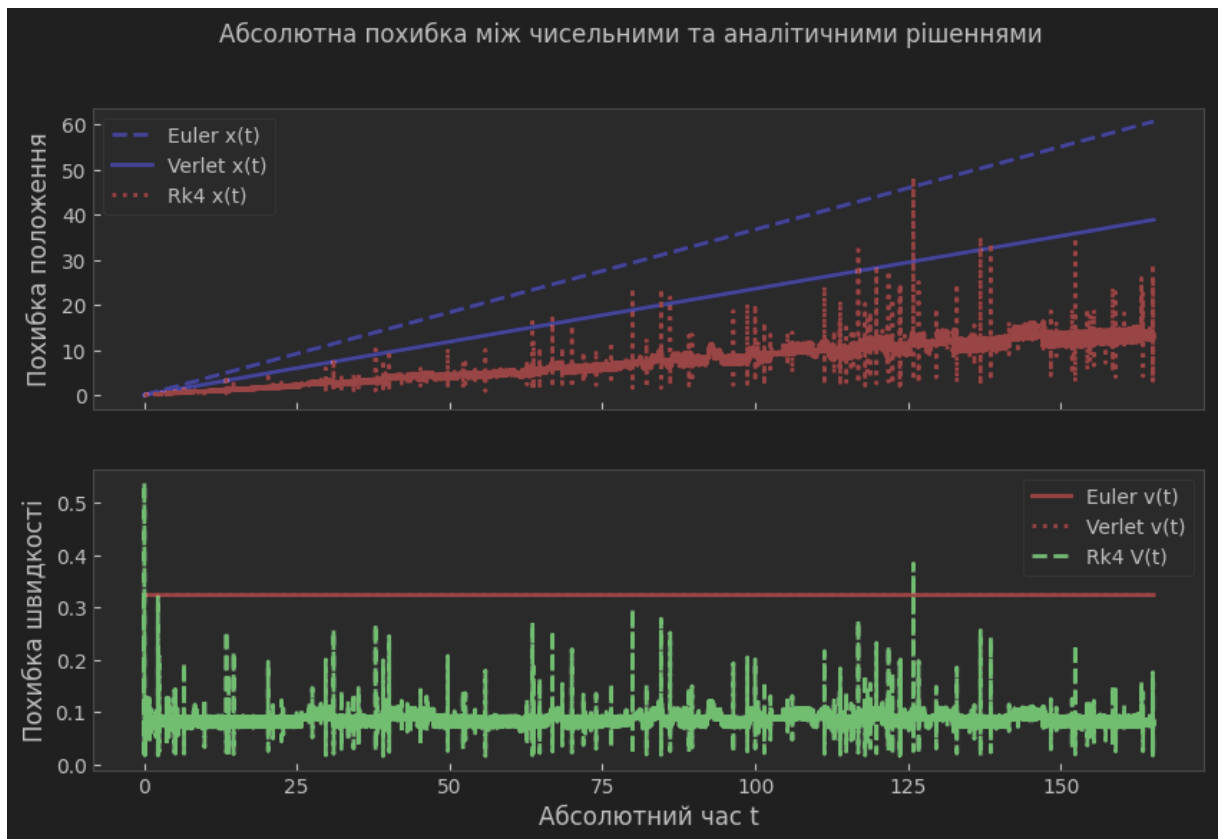


Рисунок 3.33 – Демонстрація похибки

Дивлячись на графіки, можна сказати, що в середньому найточнішим є метод Рунге-Кутта РК4, але метод Верле дає непогану точність, більш

чисельно стабільний для даної задачі. метод Ейлера дає більшу похибку на реальних значеннях проміжків, тому його краще не застосовувати для розв'язку нашої задачі. Далі наведено більш деталізовані графіки, які підтверджують похибку зверху.

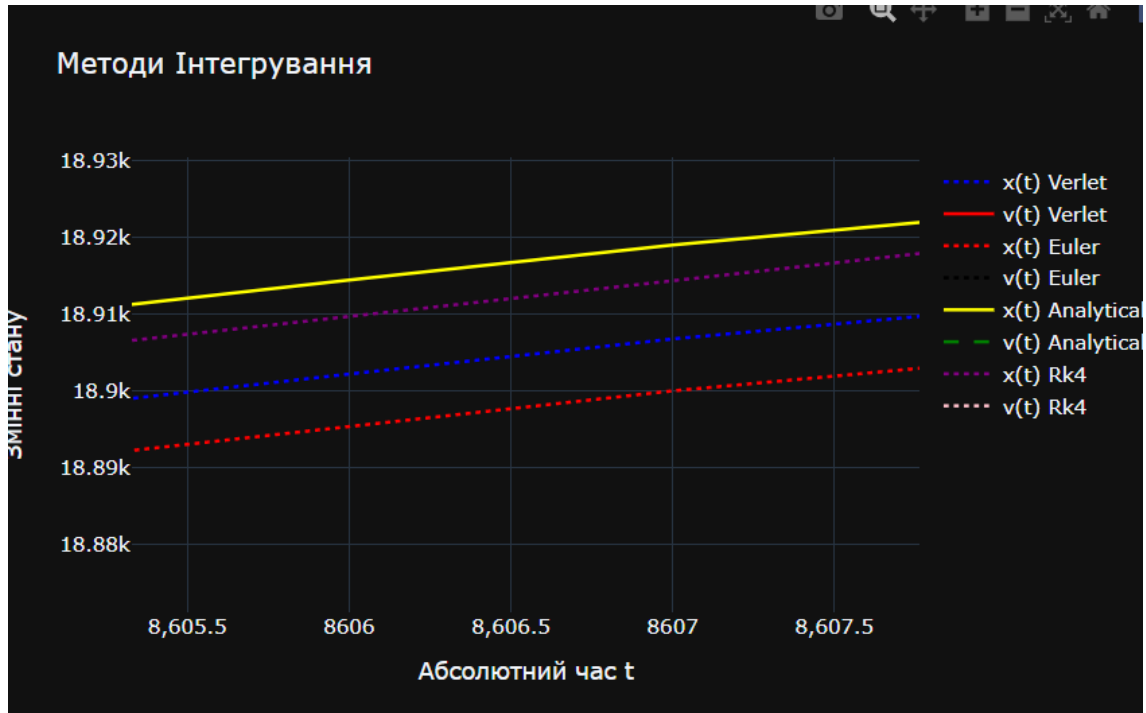


Рисунок 3.34 – Деталізовані графіки шляху

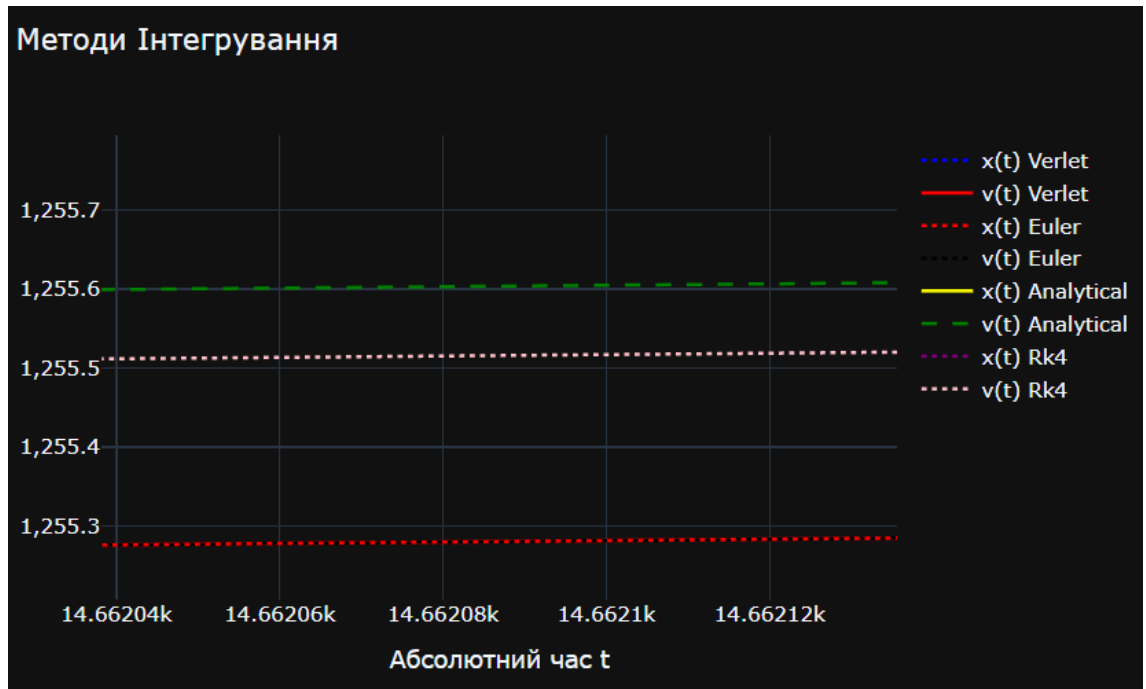


Рисунок 3.35 – Деталізовані графіки швидкості



Рисунок 3.36 – Агрегована оцінка похибки різних методів

Отже, у разі, якщо не потрібно мати дуже велику точність обчислень, але потрібно зберегти час, то метод Верле добре підходить для наближеного розв’язку диференціальних рівнянь.

3.3 Проєктування, архітектура та запуск додатка

3.3.1 Деплой артефактів

Сам рушій розгортається як артефакт на Maven Central або як окремий Jar-архів. Після цього можна налаштувати конвеєр CI/CD за допомогою будь-якого хмарного сервісу, що підтримує цей тип функціональності, наприклад, Amazon AWS. Ця процедура зменшує складність тестування та інтеграції нових змін, а також дозволяє просто підключити його до вашого проєкту і завершити розробку.

3.3.2 Фінальна розробка додатків

Розробка кінцевого користувача відбувається за певним алгоритмом:

1. Створіть проєкт на Maven або Gradle

2. Підключіть наступні артефакти [Додаток А]
3. Створіть головний клас та скопіюйте код [Додаток Б].
4. Розробіть кастомні класи для вашого варіанту використання
5. Запустіть у виробництво як .Jar файл і запустіть за допомогою вашого JDK (21 +) Запуск програми дещо відрізняється і залежить від вашої JVM, операційної системи, шляхових змінних, але має виглядати приблизно так:
Для Віндоус 11/10:

```
java -enable-preview -jar TourmalineEngine.jar
```

Рисунок 3.37 – Виконання CMD-скрипта для запуску тестового класу

На даний момент це програмне забезпечення працює і доступне як для ОС Linux, так і для ОС Windows. Такий підхід виключає дорогі перезбірки C++ і вивільняє інтелектуальні ресурси, необхідні для забезпечення кросплатформної сумісності.

Моїм конкретним додатком був авіасимулятор. Це досить складне завдання, яке вимагає тисячі рядків коду, якщо розробляти його з нуля. Початковий етап розробки – це підготовка ресурсів. Нашій програмі потрібна принаймні одна модель літака з окремими рухомими частинами, що мають правильні назви.

Після розміщення їх у потрібних папках ресурси завантажуються та збираються. Якщо потрібно, налаштовуються конфігуровані властивості. Можна налаштувати світло. Наступним кроком є розробка власних класів для створення симуляції та створення моделей, які відображатимуть зміни стану симуляції.

Ракети використовують криві Безьє для генерації траєкторій. Це простий і доступний метод апроксимації кривими. Крім того, його досить просто реалізувати для 3 вимірів.

Це те, як виглядає апроксимація в більшості випадків. Числова похідна використовується для керування напрямком боєприпаса.

```
public Vector3f interpolate(float t) {
    if (t <= 0) return new Vector3f(referencePoints.getFirst());
```

```

    if (t >= 1) return new Vector3f(referencePoints.getLast());

    Vector3f result = new Vector3f();
    int n = referencePoints.size() - 1;

    for (int i = 0; i <= n; i++) {
        float binomialCoeff = (float) binomialCoefficient(n, i);
        float term = binomialCoeff * (float) (Math.pow(1 - t, n - i) *
            Math.pow(t, i));
        result.add(new Vector3f(referencePoints.get(i)).mul(term));
    }

    return result;
}

public Vector3f derivative(float t0, float dt){
    Vector3f f0 = interpolate(t0);
    Vector3f f1 = interpolate(t0 + dt);

    Vector3f derivative = f1.sub(f0);
    derivative.div(dt);

    return derivative;
}

```

Рисунок 3.38 – Код кривої Безьє

Для того, щоб створити більш реалістичну симуляцію, тензор інерції та маса вираховуються повторно.

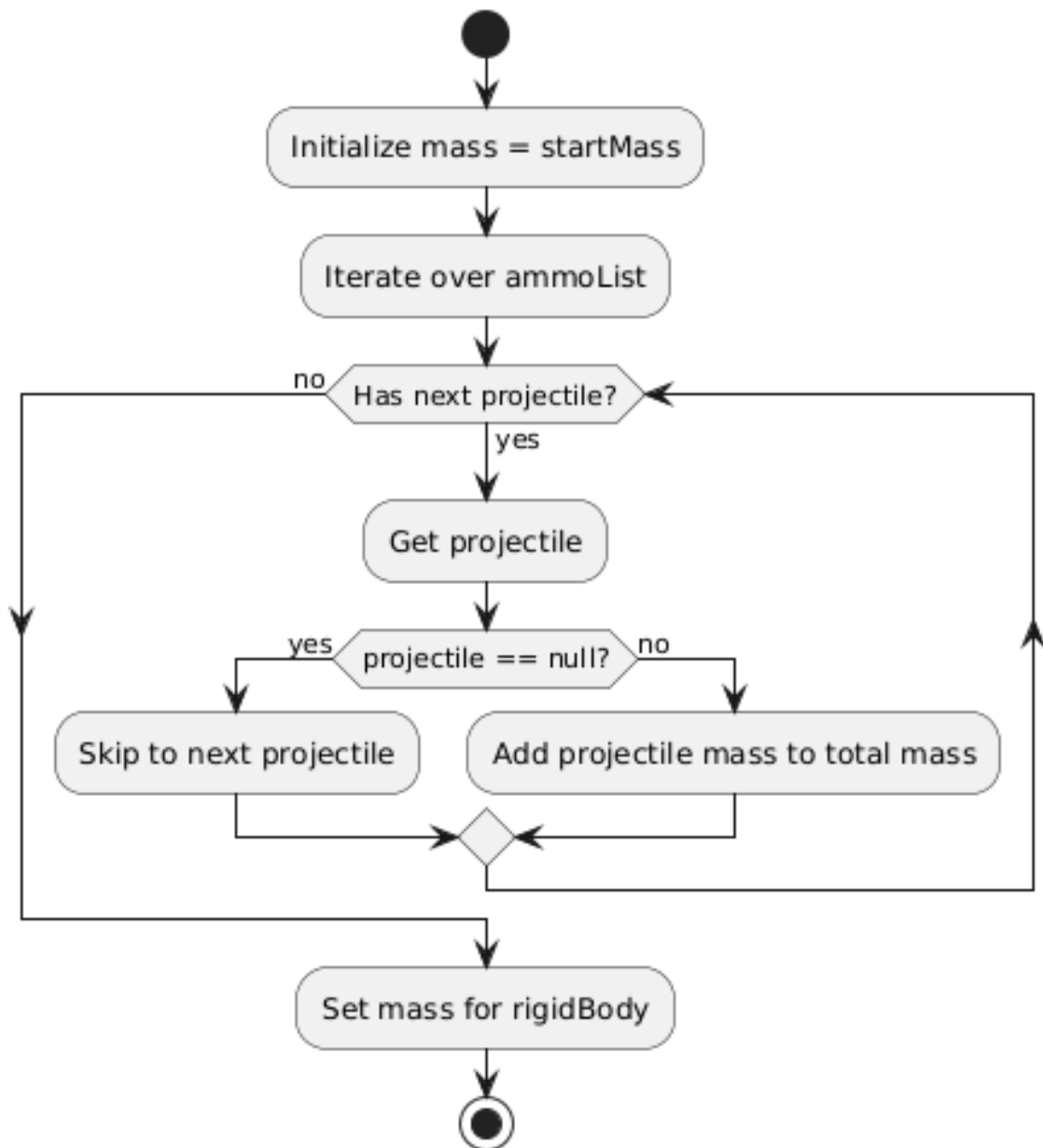


Рисунок 3.39 – Оновлення маси

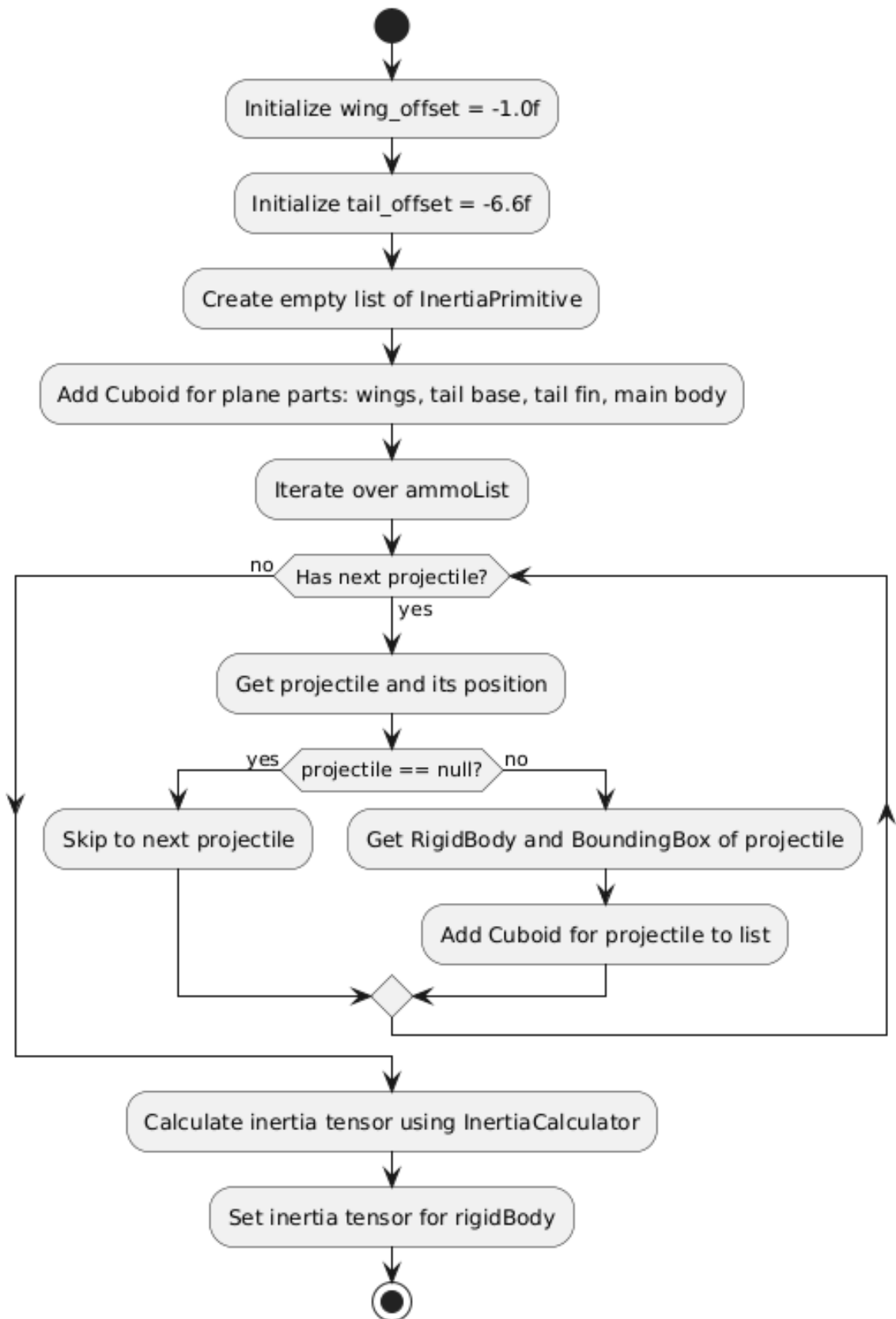


Рисунок 3.40 – Оновлення інерції

Така концепція дозволяє наочно побачити, що літак прискорюватиметься швидше після скиду об'єктів. Це також стосується того, як просто літаку повертатися в ту чи іншу сторону.



Рисунок 3.41 – Демонстрація тестового додатка

3.3.3 Проєктне планування

Під час розробки я використовував типовий Agile-підхід з тим лише винятком, що я мав дуже детальну концепцію проєкту і намагався врахувати всі нюанси та особливості з самого початку. Ця детальність робить його більш схожим на WaterFall. Я значною мірою покладався на YouTrack від JetBrains, щоб відстежувати прогрес, ставити завдання, створювати історії користувачів, епопеї та перелічувати помилки. Оскільки цей фреймворк великий, ми змушені думати про принципи SOLID, щоб забезпечити юзабіліті, ремонтпридатність.

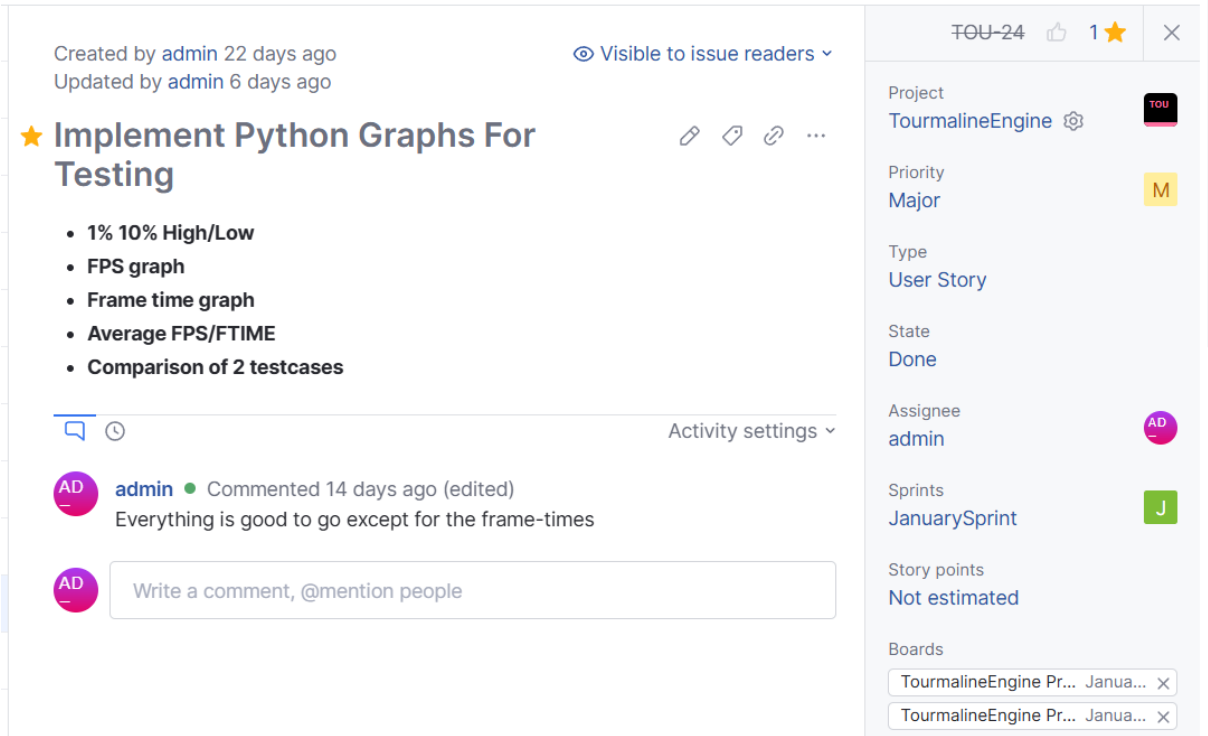


Рисунок 3.42 – Приклад завдання

Проект реалізує тип архітектури, який називається модульний моноліт. Це дає завжди перевагу над монолітом, оскільки проект є більш гнучким і придатним для багаторазового використання, а також більш гнучким.

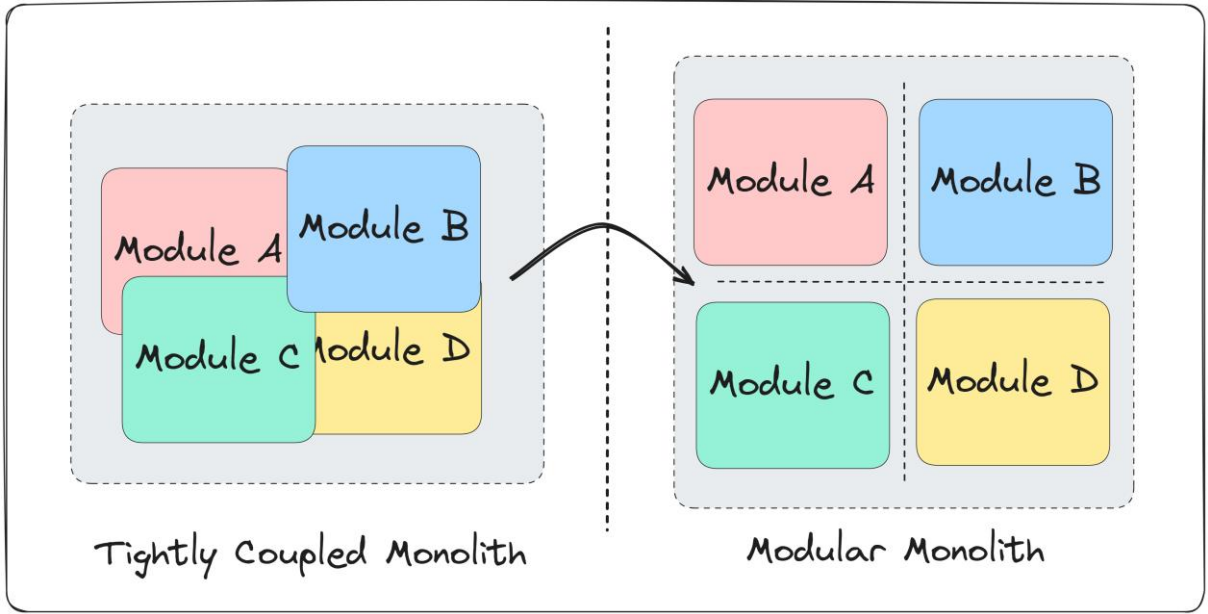


Рисунок 3.43 – Діаграма модульного моноліту [33]

РОЗДІЛ 4. ТЕСТУВАННЯ ТА АНАЛІЗ

4.1 Метрики продуктивності

Найважливішими тут є частота кадрів і тривалість кадрів. Ми повинні стежити за завантаженням GPU, CPU, а також за споживанням RAM і VRAM.

Таблиця 4.1 – Характеристики тестових комп’ютерів

Характеристика	Test Stand 1	Test Stand 2	Test Stand 3
CPU	AMD Ryzen 7 7735HS	AMD FX-4300	AMD Ryzen 7 7735HS
GPU	NVIDIA RTX 4050 Laptop GPU	NVIDIA GTX 960	AMD VEGA 8
Відеопам’ять	6 GB GDDR6	2 GB GDDR5	–
Оперативна пам’ять (RAM)	16 GB DDR5 4800 MHz	4 GB DDR3 1600 MHz	16 GB DDR5 4800 MHz
ОС	Windows 11 23H2	Ubuntu 24.10	Windows 11 23H2

Протягом тестування, було задіяно близько 3-4 гігабайтів оперативної пам'яті та близько 1200 Мбайт відеопам'яті. При цьому рушій більше користувався ресурсами відеоприскорювача, завантажуючи ЦП лише на 15-20%. Це саме по собі не є поганим, але можна оптимізувати додаток, перенесши повторювані обчислення на центральний процесор.

4.2 Результати тестів

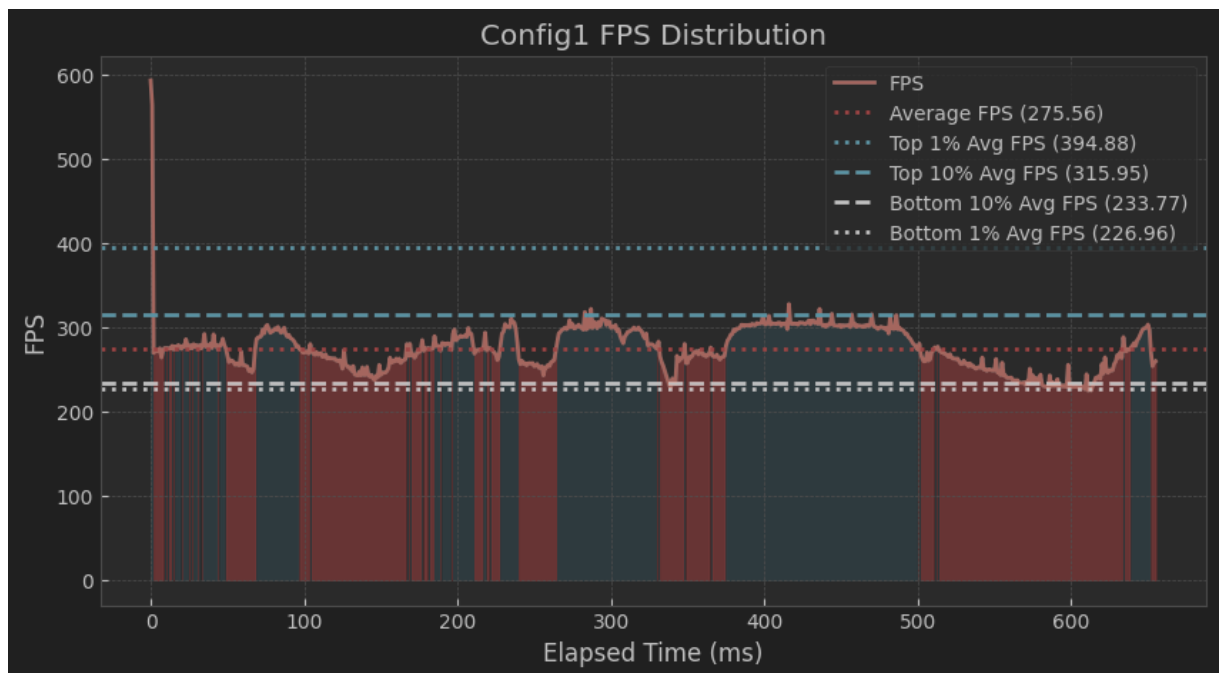


Рисунок 4.1 – Детальний аналіз продуктивності на першому тестовому ПК

Наведений вище графік відображає продуктивність гри з часом, показує середні значення fps, верхні та нижні 10 та 1 відсотки відповідно. Падіння на початку гри, ймовірно, спричинене ініціалізацією OpenGL. Періоди, коли FPS був вищим за середній, позначено синім кольором, а пропуски кадрів – червоним. Для оцінки нестабільності використовується індекс тремтіння.

Jitter Index (Windows): 10.074909472694626

Jitter Index (Linux): 5.557789510249336

Jitter Index (Windows + Vega 8): 3.8530139734883577

Такі значення є задовільними, але все ж 10.07 це трохи забагато для першого профілю. Варто було би розглянути заміну процесора або оптимізувати роботу для перекладення навантаження із нього.

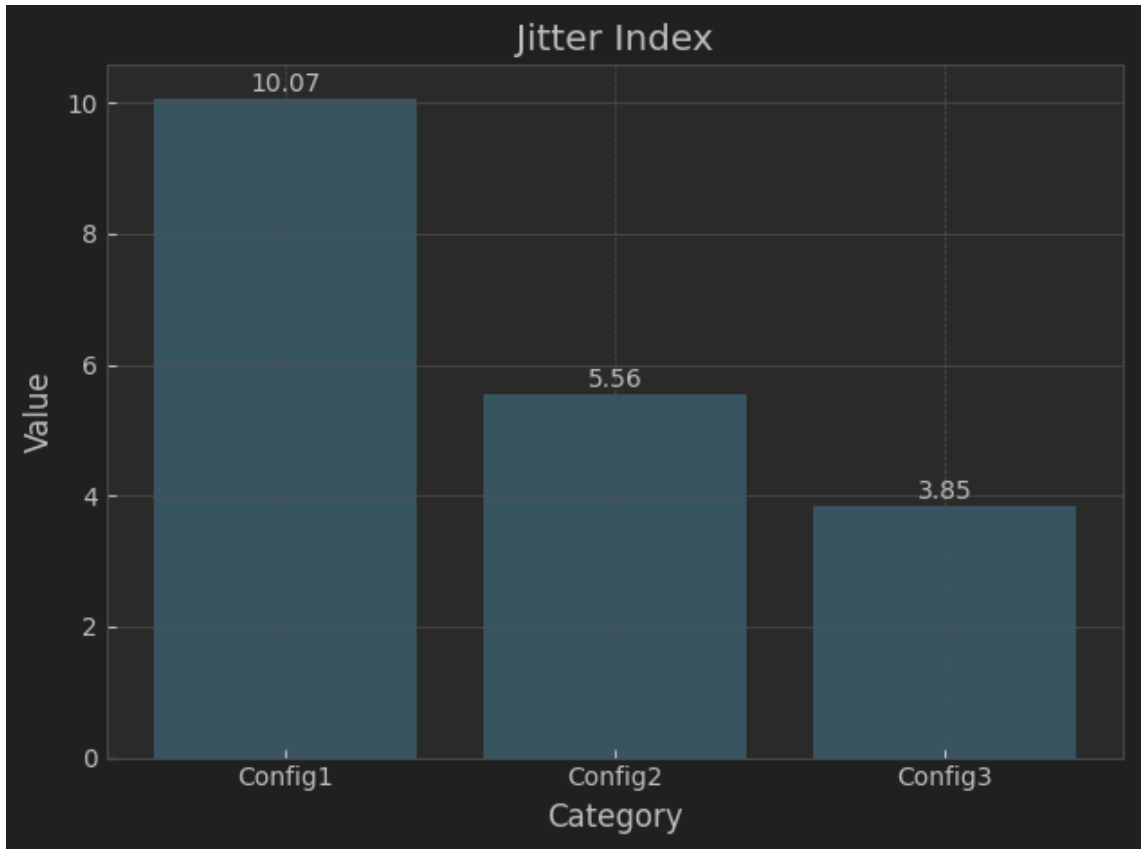


Рисунок 4.2 – Індекс Тремтіння

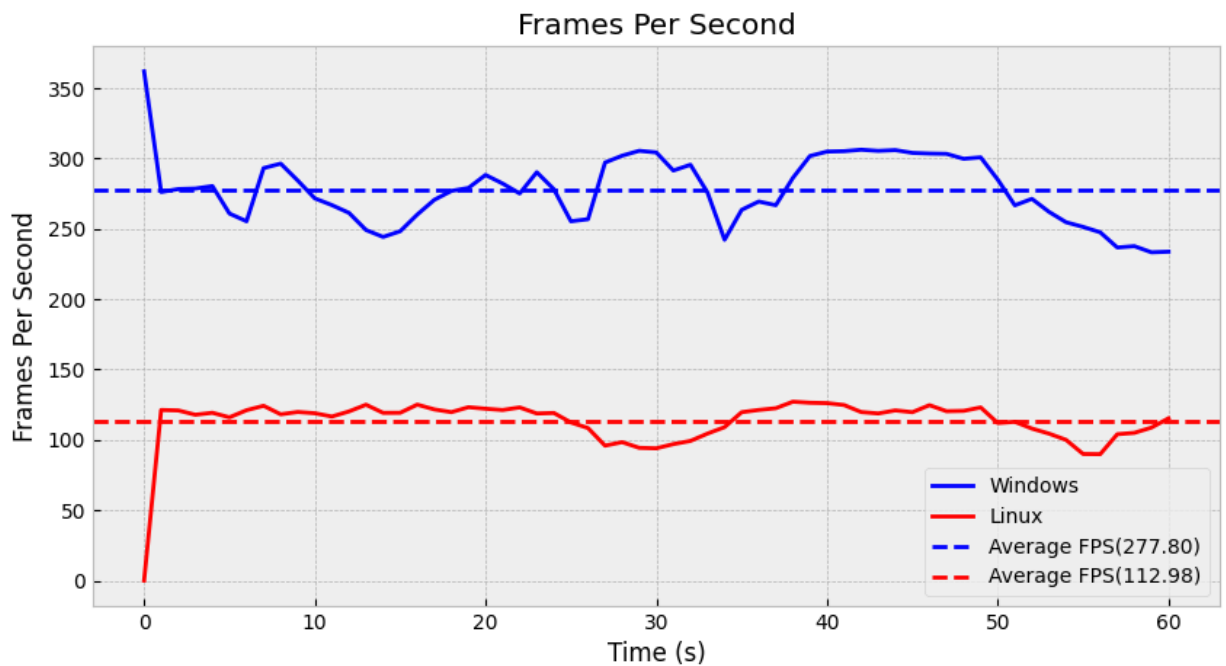


Рисунок 4.3 – Порівняння продуктивності на двох різних системах

На даному графіку наочно представлено порівняння двох операційних систем, на яких працює додаток на основі TourmalineEngine. Тестування на

різних системах дозволило нам перевірити стабільність та кросплатформеність наших систем.

Цей графік відображає періоди часу, коли FPS зростав і падав. Він може бути корисним для більш складного аналізу продуктивності.

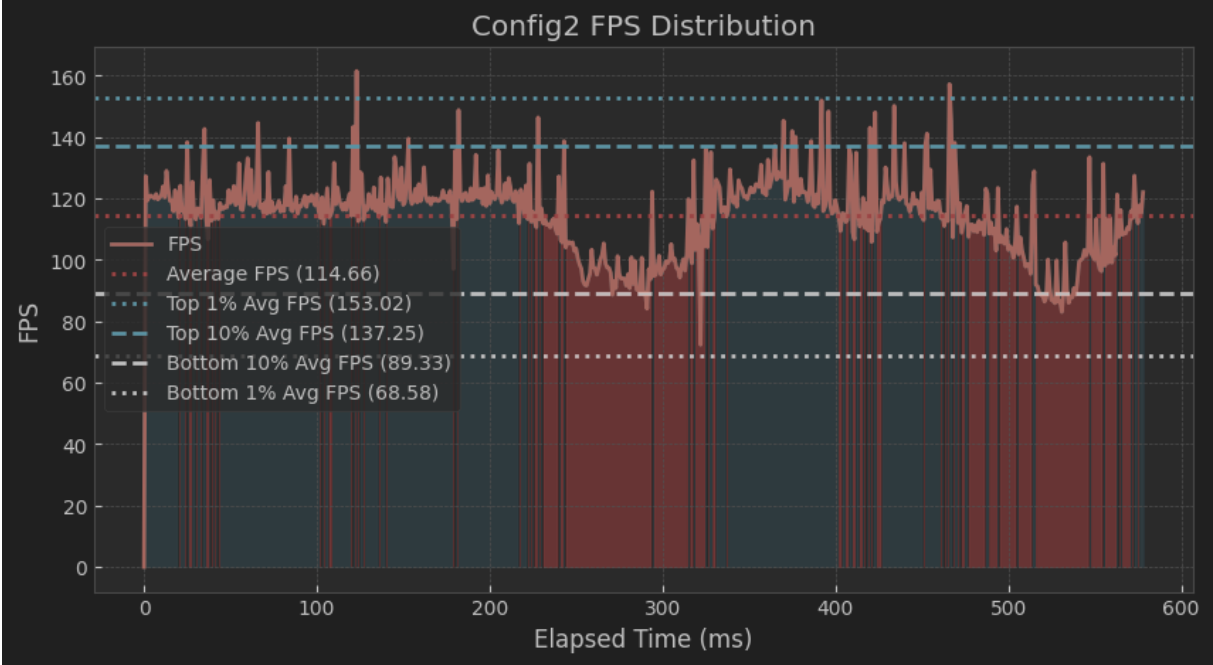


Рисунок 4.4 – Детальний аналіз продуктивності на другому тестовому ПК

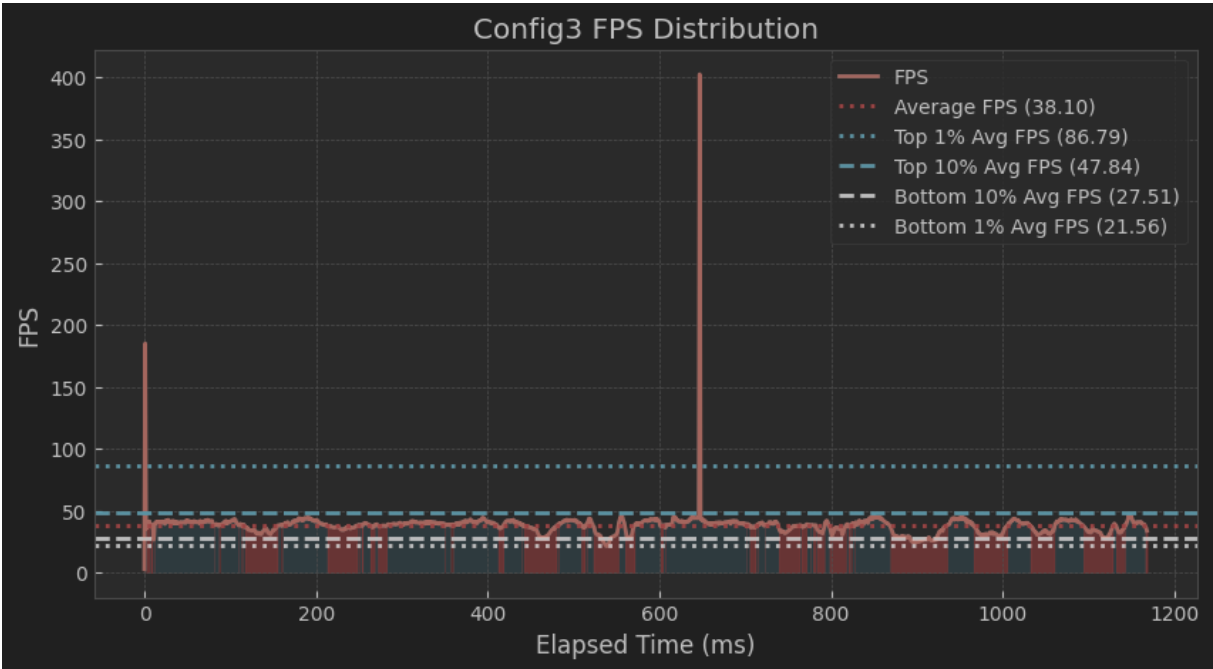


Рисунок 4.5 – Детальний аналіз продуктивності на третьому тестовому ПК



Рисунок 4.6 – Порівняння всіх трьох профілів

Тестові сценарії були не абсолютно ідентичними, тому по суті просто відображають загальну продуктивність та не відображають точної кореляції.

Швидкість завантаження ресурсів, в свою чергу, виглядала наступним чином:

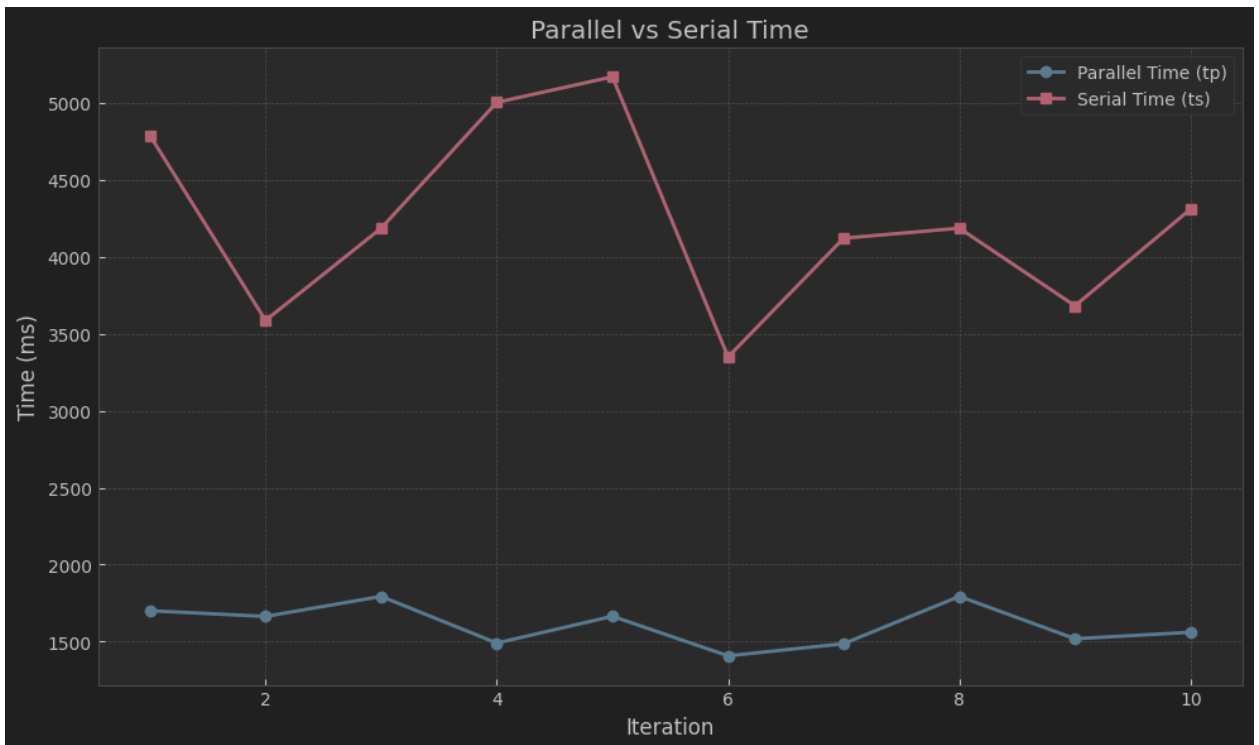


Рисунок 4.7 – Порівняння швидкості завантаження

В середньому ми вигравали 2628.8 мс, що є добрим результатом, особливо беручи до уваги використання лише потоків. Особливо вирізняється те, що перше завантаження було найдовшим, а ресурси в свою чергу

завантажуються лише один раз. Я би пояснив це внутрішніми оптимізаціями віртуальної машини або ядра ОС. Прискорення коливалося в діапазоні від 2 до 3.5, а ефективність – 0.38-0.56. Це є закономірним результатом для програми із псевдопаралельним виконанням операцій.

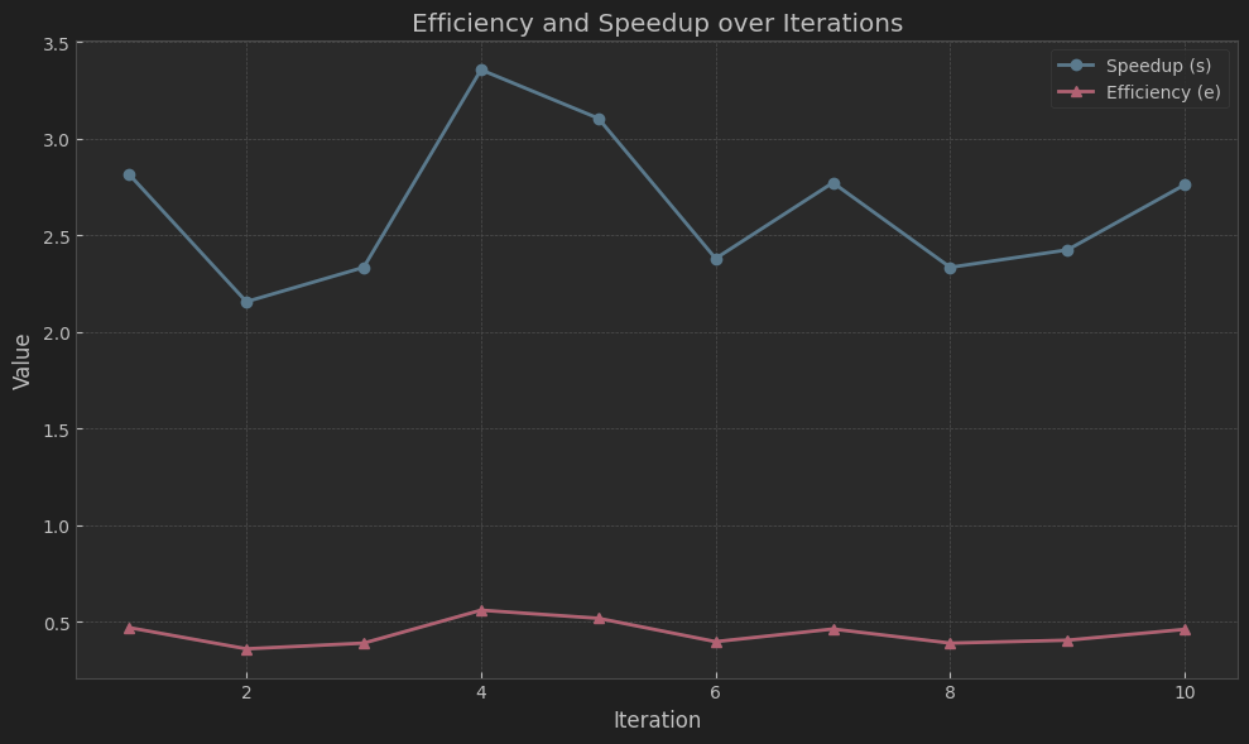


Рисунок 4.8 – Порівняння ефективності та прискорення

Таблиця 4.2 – Швидкість завантаження

Ітерація	Паралельний час	Послідовний час	Різниця часу	Прискорення	Ефективність
1	1701	4787	-3086	2.814227	0.469038
2	1664	3586	-1922	2.155048	0.359175
3	1794	4185	-2391	2.332776	0.388796
4	1491	5002	-3511	3.354795	0.559133
5	1665	5168	-3503	3.103904	0.517317
6	1408	3349	-1941	2.378551	0.396425
7	1487	4120	-2633	2.770679	0.46178
8	1794	4185	-2391	2.332776	0.388796
9	1519	3681	-2162	2.423305	0.403884
10	1561	4309	-2748	2.76041	0.460068

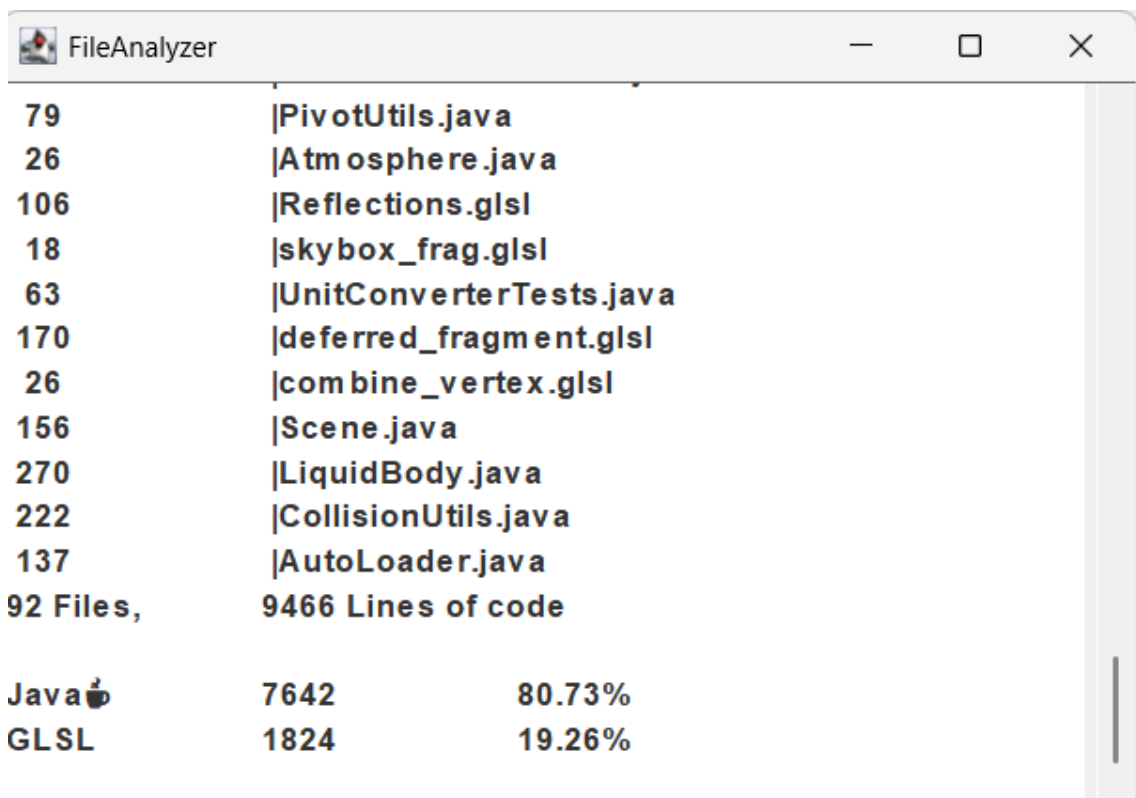
Окрім цього, було виконано цілий ряд інтеграційних та модульних тестів, які виконуються без помилок та підтверджують стабільність та якість бази коду даного продукту. Також перевіряються окремі модулі – Tourmaline Graphics та Quartz Physics.

4.3 Висновки тестування

Під час тестування не було виявлено жодних проблем із завантаженням ресурсів: всі моделі, текстури та інші необхідні дані завантажувались стабільно та без збоїв. Жодних зависань або критичних помилок виконання виявлено не було. Час показу кадру залишався досить стабільним і плавним, особливо якщо враховувати середнє значення FPS у кожному з тестових запусків. Це підтверджує ефективність поточної архітектури рендерингу та ресурс-менеджменту.

4.4 Аналіз коду

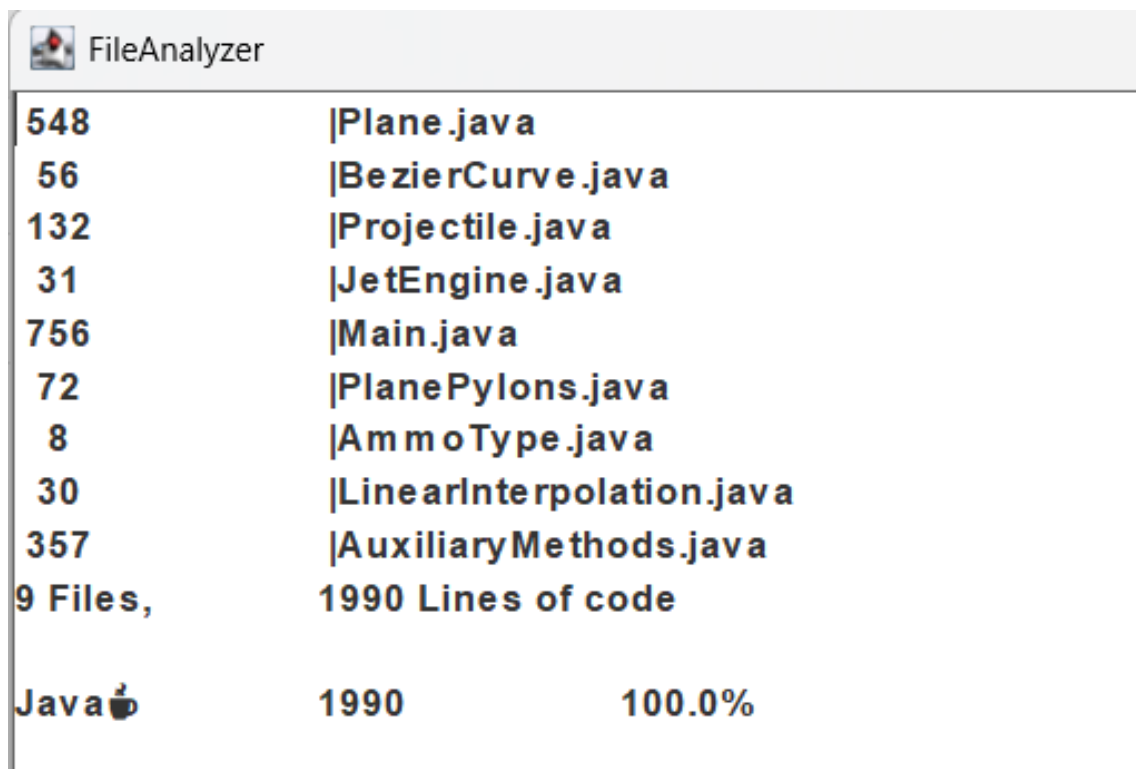
У підсумку головну задачу такого рушія – скорочення коду – вдалося успішно виконати. Використовуючи власну утиліту (Додаток А), я визначив загальну кількість коду. Тут врахований код обох модулів: графіки та фізики. Також включені тести, шейдерні програми. Але не включені моделі, конфіг-файли та інші файли, які напряду не пов'язані із процесом компіляції у якості source.



79	PivotUtils.java		
26	Atmosphere.java		
106	Reflections.glsl		
18	skybox_frag.glsl		
63	UnitConverterTests.java		
170	deferred_fragment.glsl		
26	combine_vertex.glsl		
156	Scene.java		
270	LiquidBody.java		
222	CollisionUtils.java		
137	AutoLoader.java		
92 Files,		9466 Lines of code	
Java	7642	80.73%	
GLSL	1824	19.26%	

Рисунок 4.11 – Загальна кількість коду рушія

Код фінального додатка склав 1990 рядків коду, що сильно збереже сили, час та гроші фінальним розробникам.



548	Plane.java		
56	BezierCurve.java		
132	Projectile.java		
31	JetEngine.java		
756	Main.java		
72	PlanePylons.java		
8	AmmoType.java		
30	LinearInterpolation.java		
357	AuxiliaryMethods.java		
9 Files,		1990 Lines of code	
Java	1990	100.0%	

Рисунок 4.12 – Загальна кількість коду авіасимулятора

Варто відзначити, що за рахунок вдалих ієрархій наслідування, вдалося зробити код, що добре розширюється та модифікується без зміни суперкласів. Такими зокрема були:

TreeNode → MeshTree. Використовує структуру даних n-арне дерево для представлення складених об'єктів.

Loadable, EnhancedLoadable → Mesh, Texture, CubeMap. Використовує універсальні механізми завантаження об'єктів в пам'ять із використанням багатопотокових програм. Легко можна додати ресурси і інших типів. Вони будуть так само працювати та пришвидшувати завантаження.

EventHandler → MouseEventHandler, KeyboardEventHandler. Можна легко додати нові способи керування ігровим процесом, як руль або контролер(joystick).

BaseEffect → BoundingBoxEffect, JetEffect, ExplosionEffect. Існує технічна можливість створювати нові ефекти без зміни вихідного коду

AbstractLight → PointLight, DirectionalLight. Можна підвантажувати нові типи освітлення, хоча для цього знадобиться зміна коду шейдерних програм.

Із недоліків можливо виділити наступне: у порівнянні із більш розвиненими об'єктами, графічний конвеєр досить слабо змінюється на пряму. У Unreal Engine та Unity3D є можливість легко змінювати пайплайн, користуючись вбудованими системами без написання коду. У моєму випадку це можливо лише за наявності вихідного коду та переробки багатьох методів (можливо, через наслідування, але це не дуже зручний метод).

ВИСНОВКИ

У підсумку, цей проект зайняв у мене 360 годин, що дорівнює 12 кредитам ECTS. Я використовував щонайменше три мови програмування: Java, GLSL та Python. Я отримав чудовий досвід роботи з моєю першою DSL для шейдингу. Я вивчив і застосував на практиці широкий спектр алгоритмів, характерних для комп'ютерної графіки. Я впроваджував удосконалення та власні версії існуючих алгоритмів.

Типові ігрові рушії складаються з багатьох елементів, десятків 2D і 3D текстур, безлічі шейдерів, різних об'єктів і оптимізацій. Це означає, що простір для вдосконалень величезний. Якби у мене було більше часу, я б хотів реалізувати об'ємні хмари, більше карт для покращеної підтримки PBR та більше фізичних моделей для різних об'єктів. Я хотів би покращити реалізацію блуму та вдосконалити маппінг тіней і SSR. Вивчення інших варіантів наближення рівняння PBR для пошуку найкращого також є гарною ідеєю. Я б також запровадив реалістичні денні зміни: динамічні скайбокси та зміну положення світла. Я б також приділив особливу увагу розвитку інфраструктури, оскільки вона є основою будь-якого ігрового рушія та успішного проекту в цілому. Якщо у вас відсутні плагіни або інтегровані API для роботи, то немає сенсу в тому, щоб бізнес використовував ваші рішення. Було б чудово додати SteamAPI, OpenAL та інші API. Впровадження скриптової мови, такої як Lua або Scala, може бути гарною ідеєю, щоб забезпечити більш просту модифікацію та покращення функціональності. Також може бути гарною ідеєю створити плагін для IntelliJ Idea, щоб вивести цей фреймворк на комерційний рівень. Процес тестування був успішним і довів кросплатформні можливості «Tourmaline Engine». Quartz Physics продемонстрував чисельну стабільність, точність розрахунків і простоту використання. SDK є простим і може використовуватися для наукових, ігрових та 3D-проектвань.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. David Lettier. 3D Game shaders for Beginners // Lettier. URL: <https://lettier.github.io/3d-game-shaders-for-beginners/lighting.html> (дата звернення: 02.04.2025)
2. Khronos Group. GLSL Specification // khronos. URL: [https://www.khronos.org/opengl/wiki/Core_Language_\(GLSL\)](https://www.khronos.org/opengl/wiki/Core_Language_(GLSL)). (дата звернення: 25.03.2025)
3. Wikipedia. Physically Based Rendering // Wikipedia. URL: https://en.wikipedia.org/wiki/Physically_based_rendering. (дата звернення: 10.05.2025)
4. Northwestern University. Motion from Blur // University Archive. URL: http://users.ece.northwestern.edu/~sda690/MfB/Motion_CVPR08.pdf. (дата звернення: 14.04.2025).
5. Disney. Physically Based Shading at Disney. URL: https://media.disneyanimation.com/uploads/production/publication_asset/48/asset/s2012_pbs_disney_brdf_notes_v3.pdf. (дата звернення: 26.04.2025).
6. Jakob Maier. Flight Simulation // jakobmaier. URL: <https://www.jakobmaier.at/posts/flight-simulation/#creating-the-flightmodel>. (дата звернення: 26.04.2025).
7. Stan Fortoński. Water Simulation // github. URL: <https://github.com/stanfortonski/Ocean-OpenGL>. (дата звернення: 26.04.2025).
8. Derek Whatley. Water Simulation // github. URL: <https://github.com/djwhatle/glsl-water-sim>. (дата звернення: 28.03.2025).
9. Jocelyn Frechot. REALISTIC SIMULATION OF OCEAN SURFACE USING WAVE SPECTRA // semantic scholar. URL: https://pdfs.semanticscholar.org/9af0/89e011f774aa2bf02643325312b0fdcbb213.pdf?_gl=1*1hm5icl*_gcl_au*MTI1ODUxNTczNS4xNzMyNTY5Mjkw*_ga*OTE1NzY5NDI5LjE3MzI1NjkyOTA.*_ga_H7P4ZT52H5*MTczMjU2OTI4OS4xLjA1uMTczMjU2OTI5MC41OS4wLjA. (дата звернення: 28.03.2025).

10. Screen Space Reflections. Casual Effects Blog. URL: <https://casual-effects.blogspot.com/2014/08/screen-space-ray-tracing.html>. (дата звернення: 14.04.2025).
11. Advanced Ligting Tecniques // LearnOpenGL. URL: <https://learnopengl.com/Advanced-Lighting/Bloom>. (дата звернення: 14.04.2025).
12. Skybox Generation AI // BlockadeLabs URL: <https://skybox.blockadelabs.com/49f50bf4fe79be07d95dec86d8014ceb>. (дата звернення: 28.03.2025).
13. Gradient Types // Geeks For Geeks URL: <https://www.geeksforgeeks.org/gradients-in-figma/>. (дата звернення: 26.04.2025).
14. Volumetric Lighting demonstrarion // Unity Forum URL: <https://discussions.unity.com/t/released-hx-volumetric-lighting/635063>. (дата звернення: 28.03.2025).
15. Gaussian Blur // Wikipedia URL: https://en.wikipedia.org/wiki/Gaussian_blur. (дата звернення: 28.03.2025).
16. Graphics and Multimedia Programming Lectures // Philipps-Marburg Universität. URL: <https://www.uni-marburg.de/en/fb12/research-groups/grafikmultimedia/lectures/graphics>. (дата звернення: 14.04.2025).
17. Vertex shader implementation of Gerstner waves // Max madblade, GitHub URL: <https://github.com/madblade/waves-gerstner>. (дата звернення: 14.04.2025).
18. Efficient Soft-Edged Shadows Using Pixel Shader Branching // GPU Gems 2, Nvidia URL: <https://developer.nvidia.com/gpugems/gpugems2/part-ii-shading-lighting-and-shadows/ПОЗДІІ-17-efficient-soft-edged-shadows-using>. (дата звернення: 28.03.2025).
19. SkyBox unwrapping // Stack Overflow URL: <https://stackoverflow.com/questions/29678510/convert-21-equirectangular-panorama-to-cube-map> (дата звернення: 08.04.2025).
20. Custom 3D model generation // Meshy URL: <https://www.meshy.ai/workspace>. (дата звернення: 28.03.2025).

21. Real-Time Collision Detection // Christer Ericson Sony Computer Entertainment America (дата звернення: 08.04.2025)
22. Introduction to Aircraft Flight Mechanics: Performance, Static Stability, Dynamic Stability, and Classical Feedback Control // Thomas R. Yechout with Steven L. Morris David E. Bossert Wayne F. Hallgren. (дата звернення: 29.03.2025).
23. Moments of inertia // javalab URL: https://javalab.org/en/moment_of_inertia_2_en/ (дата звернення: 05.04.2025)
24. Force // Wikipedia URL: <https://en.wikipedia.org/wiki/Force> (дата звернення: 28.03.2025).
25. Torque // Wikipedia URL : <https://en.wikipedia.org/wiki/Torque> (дата звернення: 28.03.2025).
26. K-ary tree illustration // Researchgate URL: https://www.researchgate.net/Рисунок/The-complete-k-ary-tree-Tkrdocumentclass12ptminimal-usepackageamsmath_fig1_334410817 (дата звернення: 20.05.2025).
27. K-means clustering // Medium URL: <https://medium.com/intro-to-artificial-intelligence/k-means-clustering-f9910da87c43>. (дата звернення: 29.03.2025).
28. PCA // Scaler URL: <https://www.scaler.com/topics/nlp/what-is-pca/>
29. Projection matrices // songho.ca URL: https://www.songho.ca/opengl/gl_projectionmatrix.html. (дата звернення: 29.03.2025).
30. Attenuation function // lisyarus.github.io URL: <https://lisyarus.github.io/blog/posts/point-light-attenuation.html>. (дата звернення: 29.03.2025).
31. Hello Triangle. // learnopengl.com URL: <https://learnopengl.com/Getting-started/Hello-Triangle> (дата звернення: 05.04.2025)
32. Substance Painter Guide // adobe.com URL: <https://helpx.adobe.com/substance-3d-painter/home.html> (дата звернення: 05.04.2025)

33. What is Spring Modulith? // linkedin.com URL: <https://www.linkedin.com/pulse/what-spring-modulith-hushen-savani-kfkjf/>
(дата звернення 01.06.2025)
34. Comparing Popular Game Engines // pubnub URL: <https://www.pubnub.com/blog/comparing-popular-game-engines/>
(дата звернення 01.03.2025)
35. Unreal Engine Information // unrealengine.com URL: <https://www.unrealengine.com/en-US> (дата звернення 01.03.2025)
36. Godot Engine Information // godotengine.org URL: <https://godotengine.org/>
(дата звернення 01.03.2025)
37. Unity Engine Information // unity.com URL: <https://unity.com/> (дата звернення 01.03.2025)
38. CryEngine Information // cryengine.com URL: <https://www.cryengine.com/>
(дата звернення 01.03.2025)

ДОДАТОК А

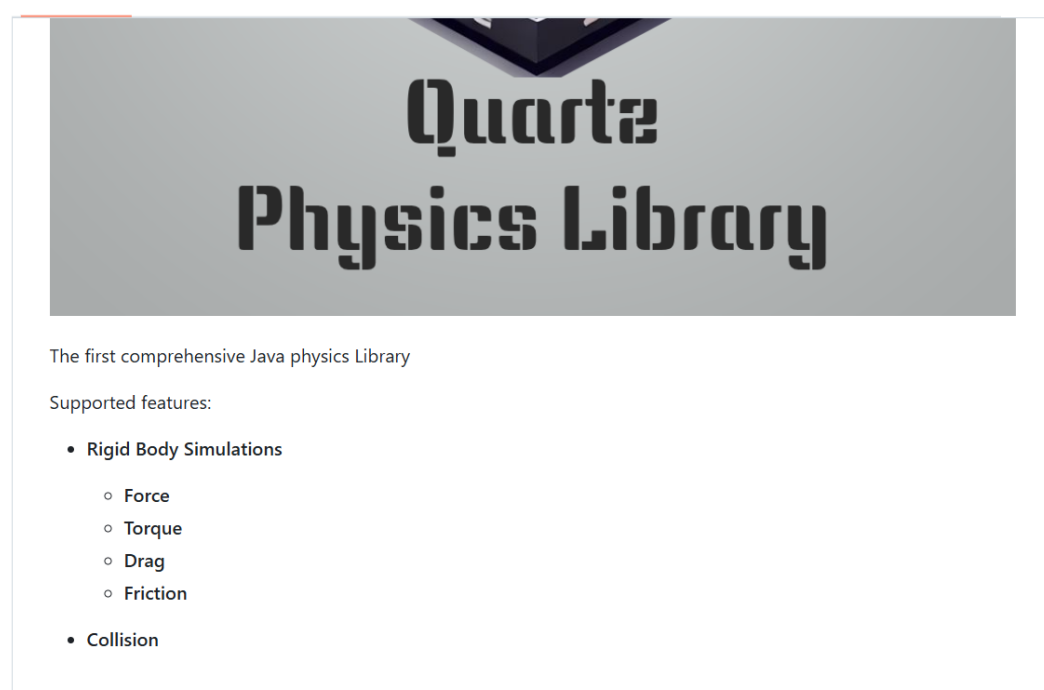
Додаток А.1 – Посилання репозиторій із графічним рушієм

<https://github.com/emeraldDevUA/TourmalineEngine>



Додаток А.2 – Посилання репозиторій із фізичною бібліотекою

<https://github.com/emeraldDevUA/QuartzPhysics>



Додаток А.3 – Посилання репозиторій із утілтою для аналізу файлів.

<https://github.com/emeraldDevUA/FileAnalyzer>

 **emeraldDevUA** Add files via upload

7031fe3 · 2 years ago 8 Commits

 FileAnalyzer.jar	Add files via upload	2 years ago
 LICENSE	Initial commit	2 years ago
 Main.java	Add files via upload	2 years ago
 preview1.jpg	Add files via upload	2 years ago
 preview2.jpg	Add files via upload	2 years ago

 README

 MIT license



Add a README

Help people interested in this repository understand your project by adding a README.

Add a README

ДОДАТОК Б

Рисунок Б.1 – Перелік принципів SOLID

S — Single Responsibility Principle (Принцип єдиної відповідальності)

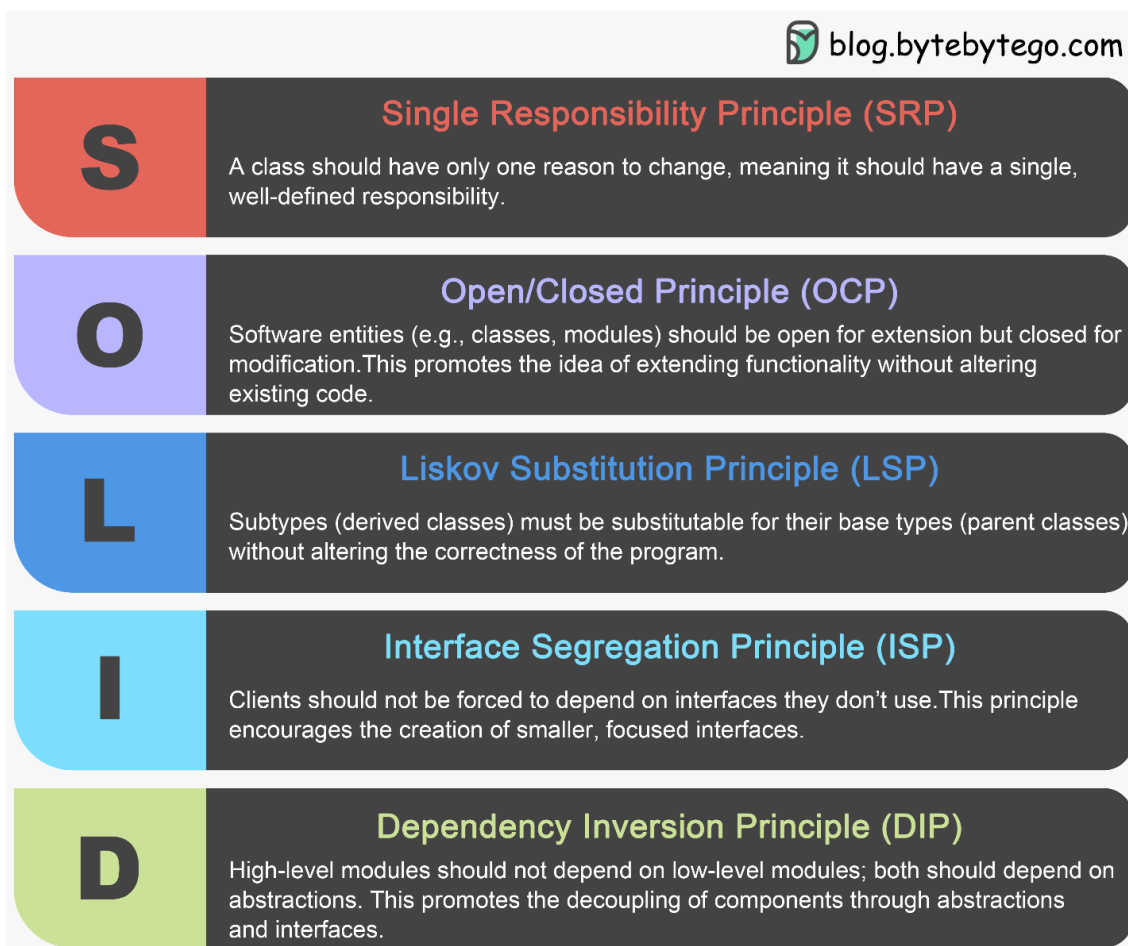
O — Open/Closed Principle (Принцип відкритості/закритості)

L — Liskov Substitution Principle (Принцип підстановки Лісков)

I — Interface Segregation Principle (Принцип розділення інтерфейсу)

D — Dependency Inversion Principle (Принцип інверсії залежностей)

Рисунок Б.2 – Візуалізація принципів SOLID



[<https://blog.bytebytego.com/p/mastering-design-principles-solid>]

ДОДАТОК В

Лістинг В.1 – Приклад основного класу.

```

package org.tourmaline;
import Annotations.BasicWindow;
import Controls.Keyboard;
import Rendering.Camera;
import Rendering.Lights.DirectionalLight;
import Rendering.Lights.LightingConfigurator;
import Rendering.Scene;
import Rendering.SkyBox;
import ResourceImpl.*;
import ResourceLoading.AutoLoader;
import ResourceLoading.ResourceLoadScheduler;

import org.joml.Vector2f;
import org.joml.Vector3f;
import static org.joml.Math.cos;
import static org.joml.Math.sin;

import static org.lwjgl.glfw.GLFW.*;
import static org.lwjgl.opengl.GL11.*;
import static org.lwjgl.opengl.GL30.GL_FRAMEBUFFER;
import static org.lwjgl.opengl.GL30.glBindFramebuffer;

import Annotations.OpenGLWindow;
@OpenGLWindow(windowName = "Complex Example", defaultDimensions
= {1920,1018},
    windowHints = {GLFW_DECORATED},
windowHintsValues={GLFW_TRUE}, shadowMapResolution = 8192)

public class RotatingPlaneDemo extends BasicWindow {

    private static final ResourceLoadScheduler
resourceLoadScheduler = new ResourceLoadScheduler();
    private static final AutoLoader autoLoader =
        new AutoLoader("src/main/resources/3D_Models",
resourceLoadScheduler);
private static float ro = 200;
private static float phi = 0;
public static void main(String[] args) throws Exception {

    init(RotatingPlaneDemo.class);
    scene = new Scene();

    Texture.setVerticalFlip(true);
    Mesh.setUseAssimp(true);
    autoLoader.loadTrees();
    autoLoader.asyncLoad();

```

Продовження 1 – Лістингу В.1

```

while (autoLoader.getReadiness() < 1){
    Thread.onSpinWait();
}
resourceLoadScheduler.reset();
MeshTree Fl6Tree = autoLoader.get("MIG29");
Fl6Tree.compile();
Vector3f scaleVector = new Vector3f(50);
Material mainMat = Fl6Tree.getNodeValue().getMaterial();
mainMat.addProperty(Material.METALNESS, 0.0);
mainMat.addProperty(Material.ROUGHNESS, 0.8);
Fl6Tree.traverse(mesh -> {
    mesh.setScale(scaleVector);
    mesh.setEnableReflection(false);
});
Fl6Tree.getChildNodes().forEach(meshTreeNode -> {
    if(!meshTreeNode.getNodeName().equals("%s_glass.obj".formatted("
Fl6"))) {
        ((MeshTree) (meshTreeNode))
            .traverse(mesh -> {
                mesh.getMaterial().close();
                mesh.setMaterial(mainMat);
                mesh.setIncidenceAtZero(new
Vector3f(0.04f));
            });
        } else {
            meshTreeNode.getNodeValue().setEnableBlending(true);
        }
    })
Fl6Tree.setPosition(new Vector3f(0,-20,0));
deferredShader = new Shader(
"src/main/glsl/deferred_shaders/deferred_vertex.glsl",
"src/main/glsl/deferred_shaders/deferred_fragment.glsl");
postprocessingShader = new Shader(
"src/main/glsl/postprocessing_shaders/postprocessing_vertex.glsl",
"src/main/glsl/postprocessing_shaders/postprocessing_fragment.glsl");
combineShader = new Shader(
"src/main/glsl/combine_shaders/combine_vertex.glsl",
"src/main/glsl/combine_shaders/combine_fragment.glsl");
shadowMappingShader = new Shader(
"src/main/glsl/shadow_shaders/shadow_vertex.glsl",
"src/main/glsl/shadow_shaders/shadow_fragment.glsl");
visualEffectsShader = new Shader(
"src/main/glsl/visualeffects_shaders/visual_effects_vertex.glsl",
"src/main/glsl/visualeffects_shaders/visual_effects_fragment.glsl");

```

Продовження 2 – Лістингу В.1

```

skyBoxShader = new Shader(
"src/main/glsl/skybox_shaders/skybox_vertex.glsl",
"src/main/glsl/skybox_shaders/skybox_frag.glsl");

transparentShader = new Shader(
"src/main/glsl/vertex_test.vert",
"src/main/glsl/fragment_test.frag");

SkyBox skyBox = new SkyBox(
new CubeMap("src/main/resources/skybox/skybox", ".hdr", false),
new CubeMap("src/main/resources/skybox/radiance", ".hdr",
false),
new CubeMap("src/main/resources/skybox/irradiance", ".hdr",
false)
);
skyBox.compile();
camera = new Camera(
    new Vector3f(5,1,3).normalize().mul(180),
    new Vector3f(0,0,0));

shadowCamera = new Camera(
    new Vector3f(-90,120,20),
    new Vector3f(0,0,0));
camera.loadPerspectiveProjection((float)Math.PI/3,1.8f,
2000,0.1f);
shadowCamera.loadOrthographicProjection(-200,200, -200,
    200, -400, 400);
camera.loadViewMatrix();
shadowCamera.loadViewMatrix();
scene.setSkyBox(skyBox);
scene.addDrawItem(F16Tree);
DirectionalLight directionalLight = new DirectionalLight(
    new Vector3f(-90,120,20).negate().normalize());
directionalLight.setLightColor(new Vector3f(2.8f, 2.8f,
2.8f).div(10));
directionalLight.setLightIntensity(15);
scene.addLightSources(directionalLight, true);
LightingConfigurator.setLights(scene.getLights(),
combineShader);
Keyboard keyboard = new Keyboard();
keyboard.setWindow_pointer(window_handle);
keyboard.init();
Vector2f screenDimensions = new Vector2f(windowWidth,
windowHeight);
postprocessingShader.setUniform("uViewportSize",
screenDimensions);
postprocessingShader.setUniform("enableFXAA", true);
postprocessingShader.setUniform("gamma", 1.8f);
transparentShader.setUniform("uViewportSize", screenDimensions);
measureTime();

```

Продовження 3 – Лістингу В.1

```

while (!glfwWindowShouldClose(window_handle)) {

    Vector3f cameraPos = camera.getPosition();
    phi += (0.001f);
    if(phi >= Math.PI*2){phi = 0;}
    cameraPos.x = ro * cos(phi);
    cameraPos.z = ro * sin(phi);
    camera.loadViewMatrix();

    glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);

    shadowPass();
    deferredPass();
    skyBoxPass();

    glBindFramebuffer(GL_FRAMEBUFFER, 0);

    glDisable(GL_DEPTH_TEST);
    postprocessingPass();
    glEnable(GL_DEPTH_TEST);
    transparentPass();

    glfwPollEvents();
    glfwSwapBuffers(window_handle);
    measureTime();

    keyboard.processEvents((key, state) -> {
        if(key == GLFW_KEY_C && state == GLFW_PRESS){
            saveFramebufferAsImage(windowWidth,
windowHeight, "Screenshot.png");
        }
    });

}

}

```