МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний аерокосмічний університет ім. М.Є. Жуковського
«Харківський авіаційний інститут»

Факультет програмної інженерії та бізнесу

Кафедра інженерії програмного забезпечення

Пояснювальна записка до дипломної роботи

магістра
(освітній ступінь)

на тему «Аналіз оптимізаційних рішень для конвеєру рендеру зображення, а також пошук найкращих методів їх використання для досягнення оптимізаційного балансу»

ХАІ.603.667п1.121.156306.20В

Виконав: студент 6 курсу групи №667п1
Спеціальність 121 – Інженерія програмного забезпечення
(код та найменування)
Освітня програма Хмарні обчислення та Інтернет речей
(найменування)
Дружинін В.Д
(прізвище й ініціали студента)
Керівник: Конорев Б.М.
(прізвище й ініціали)
Рецензент: Іващенко Г.С.
(прізвище й ініціали)

Харків – 2020

Міністерство світи і науки України
Національний аерокосмічний університет ім. М. Є. Жуковського
«Харківський авіаційний інститут»

Факультет програмної інженерії та бізнесу
(повне найменування)
Кафедра інженерії програмного забезпечення
(повне найменування)
Рівень вищої освіти другий (магістерський)
Спеціальність 121 – інженерія програмного забезпечення
(код та найменування)
Освітня програма хмарні обчислення та Інтернет речей
(найменування)

ЗАТВЕРДЖУЮ
Завідувач кафедри

_____ _____
(підпис) (ініціали та прізвище)
“_____” _____ 2020 року

З А В Д А Н Н Я
НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Дружиніну Володимиру Дмитровичу
(прізвище, ім’я, по батькові)

1. Тема дипломної роботи «Аналіз оптимізаційних рішень для конвеєру рендеру зображення, а також пошук найкращих методів їх використання для досягнення оптимізаційного балансу»
керівник дипломної роботи Конорев Борис Михайлович д.т.н., профессор
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом Університету № _____ від “____” _____ 2020 року
2. Термін подання студентом роботи _____
3. Вихідні дані до роботи виконати аналіз оптимізаційних рішень для конвеєру рендеру зображення, пошук найкращих методів досягнення оптимізаційного балансу за рахунок отриманих даних
4. Зміст пояснювальної записки (перелік питань, які потрібно розробити аналіз оптимізаційних рішень для конвеєру рендеру зображення, а також пошук найкращих методів їх використання для досягнення оптимізаційного балансу
5. Перелік графічного матеріалу (усього 98 сторінок), 29 рисунків, 5 таблиць, 38 джерел, додаток А _____

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1÷3	Конорев Б.М., професор		

Нормоконтроль _____ _____ «___»_____20__р.
(підпис) (ініціали та прізвище)

7. Дата видачі завдання «___»_____20_____ р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1	Огляд і аналіз проблеми оптимізації графічного конвеєру		
2	Аналіз можливих шляхів знаходження оптимізаційного балансу		
3	Аналіз можливих рішень оптимізації графічного конвеєру, а також розробка методів збору даних для аналізу їх ефективності		
4	Розробка, та тестування ПЗ, яким буде виконано збір даних для аналізу ефективності теоретичних методів оптимізації		
5	Аналіз отриманих даних, та пошук оптимізаційного балансу з їх використанням		
6	Оформлювання пояснювальної записки до дипломної роботи		
7	Підготовка доповіді		
8	Підготовка презентації		

Студент _____ ___Дружинін В.Д___
(підпис) (прізвище та ініціали)

Керівник проекту ____ _____Конорев Б.М.__
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Дипломна робота магістра на тему: «Аналіз оптимізаційних рішень для конвеєру рендеру зображення, а також пошук найкращих методів їх використання для досягнення оптимізаційного балансу»: 98 сторінок, 29 рисунків, 5 таблиць, 38 джерел, додаток А.

Об’єкт дослідження – конвеєр рендеру растрового зображення, шляхи його оптимізації, оптимізаційний баланс.

Предмет дослідження – існуючі методи оптимізації конвеєру рендеру, а також пошук та використання оптимізаційного балансу.

Мета – проаналізувати існуючі методи оптимізації конвеєру рендеру, знайти слабкі місця оптимізації, зібрати фактичні дані для аналізу та знаходження шляхів досягнення оптимізаційного балансу.

Мета роботи полягає у використанні методів: статистичного аналізу, планування експерименту для аналізу ефективності методів оптимізації, аналіз архітектурної логіки систем, пошуку оптимізаційного балансу.

На етапі аналізу проблеми проаналізовані системи рендеру растрового зображення, існуючі методи його оптимізації, можливі шляхи знаходження оптимізаційного балансу.

На етапі пошуку можливих рішень: були знайдені слабкі місця оптимізації конвеєру рендеру, розроблені можливі рішення оптимізації їх і зібрані дані що до їх ефективності. Використовуючи отримані дані був зроблений аналіз їх ефективності, а також проведений пошук оптимізаційного балансу для обраних оптимізаційних рішень.

Актуальність даної роботи полягає в тому, що кожного разу намагаючись отримати краще зображення ми вигадуємо більш складніші, точніші алгоритми, але потужності заліза кінцеві, і їх недостатньо для створення максимально деталізованої графіки у реальному часі для нових стандартів моніторів. До того ж збільшується навантаження за рахунок більш нових та складніших алгоритмів, які надають кращий результат але коштують дорожче. Через це потрібно використовувати ще більш складні архітектурні та обрахункові рішення для оптимізації графічного конвеєру а також поміж вибором варіантів оптимізації знаходити оптимізаційний баланс.

КОМП’ЮТЕРНА ГРАФІКА, КОНВЕЄР РЕНДЕРУ, ОПТИМІЗАЦІЯ, ОПТИМІЗАЦІЙНИЙ БАЛАНС, РІШЕННЯ ОПТИМІЗАЦІЇ РЕНДЕРУ КОМП’ЮТЕРНОЇ ГРАФІКИ

РЕФЕРАТ

Дипломная работа магистра на тему: «Анализ оптимизационных решений для конвейера рендера изображения, а также поиск лучших методов их использования для достижения оптимизационного баланса»: 98 страниц, 29 рисунков, 5 таблиц, 38 источников, дополнение А.

Объект исследования - конвейер рендера растрового изображения, пути его оптимизации, оптимизационный баланс.

Предмет исследования - существующие методы оптимизации конвейера рендера, а также поиск и использование оптимизационного баланса.

Цель - проанализировать существующие методы оптимизации конвейера рендера, найти слабые места оптимизации, собрать фактические данные для анализа и поиска путей достижения оптимизационного баланса.

Цель работы заключается в использовании методов: статистического анализа, планирования эксперимента для анализа эффективности методов оптимизации, анализ архитектурной логики систем, поиска оптимизационного баланса.

На этапе анализа проблемы проанализированы системы рендера растрового изображения, существующие методы его оптимизации, возможные пути нахождения оптимизационного баланса.

На этапе поиска возможных решений: были найдены слабые места оптимизации конвейера рендера, разработаны возможные решения оптимизации и собранные данные косательно их эффективности. Используя полученные данные был сделан анализ их эффективности, а также проведен поиск оптимизационного баланса для избранных оптимизационных решений.

Актуальность данной работы заключается в том, что каждый раз пытаясь получить лучшее изображение мы придумываем более сложные, точные алгоритмы, но мощности железа конечные, и их недостаточно для создания максимально детализированной графики в реальном времени для новых стандартов мониторов. К тому же увеличивается нагрузка за счет более новых и более сложных алгоритмов, которые предоставляют лучший результат но стоят дороже. Поэтому нужно использовать еще более сложные архитектурные и алгоритмические решения для оптимизации графического конвейера, а также между выбором вариантов оптимизации находить оптимизационный баланс.

КОМПЬЮТЕРНАЯ ГРАФИКА, КОНВЕЙЕР РЕНДЕРА, ОПТИМИЗАЦИЯ, ОПТИМИЗАЦЫОННЫЙ БАЛАНС, РЕШЕНИЯ ОПТИМИЗАЦИИ РЕНДЕРА КОМПЬЮТЕРНОЙ ГРАФИКИ

ABSTRACT

Master's thesis on the topic: «Analysis of optimization solutions for the image rendering pipeline, as well as the search for the best methods of using them to achieve an optimization balance»: 98 pages, 29 pictures, 5 tables, 38 sources, addition A.

The object of research is the bitmap rendering pipeline, ways of its optimization, optimization balance.

The subject of research is the existing methods of optimization of the render pipeline, as well as the search and use of the optimization balance.

The goal is to analyze the existing methods for optimizing the render pipeline, find optimization weaknesses, collect evidence for analysis and find ways to achieve an optimization balance.

The purpose of the work is to use methods: statistical analysis, experiment planning to analyze the effectiveness of optimization methods, analysis of the architectural logic of systems, search for an optimization balance.

At the stage of analyzing the problem, the systems for rendering the raster image, the existing methods of its optimization, possible ways of finding the optimization balance were analyzed.

At the stage of searching for possible solutions: weaknesses in the optimization of the render pipeline were found, possible optimization solutions were developed and the collected data regarding their effectiveness. Using the data obtained, an analysis of their effectiveness was made, and an optimization balance was searched for selected optimization solutions.

The relevance of this work lies in the fact that every time we try to get a better image, we come up with more complex, accurate algorithms, but the power of the hardware is finite, and they are not enough to create the most detailed graphics in real time for the new monitor standards. In addition, the load increases due to newer and more complex algorithms that provide better results but are more expensive. Therefore, it is necessary to use even more complex architectural and algorithmic solutions to optimize the graphics pipeline, and also to find an optimization balance between the choice of optimization balance.

COMPUTER GRAPHIC, RENDER PIPELINE, OPTIMIZATION, OPTIMIZATION BALANCE, SOLUTIONS FOR OPTIMIZING COMPUTER GRAPHICS RENDER

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ ВИМІРЮВАНЬ ФІЗИЧНИХ ВЕЛИЧИН, СКОРОЧЕНЬ І ТЕРМІНІВ

CPU – центральний процесор;

GPU – відеокарта, графічний процесор, пристрій для обрахунку графіки;

Render, рендер – процес створення зображення на комп’ютері за допомогою математичних алгоритмів;

Tick, тік – одиниця часу або алгоритмічна одиниця, яка обраховується як деякий повний цикл у середині програми;

Вершина – точка у просторі, яка використовуються як мінімальна одиниця для рендеру. З трьох точок складається трикутник - найпростіша фігура для малювання, яка використовується для створення растрової графіки;

Mesh, меш – набір вершин або згрупованих трикутників, які являються 3D об’єктом;

Піксель – точка відображення кольору;

Shader – алгоритм, який використовується для обрахунку графіки;

Render pipeline, графічний конвеєр, конвеєр рендеру – набір алгоритмічних та математичних рішень для малювання графіки;

Vertex shader, VS, вершинний шейдер – алгоритм, який викликається для обрахунку вершин у графічному конвеєрі;

Pixel shader, PS, піксельний шейдер – алгоритм, який викликається для обрахунку пікселів у графічному конвеєрі;

Clip, кліп, кліпінг – процес відсіювання зайвих вершин або пікселів для оптимізації розрахунків та рендеру;

Texture, текстура – зображення або масив пікселів, які несуть певні дані;

Z-буфер – текстура яка має тільки один канал зображення, який зберігає відстань до об’єкта;

Растерайзер – алгоритм перетворення математичних моделей та даних, фінальним результатом дії якого є растрове зображення;

Blend state, стан змішування пікселів – налаштування конвеєру рендеру, які встановлюють логіку змішування пікселів.

ВСТУП

Актуальність даної роботи у тому – що людство вже не може жити без кіно, ігор а також симуляторів. Людина потребує пізнання світу, але саме через це вона намагається відтворити його і поділитися цим з іншими людьми. Одним з найдавніших та надійніших способів відтворювати світ було зображення. Але завдяки технологіям людина прийшла від малюнків до рендеру графіки у реальному часі.

Кожного разу намагаючись отримати краще зображення ми вигадуємо більш складніші, точніші алгоритми, але потужності заліза кінцеві, і їх недостатньо для створення максимально деталізованої графіки у реальному часі для нових стандартів моніторів. До того ж збільшується навантаження за рахунок більш нових та складніших алгоритмів, які надають кращий результат але коштують дорожче. Через це потрібно використовувати ще більш складні архітектурні та обрахункові рішення для оптимізації графічного конвеєру а також поміж вибором варіантів оптимізації знаходити оптимізаційний баланс.

Через популярність а також постійний розвиток технологій рендеру – було розроблено багато способів створити зображення. Усі вони корисні для відтворення зображення, але усі вони працюють по різному:

- векторна графіка – використовує математичні функції для створення фігур, з яких складається зображення;
- растрова графіка – використовує трикутники для відображення графіки;
- воксельна графіка – використовує об’ємні пікселі;
- трасування променів – одна з перспективніших, та найточніших систем для створення реалістичного зображення, однак вона потребує занадто багато розрахунків, адже симулює фізичну дію світла.

Усі ці види малювання графіки використовуються у різних сферах. Деякі з них кращі для відображення даних, інші для кіно. Але найпопулярніший, та найрозвинутіший з методів – растрова графіка. Через її гнучкість (порівняно з векторною), дешевизну (порівняно з трасуванням променів), а також простоту використання – вона надає найкращі результати. За допомогою растрової графіки ми навчились робити реалістичну картинку у реальному часі, а також стилізоване зображення. У деяких випадках цей вид рендеру дозволяє комбінувати його з іншими, наприклад з трасуванням променів.

Завдяки тому що растрова графіка використовує декілька етапів створення зображення – ми можемо як налаштувати її під себе, так і оптимізувати. Ми контролюємо майже увесь цикл створення зображення, але це впливає і на етапи розробки ПЗ яке буде обслуговувати даний алгоритм. Цей вид рендеру надає нам можливість контролювати алгоритм обробки кожної вершини, трикутника, текстури яку він використовує.

Кількість тіків процесору – кінцева, кількість обрахунків відеокарт на одиницю часу – також кінцева. «З великою владою приходить велика відповідальність» - ця фраза повністю описує можливості використання

конвеєру рендеру. Ми можемо налаштовувати алгоритми створення зображення під себе, однак ми повинні пам’ятати про оптимізаційний баланс. Деякі алгоритми можуть дати кращий результат але доведеться виключити інші через брак обрахункових ресурсів, або навпаки – використати декілька дешевих обрахунків, які у купі дадуть кращий результат за один складніший та дорожчий.

Мета роботи – проаналізувати існуючі методи оптимізації рендеру растрової графіки. Знайти можливі методи додаткової оптимізації, а також проаналізувати у яких випадках краще використовувати їх для досягнення кращого оптимізаційного балансу.

Для виконання поставленої мети необхідно вирішити наступні питання:

- проаналізувати особливості та проблеми розробки рендеру, знайти слабкі оптимізаційні місця;
- зробити експериментальний аналіз існуючих рішень оптимізації рендеру;
- за допомогою тестових даних – віднайти найкращі рішення для оптимізаційного балансу;
- проаналізувати результати аналізу, і віднайти практичні рекомендації що до створення таких систем;
- за необхідності розробити ПЗ для отримання точних даних.

Об’єкт дослідження – конвеєр рендеру растрового зображення, шляхи його оптимізації, оптимізаційний баланс.

Предмет дослідження – існуючі методи оптимізації конвеєру рендеру, а також пошук та використання оптимізаційного балансу.

Методи дослідження – статистичного аналізу, планування експерименту для аналізу ефективності методів оптимізації, аналіз архітектурної логіки систем, пошуку оптимізаційного балансу.

Наукова новизна дослідження полягає у пошуку відповідей на питання оптимізаційного балансу при створені алгоритмів та систем рендеру комп’ютерної графіки. Вибору найкращих варіантів оптимізації а також пошуку кращих ситуацій для їх використання, які дадуть найбільшу вигоду під час роботи алгоритму створення зображення.

Практичне значення результатів: у розробці архітектурних рішень, для досягнення найкращого оптимізаційного балансу рендеру.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ, ТА ПОСТАНОВКА ЗАДАЧ

1.1 Актуальність роботи

Рендер графіки [1] – складний, багаторівневий процес який використовує математичні алгоритми обробки даних для отримання зображення. Саме через те, що даний напрямок розвивається дуже швидко (особливо за останні 15 років), людство не встигає виявити усі недоліки нових підходів. Зазвичай використовувати технологію на рівні «вище середнього» починають тільки перед виходом нової – це пов’язано з великою кількістю абстракцій [2], а також взаємозв’язком залізної частини та програмної. Усе це в цілому впливає на фінальний результат використання даних технологій – обов’язковий вибір між якістю та швидкістю.

Не зважаючи на швидкий розвиток технологій рендерингу людина навчилася використовувати накопленні знання для швидшого вивчення нових підходів до розробки. Завжди все впирається у три головні речі:

1) час;
2) якість;
3) вартість.

Через багатогранність, складність роботи над рендером графіки, а також фізичними і розумовими обмеженнями людини, виникає багато проблем при розробці. Починаючи з великої кількості помилок які важко знайти, та закінчуючи неможливістю правильно вести менеджмент часу. Навідь після більш ніж 40 років, людство все ще не знає ідеального алгоритму організації розробки ПЗ. На це впливає швидкий розвиток нових підходів та алгоритмів у даній галузі, а також відсутність строгої типізації. Усі правила які використовуються під час розробки можуть бути корисними в одних випадках, а також шкідливими у інших.

До того ж є інші фактори які додають проблем при розробці ПЗ яке рендерить зображення:

- тісна залежність програмного забезпечення від заліза, а також фізичні обмеження обрахункових можливостей;
- велика кількість абстракцій, які виникли через загальну складність розробки ПЗ, це впливає на потенційні помилки або ситуації які змушують відмовлятися від деяких підходів розробки на користь дешевизні або простоті;
- людський фактор, який є одним з непередбачуваних підводних каменів при розробці та оптимізації алгоритмів рендеру зображення.

Саме через усе вище написане потрібно використовувати алгоритми пошуку оптимізаційного балансу [3]. Оптимізаційний баланс багато використовують на підприємствах, для досягнення максимальної вихідної потужності, також його використовують у фінансовій сфері, де його задача є –

знаходження кращої тактики роботи певного алгоритму, який використовує організація.

Загалом оптимізаційний баланс знаходиться через аналіз певного алгоритму дій а також статистики цього алгоритму. Безпосередньо при розробці ПЗ його використовують тільки для менеджменту часу та ресурсів, рідше – для знаходження кращих архітектурних рішень а також алгоритмічних підходів при розробці. Через часту зміну напрямку, відмінність уявлення архітектури кожного з членів команди а також важкості «проектування на майбутнє», оптимізаційний баланс можуть використовувати тільки на фінальній стадії розробки ПЗ. Однак через монолітність архітектури на даній стадії розробки, воно не дає великого впливу.

Найкращі оптимізаційні рішення зазвичай виносяться за допомогою досвіду архітекторів ПЗ [4]. Вони дуже рідко опираються на статистичні дані та аналіз – і частіше на свій досвід, таким чином породжуючи хибні рішення при розробці систем які навіть трохи відрізняються від тих, з якими вони мали справу. Саме через це багато систем рендерингу не надають максимальної ефективності – що натомість впливає на якість кінцевого результату розробки.

Знаходження оптимізаційного балансу - один з непопулярних, та дорогих методів, який надає кращий результат, порівняно від звичайних рішень які базуються на досвіді.

1.2 Виявлення проблем та актуалізація рішень

Найбільшою проблемою при розробці ПЗ для рендерингу – є людський фактор [5]. Людина все ще не може тримати велику кількість даних, саме для цього вона використовує абстракції. Але абстракції мають великий недолік – через спрощення вони можуть розумітися декількома людьми по різному. Різне розуміння – різне прийняття рішень, що приводить до виникнення розбіжностей або помилок. Дана ситуація дуже критична коли мова йде про розробку ПЗ. На відміну від людини машина – виконує те що їй наказали. Якщо через велику кількість взаємозв’язаних абстракцій прокрадеться помилка – вона може вплинути на кінцеву оптимізацію, що не є бажаним результатом розробки. Завжди є ідеальний алгоритм, який виконує задачу максимально чітко та швидко, але через складність та комплексність ми ще не можемо легко винаходити його.

У фінансовій сфері через велику кількість зв’язаних об’єктів а також даних (цифр) використовують математичну статистику для знаходження оптимізаційного балансу. Оптимізаційний баланс – це не ідеальний алгоритм, а лише його наближене грубе рішення.

Через аналогії [6] – людина використовує схожі рішення усюди, це також стосується і розробки ПЗ з фінансовою сферою. Методи пошуку оптимізаційного балансу можливо частково позичити з відти. Через можливість

отримувати усі необхідні дані для тестів – збирають статистику для певних оптимізаційних рішень, а вже завдяки їй – знаходять оптимізаційний баланс для будування архітектури ПЗ і пошуку кращих алгоритмів. Рендер графіки найперспективніша з напрямків розробки ПЗ для використання даного методу, через її залежність від кінцевих обрахункових ресурсів, а також її необхідності у сучасний час.

1.3 Конвеєр рендеру, а також шляхи його оптимізації

Конвеєр рендеру (Рисунок 1.1) - фіксована частина яка відповідає саме за обрахунок графіки. Вона дуже залежить від архітектурних рішень фізичної частини (заліза), адже фізична оптимізація завжди діє краще програмної. Через її залежність від фізичної частини – зменшується її гнучкість, але надається можливість точно обраховувати методи створення зображення [7].

Рисунок 1.1 – Схема роботи конвеєру рендеру

Загалом конвеєр рендеру - це логічна сукупність обчислень, що виконуються послідовно і дають на виході синтезовану сцену. Обчислення в конвеєрі розділені на кілька етапів, у кожному з яких апаратно або програмно виконується визначена функція.

Даний поетапний алгоритм використовується для створення растрової графіки. Растрова графіка – це математичний алгоритм який дозволяє створювати відображення моделей за допомогою примітивів.

Стадії геометричних перетворень:

- розбиття геометричних моделей на примітиви;
- етап модельних перетворень;
- освітлення;
- видові перетворення;
- видалення невидимих поверхонь.

1.3.1 Представлення геометричних моделей через примітиви

Основними графічними примітивами [8] які використовуються у растровій графіці є:

1) вершина;
2) лінія;
3) трикутник.

Найчастіше використовується саме трикутник – як найпростіша фігура (Рисунок 1.2). За допомогою трикутників можливо створювати як 2D, так і 3D зображення. З трикутників можливо створити об’ємну фігуру. Коли об’єднані трикутники створюють деяку фігуру у просторі – їх називають сіткою, полігональною моделлю, або мешем [9] (mesh) (Рисунок 1.2). Окремий трикутник який використовується у меші називають полігоном.

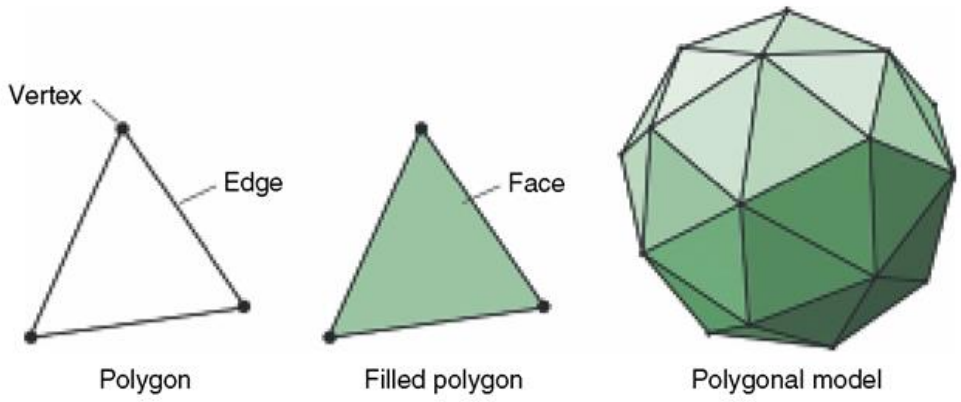

Рисунок 1.2 – Примітив трикутник, полігон, полігональна модель

На початковому етапі виробляється опис тривимірної сцени, зображення якого необхідно синтезувати. Тривимірні об'єкти зазвичай конструюються переважно з трикутників, оскільки трикутник є найпростішим полігоном, та однозначно задає найпростішу площину в просторі і гарантовано забезпечує рішення задачі декомпозиції [10].

Будь-яка поверхня може бути апроксимована сіткою трикутників і, якщо така сітка досить добре складена, то за її допомогою можна представити будь-яку поверхню з потрібною точністю.

Поверхня апроксимується набором полігональних граней (face, polygon). Границі граней описуються ребрами (edge). Частина відрізка, що формує ребро, закінчується вершинами (vertex) (Рисунок 1.2).

Далі іде процес тріангуляції. Тріангуляція [11] - процес розбиття модельованого об'єкту на трикутники.

Як правило, вона виконується програмно. Найпростішим рішенням задачі тріангуляції являється розщеплення полігона вздовж деякої хорди [12] на два полігона і подальше рекурсивне розбиття їх до ситуації, коли підлягаючий тріангуляції полігон являється трикутником.

Даний алгоритм може бути застосований лише для опуклих полігонів. Кожний пакет 3D моделювання [13] здійснює тріангуляцію декількома способами, залежно від поставленої задачі.

Наприклад, при моделюванні віддалених об'єктів немає сенсу застосовувати дуже детальну мережу для їх опису і, навпаки, для близьких об'єктів використовують більшу кількість трикутників. Це зменшує навантаження при обрахунку графіки – менше полігонів – менше обрахунків.

1.3.2 Моделі перетворення

Усі моделі для обрахунку використовують два типи координат [14]:

- локальні (об’єктні OCS - Object Coordinate System) – це координати вершин у локальному просторі фігури, відносно її умовного центру;
- глобальні (WCS – World Coordinate System) – це координати у світовому просторі – просторі де будуть розміщуватися фігури один відносно одного.

Усі моделі збираються з примітивів, які у свою чергу можуть бути змінені за допомогою матриць перетворень. Матриці перетворення [15] дозволяють проводити наступні операції над об’єктом:

- переносити;
- масштабувати;
- повертати.

1.3.3 Моделі освітлення

Об’єм фігури майже завжди задається завдяки освітленню, тому наступний етап який виконується – це обрахунок освітлення об’єктів [16]. Через те що трасування променів занадто дороге, ми будемо розглядати більш примітивні способи освітлення полігональних об’єктів, які використовуються вже багато років.

На освітлення об’єкта впливають 3 фактори:

1) матеріал об’єкта [17];

2) тип джерела світла;
3) алгоритм освітлення.

Матеріал об’єкта задає властивості відбивання або поглинання світла, а також можливу зміну кольору у майбутньому.

Джерела світла відрізняються лише типом розповсюдження світла (розсіяне, спрямоване, точкове), а також їх інтенсивністю (постійні, змінна).

Алгоритми освітлення – одна з найважчих операцій при обрахунку графіки. Через це існує декілька алгоритмів які надають різний результат:

Дзеркальне (полігональне) освітлення (Рисунок 1.3) – надає добрий результат при простій графіці, але виділяє полігони. Використовує мінімум ресурсів для обрахунків. Для цього типу освітлення достатньо обрахувати кут між нормаллю полігона та напрямком зору камери відображення.

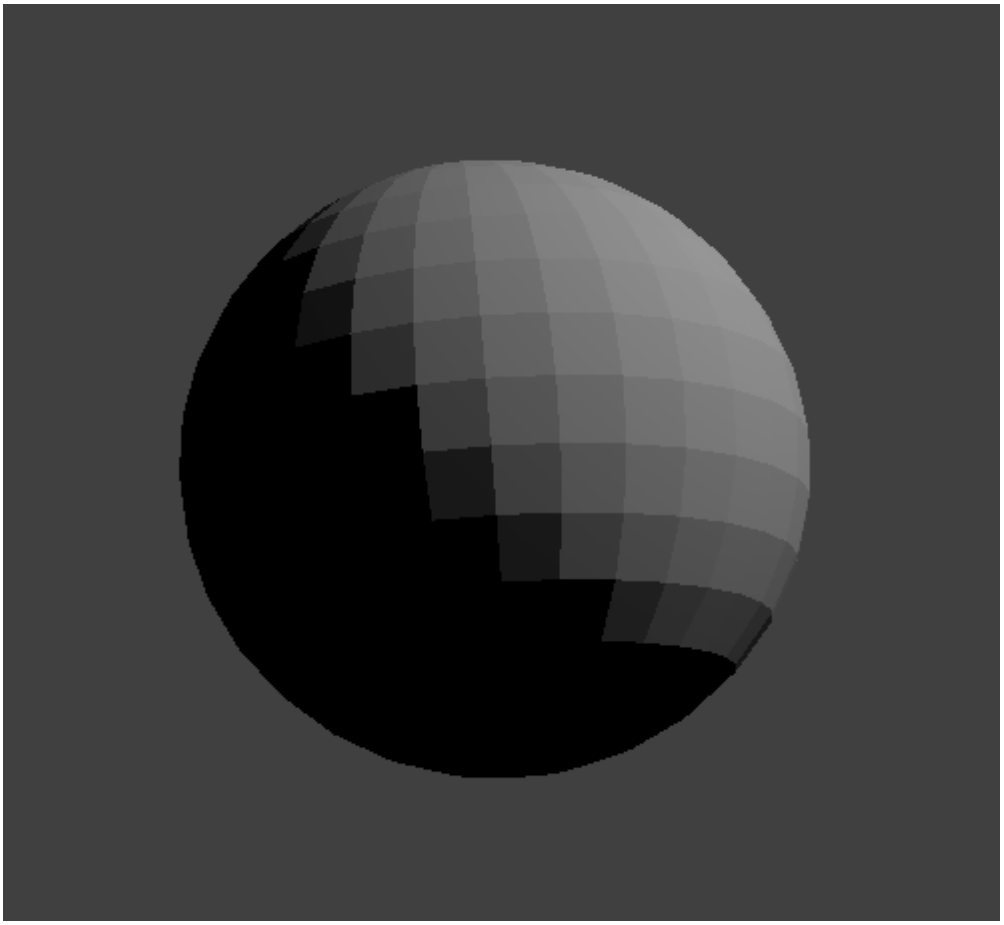

Рисунок 1.3 – Приклад дзеркального (полігонального) освітлення

Дифузне освітлення (Рисунок 1.4) – використовую більше обрахунків а також може додатково використовувати дані з текстур для покращення фінального результату (карти нормалей або normal map) (Рисунок 1.5). Однак надає реалістичне та м’яке освітлення. Цей тип освітлення використовується частіше за все через його дешевизну відносно результату.

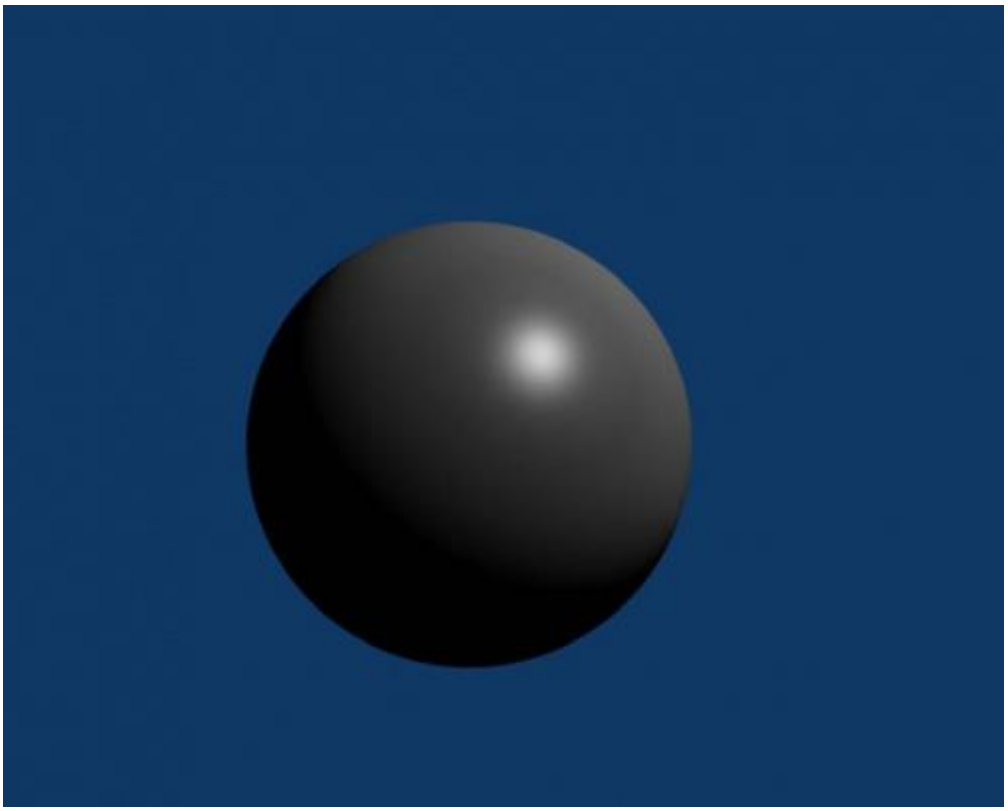

Рисунок 1.4 – Приклад дифузного освітлення

Також до освітлення можемо віднести ще інші техніки, які покращують вигляд картинки, але зазвичай використовуються на пізніх етапах формування зображення (етап пост-обробки). Це можуть буди:

- глобальне освітлення;
- розсіяне освітлення;
- під об’єктне розсіювання світла;
- запечені карти освітлення;
- свічення об’єктів;
- техніки обрахунку віддзеркалень та бліків.

Окремо потрібно виділити техніку покращення якості та плавності відображення полігональної моделі завдяки додатковій текстурі – карті нормалей [18] (Рисунок 1.5). Вона не тільки дозволяє зменшити навантаження при створення зображення за рахунок зменшення кількості полігонів у рази, але ще й надає дуже добрий фінальний результат. Обрахунок карти нормалей може бути використаний для значного покращення вигляду моделей, однак слід пам’ятати, що текстура потребує багато пам’яті.

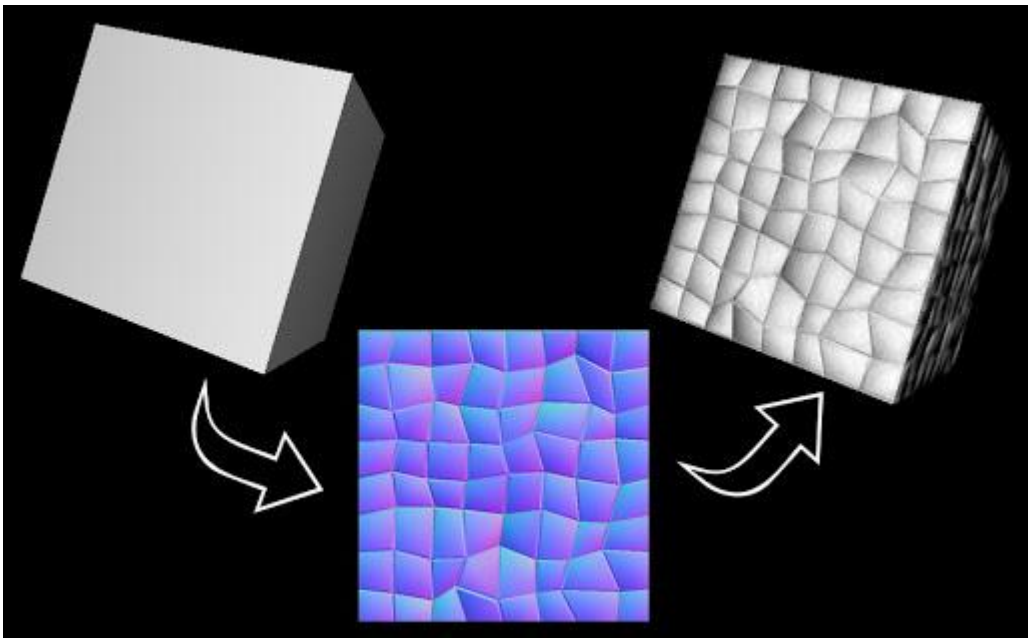

Рисунок 1.5 – Приклад використання карти нормалей (normal map) для низько полігональної моделі

1.3.4 Видові перетворення

Для створення 2D зображення з об’ємних фігур використовують видові перетворення. Координати об’єктів переводяться у глобальні координати а потім у координати відображення.

Перспективна проекція (Рисунок 1.6) – фігури при відображенні змінюють свої пропорції залежно від знаходження у просторі.

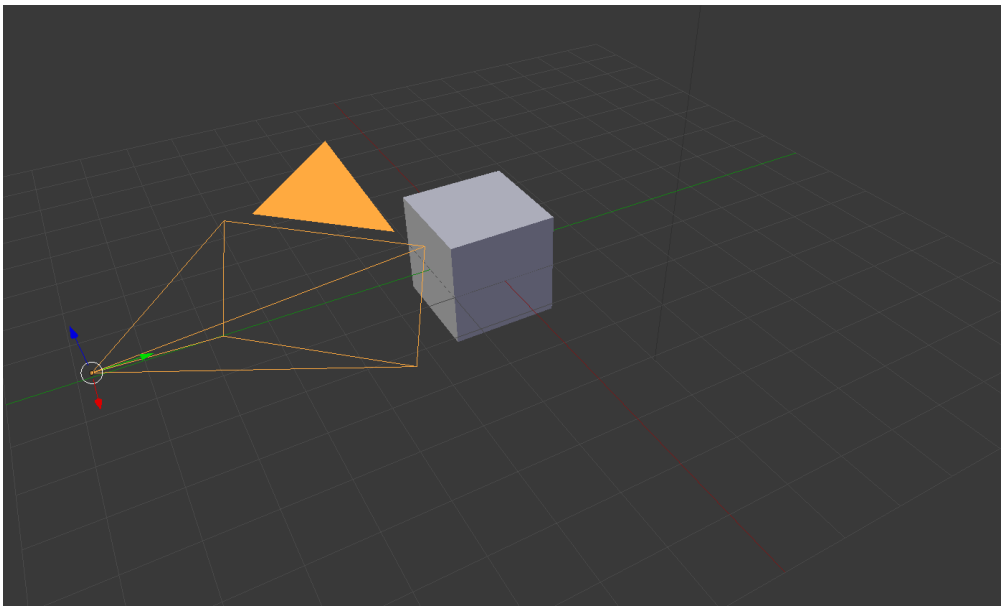

Рисунок 1.6 – Перспективна проекція

Орографічна проекція (Рисунок 1.7) – усі фігури відображаються зберігаючи свої пропорції.

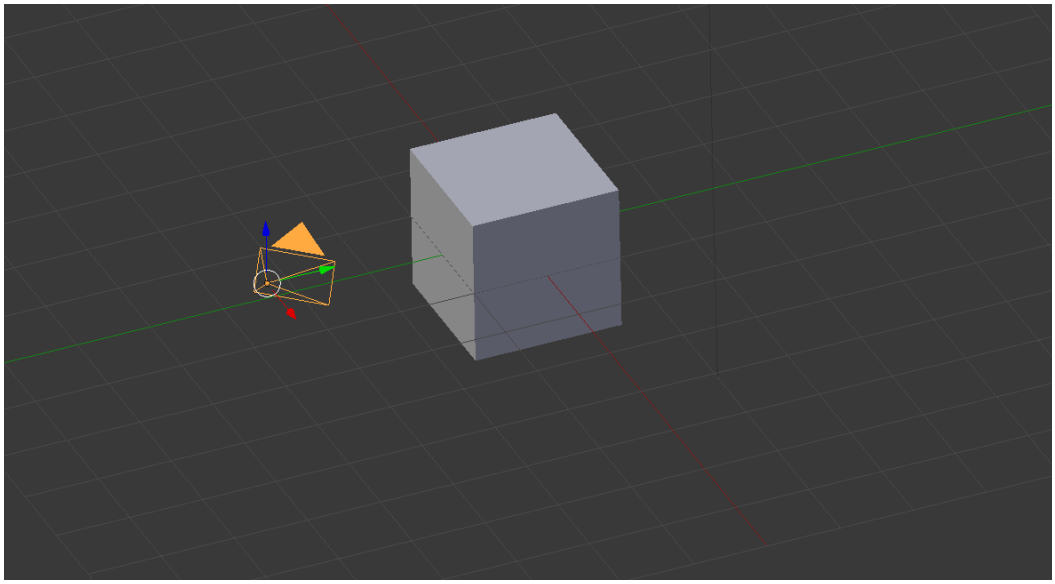

Рисунок 1.7 – Орографічна проекція

1.3.5 Відсікання невидимих поверхонь

Через те що обробка 3D об’єктів виконується у 2D зображення виникають ситуації коли нам потрібно відсіяти непрозорі об’єкти які нам непотрібно малювати (Рисунок 1.8). Для цього лише перевіряється чи розвернута нормаль грані до камери відображення, чи ні.

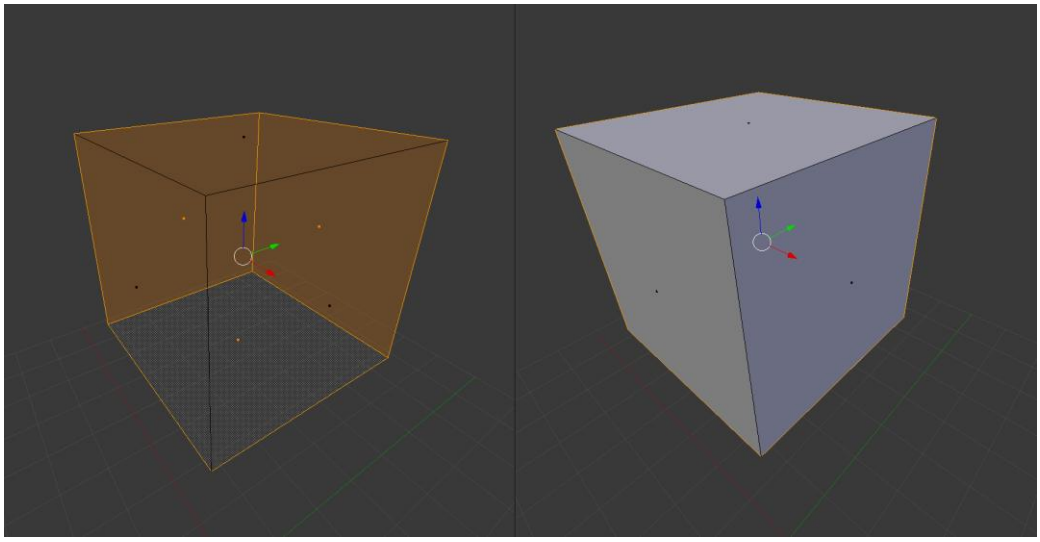

Рисунок 1.8 – Приклад відсіювання зайвих полігонів - грані які не будуть відображені розмальовані жовтим

1.3.6 Метод Z–буферу

Після геометричних перетворень виконується стадія візуалізації - піксельної обробки зображення. На даному етапі виконується відсіювання зайвих обробок пікселів за допомогою Z-буферу [19].

Z-буферу – це окремий буфер зображення який має лише один канал. Його задача відсіювати зайві операції малювання пікселів на екрані для оптимізації рендеру зображення а також коректного відображення перекриття об’єктів. При створенні (візуалізації) 3D-об'єкту, його глибина генерується на осі Z-координат і зберігається у Z-буфері. Сам буфер, зазвичай, являє собою двомірний масив X-Y координат із одним елементом (глибиною) для кожного екранного пікселя. Коли інший об'єкт сцени повинен бути відображений у цьому пікселі зараз, тоді порівнюється дві глибини та перекривається поточний піксель, якщо об'єкт знаходиться ближче до спостерігача. Обрана глибина зберігається в Z-буфері і замінює попередню. Зрештою, Z-буфер дозволяє правильно відтворювати звичне для нас сприйняття глибини: ближчий до нас об'єкт перекриває наступні, що розташовуються за ним — невидимих поверхонь. Даний процес можна добре побачити на рисунках 1.9 та 1.10.

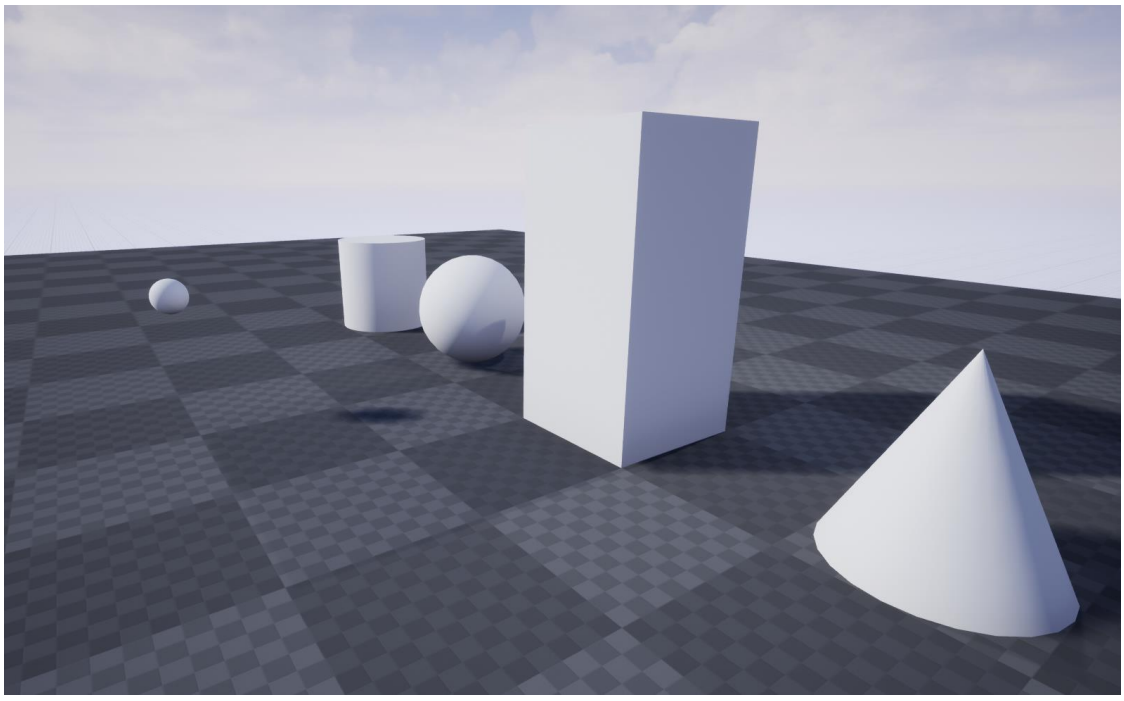

Рисунок 1.9 – Просте відображення сцени

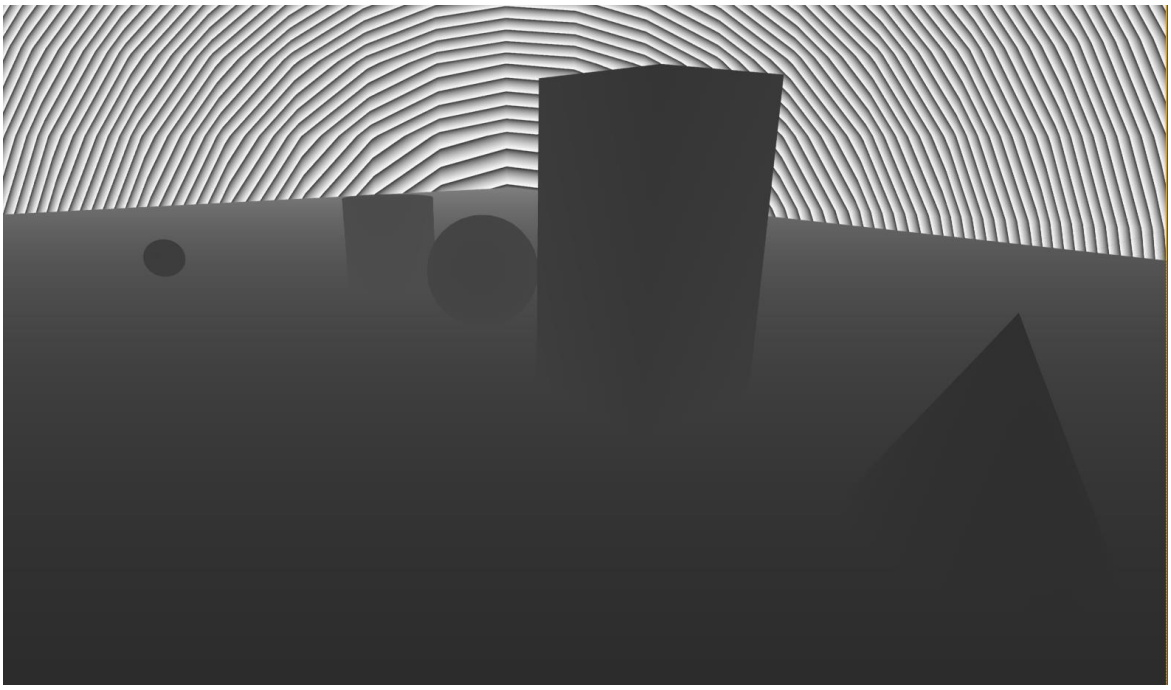

Рисунок 1.10 – Представлення сцени у Z-Буфері

1.3.7 Обробка кожного пікселя та текстурування

Один з останніх етапів формування зображення – етап піксельного шейдеру [20]. На цьому етапі проходить обробка усіх пікселів для об’єкту з використанням додаткових карт, текстур, а також Z-буферу. Саме на цьому етапі відображення об’єкту отримує свій колір, до якого додається освітлення а також інші ефекти покращення зображення.

Для кожного пікселю об’єкту викликається алгоритм обробки, який визначає його фінальний колір. При обробці використовуються дані освітлення, текстури, а також інші вхідні дані які потрібні для алгоритму.

Для використання потрібних даних з текстур використовуються текстурні координати, які прив’язані до вершин. В деяких випадках можливо зберігати дані кольору у вершинах, та інтерполювати їх за необхідності.

Приклади відображення сцени з використанням текстур кольору (Рисунок 1.11) а також сцени без них (Рисунок 1.12).

Рисунок 1.11 – Відображення сцени з повним набором текстур.

Рисунок 1.12 – Відображення сцени без текстур кольору, використовуючи тільки технології освітлення а також карти нормалей.

1.4 Постановка задач

Через досить фіксовану технічну архітектуру рендеру зображення, при створенні алгоритму є невеликий простір для зміни алгоритму, але у свій час це можливо використати собі на користь. Фіксованість архітектури зменшує потенційні відхилення архітектурних рішень, що дозволяє аналізувати даний процес розбивши його на невелику кількість типових алгоритмів обробки даних.

Це в свою чергу дозволить нам легше збирати статистику а також усі необхідні дані для знаходження оптимізаційного балансу, а також спростить прийняття фінальних оптимізаційних рішень у деяких частинах алгоритму створення зображення. Для цього нам потрібно:

- проаналізувати існуючі принципи оптимізації рендеру;
- виявити слабкі місця в алгоритмі формування зображення, а також знайти можливі рішення для покращення їх роботи;
- проаналізувати можливі шляхи знаходження оптимізаційного балансу, а також ситуації у яких він може використовуватись.

1.5 Висновки з розділу 1

У даному розділі була описана актуальність даної роботи, було виявлено проблеми, а також була проведена актуалізація рішень. Був описаний повний конвеєр рендеру зображення, а також його додаткові можливості включно з описом їх використання при створення зображення.

Була поставлена задача для даної роботи, а саме:

- проаналізувати існуючі принципи оптимізації рендеру;
- виявити слабкі місця в алгоритмі формування зображення, а також знайти можливі рішення для покращення їх роботи;
- проаналізувати можливі шляхи знаходження оптимізаційного балансу, а також ситуації у яких він може використовуватись.

2 АНАЛІЗ ПРОБЛЕМ ОПТИМІЗАЦІЇ РЕНДЕРУ, ПОШУК ОПТИМІЗАЦІЙНОГО БАЛАНСУ

2.1 Основні принципи оптимізації

Оптимізація – один з невід'ємних етапів розробки ПЗ [21]. Навідь якщо оптимізація не виноситься окремою задачею – вона виконується повсякчасно та усюди. Головна задача оптимізації, так само як і її сутність – зробити максимум при найменших затратах.

Якщо ми говоримо про оптимізацію під час розробки складних взаємозв'язаних процесів – ми повинні розуміти чого ми бажаємо отримати у результаті. Немає ідеального алгоритму який дозволить оптимізувати усе, можливо лише оптимізувати одну частину за рахунок іншої. Наприклад прискорити час виконування програми, витративши більше часу на розробку, або знехтувати якістю на користь меншого кінцевого навантаження. Звичайно до оптимізації можливо приписати правильні рішення, але навіть у цьому випадку ми отримуємо більшу продуктивність за рахунок досвіду людини яка прийняла рішення.

Оптимізація при розробці ПЗ проводиться повсякчасно – на етапі планування, проектування, розробки, пошуку помилок, під час написання коду, чи алгоритму. Неправильно вважати оптимізацію певним набором правил – які роблять краще.

Оптимізація графічних процесів формування зображення новою формою умовної абстракції – кінцевим зображенням. Інколи ми можемо знати точні дані які хочемо отримати, але при створенні алгоритмів формування зображення найчастіше доводиться оцінювати фінальний результат суб'єктивно. Саме це може як допомогти та і нашкодити при прийнятті рішень.

2.1.1 Завчасна оптимізація і оптимізація у процесі, при розробці ПЗ

Вже досить довго ведеться сперечання: «Чи потрібна завчасна оптимізація, або краще виконувати її у процесі?» - на дане питання не можливо відповісти однозначно, ми можемо тільки проаналізувати усі плюси та мінуси обох випадків [22].

Завчасна оптимізація – дозволяю спроектувати основні рішення розробки ПЗ на перед, а також деякі важливі моменти. Це потребує ресурсів, але якщо не відхилятися від поставленої задачі – відбиває їх у процесі розробки. ЇЇ краще використовувати для виконання завчасно спланованих задач, які не будуть змінюватися у процесі розробки. Дана оптимізація при розробці ПЗ дозволяє прийняти корисні базові які можуть прискорити спланову розробку у майбутньому. Найкраще дана методика себе проявляє при розробці архітектури ПЗ, а також при плануванні основних алгоритмів, які не будуть змінюватися.

Оптимізація в процесі – не потребує завчасних дій, виконується безпосередньо під час процесу розробки. Принцип цієї оптимізації – правильний вибір серед декількох можливих варіантів рішення задачі. Це дозволяє зекономити час перед початком роботи, а також збільшує гнучкість проекту в цілому. Вона найкраще розкриває себе під час розробки проектів, які не мають фіналізованого повного ТЗ, та експериментують під час розробки. Найважливіша частина даного підходу залежить від вмінь людей які розробляють проект. Чим більший досвід у людини яка робить рішення, тим більша ймовірність що дана оптимізація буде вірною.

Виходячи з вищевикладеного, ми можемо зробити висновок, що завчасна оптимізація потрібна на початкових етапах проекту, а також під час конструювання архітектури. Також оптимізація у процесі це зазвичай пасивна дія яка залежить від рівня досвіду розробника.

2.1.2 Оптимізація, а також її вплив на чистоту коду

Чистота коду [23] – абстрактне значення, але можливо відрізнити «поганий - брудний» код від «гарного - чистого». Зазвичай під це поняття підводиться стилістика коду. Через те що код – це набір складних архітектурно-алгоритмічних рішень виражених зазвичай у вигляді тексту, який пишуть, читають, а також підтримують люди – він повинен бути структурованим, та зрозумілим. Це економить велику частину часу при роботі з ним, а також при його вивченні.

Правильний вибір оптимізаційного рішення вливає на багато речей, але найчастіше від цього страдає лаконічність а також мінімізація коду. Через архітектуру комп’ютерів а також написаних для них інтерфейсів, у багатьох випадках людина використовує готові елементні конструкції, які складаються с декількох елементів у середини. Ці елементи при правильній структуризації дуже легко зчитуються, цим самим економлячи час розробки. Звісно при такому підкоді виникають випадки які потребують замінити готовий набір елементів на свій, за для досягнення кращої оптимізації. Саме у цей момент з’являється проблеми інтеграції цієї частини у використовуємий стиль коду. У багатьох випадках це додає декілька строк оформлених у прийнятому стилі, але існують випадки коли потрібно використати досить архаїчні а також громіздкі конструкції, які будуть дуже ускладнювати читабельність коду, але дадуть бажаний результат.

Поганий стиль коду, або велика кількість оптимізаційних рішень які його ускладнюють у кінцевому випадку можуть вплинути на час вирішення задачі. Саме через це найкраще рішення – використовувати шаблони для вирішення задачі, а у більш унікальних випадках виділяти оптимізований код у окремий блок, додаючи додаткові роз’яснення. Також оптимізацію коду можливо проводити після написання окремого елементу, але це потребую більше часу через додавання ще одного етапу тестування. Залежно від команди, задачі а

також оптимізаційного рішення можливо лише виходячи зі свого досвіду робити висновки що до такої оптимізації.

Однак, зважаючи на те що людина не може зберігати усі дані а також зв’язки під час розробки, а також схильна робити помилки – менше коду зазвичай буде працювати краще ніж більше. Це також додає додаткові аргументи, щоб робити додаткову а також громіздку чи складну оптимізацію після розробки компоненту.

2.1.3 Кожна оптимізація діє але ускладнює наступні

Через те що розробка ПЗ – комплексний та багатогранний процес, зміна однієї частини майже завжди вплине на іншу. Через такий розклад речей потрібно завжди пам’ятати що оптимізуючи одну частину – ми можемо ускладнити або погіршити оптимізацію іншої частини.

Кращі, та безпечніші місця оптимізації – кінцеві алгоритми (частини ПЗ які роблять одну дію яка впливає тільки на вихідні дані цієї частини). Через це ООП, а також інші архітектурно конструкторські рішення використовуються дуже часто. Вони дозволяють легше обробляти а також відокремлювати абстракції, роблячи кінцеве рішення модульним або секційним. У даному випадку більшість оптимізаційних рішень не буде впливати на інші частини проекту, але зміна загальної архітектури таких проектів дуже складна, та потребує завчасного продумування.

2.1.4 Оптимізація ЦП у сучасних системах

Найпростіший а також найефективніший метод оптимізації сучасних ЦП [24] – зменшення кількості інструкцій. Цей метод оптимізації найпростіший у реалізації, а також не потребує багато часу та знань. Але існують випадки коли потрібно використовувати потужності ЦП на максимум – у цих випадках потрібно розуміти роботу а також оптимізаційні рішення сучасних ЦП.

Сучасні ЦП оптимізуються за рахунок декількох підходів:

2.1.4.1 Виконання декількох команд за 1 інструкцію

Оптимізація за рахунок виконання складних команд за одну інструкцію, це дозволяє виконувати більший набор команд за такий самий набір інструкцій. Ця оптимізація реалізується на рівні заліза, але у випадку розробки ПЗ – її потрібно додатково використовувати під час компіляції коду. Використовуючи правильно налаштований компілятор який підтримує нові інструкції ми можемо отримати оптимізацію використавши мінімум часу не вносячи зміну до коду. Але слід окремо виділити випадки, що дана оптимізація буде працювати тільки на процесорах які її підтримують, на всіх інших процесорах це може не лише не надати результату, а й погіршити оптимізацію.

2.1.4.2 Кеш ЦП

Досить давно ЦП почав використовувати кеш [25] для оптимізації роботи, але слід розуміти саму природу а також можливості цієї оптимізації. Через те що швидкість передачі інформації кінцева, а також частота обробки різних компонентів системи різна виникає затримка. Для зберігання набору інструкцій виконання використовується оперативна пам’ять – як найшвидше та найпростіше рішення для зберігання достатньої кількості даних, а також швидкості їх передачі. Запит до оперативної пам’яті викликає затримки, які змушують ЦП «працювати в холосту». Для мінімізації цих запитів використовують процесорний кеш – пам’ять яка знаходить у ЦП, та використовується як буфер. Цей буфер зменшує затримки запитів на нові інструкції, але все одно потребує оновлення даних.

Це досить складний процес з технічної точки зору – де використовуються багато алгоритмів передбачення, але для програмної частини потрібно знати лише поверхневу роботу на яку можливо вплинути (рисунок 2.1):

1) ЦП дає запит до свого кешу на нові інструкції;
2) кеш центрального процесору перевіряє чи є лінія інструкцій яку потребує процесор;
3) якщо виконується «попадання» - кеш ЦП надає процесору інструкції, але якщо виконується «промах» - кеш ЦП оновлює дані з оперативної пам’яті, і тільки після цього надає їх процесору.

Усі ці етапи зав’язані на кеш-блоках – одиниці зберігання інформації у кеші. Якщо ваша програма максимально ефективно використовує кеш-блоки а також мінімізує їх оновлення з оперативної пам’яті – можливо отримати оптимізаційний приріст. Цей приріст зазвичай залежить від архітектури процесору а також багатьох інших параметрів.

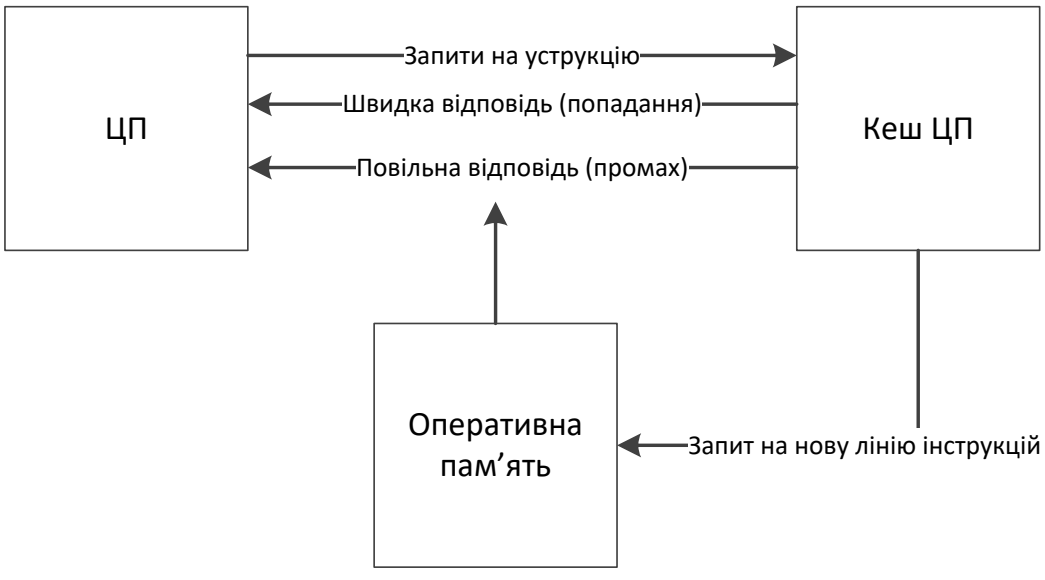

Рисунок 2.1 – Принцип роботи оптимізації ЦП за рахунок кешу ЦП

Даний тип оптимізації може буду непередбачуваний [26], а також потребує особливих знань при розробці та прийнятті алгоритмічно-архітектурних рішень. Використовувати його потрібно тільки у особливих випадках (наприклад при обробці великої кількості однотипних даних за мінімальний час). Але потрібно завжди пам’ятати про цю можливість оптимізації, так як вона інколи може дуже дешево налаштовуватися за допомогою правильних налаштувань компілятору, або правильної обробки і зберігання інформації у ПЗ.

2.1.4.3 Багатопоточність ЦП

Використання багатопоточності [27] почалось досить давно – для виконання декількох програм на одному ядрі (рисунок 2.2), але у теперішній час ми маємо процесори з великої кількістю фізичних ядер а також додатковою віртуалізацією.

Що робити з великою потужністю а також як її правильно використати? На це питання не можливо дати однозначну відповідь – через фізичні закони. Як бути - описано вище: «існує лише одне краще рішення, яке надає ідеальний результат», але поетапність обробки даних у процесі роботи ПЗ не дозволяє легко взяти та «розпиляти» її на частини які будуть виконуватися у своєму потоці. Потрібно розуміти що синхронізація даних між потоками досить дорога операція. При розробці багатопоточного ПЗ сліз завжди обережно працювати з сумісними даними у різних потоках.

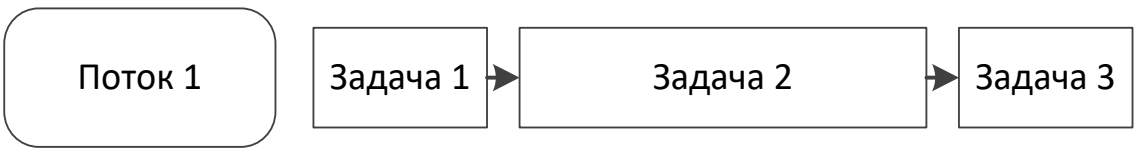

Рисунок 2.2 – Виконання задач у 1 потоці

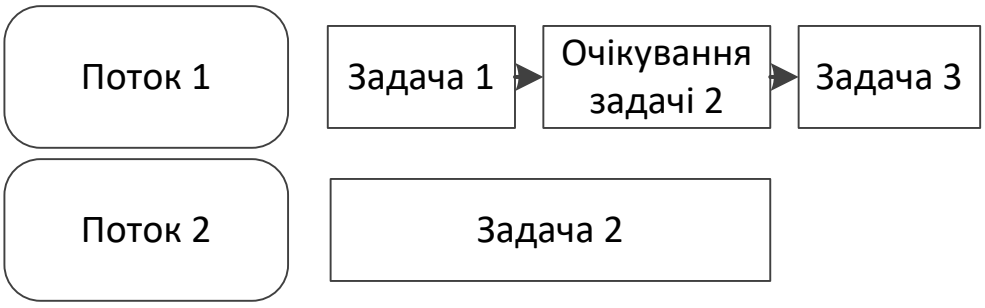

Рисунок 2.3 – Виконання задач при многопоточності у випадку правильного розподілення задач

До того ж багатопоточність добре може себе проявити при обробці незалежних даних (рисунок 2.3), але може погано вплинути на оптимізацію у випадках де виконання однієї задачі залежить від даних які може використовувати інша (рисунок 2.4).

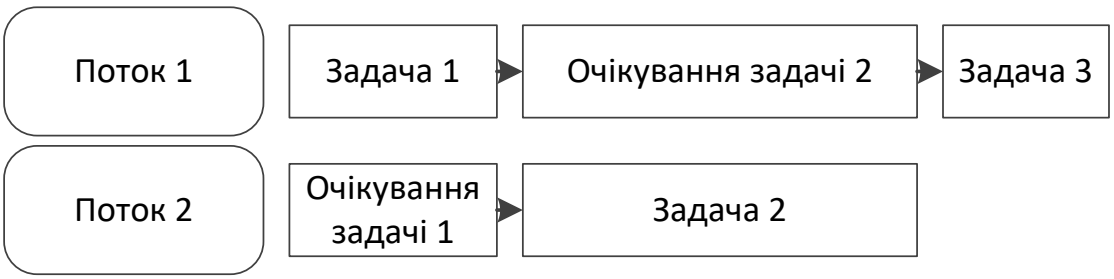

Рисунок 2.4 – Виконання задач при многопоточності у випадку не правильного розподілення задач

2.2 Оптимізаційний баланс та шляхи його досягнення

Оптимізаційний баланс – досить цікава річ при розробці ПЗ, з одного боку її намагаються досягнути усі та усюди, з іншого боку окремо її не виділяють та не обраховують. Це пов’язано з багатьма факторами, але найголовніший з них – важкість представлення ПЗ у вигляді математичної моделі а також його багатошарову абстрактність у реалізації. На відміну від фінансів, або менеджменту виробництва – процес розробки ПЗ на даний час занадто важко передбачити. Звісно ми можемо спроектувати окрему частину ПЗ, але найменші зовнішні фактори які впливають на розробку (людський фактор, різне розуміння термінів тощо…), і це може пошкодити завчасні плани розробки ПЗ і фінальний оптимізаційний баланс буде відрізнятися від початкового.

Також одним з найважливіших факторів пошуку оптимізаційного балансу є дані та статистика. Щоб зібрати дані – ми повинні розробити алгоритм, який ми і повинні визначити при використанні оптимізаційного балансу – це парадокс. Цю проблеми ми можемо вирішувати лише локально, для невеликих задач, але слід розуміти що оптимізація однієї частини може вплинути на іншу – тим самим ми можемо отримати зворотній ефект.

У фінансовій сфері при пошуку оптимізаційного балансу використовують умовні дані, і фінальний результат може буди не точним, а приблизним. Оптимізаційний баланс при розробці ПЗ повинен бути 100% точним. На це вливає фактор роботи комп’ютера – він робить те що йому скажеш. На відміну від виробництва або фінансів – оптимізаційний баланс для ПЗ намагається приблизитися до ідеалу, але тільки у випадку 100% гарантії роботи цього рішення.

З усього вищевикладеного виникає наступне:

- оптимізаційний баланс потребує даних, які ми можемо отримати тільки після розробки, що заважає завчасній оптимізації;
- при пошуку оптимізаційного балансу ми повинні опиратися тільки на фактичні дані.

Методи вирішення проблем пошуку оптимізаційного балансу у ПЗ можуть використовувати методики з фінансової сфери, але за умови використання фактичних даних. Отримання цих фактичних даних потребує розробки окремих частин чи систем, але ми можемо оптимізувати цей процес обравши найбільш залежні від оптимізації частини, тим самим економлячи час розробки. Використовуючи даний метод ми повинні у першу чергу визначити ці частини а також провести тестування їх, зібравши дані. Після цього ми проводимо аналіз отриманих даних, а також знаходимо найбільш оптимізований метод її використання – це і буде наш оптимізаційний баланс. Він не буде точним, але дозволить прийняти правильне оптимізаційне рішення, яке гарантовано дає результат.

Також слід відокремити випадки коли цей метод може нашкодити:

- через неправильну обробку даних, а також не коректний підбор даних для аналізу;
- через зміну взаємодії а також роботи елементів які були залежні від обрахованого оптимізаційного балансу;
- через хибне сприйняття алгоритмів або архітектурних рішень при обрахунку оптимізаційного балансу.

Для зменшення кількості помилок у фінальному результаті при використанні оптимізаційного балансу потрібно чітко визначити де його знаходження надає найбільший результат. Зазвичай це незмінні високонавантажені частини коду, які використовуються частіше за все, або обробляють велику кількість даних.

Наприклад при рендеру графіки оптимізаційний баланс краще шукати для частин які відповідають за логіку обробки та підготовки даних для створення зображення – ці частини будуть фіксованими на відміну від шейдерів, де краще проводити оптимізацію у процесі розробки, це у кінці дасть нам найбільшу вигоду від використання знайденого оптимізаційного балансу. Але потрібно звернути увагу не те що це актуально при нинішній архітектурній моделі відеокарт, використанні технологій які популярні та ефективні зараз. При зміні технологій та логіки обробки у майбутньому оптимізаційний баланс може змінити свій вплив на фінальну оптимізацію (Як у випадку з використання нових технологій ЦП).

2.3 Вплив часу розробки на прийняття кінцевих оптимізаційних рішень

Розробка будь якого ПЗ – потребує ресурсів. Кожна дія, кожне прийняття рішення, кожні строчка коду – потребує часу. ПЗ у першу чергу повинно вирішувати задачу, а у другу чергу вирішувати її максимально якісно.

Існує дуже багато підходів до розробки систем, які сприяють економії ресурсів. Але усі вони використовують фінансову точку зору розробки. Оптимізацію як окремий процес при розробці ПЗ використовують дуже рідко – це пов’язано з фінальною метою розробки майже усіх ПЗ. Звісно існують багато методів, коли оптимізація враховується як окремий елемент при розробці ПЗ – та найчастіше це або системи пов’язані з графікою, або високонавантажені системи, для обробки великої кількості даних.

Зважаючи на частий брак часу, а також максимальну економію, не використання оптимізаційного балансу досить логічне, адже на даний час більшість задач можливо вирішувати класичними оптимізаційними підходами. Але пошук оптимізаційного балансу цілком оправдовує себе при використанні його для критичних кінцевих алгоритмів (алгоритмів, які слабо впливають на усю систему в цілому, або тільки вирішують певну задачу). Одним з прикладів можливо вважати пошук кращих підходів при розробці алгоритму конвеєру рендеру. Через фактичну аналогічність з виробництвом, а також можливість провести не затратні експерименти для кожного етапу роботи цього конвеєру, пошук оптимізаційного балансу оправданий.

2.4 Слабкі місця оптимізації у конвеєрі рендеру

Загальна структура сучасного конвеєру рендеру растрової графіки використовує один прийнятий шаблон [28] алгоритму створення зображення. Інколи він може доповнюватися, або спрощуватися (залежно від додаткового ПЗ а також задач), але найчастіше його структура статична, та поділяється на декілька послідовних етапів. Кожен з цих етапів може бути або контрольованим ПЗ, або використовувати статичні рішення для оптимізації. Повний цикл, а також додаткові можливості роботи конвеєру рендеру описані у першому розділі цієї роботи.

Слід розуміти що усі етапи конвеєру рендеру – це лише послідовно виконувані алгоритми, які обробляють дані. Кожен з етапів цього конвеєру не може вплинути на минулий – це робить оптимізацію цього конвеєру виправданою за допомогою пошуку оптимізаційного балансу. Також слід зазначити що кожен з етапів є алгоритмом, який лише виконує свою задачу – що дозволяє швидко збирати фактичні дані для аналізу за допомогою тестових експериментів.

Основними слабкими місцями у даному конвеєрі є:

- передача а також оптимізація великої кількості даних і команд для алгоритму;
- виконання зайвих обрахунків усього конвеєру, через неможливість ним точно визначити які дані потрібно обробляти, та які обрахунки будуть зайвими (Кліпінг);
- забезпечення безпеки роботи конвеєру рендеру, за допомогою додаткових перевірок даних, які будуть оброблятися алгоритмом.

Саме ці частини мають декілька варіантів оптимізації, вибір серед яких найвигідніше робити її за допомогою пошуку оптимізаційного балансу. Кожен з цих етапів може додатково навантажити GPU або CPU, залежно від виконання задач. Також збір тестових даних буде досить швидким через локалізацію кожного етапу створення зображення, а також майже незалежність оптимізаційних підходів від попередніх етапів графічного конвеєру.

2.5 Методи збору даних для пошуку оптимізаційного балансу

Оптимізаційний баланс використовує аналіз фактичних даних, для знаходження найефективніших оптимізаційних рішень.

Для збору коректних даних ми повинні:

- виявити можливі оптимізаційні рішення;
- виявити які дані допоможуть нам оцінити ефективність прийняття цих рішень;
- визначити ПЗ, а також методи які дозволить нам зібрати точні дані.

Для коректного знаходження оптимізаційного балансу, ми повинні:

1) проаналізувати отримані дані;
2) визначивши який вплив мають можливі методи оптимізації;
3) після аналізу кожного з можливих оптимізаційних рішень, визначити який вплив вони можуть надавати один на одного;
4) визначити найкращу комбінацію оптимізаційних рішень, яка надає найліпші результати для досягнення оптимізаційного балансу.

Для коректного збору даних, було прийнято розробити ПЗ яке реалізує графічний конвеєр. Це дозволить нам повністю контролювати збір даних при різних оптимізаційних рішеннях. Також це надає змогу перевірити отриманні результати оптимізаційного балансу а також зробити точні висновки.

2.5.1 Функціональні вимоги до ПЗ для збору даних

Функціональні вимоги до ПЗ:

1) рендер зображення;
2) завантаження 3D моделей;

3) завантаження текстур;
4) ПЗ повинно мати базовий функціонал для можливості збору даних.

2.5.2 Не функціональні вимоги до ПЗ для збору даних

Не функціональні вимоги до ПЗ:
5) ПЗ повинно реалізовувати графічний конвеєр для растрової графіки;
6) ПЗ повинно бути максимально оптимізованим, для зменшення впливу реалізації на отримані тестові дані;
7) ПЗ повинно бути гнучким, для зменшення часу створення тестових ситуацій;
8) ПЗ повинно мати базовий функціонал для можливості збору даних.
9) ПЗ повинно працювати на операційній системі Windows 10 [29]

2.5.3 Виявлення можливих оптимізаційних рішень а також даних, які потрібні для їх аналізу

Навідь враховуючи сильну залежність графічного конвеєру від GPU, ми маємо досить багато можливостей оптимізації. Оптимізацію алгоритмів обробки даних ми проводити не будемо, так як вона не вливає на фактичну оптимізацію графічного конвеєру, а впливає лише на швидкість знайдення кінцевого результату. Зважаючи на це ми повинні виявити найслабші місця графічного конвеєру, які впливають на підготовку даних, або можуть знизити навантаження на обрахунки GPU.

2.5.3.1 Швидкості кліпінгу на GPU та CPU

Одним з найдешевших методів знизити кількість викликів на GPU – сортування даних, які використовуються при формуванні зображення. Через те що створення зображення використовує тривимірне зображення виникають випадки коли об’єкт який алгоритм обробляє не буде відображений на фінальному зображенні. На це може вплинути його розташування, перекриття іншим об’єктом, або особливі властивості. Слід розуміти що при малюванні векторної графіки використовуючи полігони ми завжди маємо дві сторони – повернуту до нас, або навпаки розвернуту у інший бік.

Для не викликання зайвих команд обробки даних, використовують алгоритми кліпінгу [30]. Ці алгоритми сортують дані які можуть відобразитися на екрані а також видаляють пропускають усі інші. Даний алгоритм можливо реалізувати як на стороні CPU так і на стороні GPU. Саме у питанні де краще реалізовувати алгоритм кліпінгу полягає оптимізаційна дилема.

Реалізація на стороні GPU обробляє усі трикутники і якщо який-небудь з них не потрапляє у поле зору камери – пропускає його.

Реалізація на стороні CPU – обробляє об’єкти які не потрапляють у поле зору камери, тим самим не відсилаючи до GPU команд малювати ці об’єкти.

Для аналізу обох методів ми повинні провести експерименти з наступними змінними даними:

- кількість команд малювання відправлених до GPU;
- кількість трикутників у моделі, які будуть оброблятися однією командою малювання;
- кількість трикутників які ми буде бачити камера;
- кількість трикутників які ми знаходяться за полем зору камери.

Усі тести ми повинні провести для наступної кількості команд рендеру:

- 10;
- 1 000;
- 100 000;
- 1 000 000;
- 1 000 000 000;

Усі тести ми повинні провести для наступної кількості трикутників у одній моделі:

- 1;
- 100;
- 1000;

Усі тести ми повинні провести у випадках коли:

- усі трикутники потрапляють у поле зору камери;
- усі трикутники не потрапляють у поле зору камери.

Вихідними даними для цих експериментів будуть час виконання циклу відправки усіх команд малювання на GPU з урахуванням додаткових алгоритмічних дій на стороні CPU (якщо це буде необхідно).

Ми можемо побачити на рисунку 2.5 - куб окрашений у жовтий колір, який знаходиться у середині оранжевої піраміди, яка представляє собою поле зору камери. А також оранжевий куб, який знаходиться осторонь – тим самим при формуванні зображення він не буде використовуватися за допомогою алгоритмів кліпінгу, тим самим оптимізувавши навантаження не GPU. На рисунку 2.6 ми можемо побачити кінцевий результат рендеру даної сцени – відображення кубу який потрапив до поля зору.

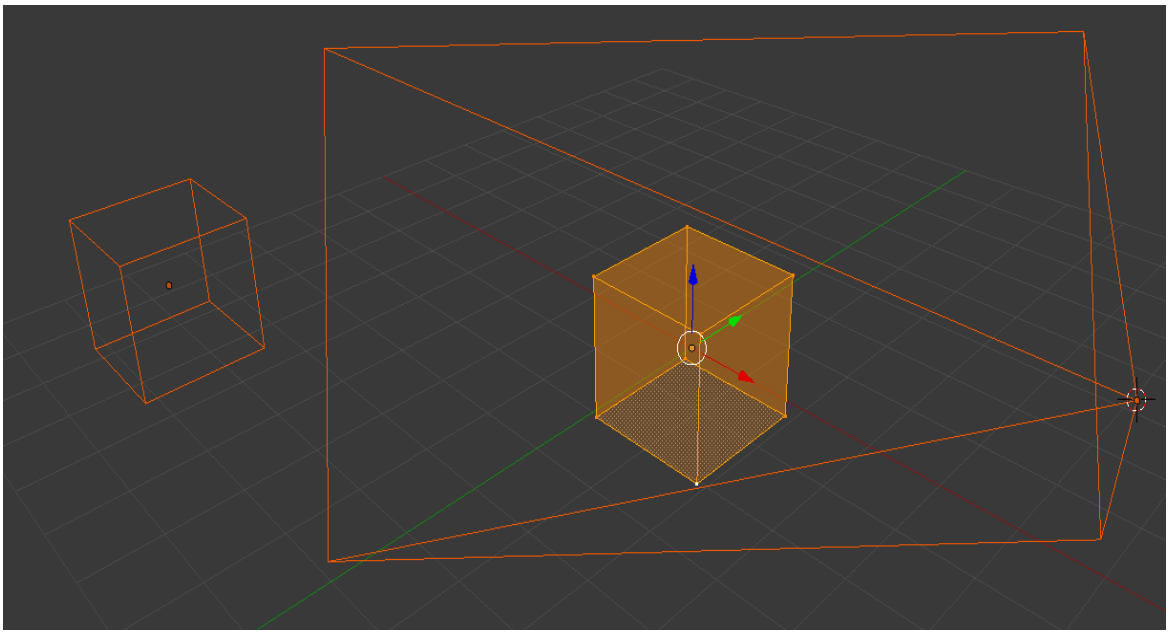

Рисунок 2.5 – Об’єкт який потрапляє у поле зору камери, а також об’єкт який туди не потрапляє

Рисунок 2.6 – Об’єкт який потрапив у поле зору камери був відображений

2.5.3.2 Необхідність зменшення кількості команд налаштування GPU для рендеру об’єкта за рахунок додаткових розрахунків CPU

Через залежність а також складність алгоритму рендеру зображення виникає потреба у налаштуваннях кожного з етапів конвеєру. Особливо це відчувається при використанні ПЗ яке використовується як інтерфейс між CPU та GPU. Дане ПЗ пришвидшує розробку кінцевого продукту реалізуючи основні процеси роботи з GPU. Звісно можливо контролювати процес роботи GPU самостійно, а також підлаштовувати алгоритми взаємодії під себе, але це потребує дуже багато ресурсів, знать та умінь. Це не оправдано дорого для розробки більшості програмних продуктів.

Кожна абстракція яка надає інтерфейс зазвичай має внутрішні налаштування для збільшення гнучкості його використання. Але слід пам’ятати, що при роботі з зовнішнім приладом (GPU у нашому випадку) існують дуже великі затримки при посиланні команд.

Тільки у дуже рідкісних випадках можливо точно виявити які дані буде використовувати GPU, а також завчасно їх підготувати і згрупувати. Але у абсолютній більшості для кожної команди малювання ми повинні виставляти налаштування рендеру – що змушує нас відсилати зайві, дорогі для оптимізації команди.

Ми можемо частково перенести перевірку а також сортування команд налаштування на CPU. З однієї сторони це повинно більше навантажити CPU зайвими обрахунками, які будуть потребувати додаткового використання пам’яті для зберігання налаштувань а також додаткової архітектури що буде робити ці обрахування. Слід також зазначити що даний метод має деяке «плато», від якого залежить чи принесе від оптимізацію чи погіршить час виконання алгоритму.

Для аналізу даного випадку у проведенні експерименту ми повинні заміряти наступні ситуації:

- дані для моделі були встановлені один раз;
- дані для моделі встановлювались кожен раз;
- дані для моделі оновлювались, якщо були встановлені інші (перевірка на стороні CPU).

Усі тести ми повинні провести для наступної кількості команд рендеру:

- 10;
- 1 000;
- 100 000;
- 1 000 000;
- 1 000 000 000;

Вихідними даними для цих експериментів будуть час виконання циклу відправки усіх команд малювання на GPU з урахуванням додаткових алгоритмічних дій на стороні CPU (якщо це буде необхідно).

2.5.3.3 Вплив оптимізаційних методів на стабільність роботи програми.

Під час розробки багато часу використовується на відлов помилок, а також інколи ці помилки можуть критично впливати на хід роботи програми. Інколи достатньо бути впевненими у коректності даних, а інколи їх краще перевіряти. Існує дуже багато ситуацій при створенні ПЗ які можуть себе проявити двояко. У випадку використання складних алгоритмів рендеру графіки ми повинні гарантувати якість даних, або робити додаткові перевірки, за для стабільності роботи програми.

Звичайно кожна перевірка яка використовує ресурси – це зазвичай рішення яке використовується через людський фактор. Велика кількість логіки та абстракцій впливає на важкість розуміння та сприйняття системи або алгоритму. Інколи помилки у даних можуть призвести до пошкодження фінального зображення, інколи до непередбачуваних випадків.

У ситуаціях коли поведінка програми непередбачувана ніколи не можливо дати гарантії правильності її роботи, але гарантування правильності виконання потребує додаткових інструкцій – перевірок.

Чи потрібно нам залишати додаткові перевірки для гарантування коректності роботи, або ми повинні їх видаляти для оптимізації? – на це питання ми можемо відповісти, тільки у випадку проведення експерименту, а також аналізу впливу додаткових перевірок на оптимізацію.

Для аналізу даного випадку у проведенні експерименту ми повинні заміряти наступні ситуації:

- перевірка даних буде повною;
- перевірки даних не буде;
- перевірка даних буде частковою (перевірка тільки на nullptr).

Усі тести ми повинні провести для наступної кількості команд рендеру:

- 10;
- 1 000;
- 100 000;
- 1 000 000;

Вихідними даними для цих експериментів будуть час виконання циклу відправки усіх команд малювання на GPU з урахуванням додаткових алгоритмічних дій на стороні CPU (якщо це буде необхідно).

2.5.4 Визначення необхідності розроблення ПЗ, а також методи які ми будемо використовувати для коректного збору даних

Для скорочення часу розробки ми будемо використовувати графічне ПЗ DirectX11 [31], який виступає інтерфейсом між драйверами GPU і розробником.

DirectX11 використовує архітектуру графічного конвеєру. Також він надає певний контроль над конвеєром створення зображення через шейдери. DirectX використовує два види шейдерів, які являються програмованими алгоритмами для обробки даних:

- вершинний шейдер [32] – викликається для обробки даних вершин;
- піксельний шейдер – викликається для обробки кольору кожного пікселю об’єкту.

Архітектура взаємодії DirectX11 з GPU намагається автоматизувати а також оптимізувати весь графічний конвеєр. При використанні його, потрібно лише встановити налаштування графічного конвеєру, шейдери а також встановити необхідні дані, які будуть використовуватися для створення зображення.

Виходячи з вищеописаного ми можемо зробити висновок, що DirectX надає нам можливість зекономити багато часу на розробку, використавши основні алгоритми, які були максимально оптимізовані за багато років підтримки. Це також буде сприяти на точність зібраних даних мінімізувавши погрішності, через перевірену стабільність DirectX11.

Для створення ПЗ буде використана Microsoft VisualStudio2019 [33], а також набір стандартних математичних бібліотек, і бібліотек які надають можливість створювати вікна під операційною системою Windows.

Для отримання часу ми будемо використовувати бібліотеку time.h, а також функцію з неї timeGetTime() – яка повертає час.

Для заміру часу роботи алгоритмів ми будемо використовувати наступний алгоритм:

1) зберігаємо час за допомогою функції timeGetTime();
2) виконання алгоритму;
3) використовуючи збережений час, також оновлений час отриманий за допомогою timeGetTime(), ми вираховуємо різницю між ними – це і буде проміжок часу виконання алгоритму;

Цей метод не є повністю точним, через його залежність від абстракцій CPU і інших факторів. Але він надає досить точний результат для проведення даних експериментів, забираючи мінімум часу на інтеграцію а також роботу.

Шейдери для тестових випадків будуть написані на мові HLSL [34].

2.6 Висновки з розділу 2

У другому розділі даної роботи було проаналізовано основні принципи оптимізації, можливі методи а також випадки оптимізації за рахунок CPU, оптимізаційні можливості сучасних CPU.

Було проаналізовано можливості використання оптимізаційного балансу для розробки ПЗ, а також випадки коли це необхідно.

Були виявлені слабкі місця оптимізації конвеєру рендеру для сучасного ПЗ. Можливі рішення оптимізації у даних випадках потребують додаткового аналізу та експериментів, через які ми повинні отримати фактичні дані які допоможуть нам визначити випадки їх використання.

Було визначено експерименти які будуть проведені:

- швидкості кліпінгу на GPU та CPU;
- необхідності зменшення кількості команд налаштування GPU для рендеру об’єкта за рахунок додаткових розрахунків CPU;
- впливу оптимізаційних методів на стабільність роботи програми.

Було визначено критерії аналізу даних випадків, а також які тестові дані будуть використовуватися для експериментів.

Для проведення експериментів було прийняте рішення розробити ПЗ – яке дозволить нам швидко, точно збирати потрібні нам дані, гарантуючи максимальний контроль над циклом рендеру. Дане ПЗ буде використовувати DirectX11 для роботи з GPU. Був визначений алгоритм заміру даних для експерименту.

3 ЗБІР РЕЗУЛЬТАТІВ ТА АНАЛІЗ ОТРИМАНИХ ДАНИХ

3.1 Розроблення ПЗ для збору даних

Для коректного збору даних було вирішено розробити ПЗ. Дане ПЗ повинно реалізовувати функціональні вимоги описані у розділі 2.5.1 даної роботи, не функціональні вимоги описані у розділі 2.5.2, а також які використовувати технології описані у розділі 2.5.3.

3.1.1 Проектування

Розробка ПЗ спрямована на рендер графіки у реальному часі. Тому ми повинні використовувати цикл для усіх необхідних операцій (Tick [35]). На рисунку 3.1 зображена схема взаємодії компонентів [36] програми, яка забезпечить максимальну гнучкість а також модульність для полегшення модифікування і збору даних для тестів.

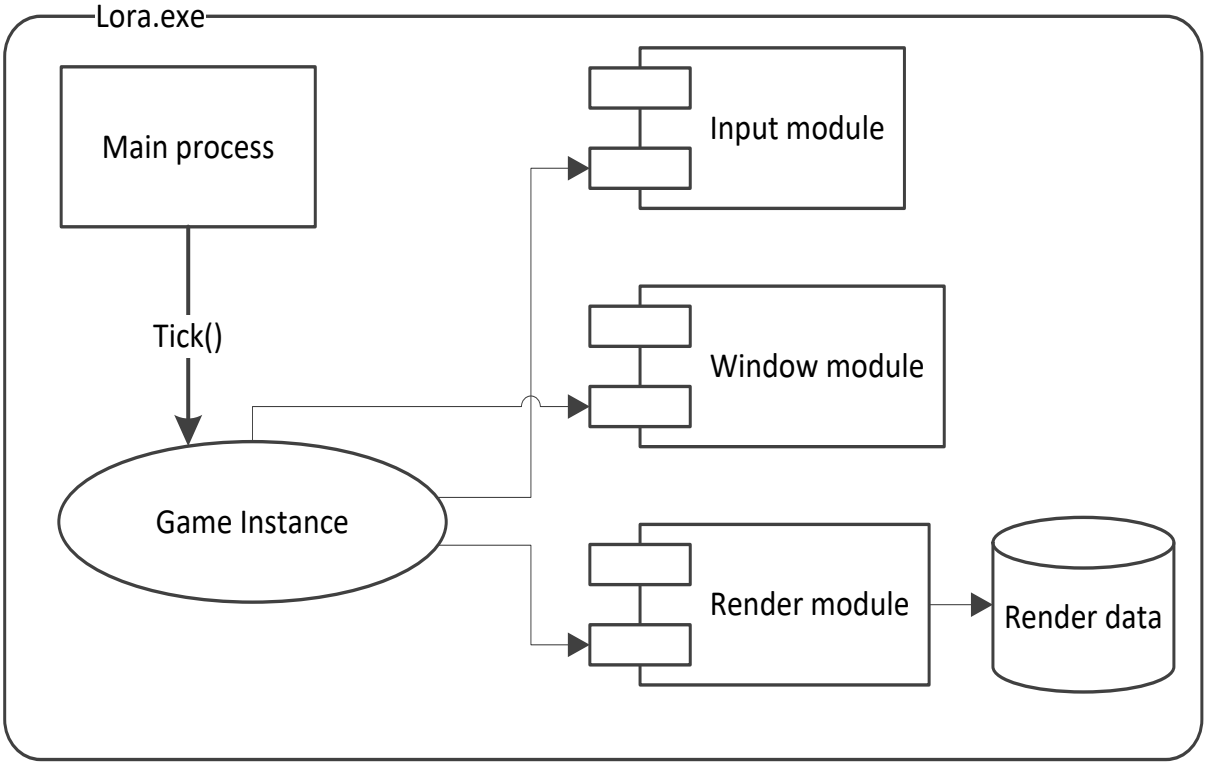

Рисунок 3.1 – Схема взаємодії компонентів ПЗ для збору даних

Кожну з цих компонентів буде представляти клас – інтерфейс [37], який буде надавати доступ до його функціоналу.

Першим етапом роботи програми буде ініціалізація усіх компонентів, на рисунку 3.2 ви можете побачити взаємодію усіх підсистем при ініціалізації.

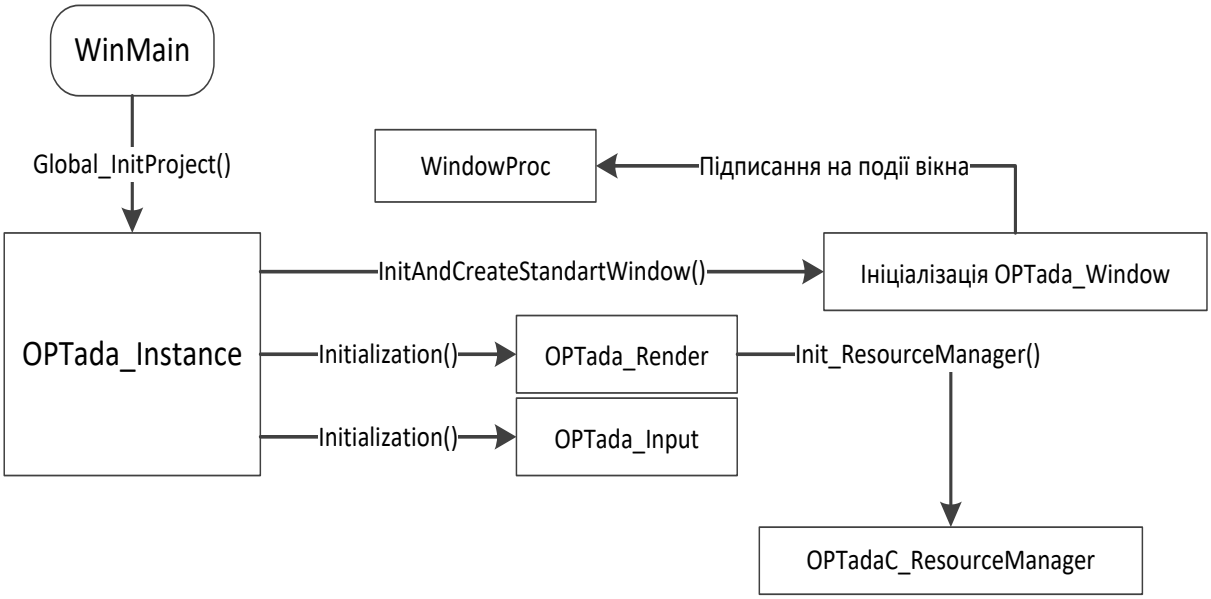

Рисунок 3.2 – Взаємодія підсистем при ініціалізації

Після даного етапу усі підсистеми будуть незалежні та зможуть взаємодіяти тільки через статичний клас, який буде інтерфейсом роботи програми OPTada_Instance. Як ми можемо побачити на рисунку 3.3, тільки компонент контроля вікна може надавати зворотні команди до інтерфейсу OPTada_Instance для зміни налаштувань, усі інші компоненти отримують нові дані через методи викликані у OPTada_Instance.

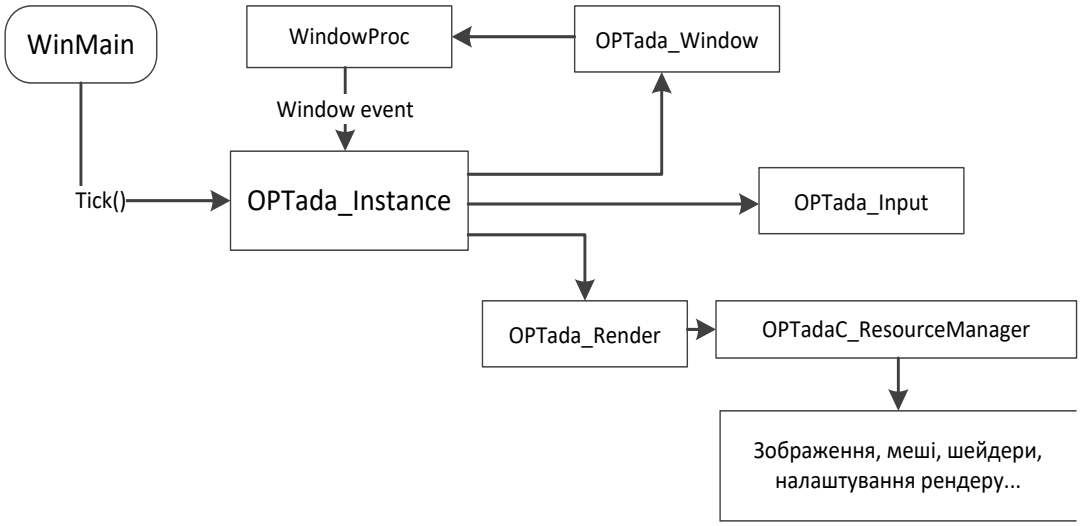

Рисунок 3.3 – Взаємодія класів-компонентів під час тіку

3.1.2 Тестування розробленого ПЗ

Для тестування розробленого ПЗ буде проведено автономне тестування. Цей метод тестування буде гарантувати відсутність критичних помилок при роботі програми, що дозволить нам бути впевненими в її стабільності при проведенні тестів.

3.1.2.1 План проведення автономного тестування

Автономне тестування буде проводитись для тестування роботи основних створених класів ПЗ.

3.1.2.2 Вибір способу виконання тестування

Тестування буде проводитись для усіх основних класів, за допомогою допоміжних інструментів тестування VisualStudio2019.

3.1.2.3 Визначення стратегії реалізації тестування

Тестування буде проводитися для класів:
- OPTada_Instance;
- OPTada_Render;
- OPTada_Window;
- OPTada_Input;
- GameLevel;
- OPTadaC_ResourceManager;
- OPTada_MemoryManager.

Усі методи які тестуються будуть проходити тести на базі позитивних вхідних даних.

3.1.2.4 Специфікація автономного тестування

Специфікація автономних тестів представлена нижче, в таблиці 3.1.

Таблиця 3.1 – Специфікація автономних тестів

Код тесту	Ім’я класу	Метод	Вхідні дані	Очікуваний результат
I1	OPTada_Instance	bool Global_InitProject	HINSTANCE hinstance_, WNDPROC windowProc_	Виконується повний цикл ініціалізації програми

Продовження таблиці 3.1

Код тесту	Ім’я класу	Метод	Вхідні дані	Очікуваний результат
I1	OPTada_Instance	bool Global_SetupProject	-	Виконується повний цикл налаштування програми
		void Global_ShutdownProject	-	Виконується повний цикл завершення роботи програми
		int Tick	float deltaTime_	Виконується тік програми
		int DrawScene	float deltaTime_	Виконується алгоритм рендеру зображення програми
		bool Do_Change_WindowSettings	OPTadaE_WindowState_ForClassWindow newWindowState_, OPTadaS_Window_Size& newWindowSize_, bool vSinc_, int countOfBackBuffers_	Виконується повний цикл обробки події зміни розміру вікна програми
		void Do_Reaction_LooseFocus	-	Реакція на подію втрати фокусу вікна програми
		void Do_Reaction_TakeFocus	-	Реакція на подію отримання або відновлення фокусу вікна програми
		void Do_Reaction_AltTab	-	Реакція ка комбінацію клавіш alt + tab

Продовження таблиці 3.1

Код тесту	Ім’я класу	Метод	Вхідні дані	Очікуваний результат
I1	OPTada_Instance	void Do_Reaction_AltEnter	-	Реакція ка комбінацію клавіш alt + tab (перехід до повноекранного відображення, та навпаки)
I2	OPTada_Render	bool Initialization	HWND hwnd_, int countOfBackBuffers_, int workspaceWidth_, int workspaceHeight_, bool vSinc_, bool isWindowedMode_	Виконання повного циклу ініціалізації модулю рендеру
		bool Init_AllRasterizerState	-	Ініціалізація усіх налаштувань растеризатора рендеру
		bool Init_AllBlendState	-	Ініціалізація усіх станів змішування пікселів
		bool InitializeSecondaryResources	int workspaceWidth_, int workspaceHeight_	Ініціалізація додаткових ресурсів модулю рендеру
		void ShuttingDown	-	Звільнення усієї пам’яті а також усіх компонентів модулю

Продовження таблиці 3.1

Код тесту	Ім’я класу	Метод	Вхідні дані	Очікуваний результат
I2	OPTada_Render	bool Setup_NewSettingsForRender	int workspaceWidth_, int workspaceHeight_, bool vSinc_, int countOfBackBuffers_	Змінення розміру кінцевого рендеру, а також допоміжних буфері і компонентів
		void Setup_FullScreenMode	bool isFullScreen_	Встановлення режиму повновіконного відображення, або його вимкнення
		void PrepareBuffersForNewFrame	const FLOAT clearColor[4], FLOAT clearDepth, UINT8 clearStencil	Підготовка усіх буферів які використовуютьс я у рендері зображення до нового рендеру
		void Setup_NewRasterizer	OPTadaE_RasterizerMass_ForRender rasterizaerEnum_	Встановлення нового растерайзеру
		void Setup_NewBlendState	OPTadaE_BlendStateMass_ForRender blendStateEnum_, float* blendFactor_	Встановлення новогу стану змішування пікселів для конвеєру рендеру
		void PresentFrame	-	Відобразити зображення

Продовження таблиці 3.1

Код тесту	Ім’я класу	Метод	Вхідні дані	Очікуваний результат
І3	OPTada_Window	void Update_WindowSizeWithBorders	-	Оновлення розміру вікна з урахуванням бортів відображення
		bool InitAndCreateStandartWindow	HINSTANCE hinstance_, WNDPROC windowProc_	Створення стандартного вікна, а також його первинна ініціалізація
		bool Change_DisplayOfWindow	OPTadaE_WindowState_ForClassWindow new_WindowState_, OPTadaS_Window_Size& new_WorkplaceSize_	Зміна стану відображення вікна, а також зміна його розміру
		void Get_MonitorSize	OPTadaS_Window_Size& monitorSize_	Отримання розміру монітору на якому відображається вікно
		void Get_WindowState	OPTadaE_WindowState_ForClassWindow& windowState_	Отримання стану вікна
		void Get_WindowSize	OPTadaS_Window_Size& windowSize_	Отримання розміру вікна
		void Get_WorkplaceSize	OPTadaS_Window_Size& workplaceSize_	Отримання розміру робочого простору вікна
		HWND Get_MainWindowHandle	-	Отримання дескриптору вікна

Продовження таблиці 3.1

Код тесту	Ім’я класу	Метод	Вхідні дані	Очікуваний результат
I3	OPTada_Instance	bool Do_SwapMode_Fullscreen_LastWindowed	-	Виконання реакції зміни повновіконного відображення на віконне, та навпаки
		bool Do_LooseFocusInFullscreenMode	-	Виконання реакції вікна на втрату фокусу
		bool Do_AltTabLooseFocusInFullscreenMode	-	Виконання реакції вікна на втрату фокусу при використанні комбінації alt + tab
		bool Do_RestoreFocusInFullscreenMode	-	Виконання реакції для відновлення фокусу вікна
I4	OPTada_Input	bool ReadKeyboard	-	Зчитування даних з клавіатури
		bool ReadMouse	-	Зчитування даних з миші
		bool Initialization	HINSTANCE hinstance_, HWND hwnd_, int newWorkspace_Width_, int newWorkspace_Height_, bool showCursor_, bool mouseIsLocked_	Ініціалізація модулю вводу

Продовження таблиці 3.1

Код тесту	Ім’я класу	Метод	Вхідні дані	Очікуваний результат
I4	OPTada_Instance	void ShuttingDown	-	Виконання реакції на завершення програми, а також звільнення усіх ресурсів
		bool Update	-	Оновлення даних усіх приладів вводу
		void Get_Input_8Mouse_256Keyboard	-	Зчитування та обробка даних для клавіатури
		void Set_Workspace	int newWorkspace_Width_, int newWorkspace_Height_	Встановлення розміру робочої частини вікна
		void Set_HaveFocus	bool isInFocus_	Реакція а також встановлення стану модулю вводу, а також зберігання стану фокусу
		void Set_LockMouse	bool isLocked_	Встановлення стану блокування миші
		void Set_ShowMouseCursor	bool isShow_	Встановлення стану відображення миші
I5	GameLevel	bool Init	-	Ініціалізація усіх ресурсів а також необхідних компонентів для створення зображення
		void Free	-	Звільнення усіх ресурсів
		bool Tick	float deltaTime_	Виконання логіки тіку

Продовження таблиці 3.1

Код тесту	Ім’я класу	Метод	Вхідні дані	Очікуваний результат
I6	OPTadaC_ ResourceM anager	bool Init_ResourceMana ger	ID3D11Device* gDevice_, std::vector<UIN T> constantBufferSi zeList_	Ініціалізацію менеджеру ресурсів
		void FreeAll	-	Звільненя усіх ресурсів якими володіє менеджер
		bool Create_PixelShader _FromBinaryFile	OPTadaE_Pixel ShaderList_For ResourceManag er shaderEnum_, const std::wstring& fileName_, ID3D11Device* gDevice_, std::vector<OPT adaE_SamplerSt ateList_ForReso urceManager> SampleStateList _	Створення піксельного шейдеру з файлу, а також його компіляція
		OPTadaS_PixelSha derStructure* Get_PixelShader_C ell	OPTadaE_Pixel ShaderList_For ResourceManag er shaderEnum_	Отримання піксельного шейдеру
		bool Use_PixelShader	OPTadaE_Pixel ShaderList_For ResourceManag er shaderEnum_, ID3D11DeviceC ontext* gDeviceContext _	Встановлення піксельного шейдеру
		bool Delete_PixelShader	ForResourceMa nager shaderEnum_	Видалення піксельного шейдеру

Продовження таблиці 3.1

Код тесту	Ім’я класу	Метод	Вхідні дані	Очікуваний результат
I6	OPTadaC_ ResourceM anager	bool Create_VertexShad er_FromBinaryFile	OPTadaE_Verte xShaderList_For ResourceManag er shaderEnum_, const std::wstring& fileName_, ID3D11Device* gDevice_, D3D11_INPUT _ELEMENT_D ESC* vertexLayoutDe sc_, UINT countOfvertexL ayoutDesc_	Створення вертексного шейдеру з файлу, а також його компіляція
		OPTadaS_VertexS haderStructure* Get_VertexShader_ Cell	-	Отримання вертексного шейдеру
		bool Use_VertexShader	OPTadaE_Verte xShaderList_For ResourceManag er shaderEnum_	Встановлення вертексного шейдеру
		bool Delete_VertexShad er	OPTadaE_Verte xShaderList_For ResourceManag er shaderEnum_	Видалення вертексного шейдеру

Продовження таблиці 3.1

Код тесту	Ім’я класу	Метод	Вхідні дані	Очікуваний результат
I6	OPTadaC_ ResourceM anager	bool Create_Mesh_From FileToMem	OPTadaE_Mesh List_ForResourc eManager meshName_, const std::string fileName_, ID3D11Device* gDevice_, UINT vertexStride_, UINT vertexOffset_, DXGI_FORMA T indexBufferFor mat_	Завантаження мешу з файлу
		void SetToDefault_Mes hCell	OPTadaE_Mesh List_ForResourc eManager meshName_	Очищення простору для мешу
		bool Load_ToGPU_Mes h	OPTadaE_Mesh List_ForResourc eManager meshName_, ID3D11Device* device_d3d11_	Завантаження мешу до пам’яті GPU
		void Unload_FromGPU _Mesh	OPTadaE_Mesh List_ForResourc eManager meshName_	Видалення мешу з пам’яті GPU
		bool Use_Mesh_WithIn dexBuffer	OPTadaE_Mesh List_ForResourc eManager meshName_, ID3D11DeviceC ontext* gDeviceContext _	Використання мешу, та встановлення його у якості ресурсу

Продовження таблиці 3.1

Код тесту	Ім’я класу	Метод	Вхідні дані	Очікуваний результат
I6	OPTadaC_ ResourceM anager	OPTadaS_MeshStr ucture* Get_MeshCell	OPTadaE_Mesh List_ForResourc eManager meshName_	Отримання даних про меш
		OPTadaS_MeshStr ucture* Get_MeshCell_IfIn GPU	OPTadaE_Mesh List_ForResourc eManager meshName_	Отримання ресурсу мешу
		bool Create_Texture_Lo adFromFile	OPTadaE_Textu reList_ForResou rceManager textureEnum_, const std::wstring& fileName_, ID3D11Device* gDevice_	Створення та завантаження текстури з файлу
		OPTadaS_TextureS tructure* Get_Texture_Cell	OPTadaE_Textu reList_ForResou rceManager textureEnum_	Отримання структури даних текстури
		bool Use_Texture	OPTadaE_Textu reList_ForResou rceManager textureEnum_, ID3D11DeviceC ontext* gDeviceContext _, UINT resourceSlot_	Встановлення текстури як ресурсу
		bool Delete_Texture	OPTadaE_Textu reList_ForResou rceManager textureEnum_	Видалення текстури та звільнення пам’яті

Продовження таблиці 3.1

Код тесту	Ім’я класу	Метод	Вхідні дані	Очікуваний результат
I6	OPTadaC_ ResourceM anager	void UpdateSubresource	OPTadaE_Const antBufferList_F orResourceMan ager constantBufferE num_, void* linkOnData_, ID3D11DeviceC ontext* gDeviceContext _	Оновлення константного буферу
I7	OPTada_M emoryMana ger	bool Init_Mamager	int countOfBuffers_	Ініціалізація менеджеру пам’яті
		bool Free_Manager	-	Звільнення усієї захопленої пам’яті
		OPTadaS_Key_Me moryManager* CreateNewMemory Buffer	int bufferID_, size_t memoryLength_ , size_t cellBuffer_Size_ , size_t cellOfDefragme ntation_Size_, OPTadaE_Buffe rTypes_ForMem oryManager bufferType_	Створення нового буферу пам’яті
		bool DeleteMemoryBuff e	OPTadaS_Key_ MemoryManage r** key_Buffer_	Видалення буферу за ключем
		bool Clear_Buffer	OPTadaS_Key_ MemoryManage r* key_Buffer_	Очищення буферу за ключем
		void* GetMemory	OPTadaS_Key_ MemoryManage r* key_Buffer_, size_t size_	Отримання пам’яті певного розміру у буфера за ключем

Закінчення таблиці 3.1

Код тесту	Ім’я класу	Метод	Вхідні дані	Очікуваний результат
I7	OPTada_MemoryManager	size_t Get_LockedMemory	OPTadaS_Key_MemoryManager* key_Buffer_	Отримання заблокованої пам’яті у буфері за ключем
		size_t Get_BufferMemorySize	OPTadaS_Key_MemoryManager* key_Buffer_	Отримання захопленої пам’яті буфером за ключем
		size_t Get_AllModulesLockedMemory	-	Отримання усієї захопленої пам’яті менеджером пам’яті

3.1.2.5 Проведення автономного тестування

Результат проходження автономних тестів представлений нижче, в таблиці 3.2.

Таблиця 3.2 – Результат проведення автономного тестування

Код тесту	Ім’я класу	Метод	Вхідні дані	Отриманий результат	Тест пройдено
II1	OPTada_Instance	bool Global_InitProject	HINSTANCE hinstance_, WNDPROC windowProc_	Виконується повний цикл ініціалізації програми	+
		bool Global_SetupProject	-	Виконується повний цикл налаштування програми	+

Продовження таблиці 3.2

Код тесту	Ім’я класу	Метод	Вхідні дані	Отриманий результат	Тест пройдено
II1	OPTada_Instance	void Global_ShutdownProject	-	Виконується повний цикл завершення роботи програми	+
		int Tick	float deltaTime_	Виконується тік програми	+
		int DrawScene	float deltaTime_	Виконується алгоритм рендеру зображення програми	+
		bool Do_Change_WindowSettings	OPTadaE_WindowState_ForClassWindow newWindowState_, OPTadaS_Window_Size& newWindowSize_, bool vSinc_, int countOfBackBuffers_	Виконується повний цикл обробки події зміни розміру вікна програми	+
		void Do_Reaction_LooseFocus	-	Реакція на подію втрати фокусу вікна програми	+

Продовження таблиці 3.2

Код тесту	Ім’я класу	Метод	Вхідні дані	Отриманий результат	Тест пройдено
II1	OPTada_Instance	void Do_Reaction_TakeFocus	-	Реакція на подію отримання або відновлення фокусу вікна програми	+
		void Do_Reaction_AltTab	-	Реакція ка комбінацію клавіш alt + tab	+
		void Do_Reaction_AltEnter	-	Реакція ка комбінацію клавіш alt + tab (перехід до повноекранного відображення, та навпаки)	+
II2	OPTada_Render	bool Initialization	HWND hwnd_, int countOfBackBuffers_, int workspaceWidth_, int workspaceHeight_, bool vSinc_, bool isWindowedMode_	Виконання повного циклу ініціалізації модулю рендеру	+

Продовження таблиці 3.2

Код тесту	Ім’я класу	Метод	Вхідні дані	Отриманий результат	Тест пройдено
II2	OPTada_Render	bool Init_AllRasterizerState	-	Ініціалізація усіх налаштувань растеризатора рендеру	+
		bool Init_AllBlendState	-	Ініціалізація усіх станів змішування пікселів	+
		bool InitializeSecondaryResources	int workspaceWidth_, int workspaceHeight_	Ініціалізація додаткових ресурсів модулю рендеру	+
		void ShuttingDown	-	Звільнення усієї пам’яті а також усіх компоненті в модулю	+
		bool Setup_NewSettingsForRender	int workspaceWidth_, int workspaceHeight_, bool vSinc_, int countOfBackBuffers_	Змінення розміру кінцевого рендеру, а також допоміжних буфері і компоненті в	+

Продовження таблиці 3.2

Код тесту	Ім’я класу	Метод	Вхідні дані	Отриманий результат	Тест пройдено
II2	OPTada_Render	void Setup_FullScreenMode	bool isFullScreen_	Встановлення режиму повновіконного відображення, або його вимкнення	+
		void PrepareBuffersForNewFrame	const FLOAT clearColor[4], FLOAT clearDepth, UINT8 clearStencil	Підготовка усіх буферів які використовуються у рендері зображення до нового рендеру	+
		void Setup_NewRasterizer	OPTadaE_RasterizerMass_ForRender rasterizaerEnum_	Встановлення нового растерайзеру	+
		void Setup_NewBlendState	OPTadaE_BlendStateMass_ForRender blendStateEnum_, float* blendFactor_	Встановлення новогу стану змішування пікселів для конвеєру рендеру	+
		void PresentFrame	-	Відображення зображення	+
II3	OPTada_Window	void Update_WindowSizeWithBorders	-	Оновлення розміру вікна з урахуванням бортів відображення	+

Продовження таблиці 3.2

Код тесту	Ім’я класу	Метод	Вхідні дані	Отриманий результат	Тест пройдено
ІІ3	OPTada_Window	bool InitAndCreateStandartWindow	HINSTANCE hinstance_, WNDPROC windowProc_	Створення стандартного вікна, а також його первинна ініціалізація	+
		bool Change_DisplayOfWindow	OPTadaE_WindowState_ForClassWindow new_WindowState_, OPTadaS_Window_Size& new_WorkplaceSize_	Зміна стану відображення вікна, а також зміна його розміру	+
		void Get_MonitorSize	OPTadaS_Window_Size& monitorSize_	Отримання розміру монітору на якому відображається вікно	+
		void Get_WindowState	OPTadaE_WindowState_ForClassWindow& windowState_	Отримання стану вікна	+
		void Get_WindowSize	OPTadaS_Window_Size& windowSize_	Отримання розміру вікна	+
		void Get_WorkplaceSize	OPTadaS_Window_Size& workplaceSize_	Отримання розміру робочого простору вікна	+
		HWND Get_MainWindowHandle	-	Отримання дескриптору вікна	+

Продовження таблиці 3.2

Код тесту	Ім’я класу	Метод	Вхідні дані	Отриманий результат	Тест пройдено
II3	OPTada_Window	bool Do_SwapMode_Fullscreen_LastWindowed	-	Виконання реакції зміни повновіконного відображення на віконне, та навпаки	+
		bool Do_LooseFocusInFullscreenMode	-	Виконання реакції вікна на втрату фокусу	+
		bool Do_AltTabLooseFocusInFullscreenMode	-	Виконання реакції вікна на втрату фокусу при використанні комбінації alt + tab	+
		bool Do_RestoreFocusInFullscreenMode	-	Виконання реакції для відновлення фокусу вікна	+
II4	OPTada_Input	bool ReadKeyboard	-	Зчитування даних з клавіатури	+
		bool ReadMouse	-	Зчитування даних з миші	+

Продовження таблиці 3.2

Код тесту	Ім’я класу	Метод	Вхідні дані	Отриманий результат	Тест пройдено
II4	OPTada_Input	bool Initialization	HINSTANCE hinstance_, HWND hwnd_, int newWorkspace_Width_, int newWorkspace_Height_, bool showCursor_, bool mouseIsLocked_	Ініціалізація модулю вводу	+
		void ShuttingDown	-	Виконання реакції на завершення програми, а також звільнення усіх ресурсів	+
		bool Update	-	Оновлення даних усіх приладів вводу	+
		void Get_Input_8Mouse_256Keyboard	-	Зчитування та обробка даних для клавіатури	+
		void Set_Workspace	int newWorkspace_Width_, int newWorkspace_Height_	Встановлення розміру робочої частини вікна	+

Продовження таблиці 3.2

Код тесту	Ім’я класу	Метод	Вхідні дані	Отриманий результат	Тест пройдено
II4	OPTada_Input	void Set_HaveFocus	bool isInFocus_	Реакція а також встановлення стану модулю вводу, а також зберігання стану фокусу	+
		void Set_LockMouse	bool isLocked_	Встановлення стану блокування миші	+
		void Set_ShowMouseCursor	bool isShow_	Встановлення стану відображення миші	+
II5	GameLevel	bool Init	-	Ініціалізація усіх ресурсів а також необхідних компонентів для створення зображення	+
		void Free	-	Звільнення усіх ресурсів	+
		bool Tick	float deltaTime_	Виконання логіки тіку	+

Продовження таблиці 3.2

Код тесту	Ім’я класу	Метод	Вхідні дані	Отриманий результат	Тест пройдено
II6	OPTadaC_ ResourceM anager	bool Init_ResourceM anager	ID3D11Devic e* gDevice_, std::vector<U INT> constantBuffe rSizeList_	Ініціалізаці ю менеджеру ресурсів	+
		void FreeAll	-	Звільненя усіх ресурсів якими володіє менеджер	+
		bool Create_PixelSha der_FromBinary File	OPTadaE_Pix elShaderList_ ForResource Manager shaderEnum_, const std::wstring& fileName_, ID3D11Devic e* gDevice_, std::vector<O PTadaE_Sam plerStateList_ ForResource Manager> SampleStateL ist_	Створення піксельного шейдеру з файлу, а також його компіляція	+
		OPTadaS_Pixel ShaderStructure * Get_PixelShader _Cell	OPTadaE_Pix elShaderList_ ForResource Manager shaderEnum_	Отримання піксельного шейдеру	+

Продовження таблиці 3.2

Код тесту	Ім’я класу	Метод	Вхідні дані	Отриманий результат	Тест пройдено
II6	OPTadaC_ResourceManager	bool Use_PixelShader	OPTadaE_PixelShaderList_ForResourceManager shaderEnum_, ID3D11DeviceContext* gDeviceContext_	Встановлення піксельного шейдеру	+
		bool Delete_PixelShader	OPTadaE_PixelShaderList_ForResourceManager shaderEnum_	Видалення піксельного шейдеру	+
		bool Create_VertexShader_FromBinaryFile	OPTadaE_VertexShaderList_ForResourceManager shaderEnum_, const std::wstring& fileName_, ID3D11Device* gDevice_, D3D11_INPUT_ELEMENT_DESC* vertexLayoutDesc_, UINT countOfvertexLayoutDesc_	Створення вертексного шейдеру з файлу, а також його компіляція	+
		OPTadaS_VertexShaderStructure* Get_VertexShader_Cell	-	Отримання вертексного шейдеру	+

Продовження таблиці 3.2

Код тесту	Ім’я класу	Метод	Вхідні дані	Отриманий результат	Тест пройдено
II6	OPTadaC_ ResourceM anager	bool Use_VertexShad er	OPTadaE_Ve rtexShaderLis t_ForResourc eManager shaderEnum_	Встановлен ня вертексного шейдеру	+
		bool Delete_VertexS hader	OPTadaE_Ve rtexShaderLis t_ForResourc eManager shaderEnum_	Видалення вертексного шейдеру	+
		bool Create_Mesh_Fr omFileToMem	OPTadaE_Me shList_ForRe sourceManag er meshName_, const std::string fileName_, ID3D11Devic e* gDevice_, UINT vertexStride_, UINT vertexOffset_, DXGI_FOR MAT indexBufferF ormat_	Завантажен ня мешу з файлу	+
		void SetToDefault_M eshCell	OPTadaE_Me shList_ForRe sourceManag er meshName_	Очищення простору для мешу	+

Продовження таблиці 3.2

Код тесту	Ім’я класу	Метод	Вхідні дані	Отриманий результат	Тест пройдено
II6	OPTadaC_ ResourceM anager	bool Load_ToGPU_ Mesh	OPTadaE_Me shList_ForRe sourceManag er meshName_, ID3D11Devic e* device_d3d11 _	Завантажен ня мешу до пам’яті GPU	+
		void Unload_FromG PU_Mesh	OPTadaE_Me shList_ForRe sourceManag er meshName_	Видалення мешу з пам’яті GPU	+
		bool Use_Mesh_With IndexBuffer	OPTadaE_Me shList_ForRe sourceManag er meshName_, ID3D11Devic eContext* gDeviceConte xt_	Використан ня мешу, та встановлен ня його у якості ресурсу	+
		OPTadaS_Mesh Structure* Get_MeshCell	OPTadaE_Me shList_ForRe sourceManag er meshName_	Отримання даних про меш	+
		OPTadaS_Mesh Structure* Get_MeshCell_I fInGPU	OPTadaE_Me shList_ForRe sourceManag er meshName_	Отримання ресурсу мешу	+

Продовження таблиці 3.2

Код тесту	Ім’я класу	Метод	Вхідні дані	Отриманий результат	Тест пройдено
II6	OPTadaC_ ResourceM anager	bool Create_Texture_ LoadFromFile	OPTadaE_Te xtureList_For ResourceMan ager textureEnum_ , const std::wstring& fileName_, ID3D11Devic e* gDevice_	Створення та завантажен ня текстури з файлу	+
		OPTadaS_Textu reStructure* Get_Texture_Ce ll	OPTadaE_Te xtureList_For ResourceMan ager textureEnum_	Отримання структури даних текстури	+
		bool Use_Texture	OPTadaE_Te xtureList_For ResourceMan ager textureEnum_ , ID3D11Devic eContext* gDeviceConte xt_, UINT resourceSlot_	Встановлен ня текстури як ресурсу	+
		bool Delete_Texture	OPTadaE_Te xtureList_For ResourceMan ager textureEnum_	Видалення текстури та звільнення пам’яті	+

Продовження таблиці 3.2

Код тесту	Ім’я класу	Метод	Вхідні дані	Отриманий результат	Тест пройдено
II6	OPTadaC_ ResourceM anager	void UpdateSubresou rce	OPTadaE_Co nstantBufferL ist_ForResour ceManager constantBuffe rEnum_, void* linkOnData_, ID3D11Devic eContext* gDeviceConte xt_	Оновлення константног о буферу	+
II7	OPTada_M emoryMan ager	bool Init_Mamager	int countOfBuffe rs_	Ініціалізація менеджеру пам’яті	+
		bool Free_Manager	-	Звільнення усієї захопленої пам’яті	+
		OPTadaS_Key_ MemoryManage r* CreateNewMem oryBuffer	int bufferID_, size_t memoryLengt h_, size_t cellBuffer_Si ze_, size_t cellOfDefrag mentation_Siz e_, OPTadaE_Bu fferTypes_For MemoryMana ger bufferType_	Створення нового буферу пам’яті	+
		bool DeleteMemoryB uffe	OPTadaS_Ke y_MemoryMa nager** key_Buffer_	Видалення буферу за ключем	+

Закінчення таблиці 3.2

Код тесту	Ім’я класу	Метод	Вхідні дані	Отриманий результат	Тест пройдено
II7	OPTada_MemoryManager	bool Clear_Buffer	OPTadaS_Key_MemoryManager* key_Buffer_	Очищення буферу за ключем	+
		void* GetMemory	OPTadaS_Key_MemoryManager* key_Buffer_, size_t size_	Отримання пам’яті певного розміру у буфера за ключем	+
		size_t Get_LockedMemory	OPTadaS_Key_MemoryManager* key_Buffer_	Отримання заблокованої пам’яті у буфері за ключем	+
		size_t Get_BufferMemorySize	OPTadaS_Key_MemoryManager* key_Buffer_	Отримання захопленої пам’яті буфером за ключем	+
		size_t Get_AllModulesLockedMemory	-	Отримання усієї захопленої пам’яті менеджером пам’яті	+

3.1.2.6 Результати тестування

Усі тести були пройдені успішно, ПЗ можливо вважати достатньо стабільним для проведення експериментів.

3.1.3 Характеристики розробленого ПЗ

Результатом розробки є програмний проект Lora. Дане програмне забезпечення допоможе зібрати точні дані для аналізу ефективності методів оптимізації, а також перевірити гіпотези що до знаходження оптимізаційного балансу

Розроблене ПЗ представляє собою комплекс підсистем та архітектурних рішень для малювання та відображення графіки (надалі ми будемо називати даний процес «рендер» або «render»), на операційній системі Windows 10 x64. Взаємодія даних підсистем спрямована на реалізацію принципу проекту реального часу. Основою даного проекту є циклічність роботи підсистем з орієнтацією на максимальну продуктивність циклу (Надалі даний цикл буде називатися «тік» або «tick»).

Кожна з підсистем розроблених у даному ПЗ виконує певні задачі, усі системи є модульними та не використовують ресурси один одного. У разі потреби усі операції синхронізації а також виклик подій які можуть передавати дані, виконуються через методи класу-інтерфейсу OPTada_Instance (рисунок 3.4). Даний клас реалізує у собі ідею контролю основних процесів а також систем через події. Усі методи окрім конструктора а також метода Tick – використовуються для виклику певних реакцій які впливають на всю програму глобально. У свою чергу метод Tick викликається через базовий клас WinMain кожного циклу програми. Він приймає аргумент функції – дельту часу (час який пройшов у програмі з моменту початку минулого Tick).

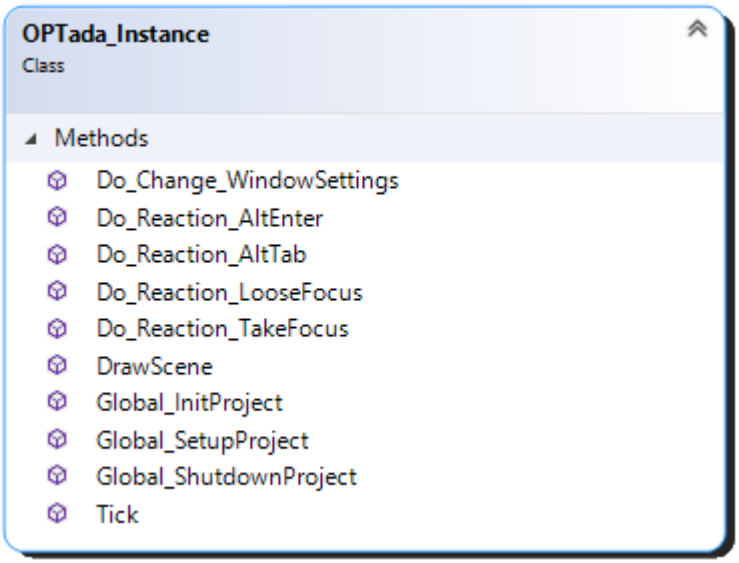

Рисунок 3.4 - Сигнатура головного класу-інтерфейсу програми

Програмні модулі які були розроблені для збору даних у даній роботі:

3.1.3.1 Модуль вікна програми

Основним класом даного модуля є OPTada_Window (рисунок 3.5). Цей клас використовує бібліотеки Windows, для створення та контролю вікна відображення у даній операційній систем. Через принципи архітектури операційної системи Windows, а також методи роботи програм під нею, для

перехоплення деяких подій які можуть бути викликані ззовні програми, використовуються клас WindowProc. За допомогою даного класу перехоплюються події які посилаються вікну операційною системою, а також іншими програмами. Через це, для коректного використання даного модуля, клас WindowProc потрібно помістити над основною функцією програми (main-функцією).

Даний модуль є підійно-опросним, та не потребує оновлення кожного tick програми, однак він потребує ініціалізації для створення вікна програми. Цю ініціалізацію потрібно робити через метод InitAndCreateStandartWindow основного класу даного модуля. Після успішної ініціалізації буде створене вікно, яке ми будемо використовувати для виводу інформації на екран.

Даний модуль також реалізує у собі методи для швидкого змінення параметрів вікна, а також реалізацію переходу вікна у повно-віконне відображення («full-screen»), або віконне («windowed»).

Загалом даний модуль буде використаній для створення вікна, його налаштування а також для отримання інтерфейсу вікна – за допомогою якого ми зможемо відображати у ньому графіку.

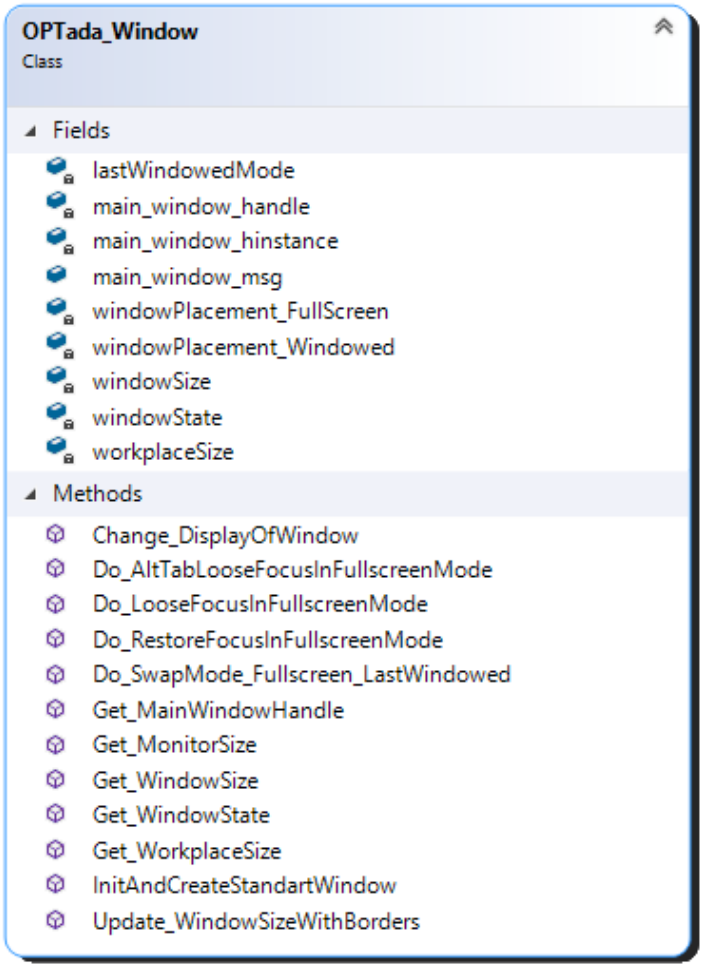

Рисунок 3.5 - Сигнатура класу для контроля вікна

3.1.3.2 Модуль вводу

Основним класом даного модуля є OPTada_Input (рисунок 3.6). Цей клас використовує бібліотеку DirectInput, для отримання команд вводу від користувача. Для використання цього модулю необхідно об’явити клас OPTada_Input, а також викликати його метод Initialization перед використанням. Даний метод повертає код помилки у разі виникнення проблем при ініціалізації. Після успішної ініціалізації потрібно оновити дані вводу користувача викликавши метод Update. Метод Update потрібно викликати кожен раз, через те що принцип збору оновлених команд від користувача є опитування.

За допомогою інших методів можливо отримати дані вводу від різних пристроїв вводу, таких як:

- комп’ютерна мишка (8 кнопок, координати у вікні, дельта руху миші);
- клавіатура на 256 кнопок.

Також даний модуль дозволяє блокувати курсор миші по середині екрану, а також керувати його відображенням.

Усі ці функції даного модуля будуть використані при розпізнаванні команд вводу, для керування станом роботи програми.

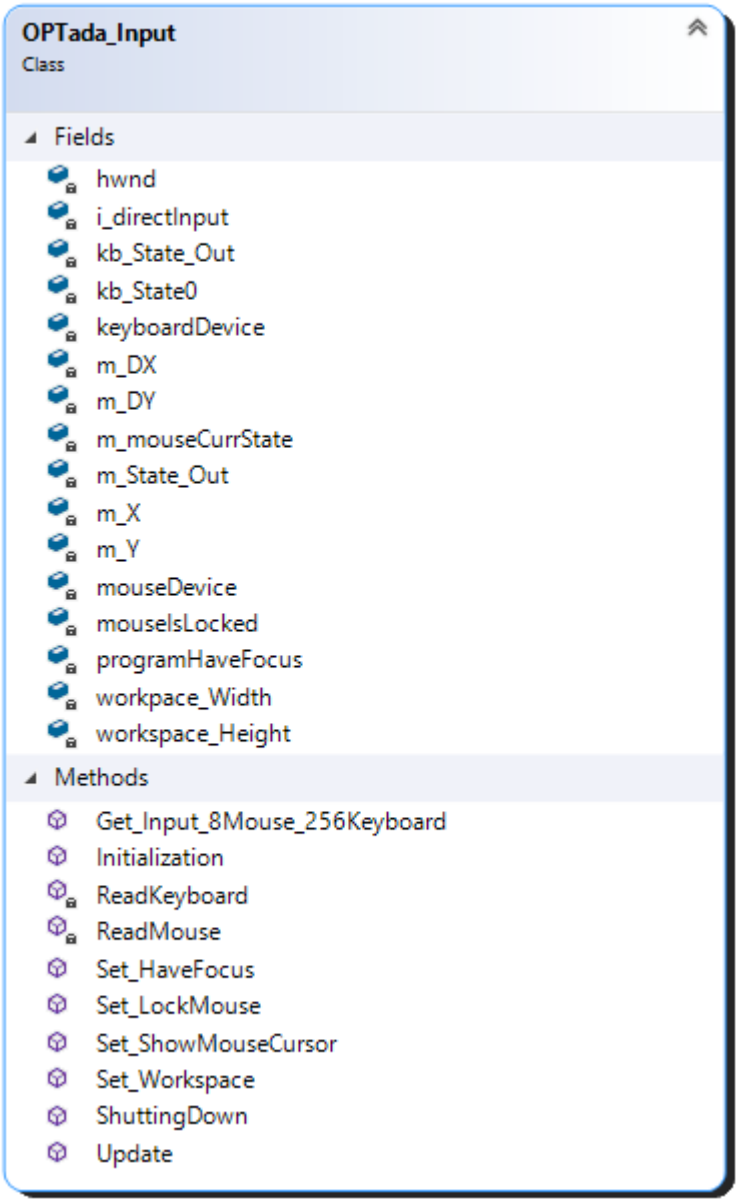

Рисунок 3.6 - Сигнатура класу для отримання вводу користувача

3.1.3.3 Модуль для рендеру графіки

Основним класом даного модуля є OPTada_Render (рисунок 3.7). Цей модуль використовує графічний API DirectX11 для рендеру 3D графіки, та відображення її у вікні. Даний модуль має у собі каскад підсистем, направлених на керування даними які використовуються у конвеєрі створення зображення «render pipeline». Усі архітектурні рішення у даному модулі спрямовані на оптимізацію, відкритість та легко доступність даних, автоматизацію а також стабільність (безперебійність) роботи програми у разі помилок.

Підсистеми які має у собі даний модуль спрямовані на вирішення певних задач які стосуються керування ресурсами. OPTada_Render потребує

ініціалізації після якої, у випадку її успішності, може використовуватися для надсилання команд до GPU.

Через комплексність архітектури систем графіки, вони потребують певного менеджменту ресурсів різних рівнів:

1) ресурси налаштування GPU
2) допоміжні ресурси GPU які створюються для вирішення певних проблем
3) ресурси які використовуються для створення зображення (текстури, моделі, шейдери)

Перші 2 рівні обробляються у класі OPTada_Render а також частково у OPTadaC_ResourceManager, який у нього інтегровано.

Саме 3 рівень обробляється у OPTadaC_ResourceManager. Даний клас – менеджер ресурсів, має у собі менеджер пам’яті для більш швидкої ті стабільної роботи з оперативною пам’яттю комп’ютера, а також системи контроля пам’яті GPU.

Для створення різних ресурсів на GPU треба використовувати різні алгоритми. Усі ці проблеми автоматично обробляє менеджер ресурсів, надаючи простий інтерфейс, а також повний доступ до усіх ресурсів у разі необхідності.

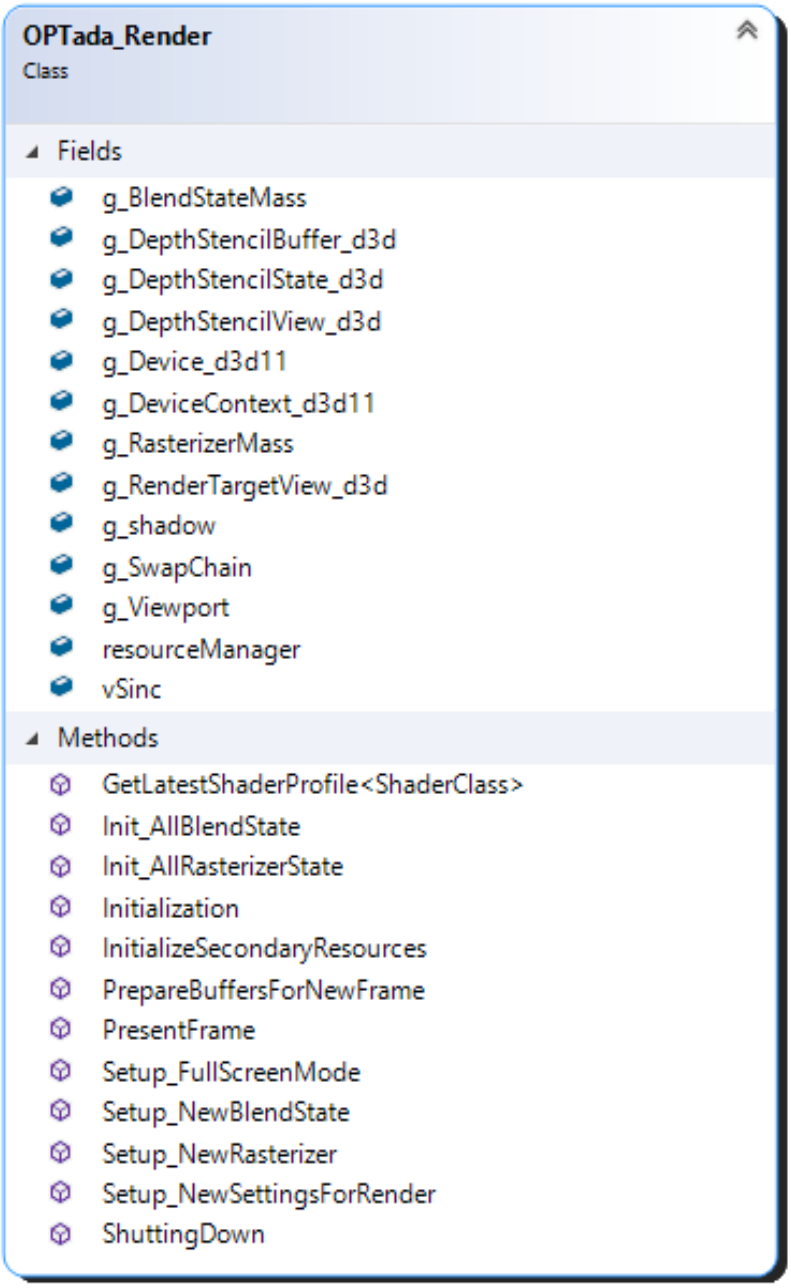

Рисунок 3.7 - Сигнатура класу для рендеру зображення

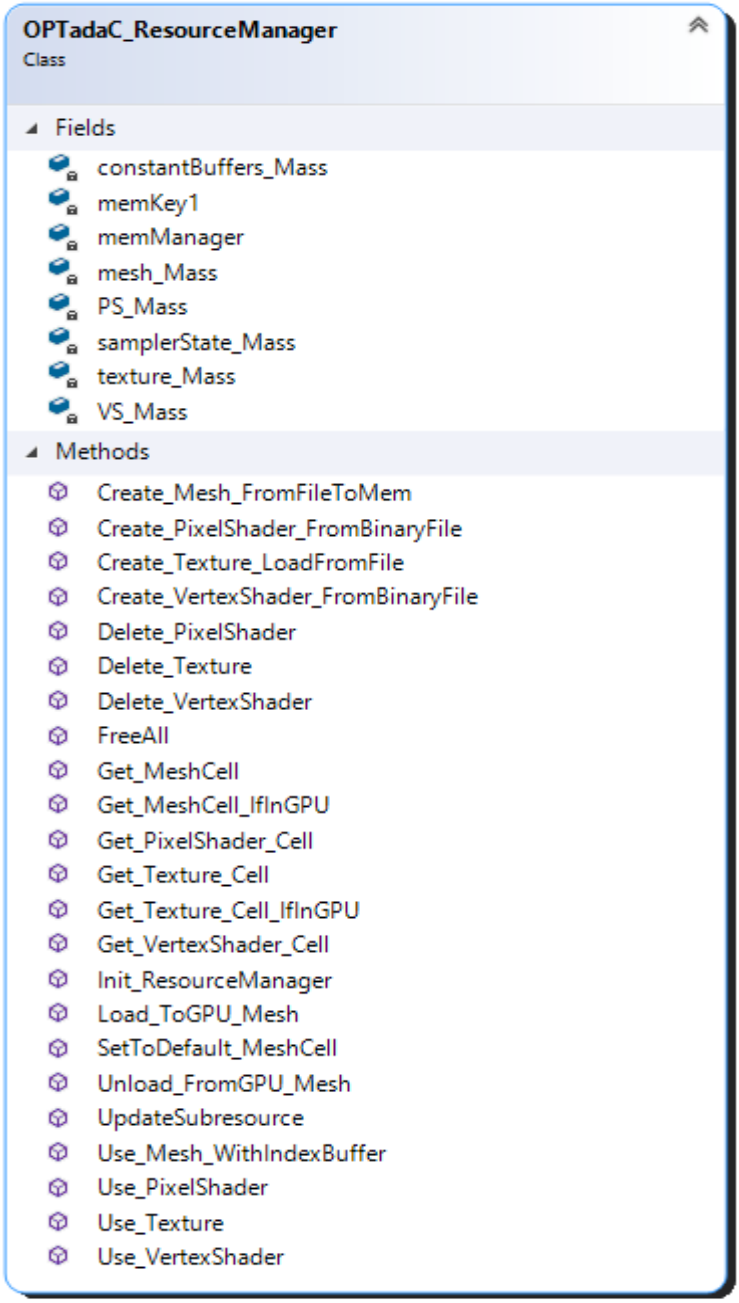

Рисунок 3.8 - Сигнатура класу менеджеру ресурсів

Також модуль для рендеру включає у собі реалізацію деяких допоміжних класів, які необідні або полегшують керування процесом створення зображення, такі як (рисунки 3.9 – 3.11):

- структура для навігації у 3D просторі OPTadaS_WorldNavigationData
- клас камери OPTadaC_Obj_Camera
- клас об’єкта який потрібно відобразити OPTadaC_Obj_Draw
- клас об’єкта джерела світла OPTadaC_Obj_Light
- клас для обрахунку колізій OPTadaC_Collision
- клас об’єкта для перевірки колізії OPTadaC_Obj_Collision

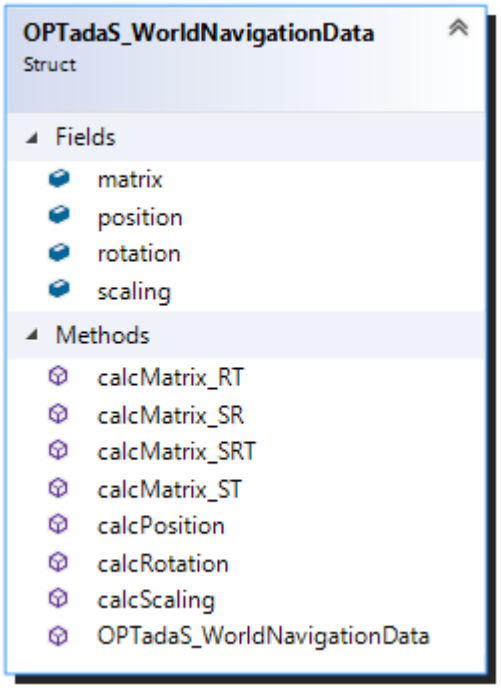

Рисунок 3.9 - Сигнатура структури навігації у 3D просторі

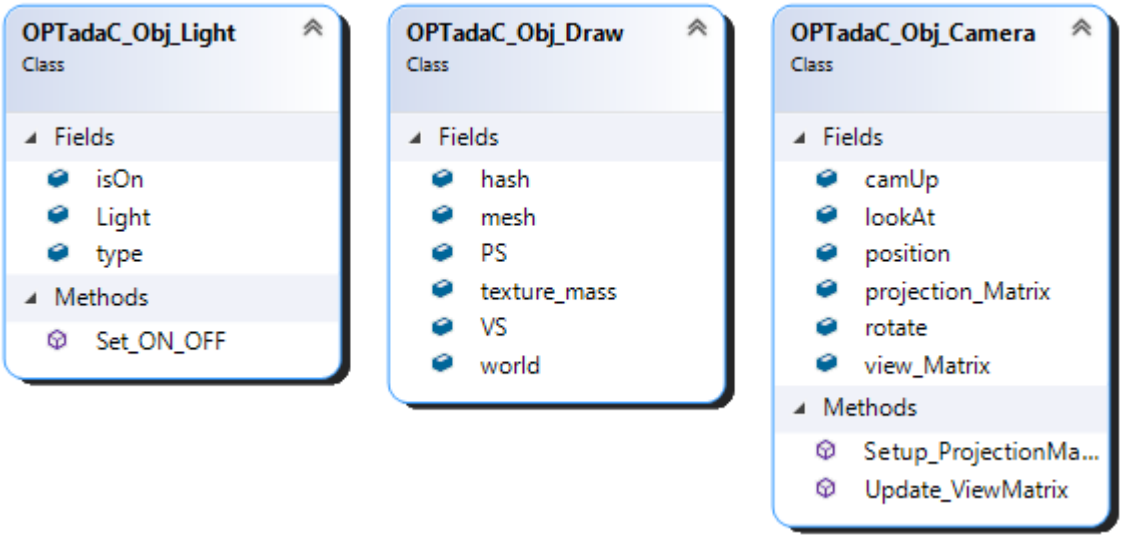

Рисунок 3.10 - Сигнатура класів джерела світла, об’єкта який потрібно відобразити та камери

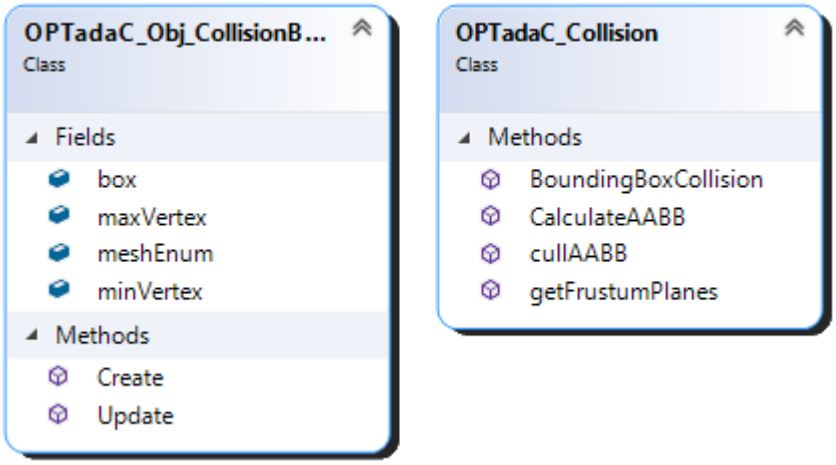

Рисунок 3.11 - Сигнатура класу об’єкта для перевірки колізії та класу для обрахунку колізій

3.2 Збір та аналіз даних

За допомогою розробленого ПЗ Lora було виконано наступне:

– зібрано аналітичні дані що до швидкості виконування відсіву невідображаємих трикутників («clip» або «кліпинг») на стороні GPU та CPU;
– зібрано аналітичні дані що до необхідності зменшення кількості команд налаштування GPU для рендеру об’єкта;
– зібрано аналітичні дані впливу оптимізаційних методів на стабільність роботи програми.

Збір даних проводився на обладнанні:

– GPU 1050ti, з 4 гігабайтами пам’яті DDR5;
– CPU i3-2100, з тактовою частотою 3.1Ггрц, а також 2-ома фізичними і 2-ома віртуальними ядрами;
– 8 гігабайт оперативної пам’яті DDR3;
– шина для передачі даних на GPU – PCI-express 3.0;
– Windows 10 x64.

3.2.1 Швидкості кліпінгу на GPU та CPU

Через те що GPU – лише механізм для обрахунку математичних задач - він не розуміє які об’єкти та їх частини будуть відображені на екрані, а які – ні. Логічно припустити, що виконання зайвих обрахунків даних, які не будуть використані у майбутньому – лише навантажать систему. Через це потрібно відрізнити та відсіяти зайві обрахунки використавши мінімум часу та ресурсів

системи. Це відсіювання можливо зробити за допомогою обрахунків на CPU або користуючись механізмами GPU.

Збір даних для аналізу швидкості кліпінгу на GPU та CPU, проводився для різної кількості моделей, з різною кількістю трикутників, а також використовуючи спрощений шейдер для вершин, для зменшення пливу поточних розрахунків на результат.

Код спрощеного шейдеру виглядає так:

```
VertexShaderOutput VS_Material_Default(AppData IN)
{
   return mul(WVP, float4(IN.position, 1.0f));
}
```

Вхідні дані для кожної з вершин:

- її координати у локальному просторі, представлені у вигляді матриці (x, y, z).
- матриця відображення (4x4)
- вихідні дані для кожної з вершин:
- її обраховані координати у тривимірному простору, представлені у вигляді матриці (x, y, z, 0).

Даний алгоритм – є мінімально необхідним для виконання вершинного шейдеру. Він лише обраховую координати вершини у світовому просторі.

Усі необхідні дані буди завчасно завантажені та встановлені. Піксельний шейдер не виконував обрахунків, а тільки повертав колір у вигляді RGBA масиву.

Кліпінг даних на стороні CPU, виконується відсіюючи зайві моделі, які не потрапляють на екран, у свій час GPU проводить кліпінг автоматично на етапі растерізації, визначаючи які трикутники будуть відображатися, а які – ні.

Ключовим фактом відображення GPU – є потрапляння хоча б однієї вершини трикутника на екран, у свій час CPU використовує поточні данні хітбоксу моделей (паралелепіпеду який оточує всю модель) для вирішення, чи буде дана модель відображатися, або ні.

Таблиця 3.3 – Дані швидкості кліпінгу GPU та CPU

Номер тесту	Кількість прямих команд рендеру	Кількість трикутників У моделі	Кількість трикутників які ми бачимо	Кількість трикутників які ми не бачимо	Час виконання у секундах для обрахунку на:	
					GPU	CPU
1	1.000	1	1.000	0	0,0001	0,0009
2	1.000	100	100.000	0	0,0004	0,0008
3	1.000	1.000	1.000.000	0	0,0005	0,0006
4	1.000	1	0	1.000	0,0001	0,0007
5	1.000	100	0	100.000	0,0002	0,0005
6	1.000	1.000	0	1.000.000	0,0002	0,0008
7	1.000.000	1	1.000.000	0	0,0011	0,0596
8	1.000.000	100	100.000.000	0	0,0422	0,0612
9	1.000.000	1.000	1.000.000.000	0	0,2178	0,0608
10	1.000.000	1	0	1.000.000	0,0007	0,0630
11	1.000.000	100	0	100.000.000	0,0357	0,0602
12	1.000.000	1.000	0	1.000.000.000	0,2100	0,0607

Зважаючи на отримані результати, можливо зробити наступні висновки:

CPU використовує статичні дані, які заздалегідь обраховуються для моделі.

Ми бачимо що на завантаженість CPU впливає тільки кількість моделей, збільшуючи кінцевий час формування зображення. В першу чергу на це впливає те що результат рішення відсіювання приходить для кожної моделі окремо, використовуючи дані хідбоксів, які формуються завчасно.

Також слід помітити, що для обрахунку хітбоксу використовується дуже важкий алгоритм, який повинен перебрати усі точки моделі для виявлення розмірів та координат його крайніх точок. Даний алгоритм занадто важкий для роботи «на льоту», тому він завжди обраховуються завчасно, та зберігаються для використання займаючи певну частину пам’яті програми.

У свій час є певна кількість трикутників у моделі, після якої дешевше робити обрахунки на стороні CPU (див. таб. 3.3 - номера тесту 9 та 12), але треба враховувати що від кількості команд рендеру також росте навантаження на нього.

GPU може швидше обробляти багато моделей з малою кількістю трикутників.

При збільшенні трикутників ми бачимо тенденцію, коли GPU починає робити занадто багато розрахунків, тим самим збільшуючи час формування

зображення. Це ми можемо побачити з даних – які завжди давали значне збільшення часу формування кадру, при збільшенні кількості трикутників у моделі. Вплив кількості команд рендеру не значний, через те що не відбувається додаткових розрахунків.

Також, ми бачимо незначне збільшення швидкості рендеру кадру (таблиця 3.3) у тестах 4-6 і 10-12 порівняно з 1-3 і 7-9, це пов'язана з впливом кількості викликів піксельного шейдеру, який не викликається при прийнятті рішення – не малювати трикутник.

Зважаючи на все вище сказане, можливо зробити наступний висновок: для ефективного кліпінгу великої кількості низько-полігональних моделей краще використовувати GPU, який може швидко обробляти велику кількість команд рендеру. У свій час є певна кількість трикутників у моделі, після якої дешевше робити обрахунки на стороні CPU. Для знаходження кращого рішення у даній ситуації потрібно проводити тестування для кожного з можливих архітектурних випадків окремо.

3.2.2 Необхідність зменшення кількості команд налаштування GPU для рендеру об'єкта за рахунок додаткових розрахунків CPU

Через те що GPU – окремий прилад, надсилання до нього будь-яких команд через інтерфейс викликає великі затримки. Кожне звернення до GPU – виконується від 5 до 100 тіків процесора. Зазвичай прості команди налаштування коштують дешево, однак для рендеру зображення потрібно передавати і інші дані. Майже усі основні дані які використовуються для обрахунку зображення зберігаються у пам'яті GPU, та все одно є дані які потрібно передавати для кожного об'єкту, наприклад матрицю перетворення, яка дозволяє встановити об'єкт у 3D просторі глобальних координат, або дані про джерела світла. Всі дані до GPU передаються пакетами по 64 байта. Якщо ви передаєте дані більшого розміру – вони будуть передані більшою кількістю пакетів чергою, що значно збільшить затримку до завершення команди.

Для збору даних було проведено декілька тестів з мінімальним алгоритмічним навантаженням у шейдерах. Інформація яка передається до GPU була обрахована завчасно.

Спрощений вершинний шейдер, такий самий як і при тестуванні швидкості кліпінгу.

Вхідні дані для кожної з вершин:

- її координати у локальному просторі, представлені у вигляді матриці (x, y, z).
- матриця відображення (4x4)
- вихідні дані для кожної з вершин:
- її обраховані координати у тривимірному простору, представлені у вигляді матриці (x, y, z, 0).

Усі необхідні дані буди завчасно завантажені та встановлені. Піксельний шейдер не виконував обрахунків, а тільки повертав колір у вигляді RGBA масиву.

Таблиця 3.4 – Дані швидкості рендеру залежно від частоти команд на GPU

Номер тесту	Кількість прямих команд рендеру	Час у секундах для випадку коли:		
		Дані для моделі були встановлені один раз	Дані для моделі встановлювались кожен раз	Дані для моделі оновлювались якщо були встановлені інші (перевірка на стороні CPU)
1	10	0,0001	0,0001	0,0001
2	1.000	0,0001	0,0002	0,0002
3	100.000	0,0057	0,0318	0,0199
4	1.000.000	0,0601	0,3214	0,2010
5	1.000.000.000	7,2091	30,4401	19.3190

Зважаючи на отримані результати, можливо зробити наступні висновки:

В потрібно щоб зайві команди до GPU не відсилались. Один з найкращих варіантів буде – перевіряти чи потрібно оновляти такі самі дані на GPU, чи ні. Це потребує додаткових обрахунків на CPU, а також використання пам’яті на зберігання даних, для порівняння. Однак ми бачимо з таблиці 3.4 – після невеликого плато де ми можемо знехтувати оптимізацією (тести 1-2) – з’являється різниця між постійним оновленням даних, а також перевіркою – чи потрібно їх оновляти на CPU. Ця різниця може складає приблизно (15-30%), та може буди корисна для інших обрахунків. Тож алгоритми перевірки необхідності оновлення даних на GPU можуть вважатися дуже корисними для оптимізації.

3.2.3 Вплив оптимізаційних методів на стабільність роботи програми.

Оптимізаційні не завжди бувають універсальними шаблонними або архітектурними рішеннями. В ідеальній програмі усі процеси повинні йти один за одним, не відхиляючись від задачі на варіативність, але через складність створення, доводиться користуватися гнучкими абстракціями для створення будь-якої логіки. Саме через це і з’являються двоякі ситуації, через які потрібно розробляти безпечні алгоритми вирішення завдання.

Таблиця 3.5 – Дані швидкості рендеру залежно від кількості перевірених даних на стороні CPU

Номер тесту	Кількість прямих команд рендеру	Час у секундах за tick для випадку коли:		
		Перевіряється коректність даних	Дані вважаються коректними завжди (без перевірки)	Дані перевіряються частково (перевірка на nullptr)
1	10	0,0001	0,0001	0,0001
2	1.000	0,0001	0,0001	0,0001
3	100.000	0,0057	0,0041	0,0041
4	1.000.000	0,0729	0,0593	0,0600

Зважаючи на отримані результати, можливо зробити наступні висновки:

Перевірки даних не використовують забагато ресурсів у випадку перевірки ссилок, через їх незначну кількість а також перевірку у 1 операцію – їх можливо вважати незначними (див. таб. 3.5).

Для перевірки коректності усіх даних потрібно на 10% більше обрахунків на тік. У випадку коли перевірка стосується даних від яких залежить робота програми – бажано залишити ці перевірки, але у випадках де не коректні дані можуть лиже надати небажаних результатів які не вплинуть на роботу програми у майбутньому – видалити.

3.3 Оптимізаційний баланс

Використовуючи дані а також висновки з проведених експериментів ми можемо знайти оптимізаційний баланс. Для цього потрібно визначити критерії оптимізації. У даному випадку усі 3 експерименти впливають один на одного, однак не змінюють алгоритм обробки даних. Їх вплив пов’язаний у цілому з аналізом а також сортуванням даних, які будуть використовуватися для створення зображення.

Виходячи з результатів експериментів у цілому, ми можемо зробити наступні висновки, які і будуть результатом пошуку оптимізаційного балансу у даному випадку. Через ситуативність отриманих даних ми можемо робити тільки загальні висновки, які у деяких випадках можуть бути не результативними. Існують випадки для яких ми можемо використати дані результати:

1) за необхідності, опираючись на дані проведених експериментів ми можемо частково перенести обрахункові навантаження з GPU на CPU і покращити оптимізацію;

2) враховуючи що кількість команд для досягнення кращого фінального зображення зазвичай максимально велика, а також використовується максимальна кількість трикутників, кращім оптимізаційним балансом у даному випадку буде - перенесення обрахунків кліпінгу на CPU у багатьох випадках надає дуже велику оптимізацію, та оправдано.
3) за умови що дані які використовуються для моделей дублюються достатньо часто (моделі використовують спільні дані) – перенесення перевірки встановлених даних на сторону CPU надає невелику оптимізацію за рахунок зменшення команд до GPU. Виходячи з економії при розробці ПЗ, а також економії пам’яті ПЗ для зберігання даних, перенесення навантаження на CPU буде кращим рішенням оптимізаційного балансу.
4) оптимізацію за рахунок скорочення перевірок даних у системах рендеру зображення можливо тільки у випадках коли ми гарантуємо коректність даних. Оптимізаційний баланс у даному випадку буде найкращим при гарантії коректності даних.

Зважаючи на висновки найкращий оптимізаційний баланс для більшості випадків буде – перенесення перевірок кліпінгу на CPU, виконання оптимізації за рахунок сортування ресурсів і зменшення команд до GPU, використовуючи потужності CPU, а також прибирання додаткових перевірок даних, що надає невелику оптимізацію. Цей висновок буде надавати максимальний оптимізаційний баланс для більшості ситуацій.

3.4 Висновки з розділу 3

У розділі 3, даної роботи, були винайдені найбільш вдалі рішення для використання деяких оптимізаційних рішень у конвеєрі рендеру. Усі висновки підтверджуються даними, які були зібрані за допомогою додатково розробленого ПЗ Lora.

Враховуючи дані які було проаналізовано ми можемо зробити наступні висновки:

- для кліпінгу зайвих моделей з малою кількістю полігонів бажано використовувати ресурси GPU, і також навпаки – коли полігонів багато використовувати кліпінг на стороні CPU;
- потреба у додаткових перевірках для зменшення команд на GPU цілком оправдовує себе за рахунок на 30% меншого часу тіку.
- перевірка усіх даних які використовуються у конвеєрі рендеру – є зайвою, для стабільності програми достатньо перевіряти тільки дані які можуть впливати на наступні тіки та ссилки на nullptr посилання.

Було визначено можливі випадки використання оптимізаційного балансу для проведених експериментів, а також визначено найкраще рішення оптимізаційного балансу для них.

ВИСНОВКИ

В результаті написання дипломного проекту магістра на тему: «Аналіз оптимізаційних рішень для конвеєру рендеру зображення, а також пошук найкращих методів їх використання для досягнення оптимізаційного балансу» було виконано аналіз роботи конвеєру рендеру растрової графіки, також були проаналізовані вже існуючі методи його оптимізації. Були поставлені задачі для даної роботи, а саме:

- проаналізувати існуючі принципи оптимізації рендеру;
- виявити слабкі місця в алгоритмі формування зображення, а також знайти можливі рішення для покращення їх роботи;
- проаналізувати можливі шляхи знаходження оптимізаційного балансу, а також ситуації у яких він може використовуватись.

Далі було проведено аналіз можливих рішень оптимізації у сучасних системах. Виявлено слабкі місця конвеєру рендеру, а також розроблені можливі рішення покращення їх оптимізації. Було проаналізовано методи знаходження оптимізаційного балансу для ПЗ.

Для аналізу оптимізаційних рішень було проведено повний цикл розробки ПЗ – результатом якого став проект Lora, яке забезпечило гнучкий та точний збір даних. Дане ПЗ – використовує концепцію рендеру растрової графіки у реальному часі, що дозволило нам гнучко налаштовувати графічний конвеєр для збору даних.

Використовуючи отримані дані - проведено аналіз оптимізаційних рішень, а також на основі даних аналізу було знайдено можливі варіанти оптимізаційного балансу.

Враховуючи проаналізовані дані було зроблено наступні висновки:

- для кліпінгу зайвих моделей з малою кількістю полігонів бажано використовувати ресурси GPU, і також навпаки – коли полігонів багато використовувати кліпінг на стороні CPU;
- потреба у додаткових перевірках для зменшення команд на GPU цілком оправдовує себе за рахунок на 30% меншого часу тіку.
- перевірка усіх даних які використовуються у конвеєрі рендеру – є зайвою, для стабільності програми достатньо перевіряти тільки дані які можуть впливати на наступні тіки та ссилки на nullptr посилання.

ПЕРЕЛІК ПОСИЛАНЬ

1. Рендер графіки [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%A0%D0%B5%D0%BD%D0%B4%D0%B5%D1%80%D0%B8%D0%BD%D0%B3
2. Абстракція [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%86%D1%96%D1%8F.
3. Оптимізаційний баланс [Електронний ресурс] – Режим доступу до ресурсу: http://nbuv.gov.ua/j-pdf/ape_2012_5_14.pdf.
4. Архітектор програмного забезпечення [Електронний ресурс] – Режим доступу до ресурсу: https://poprofessii.in.ua/uk/software_architect_arhitektor_programnogo_zabezpechennja.
5. Людський фактор [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%9B%D1%8E%D0%B4%D1%81%D1%8C%D0%BA%D0%B8%D0%B9_%D1%84%D0%B0%D0%BA%D1%82%D0%BE%D1%80.
6. Аналогія [Електронний ресурс] – Режим доступу до ресурсу: https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B0%D0%BB%D0%BE%D0%B3%D0%B8%D1%8F.
7. Створення растрового зображення у реальному часі [Електронний ресурс] – Режим доступу до ресурсу: http://www.sccg.sk/~cervenansky/rtr/pr2_optimization.pdf.
8. Графічні примітиви [Електронний ресурс] – Режим доступу до ресурсу: https://sites.google.com/site/ngginform/8-klas/vidobrazenna-bazovih-graficnih-primitiviv.
9. Mesh [Електронний ресурс] – Режим доступу до ресурсу: https://whatis.techtarget.com/definition/3D-mesh#:~:text=A%203D%20mesh%20is%20the,with%20height%2C%20width%20and%20depth.
10. Декомпозиція [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%94%D0%B5%D0%BA%D0%BE%D0%BC%D0%BF%D0%BE%D0%B7%D0%B8%D1%86%D1%96%D1%8F.
11. Тріангуляція [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%A2%D1%80%D1%96%D0%B0%D0%BD%D0%B3%D1%83%D0%BB%D1%8F%D1%86%D1%96%D1%8F_(%D0%B3%D0%B5%D0%BE%D0%BC%D0%B5%D1%82%D1%80%D1%96%D1%8F).
12. Хорда [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%A5%D0%BE%D1%80%D0%B4%D0%B0_(%D0%B3%D0%B5%D0%BE%D0%BC%D0%B5%D1%82%D1%80%D1%96%D1%8F).

13. Програми для 3Д малювання [Електронний ресурс] – Режим доступу до ресурсу: https://sites.google.com/site/3dmodeluvana/20-bezkostovnih-program-dla-3d-modeluvanna.
14. Система координат [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D0%BA%D0%BE%D0%BE%D1%80%D0%B4%D0%B8%D0%BD%D0%B0%D1%82.
15. Матриці перетворення [Електронний ресурс] – Режим доступу до ресурсу: https://code-industry.ru/masterpdfeditor-help/transformation-matrix.
16. Освітлення у комп'ютерній графіці [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D0%BA%D0%BE%D0%BE%D1%80%D0%B4%D0%B8%D0%BD%D0%B0%30.
17. Матеріали [Електронний ресурс] – Режим доступу до ресурсу: https://docs.blender.org/manual/ru/2.79/render/cycles/materials/index.html.
18. Normal maping [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%A0%D0%B5%D0%BB%D1%8C%D1%94%D1%84%D0%BD%D0%B5_%D1%82%D0%B5%D0%BA%D1%81%D1%82%D1%83%D1%80%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F.
19. Z-buffer [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Z-%D0%B1%D1%83%D1%84%D0%B5%D1%80%D0%B8%D0%B7%D0%B0%D1%86%D1%96%D1%8F.
20. Піксельний шейдер [Електронний ресурс] – Режим доступу до ресурсу: https://www.computerhope.com/jargon/p/pixel-shader.htm#:~:text=In%20computer%20graphics%2C%20a%20pixel,known%20as%20a%20shading%20artist.
21. Процес розробки програмного забезпечення [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81_%D1%80%D0%BE%D0%B7%D1%80%D0%BE%D0%B1%D0%BA%D0%B8_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%B7%D0%B0%D0%B1%D0%B5%D0%B7%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%BD%D1%8F.
22. Render pipline optimization [Електронний ресурс] – Режим доступу до ресурсу: https://www.gamasutra.com/view/feature/1879/the_top_10_myths_of_video_game_.php?print=1.
23. Читота коду [Електронний ресурс] – Режим доступу до ресурсу: https://ilyabirman.ru/meanwhile/all/clean-code/.
24. Робота процесору [Електронний ресурс] – Режим доступу до ресурсу:

https://uk.soringpcrepair.com/how-the-processor-works-and-what-it-does/#i-2.
25. CPU Cache [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/CPU_cache.
26. Graphic engine optimization [Електронний ресурс] – Режим доступу до ресурсу: https://software.intel.com/content/www/ru/ru/develop/articles/achieving-performance-an-approach-to-optimizing-a-game-engine.html.
27. Багатопоточність [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Multithreading_(computer_architecture)#:~:text=In%20computer%20architecture%2C%20multithreading%20is,supported%20by%20the%20operating%20system..
28. Патерни проектування ПЗ [Електронний ресурс] – Режим доступу до ресурсу: https://refactoring.guru/uk/design-patterns.
29. Windows 10 [Електронний ресурс] – Режим доступу до ресурсу: https://www.microsoft.com/uk-ua/software-download/windows10.
30. Clipping [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Clipping_(computer_graphics).
31. DirectX11 [Електронний ресурс] – Режим доступу до ресурсу: https://ru.wikipedia.org/wiki/Direct3D_11.
32. Вершинний шейдер [Електронний ресурс] – Режим доступу до ресурсу: http://www.malbred.com/3d-grafika-3d-redaktory/sovremennaya-terminologiya-3d-grafiki/vertex-shader-vershinnyy-sheyder.html.
33. SDK VisualStudio2019 [Електронний ресурс] – Режим доступу до ресурсу: https://visualstudio.microsoft.com/ru/vs/.
34. HLSL [Електронний ресурс] – Режим доступу до ресурсу: https://ru.wikipedia.org/wiki/HLSL.
35. Тік у ПЗ замкнутого циклу виконання [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Tick_(software).
36. Компонент [Електронний ресурс] – Режим доступу до ресурсу: https://whatis.techtarget.com/definition/component#:~:text=In%20programming%20and%20engineering%20disciplines,are%20made%20up%20of%20modules.
37. ООП [Електронний ресурс] – Режим доступу до ресурсу: https://www.cs.utah.edu/~germain/PPS/Topics/interfaces.html#:~:text=Interfaces%20in%20Object%20Oriented%20Programming,have%20a%20start_engine()%20action.
38. Романчук О. Н. ОСОБЛИВОСТІ ФОРМУВАННЯ ТРИВИМІРНИХ ГРАФІЧНИХ ЗОБРАЖЕНЬ [Електронний ресурс] / О. Н. Романчук, Т. М. Павлик, І. Г. Бабій – Режим доступу до ресурсу: https://sworld.com.ua/konfer22/834.htm.

ДОДАТОК А

СЛАЙДИ ПРЕЗЕНТАЦІЇ

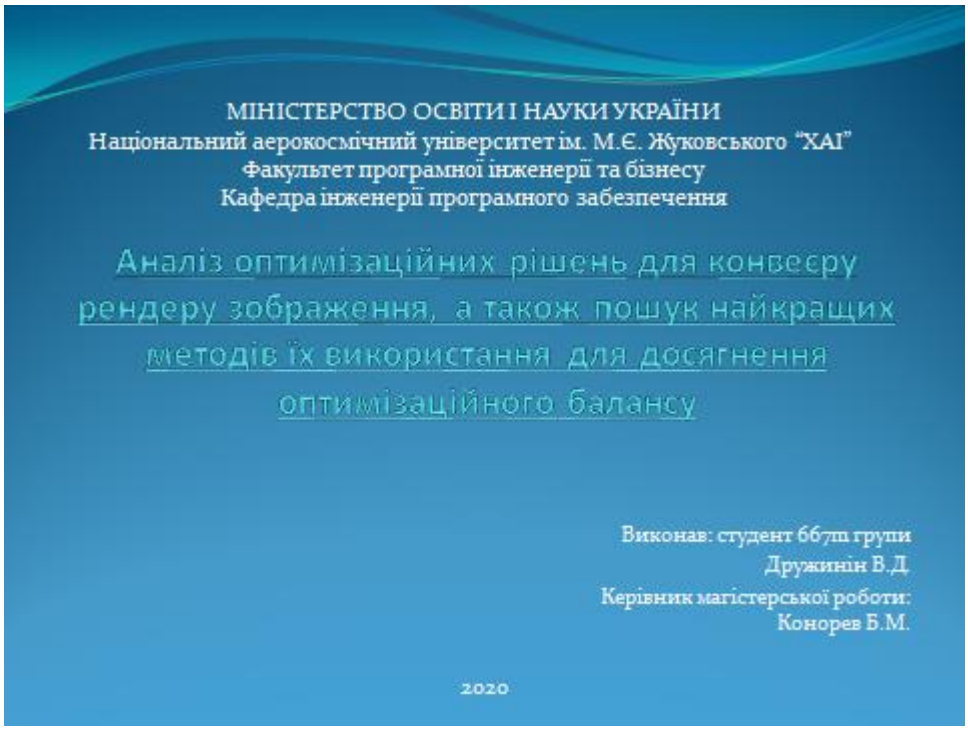

Рисунок А.1 - слайд презентації №1

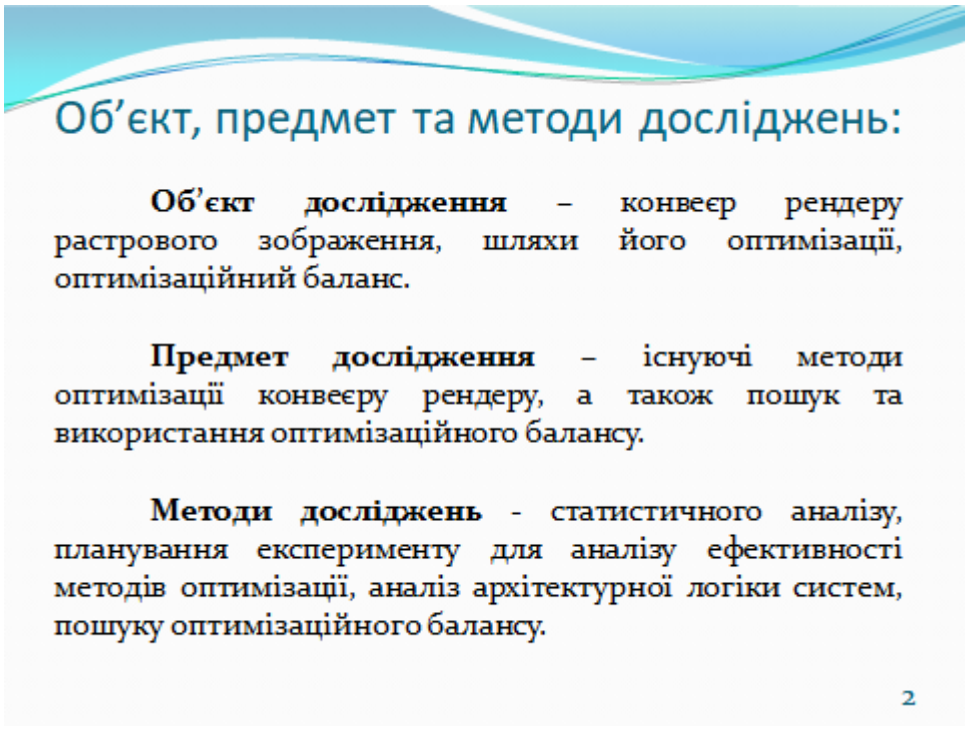

Рисунок А.2 – слайд презентації №2

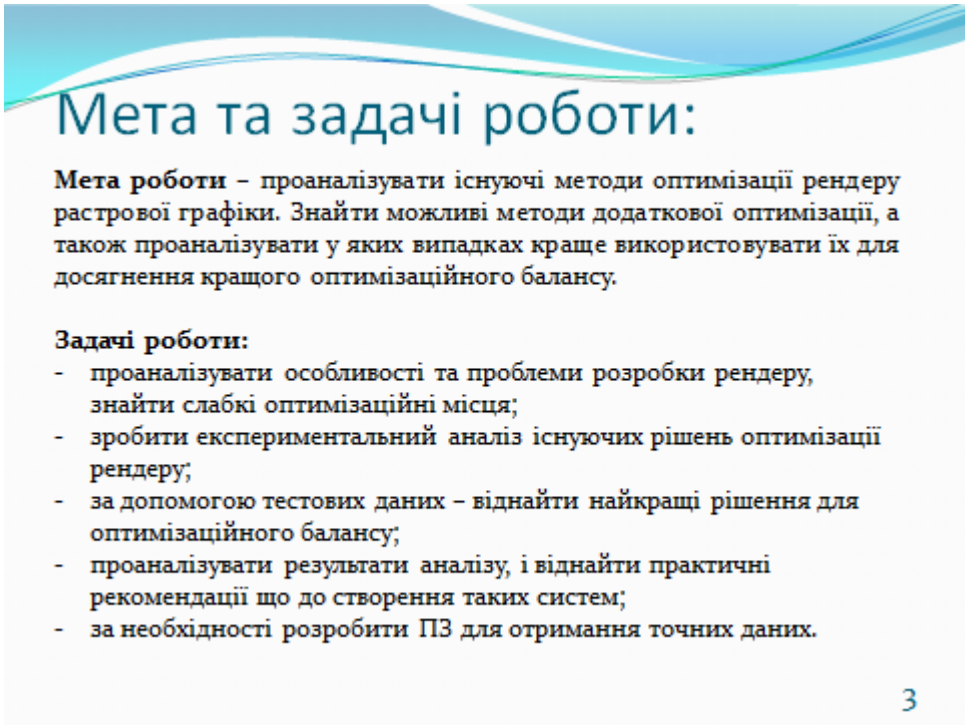

Рисунок А.3 – слайд презентації №3

Рисунок А.4 – слайд презентації №4

Рисунок А.5 – слайд презентації №5

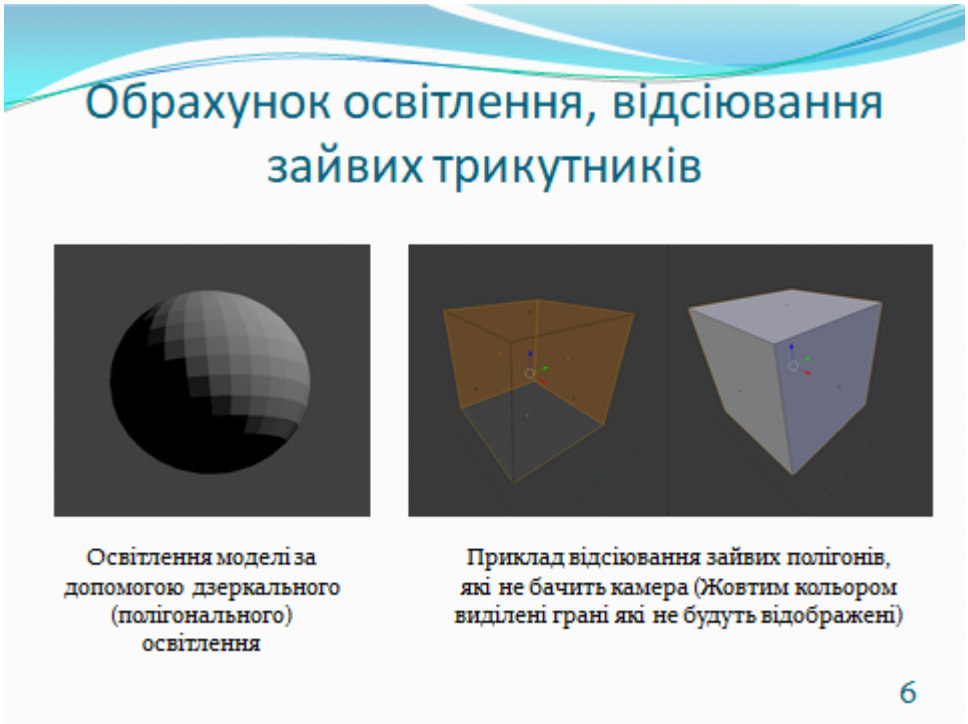

Рисунок А.6 – слайд презентації №6

Рисунок А.7 – слайд презентації №7

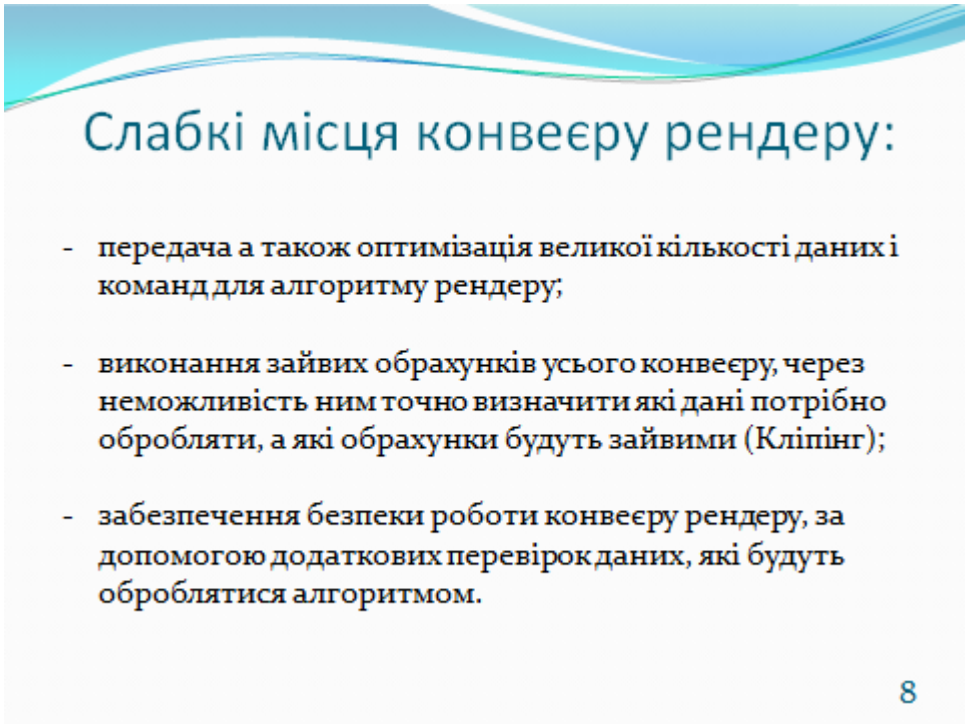

Рисунок А.8 – слайд презентації №8

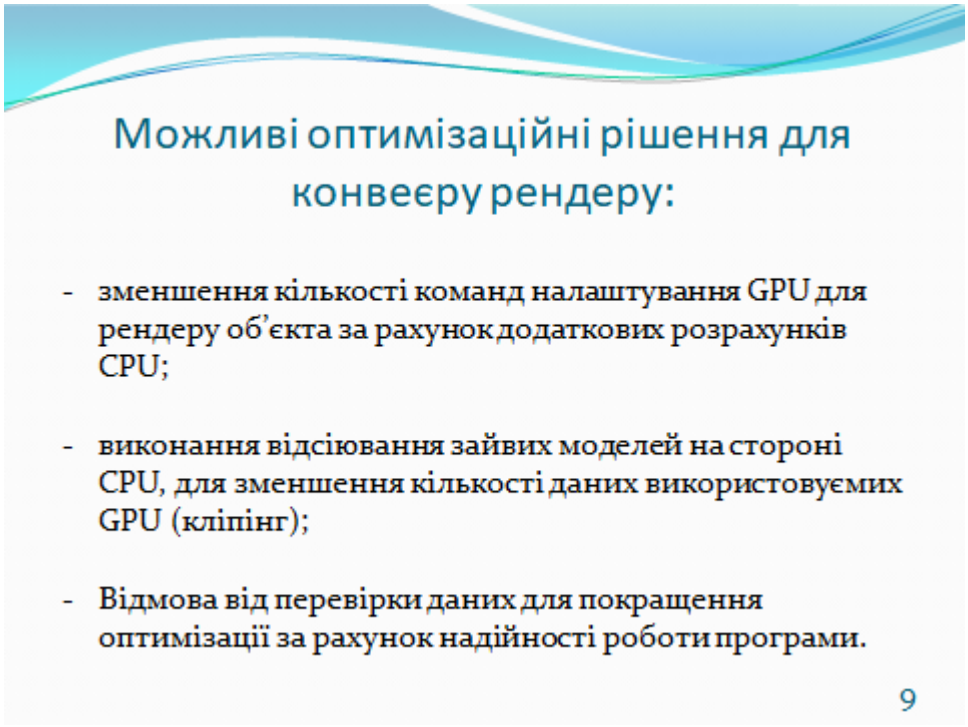

Рисунок А.9 - слайд 9

Рисунок А.10 - слайд презентації №10

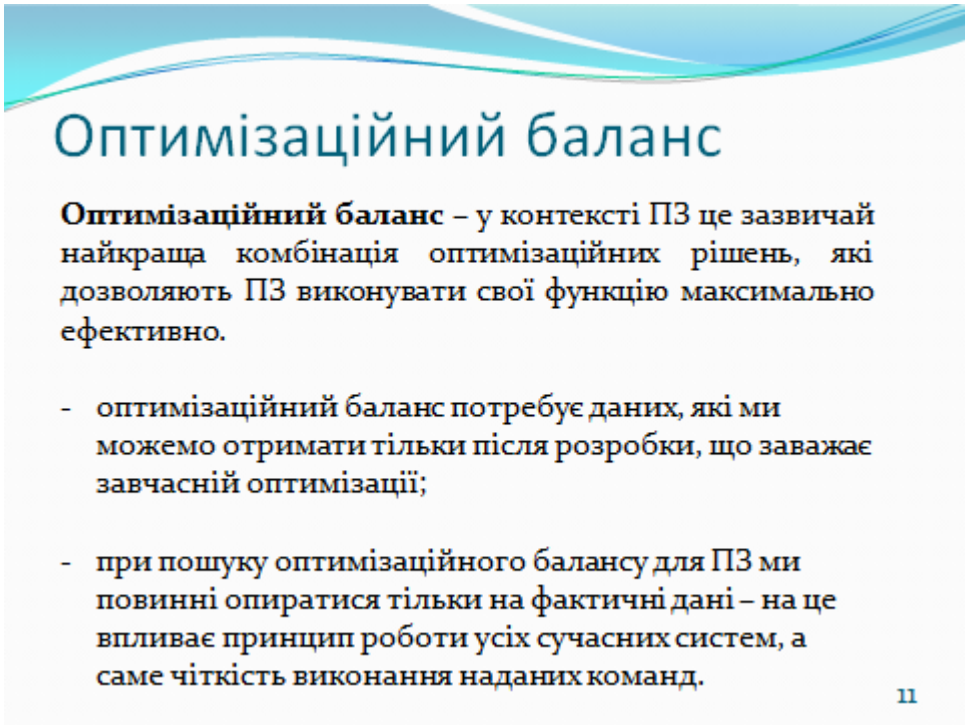

Рисунок А.11 - слайд презентації №11

Рисунок А.12 - слайд презентації №12

Дані швидкості кліпінгу на GPU та CPU

Номер тесту	Кількість прямих команд рендеру	Кількість трикутників у моделі	Кількість трикутників що не бачить камера	Кількість трикутників що можуть бачити камера	Час виконання у секундах для обрахунку на:	
					GPU	CPU
1	1.000	1	1.000	0	0,0001	0,0009
2	1.000	100	100.000	0	0,0004	0,0008
3	1.000	1.000	1.000.000	0	0,0005	0,0006
4	1.000	1	0	1.000	0,0001	0,0007
5	1.000	100	0	100.000	0,0002	0,0005
6	1.000	1.000	0	1.000.000	0,0002	0,0008
7	1.000.000	1	1.000.000	0	0,0011	0,0596
8	1.000.000	100	100.000.000	0	0,0422	0,0612
9	1.000.000	1.000	1.000.000.000	0	0,2178	0,0608
10	1.000.000	1	0	1.000.000	0,0007	0,0630
11	1.000.000	100	0	100.000.000	0,0357	0,0602
12	1.000.000	1.000	0	1.000.000.000	0,2100	0,0607

13

Рисунок А.13 - слайд презентації №13

Дані швидкості рендеру залежно від частоти команд на GPU

Номер тесту	Кількість прямих команд рендеру	Час у секундах для випадку коли:		
		Дані для моделі були встановлені один раз	Дані для моделі встановлювалась кожен раз	Дані для моделі встановлювалась якщо були встановлені інші (перевірка на стороні CPU)
1	10	0,0001	0,0001	0,0001
2	1.000	0,0001	0,0002	0,0002
3	100.000	0,0057	0,0318	0,0199
4	1.000.000	0,0601	0,3214	0,2010
5	1.000.000.000	7,2091	30,4401	19.3190

14

Рисунок А.14 - слайд презентації №14

Рисунок А.15 - слайд презентації №15

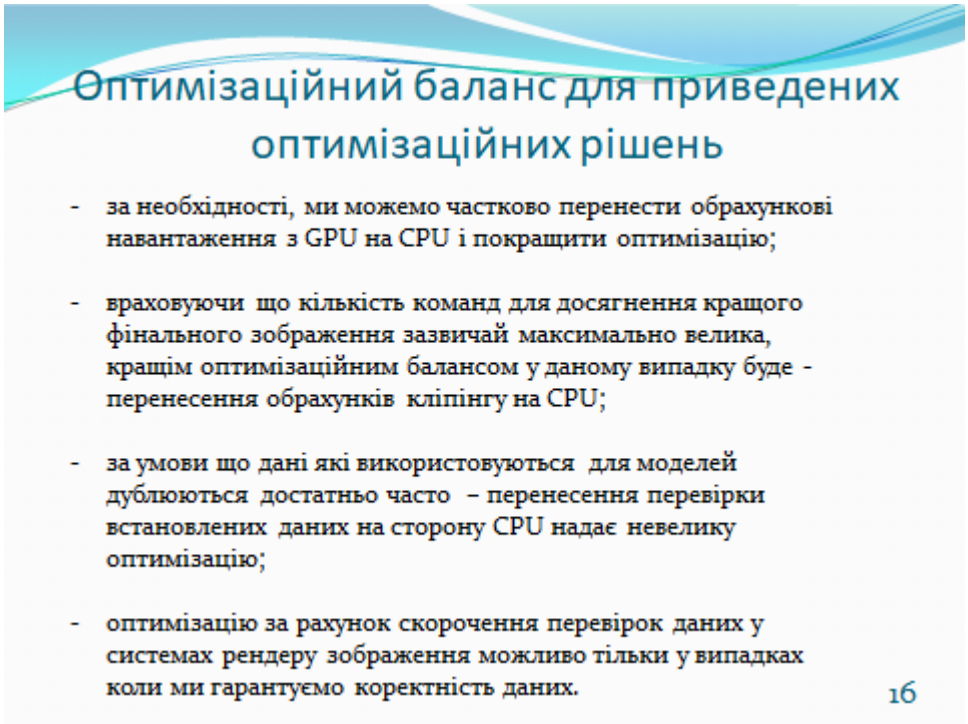

Рисунок А.16 - слайд презентації №16

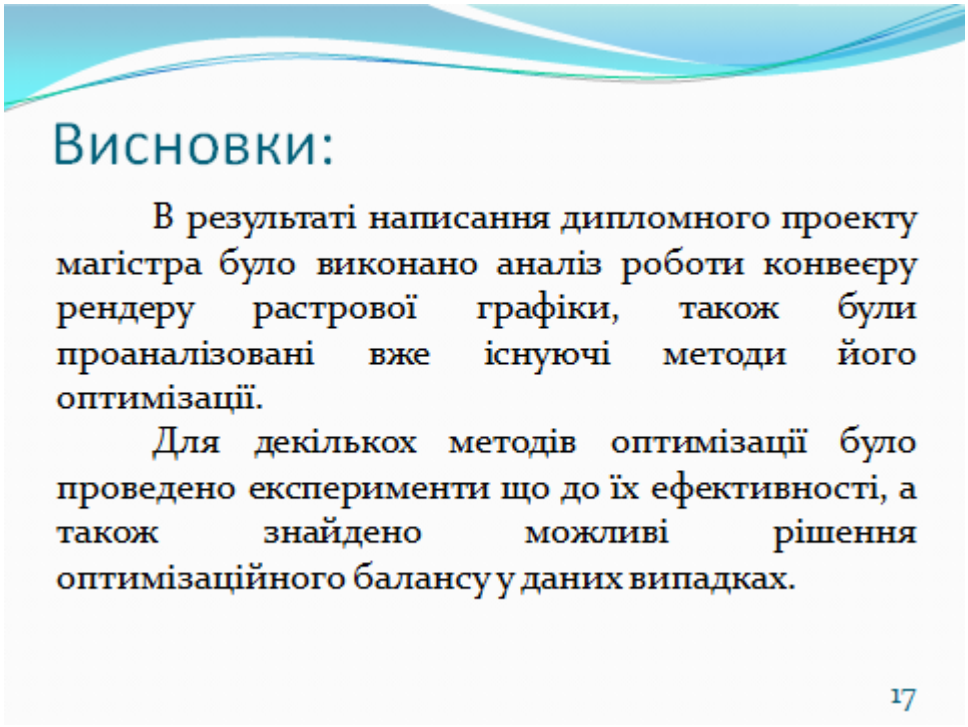

Рисунок А.17 - слайд презентації №17

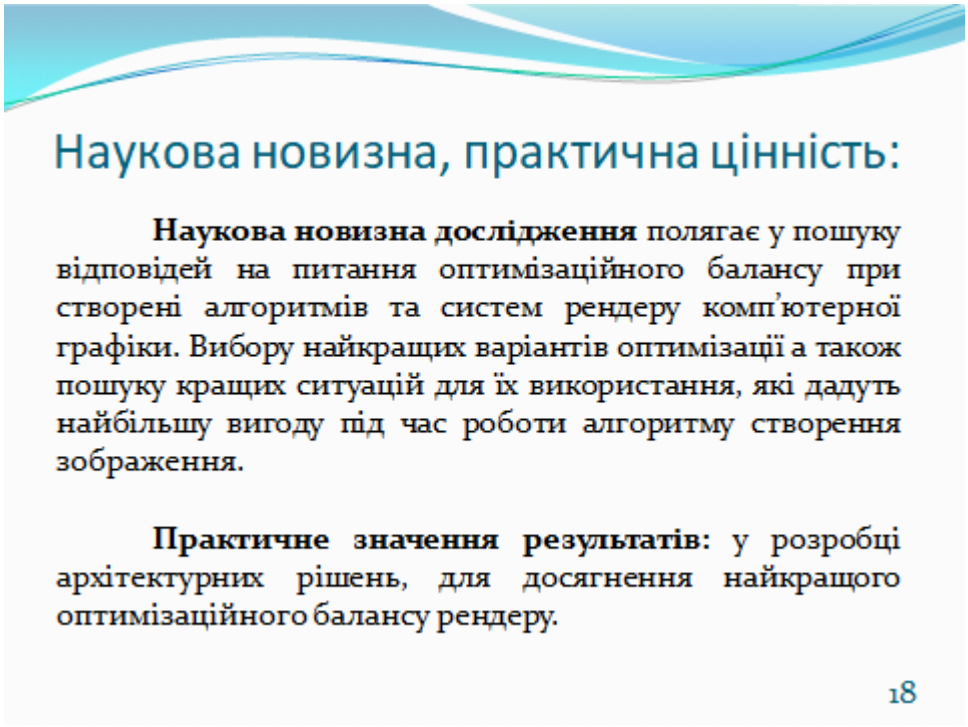

Рисунок А.18 - слайд презентації №18