Міністерство освіти і науки України

Український державний університет науки і технологій

Факультет Комп'ютерні технології та системи
Кафедра Комп'ютерні інформаційні технології

Пояснювальна записка

до кваліфікаційної роботи
магістра

на тему: «Дослідження методів оптимізації рендерінгу великої кількості об'єктів в Unity»

за освітньою програмою **«12 Інженерія програмного забезпечення»**
зі спеціальності: **«121 Інженерія програмного забезпечення»**

Виконав: студент групи «ПЗ2421»
_____ /Олег СИДОРОВ/

Керівник: _____ /доц. Олександр ІВАНОВ/

Нормоконтролер: _____ /Світлана ВОЛКОВА/

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

Дніпро – 2026 рік

Ministry of Education and Science of Ukraine

Ukrainian State University of Science and Technologies

Faculty Computer technologies and systems
Department Computer information technology

Explanatory Note

to Master's Thesis

on the topic: «Research on methods for optimizing the rendering of large numbers of objects in Unity»

according to educational curriculum **«12 Software engineering»**
in the Speciality: **«121 Software engineering»**

Done by the student of the group PZ2421: / Oleh SYDOROV/

Scientific Supervisor: /Oleksandr IVANOV/

Normative controller: /Svitlana VOLKOVA/

Dnipro – 2026

Міністерство освіти і науки України
Український державний університет науки і технологій

Факультет: Комп'ютерних технологій і систем
Кафедра: Комп’ютерні інформаційні технології
Рівень вищої освіти: магістр
Освітня програма: Інженерія програмного забезпечення
Спеціальність: Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри ____КІТ
_____ Вадим ГОРЯЧКІН
_____ грудня 2024 р.

ЗАВДАННЯ

На кваліфікаційну роботу ____Магістр____
студенту Сидорову Олегу Володимировичу____.
1. Тема дипломної роботи: «Дослідження методів оптимізації рендерінгу великої кількості об'єктів в Unity».
Керівник роботи: Іванов Олександр Петрович
затверджені наказом 1401 ст від 02.10.2025 року
2. Строк подання студентом роботи ___.01.2026 року
3. Вихідні дані до дипломної роботи:
_____.
4. Зміст пояснювальної записки (перелік питань до розробки):
4.1. Аналітична частина: Огляд технологій рендерінгу в ігровому рушії Unity;
4.2. Основна частина: Дослідження ефективності методів оптимізації в ігровому рушії Unity;
5. Перелік демонстраційного матеріалу:
5.1. презентація;
5.2. демонстраційне відео.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз сучасного стану дослідження проблеми за науковими літературними джерелами	01.09.25 – 01.10.25	10%
2	Аналіз сучасного стану програмно-апаратного забезпечення, яке потребує вдосконалення для вирішення проблем дослідження	01.10.25 – 10.10.25	
3	Постановка задачі, технічне завдання	11.10.25 – 13.11.25	30%
4	Розробка інструментальних засобів дослідження	14.11.25 – 25.11.25	
5	Виконання досліджень	26.11.25 – 05.12.25	60%
6	Оформлення пояснювальної записки	06.12.25 – 12.01.26	
7	Розробка демонстраційних матеріалів	13.01.26 – 16.01.26	100%
8	Подання кваліфікаційної роботи до кафедри	17.01.26	
9	Захист кваліфікаційної роботи на засіданні Екзаменаційної комісії	22.01.26	

Студент: _____ Олег СИДОРОВ

Керівник роботи: _____ доц. Олександр ІВАНОВ

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи магістра складається з 63 стр., 33 рис., 7 табл., 3 додатків, 16 джерел.

Об'єктом дослідження є процес рендерінгу великої кількості об'єктів в Unity з використанням Universal Render Pipeline.

Мета роботи: дослідити методи оптимізації рендерінгу великої кількості об'єктів в Unity. Провести порівняльні заміри ключових метрик продуктивності для різних методів оптимізації.

Методика дослідження: теоретичний аналіз наукових і літературних джерел, проведення серії експериментів, замір ключових метрик продуктивності, аналіз отриманих даних.

Перелік ключових слів: оптимізація рендерінгу, методи оптимізації, Unity, Universal Render Pipeline (URP), GPU Instancing, Batching, Level of Detail (LOD), Occlusion Culling, продуктивність, C#.

ЗМІСТ

Вступ 7
Розділ 1 Огляд технологій рендерінгу в ігровому рушії Unity 8
1.1 Основні поняття та принципи рендерінгу в реальному часі 8
1.2 Огляд ігрового рушія Unity та особливостей URP 9
1.3 Основні принципи графічного конвеєра 10
1.4 Порівняльна характеристика URP та HDRP 14
1.5 Поняття оптимізації в розробці ігор 15
1.6 Огляд методів оптимізації 18
Висновки до розділу 1 21
Розділ 2 вимоги до розробки 22
2.1 Основні функціональні вимоги 22
2.2 Вхідні дані 23
2.3 Вихідні дані 24
Висновки до розділу 2 24
Розділ 3 Розробка інструментальних засобів для дослідження МЕТОДІВ ОПТИМІЗАЦІЇ РЕНДЕРІНГУ 25
3.1 Мова програмування та середовище розробки 25
3.2 Особливості мови C# для використання у розробці 26
3.3 Особливості рушія Unity для використання в розробці 27
3.4 Реалізація проекту в Unity 30
3.5 Формалізація задачі 32
3.6 Проектування інтерфейсу користувача 34
Висновки до розділу 3 37
Розділ 4 Аналіз ефективності МЕТОДІВ ОПТИМІЗАЦІЇ 38
4.1 Аналіз результатів дослідження 40
Висновки за розділом 4 58
Загальний висновок 60
Бібліографічний список 62
ДОДАТОК А 65
ДОДАТОК Б 81
ДОДАТОК В 99
ДОДАТОК Д 109

ВСТУП

Сучасні відеоігри демонструють сталу тенденцію до зростання складності сцен, кількості об’єктів і вимог до якості графіки. В таких умовах питання оптимізації рендерінгу набуває особливої актуальності, оскільки саме відтворення графіки часто є вузьким місцем продуктивності.

Однією з найскладніших ситуацій для рушія є відображення великої кількості однотипних об’єктів, що одночасно присутні в сцені. Це створює навантаження як на процесор, який витрачає значні ресурси на підготовку команд рендерінгу, так і на графічний процесор, який змушений обробляти великий обсяг геометрії та пікселів.

Ігровий рушій Unity, а особливо його Universal Render Pipeline, надає розробникам широкий набір засобів для оптимізації рендерінгу. Однак на практиці постає питання не лише в наявності цих інструментів, а й в розумінні їх реального впливу на продуктивність в конкретних умовах.

Метою цієї магістерської роботи є експериментальне дослідження та порівняльний аналіз методів оптимізації рендерінгу великої кількості однотипних об’єктів в рушії Unity. В рамках роботи передбачається провести серію експериментів для різних методів оптимізації. Оцінювання ефективності методів здійснюється за допомогою таких показників, як середня частота кадрів, середній час кадру на центральному та графічному процесорах, кількість батчів та кількість трикутників.

Отримані результати можуть бути використані розробниками ігор під час розробки та проектування гри, планування цільових платформ та вимог до апаратного забезпечення.

РОЗДІЛ 1 ОГЛЯД ТЕХНОЛОГІЙ РЕНДЕРІНГУ В ІГРОВОМУ РУШІЇ UNITY

1.1 Основні поняття та принципи рендерінгу в реальному часі

Рендерінг у реальному часі — це процес формування зображення сцени з такою швидкістю, яка забезпечує інтерактивність для користувача. На відміну від офлайнового рендерінгу, де кадр може обчислюватися хвилини або години, у реальному часі кожен кадр має бути сформований за частки секунди. Типовою ціллю є частота 60 кадрів на секунду, що відповідає приблизно 16,7 мілісекунди на повний цикл оновлення і відтворення кадру. У багатьох проектах допускаються також 30 кадрів на секунду, що дає близько 33,3 мілісекунди. Усі обчислення логіки, фізики, анімацій та графіки мають вміститися в цей часовий проміжок, тому питання розподілу навантаження між центральним та графічним процесорами є ключовим.

Традиційно графічний конвеєр рендерінгу поділяють на кілька основних етапів. На стороні CPU виконується підготовка сцени: оновлення стану об’єктів, розрахунок матриць перетворень, формування списку видимих об’єктів, налаштування матеріалів та відправка команд рендерінгу на GPU. Графічний процесор відповідає за обробку геометрії, виконання шейдерів, розрахунок освітлення та формування фінального зображення в буфері кадру. Для оцінки роботи системи використовують поняття часу кадру на CPU та GPU. Якщо CPU не встигає підготувати дані за потрібний інтервал, то продуктивність вважається обмеженою процесором, якщо ж затримки створює GPU, то виходить, що GPU не встигає виконати потрібну роботу.

В рендерінгу в реальному часі також важливі структурні характеристики сцени: загальна кількість трикутників і вершин, а також кількість батчів, які визначають організаційні витрати на стороні CPU. Велика кількість окремих викликів малювання може призвести до значної втрати продуктивності, навіть якщо геометрична складність кожного об’єкта невелика.

Для забезпечення стабільної роботи використовують цілу групу принципів оптимізації. Зменшення кількості видимих об’єктів за рахунок просторового

відсікання, Occlusion Culling та різних форм LOD дозволяє скоротити обсяг роботи на GPU. Зменшення кількості Draw Call’ів досягається за допомогою статичного та динамічного батчингу, а також Instancing, коли багато копій одного й того самого меша рендеряться однією командою. Оптимізація матеріалів і шейдерів спрямована на скорочення обчислювальної вартості кожного пікселя та кожної вершини. У сукупності ці принципи дозволяють балансувати навантаження між CPU і GPU та утримувати час кадру в межах, необхідних для комфортного використання інтерактивного додатка.

1.2 Огляд ігрового рушія Unity та особливостей URP

Unity є одним із найпоширеніших ігрових рушіїв загального призначення, який використовується як для невеликих проектів, так і для комерційних ігор на різних платформах. Рушій використовує компонентно-орієнтовану архітектуру де сцена складається з ієрархії об’єктів, кожен об’єкт може містити набір компонентів, що відповідають за положення на сцені, відтворення мешів, колізії, анімації та за інший функціонал. Логіка гри реалізується у вигляді скриптів на C#, які взаємодіють з об’єктами сцени через чітко визначений API рушія. Важливою перевагою рушія є кросплатформеність, тобто один і той самий проект можна збирати для персональних комп’ютерів, мобільних пристроїв і ігрових консолей, адаптуючи налаштування якості та продуктивності під конкретну платформу.

Ігровий рушій Unity підтримує систему Scriptable Render Pipeline, яка дозволяє розробникам використовувати різні рендер-пайплайни з різними можливостями та ступенем контролю. Universal Render Pipeline (URP) розроблено як універсальне рішення для широкого спектра платформ, з акцентом на продуктивність і масштабованість. URP надає більш передбачувану та оптимізовану структуру рендерінгу, інтегрує сучасні підходи до обробки освітлення, тіней та постобробки, а також дає можливість розширювати базовий пайплайн за допомогою користувацьких рішень і шейдерів.

Однією з ключових особливостей URP є використання SRP Batcher — це механізм, який скорочує витрати на підготовку рендерінгу матеріалів і шейдерів на стороні CPU. У контексті сцен з великою кількістю однотипних об’єктів цей механізм дозволяє значно покращити ефективність обробки батчів. URP підтримує різні режими рендерінгу, включно з класичним Forward і варіантами з розширеними можливостями (наприклад, Forward+), що дає змогу гнучко налаштовувати баланс між якістю і продуктивністю. Важливим є також тісне поєднання рендерінгу та постобробки: ефекти згладжування, корекції кольору, розмиття, свічення та інші інтегровані в єдиний конвеєр, що спрощує керування ними.

В порівнянні з вбудованим рендерером, URP краще пристосований до роботи з великими сценами за рахунок сучасного підходу до батчингу, більш передбачуваної системи шейдерів і чіткіше структурованих налаштувань якості. Для задач, пов’язаних з рендерінгом великої кількості однотипних об’єктів, URP надає розробнику широкий набір інструментів — від стандартних засобів, таких як статичний батчинг, LOD та Occlusion Culling, до більш просунутих технік Instancing рендерінгу. Це робить Universal Render Pipeline найкращим вибором для експериментального дослідження методів оптимізації рендерінгу, спрямованих на підвищення частоти кадрів при збереженні нормальної візуальної якості.

1.3 Основні принципи графічного конвеєра

Для розуміння методів оптимізації рендерінгу великої кількості об'єктів необхідно проаналізувати основні принципи роботи графічного конвеєра на апаратному рівні. Процес перетворення абстрактних математичних даних про тривимірну сцену у двовимірний масив пікселів на екрані ділиться на багато стадій і базується на тісній взаємодії між центральним процесором (CPU) та графічним процесором (GPU).

Процес рендерінгу починається на стороні CPU, який відповідає за логіку програми, обчислення трансформацій об'єктів, визначення видимості та

підготовку даних для графічного чипа. Ключовим моментом цієї взаємодії є формування Draw Call.

Draw Call — це команда, яку CPU надсилає графічному API (наприклад, DirectX, Vulkan або OpenGL), щоб дати вказівку GPU відрендерити певний набір геометрії з конкретними налаштуваннями (шейдерами, текстурами, буферами). Основна проблема при візуалізації великої кількості об'єктів полягає не лише у складності самої геометрії, а й у накладних витратах на кожен такий виклик. Кожен Draw Call вимагає від CPU перемикання станів графічного конвеєра, що створює значне навантаження на потік рендерінгу. Якщо об'єктів занадто багато, CPU не встигає готувати команди для GPU, що призводить до ситуації “CPU bound”, коли потужність відеокарти простоює через очікування команд від процесора.

Перш ніж GPU зможе розпочати обробку, необхідні дані (координати вершин, індекси, текстури, нормалі) мають бути передані з оперативної пам'яті (RAM) у відеопам'ять (VRAM). Ця передача здійснюється через шину PCIe.

Оскільки пропускна здатність шини обмежена, постійна передача великих масивів даних у кожному кадрі є неефективною. Тому в сучасних ігрових рушіях, зокрема в Unity, використовується стратегія попереднього завантаження даних у буфери відеопам’яті:

- Vertex Buffer Object (VBO): зберігає дані про вершини;
- Index Buffer Object (IBO): зберігає порядок з’єднання цих вершин у трикутники.

Оптимізація на цьому етапі полягає в мінімізації змін у цих буферах та використанні технік інстансингу (GPU Instancing), коли дані про геометрію передаються один раз, а потім багаторазово використовуються для відмальовування копій об’єкта з різними параметрами трансформації.

Після отримання команди Draw Call, графічний процесор активує конвеєр, який складається з послідовних стадій обробки:

1. Вхідний асемблер (Input assembler): GPU збирає сирі дані з буферів у геометричні примітиви (точки, лінії, трикутники);
2. Вершинний шейдер (Vertex shader): це перша програмована стадія конвеєра. Вершинний шейдер виконується для кожної вершини в потоці. Його головна роль — трансформація координат вершини з локального простору об’єкта у світовий простір, а потім у простір відсікання (Clip Space) за допомогою матриць проекції. На цій стадії також можуть обчислюватися дані для освітлення на рівні вершин;
3. Теселяція/Геометричний шейдер (Tessellation/Geometry shader): створює нові вершини для деталізації або модифікує геометрію;
4. Растеризація (Rasterization): процес перетворення векторних трикутників у набір фрагментів (майбутніх пікселів), що відповідають сітці екрана. На цьому етапі система визначає, які саме пікселі покриває геометрична фігура;
5. Піксельний/фрагментний шейдер (Pixel/Fragment Shader): найбільш ресурсомістка стадія. Фрагментний шейдер виконується для кожного потенційного пікселя на екрані. Саме тут відбуваються фінальні розрахунки кольору, текстурування, обчислення попіксельного освітлення, накладання тіней та ефектів прозорості;
6. Тести та змішування (Output Merging)**:** останній етап, де перевіряється глибина (Z-test), трафарет (Stencil test) та відбувається змішування кольорів (Alpha blending). Якщо об'єкт проходить ці тести, фінальний колір записується у буфер кадру (Frame Buffer).

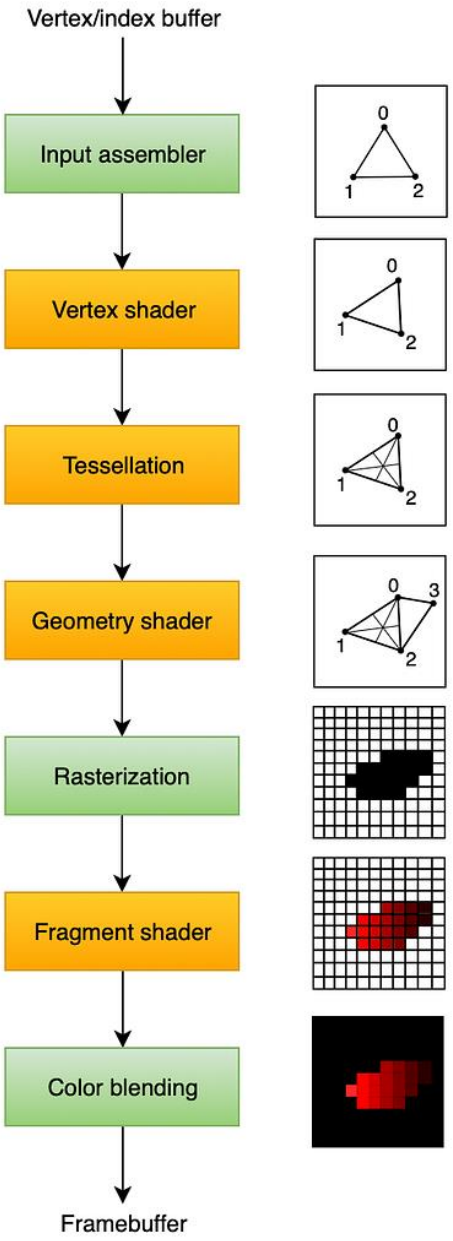


Рисунок 1.1 – схема графічного конвеєра

Таким чином, розуміння роботи конвеєра на рівні заліза дозволяє визначити вузькі місця. При роботі з великою кількістю об'єктів ми маємо справу або з перевантаженням стадії вершинного шейдера, або з надлишком викликів Draw Call, або з високою складністю піксельних обчислень.

1.4 Порівняльна характеристика URP та HDRP

У межах рушія Unity сьогодні співіснують декілька рендер-пайплайнів, серед яких Universal Render Pipeline (URP) та High Definition Render Pipeline (HDRP) є основними для нових проектів. HDRP орієнтований насамперед на високопродуктивні системи і проекти з акцентом на фотореалістичну графіку. Він пропонує розширену модель освітлення, підтримку складних ефектів глобального освітлення, об’ємних світлових ефектів, фізично коректних матеріалів і широкого набору кінематографічних можливостей. Ціна за це — набагато вища базова вартість рендерінгу, складніша конфігурація та залежність від потужного графічного обладнання. У задачах де критичною є стабільна частота кадрів за наявності великої кількості об’єктів, використання HDRP часто призводить до швидкого вичерпання ресурсів графічного процесора, навіть після застосування агресивних методів оптимізації.

Universal Render Pipeline, навпаки, розроблено як більш легкий та універсальний варіант, здатний працювати на широкому спектрі платформ — від мобільних пристроїв до ПК та ігрових консолей. URP відмовляється від частини важких візуальних ефектів, натомість пропонує більш просту й передбачувану структуру рендерінгу, оптимізовану для високої продуктивності. Для задачі рендерінгу 50 000 однакових об’єктів це має принципове значення, бо обмеження за ресурсами в першу чергу пов’язані з великою кількістю викликів рендерінгу та обсягом геометрії, а не з потребою у фотореалістичних ефектах високого рівня.

У контексті даного дослідження вибір URP як основної платформи є обґрунтованим з кількох причин. По-перше, він краще відповідає сценарію в якому головною задачею є забезпечення стабільної частоти кадрів при великій кількості об’єктів на сцені. По-друге, URP надає повний набір базових засобів оптимізації — статичний батчинг, Instancing, LOD, Occlusion Culling, спрощення матеріалів і шейдерів — які дозволяють дослідити вплив класичних прийомів на CPU та GPU без додаткових ускладнень, пов’язаних з більш важкою графічною

моделлю HDRP. По-третє, результати отримані в URP є більш репрезентативними для широкого класу проектів, де саме продуктивність, а не максимальна візуальна якість, виступає головним обмеженням. Таким чином, HDRP розглядається радше як альтернативний варіант для окремого класу застосунків з фотореалістичною графікою, тоді як URP краще підходить в якості середовища для експериментального дослідження оптимізації рендерінгу великої кількості об’єктів.

1.5 Поняття оптимізації в розробці ігор

Оптимізація в загальному розумінні трактується як процес підвищення ефективності певного об’єкта, системи або процесу з метою отримання найкращого можливого результату за наявних обмежень. У технічній літературі оптимізацію часто визначають як отримання найкращого результату за заданих умов або обмежень, тобто коли бажаний ефект досягається не взагалі, а в межах конкретних ресурсів і правил (час, пам’ять, апаратні можливості, вимоги до якості). У прикладному сенсі для програмних систем це означає приведення програми до такого стану, коли вона працює стабільніше й ефективніше, не втрачаючи необхідної функціональності.

У сфері програмного забезпечення оптимізація зазвичай розуміється як покращення продуктивності та ефективності, тобто зменшення часу виконання операцій, зниження споживання пам’яті, зменшення навантаження на CPU/GPU, оптимізація доступу до даних тощо. Важливо, що оптимізація не є разовою дією. На практиці це послідовний процес в якому спочатку визначається вузьке місце, далі застосовується певний прийом оптимізації, а потім результат перевіряється за вимірюваними метриками. Такий підхід відповідає загальній логіці оптимізації як пошуку найкращого рішення серед допустимих варіантів, а не як спроби покращити все одразу.

Хоча метою оптимізації є отримання оптимальної системи, абсолютно оптимальний варіант у реальних програмних задачах досягається далеко не завжди. Причина полягає в тому, що оцінка результату залежить від обраного

критерію, а критерії часто конфліктують між собою. У практиці розробки систем оптимізована реалізація нерідко є найкращою лише для конкретної задачі або класу сценаріїв використання. Наприклад, прискорення роботи може досягатися ціною більшого споживання пам’яті, а в задачах, де критичною є пам’ять, навпаки можуть застосовуватися менш швидкі алгоритми з меншими потребами до RAM.

Подібна ситуація характерна і для багатьох складних систем, де доводиться оптимізувати декілька параметрів одночасно. Якщо цілей кілька, то зазвичай не існує єдиного рішення, яке буде найкращим за всіма показниками. У таких випадках застосовується ідея компромісного вибору і пошуку розв’язків, які є оптимальними в балансі, тобто поліпшення одного показника неминуче погіршує інший. Це відповідає загальним принципам багатокритеріальної оптимізації, де рішення часто є компромісом між конфліктними цілями.

У контексті комп’ютерних ігор оптимізація має особливе значення через режим реального часу, бо гра повинна формувати кадри з достатньою частотою та без помітних затримок. Тому на практиці оптимізація ігрових застосунків найчастіше зводиться до того, щоб вкластися в бюджет кадру на цільовій платформі. В Unity для цього рекомендовано аналізувати не лише FPS, а й час кадру на CPU та GPU, оскільки саме ці значення показують, де виникає реальне обмеження продуктивності.

Окремо слід підкреслити, що в оптимізації рендерінгу не існує універсального методу, який би автоматично підвищував продуктивність у будь-якій сцені й за будь-яких умов. Більшість оптимізацій дають приріст лише в конкретних слабких місцях, але водночас можуть накладати обмеження або мати побічні наслідки. Наприклад, зменшення складності мешів/матеріалів може зменшити навантаження на GPU, але вплинути на візуальну якість; агресивне відсікання або спрощення може вимагати іншої організації сцени; оптимізація draw calls зменшує накладні витрати CPU на передачу команд до GPU, проте потребує дотримання умов сумісності матеріалів/об’єктів. Таким чином, оптимізація рендерінгу в іграх зазвичай є пошуком компромісу між

продуктивністю, якістю зображення та зручністю/складністю побудови контенту.

Крім того, прагнення до ідеально оптимальної програми майже завжди стикається з практичною межею, бо після певного рівня покращень кожний наступний відсоток виграшу потребує значно більших затрат часу і ускладнення рішення, тоді як реальна користь стає незначною. Через це процес оптимізації зазвичай завершують тоді, коли досягається достатній рівень продуктивності для поставлених вимог, а не тоді, коли систему вже стає неможливо поліпшити.

1.6 Огляд методів оптимізації

Unity має широкий список можливих методів оптимізації. Головною метою методів оптимізації є полегшення роботи для центрального процесора так і для графічного процесора, що в свою чергу збільшить частоту кадрів застосунку. В межах дослідження було розглянуто набір з 6 методів оптимізації, які впливають на різні складові процесу формування кадру. Нижче подано опис кожного методу та очікуваний ефект від його застосування в умовах сцени з великою кількістю однотипних об’єктів:

- статичний батчинг (Static Batching). Цей метод дозволяє об’єднати нерухомі об’єкти для зменшення кількості викликів рендерінгу. Об’єкти вважаються нерухомими, коли вони не змінюють положення, обертання та масштаб під час роботи гри. Щоб зробити об’єкти статичними вони не повинні створюватись динамічно, тобто вони повинні бути розміщенні розробником на сцені заздалегідь. Unity формує спільні буфери вершин та індексів для таких об’єктів і виконує їх відображення більш ефективно. Очікуваним результатом від цього методу оптимізації є зменшення кількості батчів і як наслідок зниження навантаження на CPU. Водночас метод може збільшувати споживання пам’яті, оскільки зберігаються об’єднані геометричні дані;
- спрощений матеріал (Light Material). Цей метод оптимізації включає в себе використання менш складного шейдера для зниження вартості обробки кожного пікселя. Під менш складним шейдером мається на увазі шейдер без складних обчислень освітлення, додаткових текстур, деталізованих масок, складних варіацій прозорості і т.д. Використання спрощених матеріалів впливає на зменшення часу кадру на GPU, оскільки спрощується обробка пікселів і знижується вартість виконання шейдера;

- спрощена модель (Light Mesh). Цей метод зменшує геометричну складність об’єктів за рахунок меншої кількості трикутників та вершин, які потрібно обробляти під час рендерінгу. Цього можна досягти використовуючи низькополігональну версію моделі об’єкта або оптимізації оригінальної моделі об’єкта. В результаті використання цього методу оптимізації очікується зниження навантаження на GPU через меншу геометрію для обробки. Найбільш помітний результат очікується у випадках, коли сцена має високу полігональну складність і GPU витрачає значний час на обробку вершин/трикутників;
- рівні деталізації (LOD). Головною ідеєю цього методу оптимізації є використання різних варіантів моделі залежно від відстані до камери. Для одного й того самого об’єкта готуються кілька варіантів моделі з різною геометричною складністю, а рушій автоматично перемикає або навіть вимикає їх залежно від відстані до камери. Очікуваний результат такий самий як і від спрощення моделей, а саме зменшення кількості трикутників і вершин у кадрі за рахунок того, що далекі об’єкти відображаються простішою геометрією. Це зменшує навантаження на GPU та може підвищити FPS у сценах, де значну частину кадру складають об’єкти на середніх/великих дистанціях;
- оклюзійне відсікання (Occlusion Culling). Цей метод оптимізації включає в себе відключення від рендерінгу об’єктів, що повністю закриті іншими або знаходяться поза полем зору. На відміну від стандартного відсікання за межами поля зору камери (frustum culling), оклюзійне відсікання додатково враховує перекриття об’єктами один одного. Очікуваним ефектом є зменшення зайвих операцій як на CPU, так і на GPU, оскільки в рендерінг не потрапляють об’єкти, що не впливають на зображення. Unity підкреслює, що метод зменшує зайві обчислення, але сам механізм також має вартість, бо частина розрахунків виконується на CPU і потрібні дані для оклюзії зберігаються в пам’яті;

– GPU Instancing Indirect. Використання цього методу оптимізації дозволяє рендерити велику кількість копій одного меша за допомогою невеликої кількості викликів рендерінгу. В Unity цей метод оптимізації реалізується за рахунок виклику DrawMeshInstancedIndirect де задаються меш, матеріал, межі та буфер аргументів на GPU. Цей метод оптимізації дозволяє різко скоротити кількість батчів і навантаження на CPU, оскільки суттєва частина рендерінгу організовується через GPU-орієнтований механізм. Це особливо показово для сцен з десятками тисяч однакових об’єктів, бо традиційний підхід рендерінгу швидко впирається в великі витрати продуктивності на центральному процесорі. Водночас метод вимагає правильної організації даних і загалом є більш складним у реалізації порівняно з базовими засобами Unity.

Висновки до розділу 1

У першому розділі було розглянуто теорію, яка стосується рендерінгу в реальному часі та особливостей його застосування в сучасних ігрових рушіях. Визначено ключові метрики експерименту, такі як: частота кадрів, час кадру, навантаження на центральний та графічний процесори, кількість батчів, кількість трикутників на сцені. Показано, що для великої кількості однотипних об’єктів вирішальне значення має не лише загальна кількість трикутників, а й організація процесу відправки команд рендерінгу, оскільки значні накладні витрати на стороні CPU можуть стати головною причиною зниження продуктивності додатка.

Проаналізовано архітектуру побудови сцени в Unity та особливості Universal Render Pipeline як сучасного рендер-пайплайна орієнтованого на продуктивність і кросплатформеність. Відзначено роль компонентного підходу, гнучкості налаштувань якості та наявності вбудованих механізмів оптимізації, зокрема SRP Batcher, статичного батчингу, Instancing, підтримки LOD та Occlusion Culling. Показано, що URP забезпечує більш передбачувану структуру рендерінгу та кращі можливості масштабування сцени за рахунок меншої вартості візуальних ефектів.

Проведено порівняльний аналіз URP та HDRP в контексті поставленої задачі. Зазначено, що HDRP доцільно використовувати для проектів із фотореалістичною графікою та складними ефектами освітлення, але його базові витрати роблять цей пайплайн менш придатним для експериментів із дуже великою кількістю однотипних об’єктів. Натомість URP, завдяки орієнтації на продуктивність і підтримку широкого спектра пристроїв, є більш відповідним вибором для дослідження методів оптимізації рендерінгу, метою яких є стабільний високий FPS.

Було детально оглянуто використані в досліджені методи оптимізації, які далі використовуються в експериментальній частині. Було визначено, який саме ефект очікується від кожного підходу.

РОЗДІЛ 2 ВИМОГИ ДО РОЗРОБКИ

2.1 Основні функціональні вимоги

Для забезпечення коректного проведення експериментів інструмент дослідження повинен виконувати наступні функціональні вимоги:

– система має забезпечувати формування тестової сцени з заданою кількістю однотипних об’єктів;

– інструмент повинен надавати можливість активації та деактивації окремих методів оптимізації рендерінгу;

– перемикання між методами оптимізації має здійснюватися через окремі сцени за допомогою інтерфейсу керування;

– програма має реалізовувати збір метрик продуктивності. Вона повинна здійснювати розігрів сцени, а потім фіксацію показників упродовж заданого інтервалу часу;

– результати вимірювань мають відображатися безпосередньо в інтерфейсі програми.

Сукупність цих функціональних вимог гарантує, що розроблений програмний засіб буде придатним для проведення серії експериментів відповідно до поставленої задачі дослідження та дозволить отримати узгоджені, порівнювані між собою результати.

2.2 Вхідні дані

До вхідних даних у межах даного дослідження відносяться:

Параметри сцени**:**

- кількість об’єктів: 50 000 однотипних сфер;
- просторове розміщення об’єктів: сітка з фіксованим кроком по осях, однакова структура для всіх сценаріїв;
- фіксоване положення та орієнтація камери під час вимірювань.

Конфігурація рендерінгу**:**

- ігровий рушій Unity (версія 6.0.49f1);
- рендер-пайплайн: URP;
- однакові налаштування якості графіки, тіней, постобробки та роздільної здатності для всіх сценаріїв.

Сценарії оптимізації**:**

- базовий варіант без спеціальних оптимізацій;
- статичний батчинг (Static Batching);
- спрощений матеріал (Light Material);
- спрощена модель сфери (Light Mesh);
- рівні деталізації (LOD);
- Occlusion Culling з частковою видимістю сцени;
- GPU Instancing Indirect.

Апаратне середовище**:**

- центральний процесор: Intel(R) Core(TM) i7-14650HX;
- графічний процесор: NVIDIA GeForce RTX 4060 Laptop GPU;
- оперативна пам’ять: 32 ГБ 5600 МТ/с;
- операційна система: Windows 11;
- версія графічного драйвера: NVIDIA 591.59.

Ці вхідні дані задаються в налаштуваннях Unity проекту та в параметрах інструменту й залишаються сталими в межах експерименту для коректного порівняння результатів.

2.3 Вихідні дані

До вихідних даних, які формуються в процесі експериментів, відносяться:

Показники:

- середнє значення FPS;
- середній час кадру на CPU (CPU Frame Time);
- середній час кадру на GPU (GPU Frame Time);
- кількість батчів рендерінгу (Batch Count);
- кількість трикутників.

Зазначені вихідні дані фіксуються тестовим інструментом й використовуються для побудови таблиць, графіків та порівняльного аналізу ефективності окремих методів оптимізації.

Висновки до розділу 2

У другому розділі були сформовані основні функціональні вимоги до програми, а саме автоматизоване формування сцени з 50 000 сфер, можливість перемикання окремих методів оптимізації, а також реалізація системи збору та відображення зазначених метрик. Окремо було визначено склад вхідних даних та вихідних даних, що забезпечує відтворюваність експериментів і коректність порівняння різних методів оптимізації.

Таким чином, у цьому розділі було сформовано основу для дослідження: визначено умови проведення експериментів, структуру вхідних та вихідних даних, а також вимоги до програмного інструменти, який реалізує ці умови.

РОЗДІЛ 3 РОЗРОБКА ІНСТРУМЕНТАЛЬНИХ ЗАСОБІВ ДЛЯ ДОСЛІДЖЕННЯ МЕТОДІВ ОПТИМІЗАЦІЇ РЕНДЕРІНГУ

Проведення експериментального дослідження потребує не лише теоретичного обґрунтування обраних методів оптимізації, а й створення спеціалізованого інструменту, який забезпечить відтворювані умови вимірювань і контроль над параметрами сцени. У попередніх розділах було визначено основні метрики продуктивності, сценарії оптимізації та вимоги до вхідних і вихідних даних. Наступним кроком є реалізація програмного інструменту, що дозволяє на практиці застосувати ці вимоги в середовищі Unity та організувати серію експериментів.

У цьому розділі розглядаються вибір мови програмування та середовища розробки, загальна архітектура створеного інструменту. Описані рішення слугують основою, на якій виконується експериментальний аналіз ефективності методів оптимізації.

3.1 Мова програмування та середовище розробки

У межах даної роботи як середовище розробки було обрано ігровий рушій Unity. Unity є кросплатформеним рушієм для розробки ігор, що широко використовується для створення двовимірних і тривимірних ігор, симуляторів та інтерактивних застосунків для ПК, мобільних пристроїв і консолей. Він базується на компонентно-орієнтованій архітектурі де сцена складається з ієрархії об’єктів, до яких додаються компоненти для відтворення геометрії, анімації, фізики та логіки, тобто скриптів. Рушій надає вбудований редактор сцен, засоби для роботи з матеріалами та шейдерами, систему збору під різні платформи та підтримку різних рендер-пайплайнів, зокрема Universal Render Pipeline, що й використовується в даному дослідженні.

Основною мовою програмування для створення ігрової логіки в Unity є C#. Скрипти на C# компілюються у керований код, який виконується на віртуальній машині .NET/Mono, що забезпечує типобезпечність, автоматичне керування

пам’яттю та розвинену стандартну бібліотеку. C# підтримує об’єктно-орієнтоване програмування, узагальнення, LINQ та інші сучасні можливості, що робить цю мову придатною як для високорівневої логіки гри, так і для побудови допоміжних інструментів, зокрема системи збору та аналізу метрик. У рамках даної роботи C# використовується для реалізації скриптів, які відповідають за генерацію сцени, перемикання методів оптимізації рендерінгу, керування камерою та автоматизований збір показників FPS, часу кадру на CPU і GPU та кількості батчів. Поєднання Unity як середовища розробки та C# як основної мови програмування забезпечує достатню гнучкість і продуктивність для реалізації інструменту для дослідження.

3.2 Особливості мови C# для використання у розробці

Мова C# є основною мовою програмування в середовищі Unity і поєднує властивості сучасної високорівневої мови з можливістю достатньо ефективної роботи з ресурсами. Завдяки статичній типізації, об’єктно-орієнтованій моделі та інтеграції з платформою .NET C# забезпечує зручну підтримку великих проектів і доступ до розвиненої бібліотеки. Це робить мову придатною як для реалізації ігрової логіки, так і для створення допоміжних інструментів, зокрема засобів збору й аналізу метрик продуктивності.

Головними особливостями мови C# є:

- статична типізація та об’єктно-орієнтована модель, що дозволяють виявляти помилки на етапі компіляції та будувати зрозумілу ієрархію класів і компонентів;
- тісна інтеграція з платформою .NET, яка надає широкий набір колекцій, засобів роботи з файлами, мережею, потоками та іншими системними ресурсами;
- підтримка сучасних мовних конструкцій, таких як узагальнення, делегати, події, лямбда-вирази та LINQ, що полегшують розробку;

- можливість комбінувати високий рівень абстракції з більш низькорівневими механізмами, зокрема через використання структур значимих типів і, за потреби, небезпечного коду для оптимізації важких ділянок коду;
- наявність засобів для роботи з багатопотоковістю та асинхронним кодом, що дає можливість виносити частину обчислень за межі основного потоку й організовувати асинхронні операції.

Головними перевагами мови C# у контексті використання з Unity є:

- сумісність з рушієм: скрипти на C# вбудовуються в компонентну модель Unity, що спрощує розробку ігрових систем та інструментів;
- читабельність і підтримуваність коду;
- інструменти для налагодження та профілювання, які полегшують пошук помилок і аналіз продуктивності;
- широка спільнота та велика кількість навчальних матеріалів, що полегшує розв’язання типових задач.

У сукупності ці особливості та переваги роблять C# доцільним вибором для реалізації інструментальних засобів дослідження в даній роботі. Мова дозволяє одночасно підтримувати достатній рівень продуктивності й зберігати структурованість та зрозумілість програмного коду.

3.3 Особливості рушія Unity для використання в розробці

Unity - це кросплатформений ігровий рушій реального часу, який широко використовується для розробки дво- та тривимірних ігор, симуляторів, інтерактивних застосунків та візуалізацій. Він надає розробнику редактор сцен, систему імпорту ресурсів, засоби збірки під різні платформи та зручну інтеграцію зі скриптами на мові C#. Така комбінація робить Unity зручним як для невеликих навчальних і інді-проектів, так і для комерційних продуктів.

Початок роботи з Unity, як правило, відбувається через утиліту Unity Hub. Спочатку користувач встановлює Unity Hub, після чого через нього завантажує потрібну версію рушія. Далі в Unity Hub створюється новий проект із вибором

шаблону (3D URP, 2D, HDRP тощо), після чого відкривається середовище рушія. У цьому середовищі розробник може додавати об’єкти на сцену, налаштовувати камери, освітлення, матеріали, підключати скрипти та збирати застосунок під цільові платформи.

Архітектура Unity базується на компонентно-орієнтованій моделі. Основні елементи:

- сцена (Scene) — одиниця, що містить набір об’єктів. Проект може складатися з однієї або багатьох сцен (меню, рівні, тестові середовища тощо);
- об’єкт сцени (GameObject) — базовий контейнер, до якого додаються компоненти. Сам по собі GameObject містить лише трансформацію (позицію, обертання, масштаб), а будь-яка корисна та унікальна поведінка реалізується через компоненти;
- компоненти (Components) — модулі функціональності. Існують стандартні компоненти (MeshRenderer, Camera, Light, Collider, Rigidbody), а також користувацькі скрипти на C#, які наслідують базовий тип MonoBehaviour і підключаються як компонент до об’єкта. Власне компоненти реалізовують всю корисну та унікальну поведінку об’єктів;
- система ресурсів (Assets) — проект містить матеріали, меші, текстури, шейдери, префаби, сцени, скрипти та інші ресурси, які організовано в структурі папок. Unity забезпечує імпорт активів із популярних форматів (FBX, PNG тощо) та їх подальше використання в сценах;
- префаби (Prefabs) — шаблони об’єктів, які дозволяють створювати багато копій однієї конфігурації GameObject з однаковим набором компонентів. Це спрощує повторне використання об’єктів і централізовану зміну їх налаштувань.

Важливою частиною архітектури є життєвий цикл скриптів і кадру. Unity викликає певні методи (Awake, Start, Update, FixedUpdate, OnDestroy тощо) в чітко визначені моменти кадру, що дозволяє розробникові організовувати логіку гри, оновлення станів, фізичні обчислення та взаємодію з користувачем.

Графічний рендерінг тісно інтегровано з цим циклом: після оновлення логіки сцена готується до відображення, формуються виклики рендерінгу та здійснюється виведення кадру.

Рушій також має розвинену систему налаштувань під різні платформи. Через вікно Build Settings та Project Settings розробник може обирати цільові платформи (Windows, Android, iOS, консолі), перемикати рендер-пайплайни (Built-in, URP, HDRP), налаштовувати якість графіки, роздільну здатність та інші параметри, що впливають на продуктивність і візуальну якість. У контексті дослідження методів оптимізації рендерінгу це дає можливість фіксувати єдині налаштування для всіх експериментів і змінювати лише ті параметри, які безпосередньо стосуються обраних методів.

Підсумовуючи, Unity є гнучким та потужним ігровим рушієм, який поєднує зручний візуальний редактор, компонентну архітектуру, підтримку сучасних рендер-пайплайнів та написання логіки на мові C#. Ці особливості роблять його придатним не лише для створення ігор, а й для побудови інструментів для дослідження.

3.4 Реалізація проекту в Unity

В проекті реалізоване головне меню, яке надає користувачу можливість вибирати методи оптимізації через графічний інтерфейс. Через інтерфейс користувача можна обрати потрібний метод, після чого завантажується відповідна сцена з реалізацією обраного методу оптимізації.

Для кожного методу оптимізації створено окрему сцену. Загалом проект містить вісім сцен, а саме головне меню та по сцені для кожного методу оптимізації включно з базовим сценарієм без методів оптимізації. Сцена “Main Menu” використовується лише для навігації і вибору сцени з відповідним методом оптимізації, тоді як решта сім сцен призначені для демонстрації та випробування конкретних методів. Сцена Basic являє собою базовий випадок без спеціальних оптимізацій, Light Mesh використовує спрощену модель сфери,

Light Material – спрощений матеріал і т.д. Такий поділ на окремі сцени забезпечує ізольоване тестування кожного методу в однакових умовах.

Кожна сцена для тесту методу оптимізації містить 50 000 сферичних об’єктів або створює їх динамічно під час запуску залежно від методу оптимізації. У всіх сценах встановлено однакові налаштування освітлення та однакове положення камери. Це зроблено для того, щоб результати вимірювань продуктивності були коректними та порівнюваними між різними методами оптимізації. Велика кількість однотипних об’єктів навантажує систему рендерінгу, що дозволяє оцінити ефективність кожного підходу в умовах близьких до граничних.

Для збору даних продуктивності в кожну з тестових сцен додано спеціальний префаб SceneTools, який відповідає за автоматичний збір і відображення ключових метрик. Зокрема, фіксуються показники FPS (кількість кадрів за секунду), часу формування кадру на CPU та GPU (CPU/GPU frame time), кількості батчів (Batch count) та кількості трикутників (Tris count). Збирання цих метрик починається із затримкою в 5 секунд після запуску сцени та триває 10 секунд, щоб отримати усереднені дані за цей проміжок часу. Така затримка запроваджена з метою уникнення впливу початкового завантаження сцени на результати вимірювань – тобто, щоб перші, нестабільні кадри не враховувалися у статистиці.

Після завершення перегляду сцени з певним методом оптимізації користувач має можливість повернутися до головного меню для вибору іншого методу. Навігація назад до “Main Menu” реалізована у кожній тестовій сцені. Таким чином, можна послідовно досліджувати всі методи оптимізації, перемикаючись між сценами в межах однієї сесії роботи застосунку.

Структура проекту організована за загальноприйнятими принципами розробки в Unity. У репозиторії створено окремі папки для різних типів ресурсів: Scenes, Scripts, Materials, Meshes, Prefabs тощо. Такий поділ спрощує підтримку проекту та допомагає швидко знаходити потрібні компоненти. Кожна сцена містить лише необхідні для її роботи об’єкти та скрипти, що підвищує ясність і керованість проекту під час експериментального дослідження.

3.5 Формалізація задачі

Формалізуємо задачі до додатку у вигляді діаграм. Для цього використаємо діаграму прецедентів на рисунку 3.1 та діаграму діяльності на рисунку 3.2

Рисунок 3.1 – діаграма прецедентів аналізу методів оптимізації

Діаграма відображає взаємодію користувача із інструментом для дослідження та описує функціональні можливості системи з точки зору користувача. В межах інструмента для дослідження визначено чотири основні варіанти використання користувачем: вибір методу оптимізації, запуск експерименту, перегляд метрик продуктивності та повернення в головне меню.

Рисунок 3.2 – діаграма діяльності аналізу методів оптимізації

Робота починається з вибору методу оптимізації. Далі починається завантаження сцени. Після завантаження сцени автоматично починається експеримент. Програма виконує прогрів протягом 5 секунд після чого починається збір даних протягом 10 секунд і потім на екрані з’являться результати експерименту з всіма метриками.

3.6 Проектування інтерфейсу користувача

Проектування інтерфейсу користувача є критично важливим етапом розробки будь-якого програмного забезпечення, оскільки саме він виступає в ролі сполучною ланки між функціональними можливостями системи та кінцевим користувачем. Ефективність інтерфейсу визначається не лише його естетичним виглядом, а й тим, наскільки інтуїтивно зрозумілою є логіка взаємодії, що безпосередньо впливає на продуктивність роботи та швидкість освоєння програми.

У сучасних програмах інтерфейс має відповідати вимогам ергономічності, забезпечувати мінімальне навантаження на користувача та надавати швидкий доступ до ключових інструментів керування. Важливим аспектом є вибір моделі діалогу, яка визначає структуру обміну інформацією між людиною та комп'ютером.

При розробці систем виділяють кілька основних моделей взаємодії, що визначають логіку побудови інтерфейсу:

- меню-орієнтований діалог. Найбільш поширений тип, де користувач обирає дії із запропонованого списку;
- командний інтерфейс. Базується на прямому введенні текстових інструкцій. Це професійний інструмент, що забезпечує максимальну швидкість роботи для підготовлених користувачів;
- безпосереднє маніпулювання. Передбачає пряму взаємодію з об'єктами на екрані (перетягування, зміна розміру). Це створює відчуття фізичного контролю та забезпечує миттєвий візуальний відгук;
- інтерфейси на основі форм. Оптимальні для введення великих масивів структурованих даних. Дозволяють користувачеві вільно переміщатися між полями та коригувати інформацію перед її фінальною обробкою.

Для сучасних ігрових рушіїв, зокрема Unity, характерним є використання розвиненого WIMP-інтерфейсу (Windows, Icons, Menus, Pointer) який адаптований під специфічні потреби розробки та візуалізації. Оскільки Unity є

багатофункціональною платформою, реалізація інтерфейсу в ній виходить за межі простого відображення кнопок і включає складні системи взаємодії з 3D-середовищем.

Основними інструментами для створення UI в Unity є два ключові фреймворки:

- Unity UI: система, що базується на ігрових об'єктах (GameObjects) та компонентах Canvas. Вона дозволяє легко створювати інтерфейси, що адаптуються під різні роздільні здатності екрана, і широко використовується для створення меню, HUD та діалогових вікон;
- UI Toolkit: новіша система, що використовує веб-технології (подібні до HTML та CSS). Вона орієнтована на підвищення продуктивності та кращий поділ логіки і стилю, що є критичним для складних професійних інструментів.

Однією з головних переваг Unity є можливість використання різних режимів візуалізації інтерфейсу (Render Mode):

- Screen Space - Overlay: UI відображається поверх усього вмісту сцени, що є стандартним для класичних програм;
- Screen Space - Camera: інтерфейс прив'язується до конкретної камери, що дозволяє додавати ефекти перспективи або взаємодії з пост-обробкою;
- World Space**:** елементи інтерфейсу розміщуються безпосередньо в 3D-просторі сцени як фізичні об'єкти. Це дозволяє створювати інтуїтивні панелі моніторингу та діагностичні інструменти, розташовані поруч із об'єктами дослідження.

Для розробленого інструменту для дослідження було обрано структуру на основі меню, коли користувач взаємодії с застосунком шляхом вибору одного з доступних пунктів (кнопок), без необхідності введення даних з клавіатури. Такий підхід гарно підходить задачі експерименту, оскільки забезпечує швидкий вибір сцени з відповідним методом оптимізації та мінімізує кількість дій для початку експерименту. Технічна реалізація цього інтерфейсу була виконана за допомогою системи Unity UI, яка дозволяє створювати гнучкі та продуктивні

графічні елементи керування. Для візуалізації було використано режим рендерінгу Screen Space - Overlay, що забезпечує відображення панелей керування та інформаційних вікон поверх усього вмісту сцени. Це гарантує, що інтерфейс залишається чітким та доступним для користувача незалежно від налаштувань камери чи складності геометрії, що візуалізується в ході дослідження методів оптимізації.

Макет інтерфейсу додатку наведено на рисунках 3.3. та 3.4.

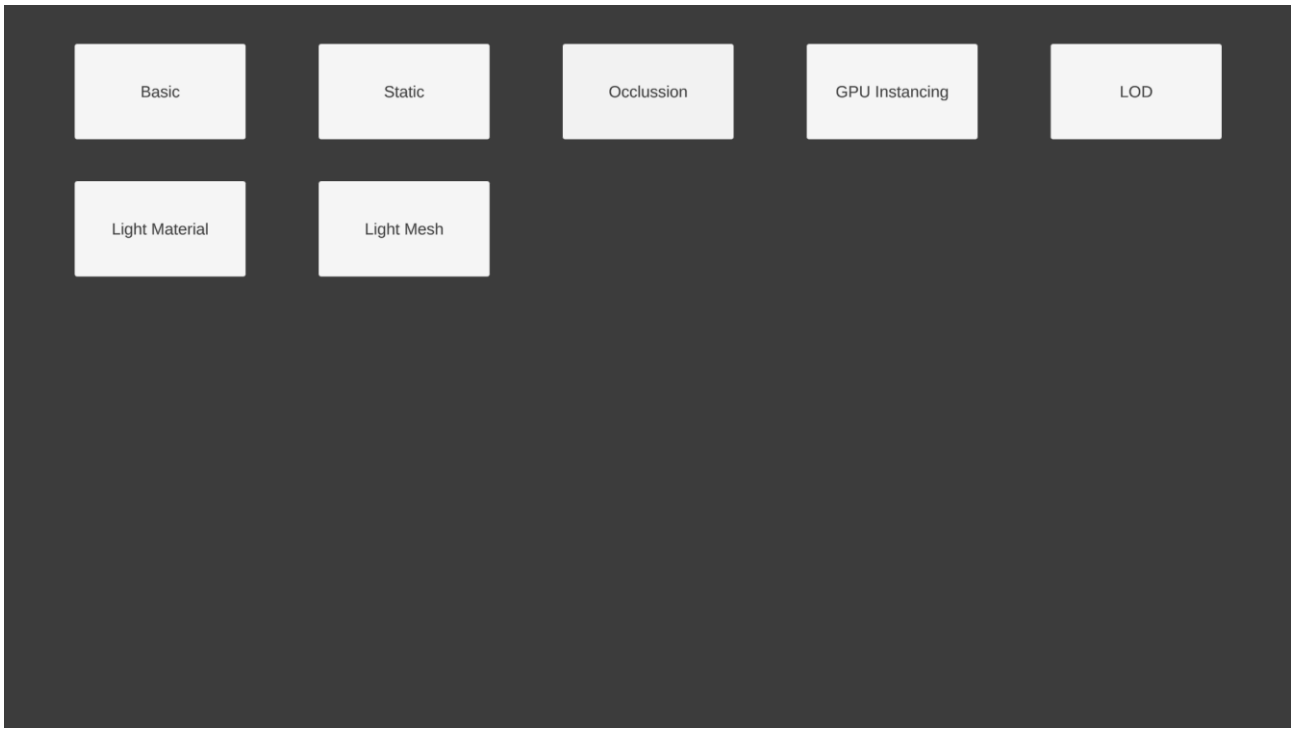


Рисунок 3.3 – інтерфейс головного меню

Головне меню на рисунку 3.3 реалізовано у вигляді простого екрану з набором кнопок, кожна з яких відповідає окремому методу оптимізації. Меню містить сім кнопок: Basic, Static, Occlusion, GPU Instancing, LOD, Light Material, Light Mesh. Натискання на кнопку ініціює перехід на сцену з відповідним методом оптимізації та запуском експерименту.

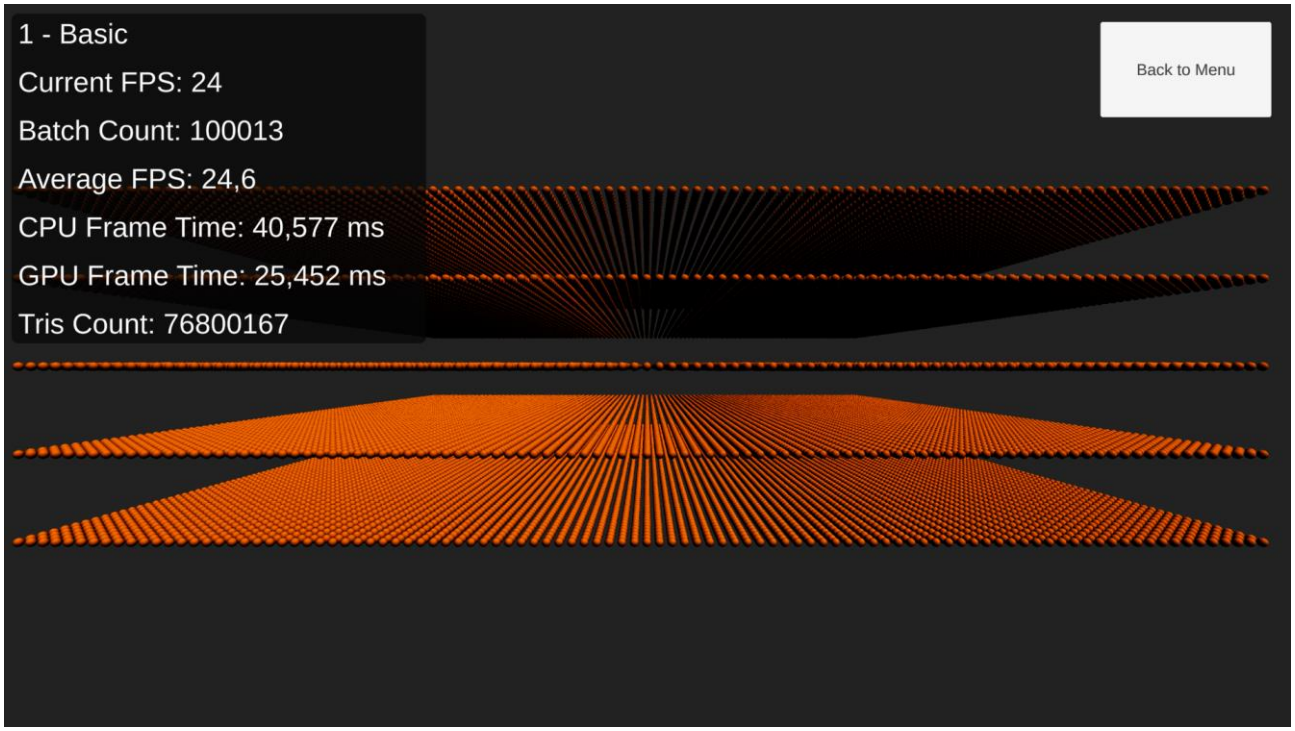


Рисунок 3.4 – інтерфейс сцени з експериментом

Екран експерименту на рисунку 3.4 містить дві ключові частини, а саме область з масивом сфер та інформаційну панель з результатами вимірювань. Панель виводить поточні та усереднені значення основних метрик продуктивності: поточний FPS, кількість батчів (Batch Count), середній FPS (Average FPS), час кадру на CPU та GPU (CPU/GPU Frame Time), а також кількість трикутників (Tris Count). Окремо передбачено кнопку “Back to Menu”, яка забезпечує повернення до головного меню для вибору іншого методу та повторення експерименту.

Висновки до розділу 3

У третьому розділі було обґрунтовано вибір технологій для дослідження та описано розробку інструменту для проведення експериментів. Описано, що поєднання рушія Unity та мови програмування C# дозволяє створити інструмент для оцінювання методів оптимізації рендерінгу. Unity виступає як гнучке середовище реального часу з компонентно-орієнтованою архітектурою, візуальним редактором сцен і підтримкою сучасних рендер-пайплайнів. Завдяки цьому стало можливим побудувати сцену з великою кількістю однотипних об’єктів, налаштувати різні сценарії оптимізації та зафіксувати єдині умови для всіх експериментів. Мова програмування C# забезпечила зручну реалізацію логіки для генерації сцен та автоматизованого збору даних, зберігаючи при цьому читабельність і структурованість коду. Обраний рушій та мова програмування є придатними для реалізації поставлених завдань, а розроблений інструмент забезпечує можливість коректного та порівнюваного аналізу ефективності методів оптимізації рендерінгу.

Додатково в межах розділу було спроектовано та реалізовано інтерфейс користувача, що базується на структурі меню. Такий підхід дозволив мінімізувати кількість дій користувача для початку експерименту та забезпечив швидку навігацію між тестовими сценами. Технічна реалізація графічного інтерфейсу була виконана за допомогою фреймворку Unity UI (uGUI), який надав необхідні інструменти для створення адаптивних елементів керування. Для забезпечення постійної видимості діагностичної інформації та панелей вибору методів було використано режим візуалізації Screen Space - Overlay, що дозволяє відображати інтерфейс поверх усього тривимірного контенту сцени без спотворень. Це забезпечило зручність моніторингу ключових показників (FPS, час кадру на CPU/GPU, кількість батчів) безпосередньо під час виконання рендерінгу великої кількості об'єктів.

РОЗДІЛ 4 АНАЛІЗ ЕФЕКТИВНОСТІ МЕТОДІВ ОПТИМІЗАЦІЇ

Цей розділ присвячено аналізу ефективності методів оптимізації рендерінгу сцени з великою кількістю однотипних об’єктів в середовищі Unity. Після розробки проекту та підготовки тестових сцен було виконано серію експериментів, метою яких є визначення впливу кожного методу на продуктивність. Щоб забезпечити коректність порівняння, всі експерименти проводилися в однакових умовах, а саме з незмінними налаштуваннями камери та освітлення, сталою структурою сцени та єдиним підходом до збору показників. Кожен метод реалізовано на окремій сцені, що дозволяє ізолювати його вплив і уникнути змішування ефектів різних методів оптимізації. Порівняння виконується відносно базового сценарію, який відображає стан сцени без застосування спеціальних засобів оптимізації та використовується як контрольна конфігурація. Для кожного методу оптимізації експеримент виконується 3 рази та береться середнє значення всіх метрик.

В межах дослідження аналізуються такі методи оптимізації:

– базовий сценарій – рендерінг без спеціальних засобів оптимізації, використовується як контрольна конфігурація;

– статичний батчинг (Static Batching) – об’єднання нерухомих об’єктів для зменшення кількості викликів рендерінгу;

– спрощений матеріал (Light Material) – використання менш складного шейдера для зниження вартості обробки кожного пікселя;

– спрощена модель кулі (Light Mesh) – зменшення геометричної складності об’єктів за рахунок меншої кількості трикутників;

– рівні деталізації (LOD) – використання різних варіантів моделі залежно від відстані до камери;

– оклюзійне відсікання (Occlusion Culling) – відключення від рендерінгу об’єктів, що повністю закриті іншими або знаходяться поза полем зору;

– GPU Instancing Indirect – використання непрямого інстансингу для рендерінгу великої кількості копій одного меша за допомогою невеликої кількості викликів рендерінгу.

Оцінювання ефективності методів здійснюється на основі ключових метрик, які відображають як загальну плавність роботи, так і розподіл навантаження між центральним та графічним процесорами:

– середня частота кадрів (FPS) – узагальнена характеристика плавності відображення сцени;

– середній час кадру на CPU (CPU frame time) – відображає навантаження на центральний процесор в межах одного кадру;

– середній час кадру на GPU (GPU frame time) – характеризує обсяг роботи графічного процесора в межах одного кадру;

– кількість батчів (Batch Count) – показує, скільки окремих викликів рендерінгу необхідно для формування кадру;

– кількість трикутників (Tris Count) – показує полігональну складність сцени.

Збір даних виконується автоматизовано в межах кожної тестової сцени. Після запуску сцени передбачено короткий інтервал стабілізації, після чого обчислюються середні значення метрик за визначений проміжок часу. Для зручності інтерпретації результати подаються у вигляді зведеної таблиці та діаграм, а також розглядаються окремо для кожного сценарію.

Аналіз результатів спрямований на визначення того, які методи забезпечують найбільший приріст частоти кадрів, зменшують час кадру на CPU та GPU і скорочують кількість батчів та трикутників. Це дає змогу не лише порівняти ефективність підходів між собою, а й сформувати практичні рекомендації щодо їх застосування в проектах, де потрібно відображати велику кількість об’єктів у сцені та підтримувати стабільну продуктивність.

4.1 Аналіз результатів дослідження

№	Метод оптимізації	Batch Count	Avg FPS	Avg CPU Frame Time, мс	Avg GPU Frame Time, мс	Tris Count
1	Basic	100 011	37,3	26,870	16,294	76.8M
2	Static Batching	1 259	51,6	19,414	16,500	76.8M
3	Occlusion Culling	788	79,8	12,551	11,157	46M
4	GPU Instancing Indirect	15	151,4	6,609	6,587	467
5	LOD	1 189	53,3	18,765	6,724	9.1M
6	Light Material	100 011	41,8	23,937	14,233	76.8M
7	Light Mesh	100 011	43,5	22,989	10,054	8M

Таблиця 4.1 – результати дослідження

Базовим варіантом для порівняння є сценарій Basic, який відображає стан сцени без спеціальних засобів оптимізації. Інші сценарії демонструють вплив окремих методів на основні метрики продуктивності.

Аналіз виконується в порядку зростання середньої частоти кадрів, виходячи з базового сценарію як контрольної точки.

4.1. Базовий сценарій (Basic)

У базовому варіанті спостерігається порівняно невисока продуктивність (Avg FPS ≈ 37) за наявності дуже великої кількості батчів (понад 100 000). Середній час кадру на CPU (≈ 26,8 мс) істотно перевищує час кадру на GPU (≈ 16,2 мс), що свідчить про домінування CPU-складової та значні витрати на підготовку рендеру.

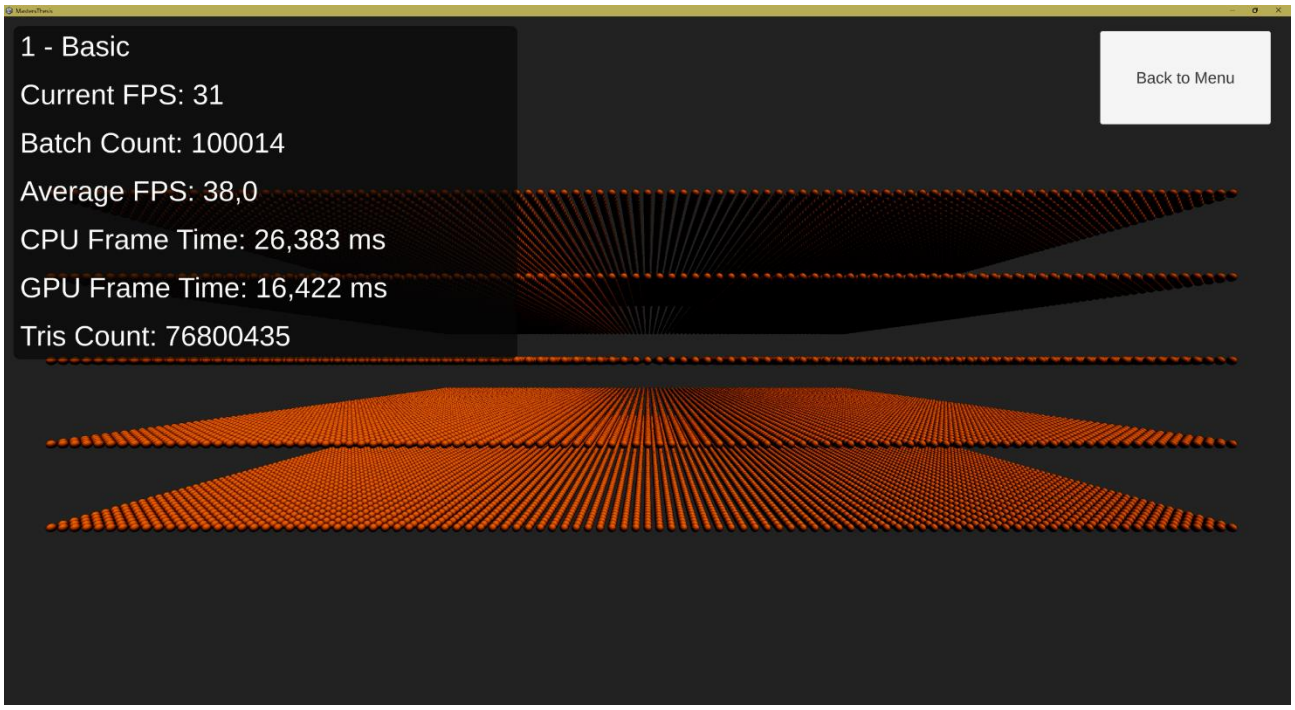


Рисунок 4.1 – експеримент базового сценарію №1

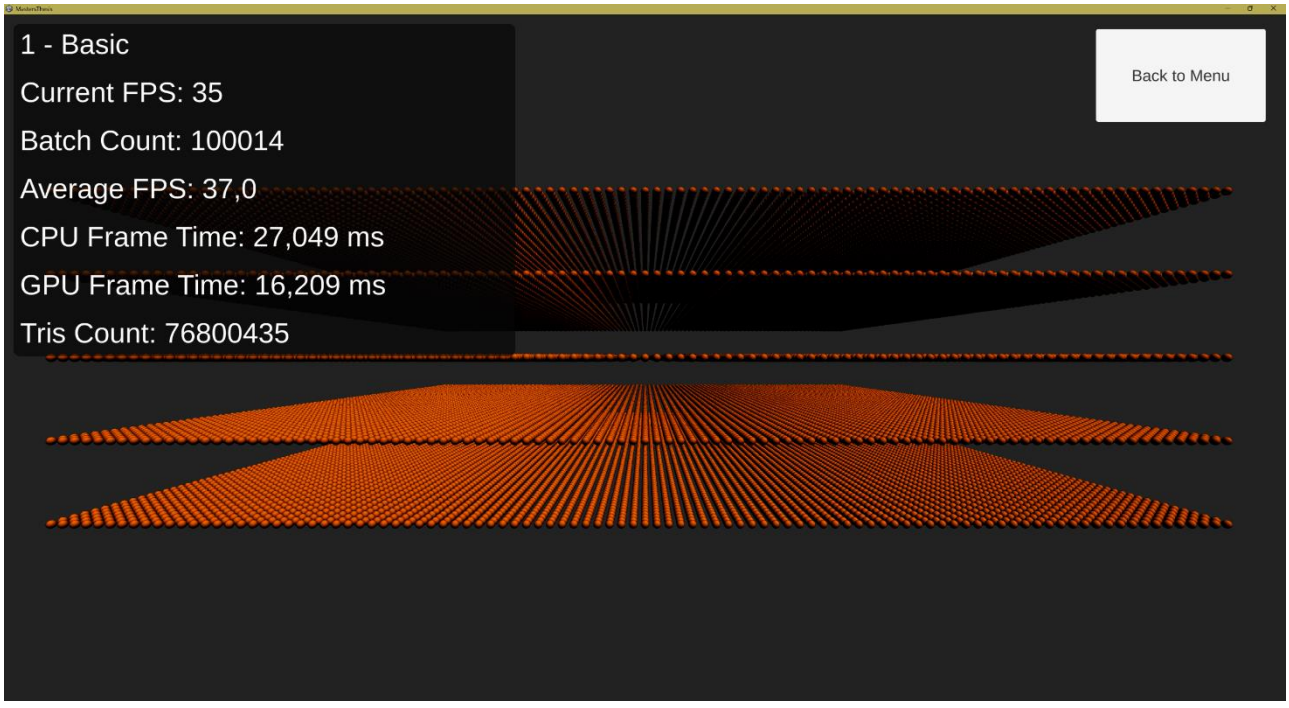


Рисунок 4.2 – експеримент базового сценарію №2

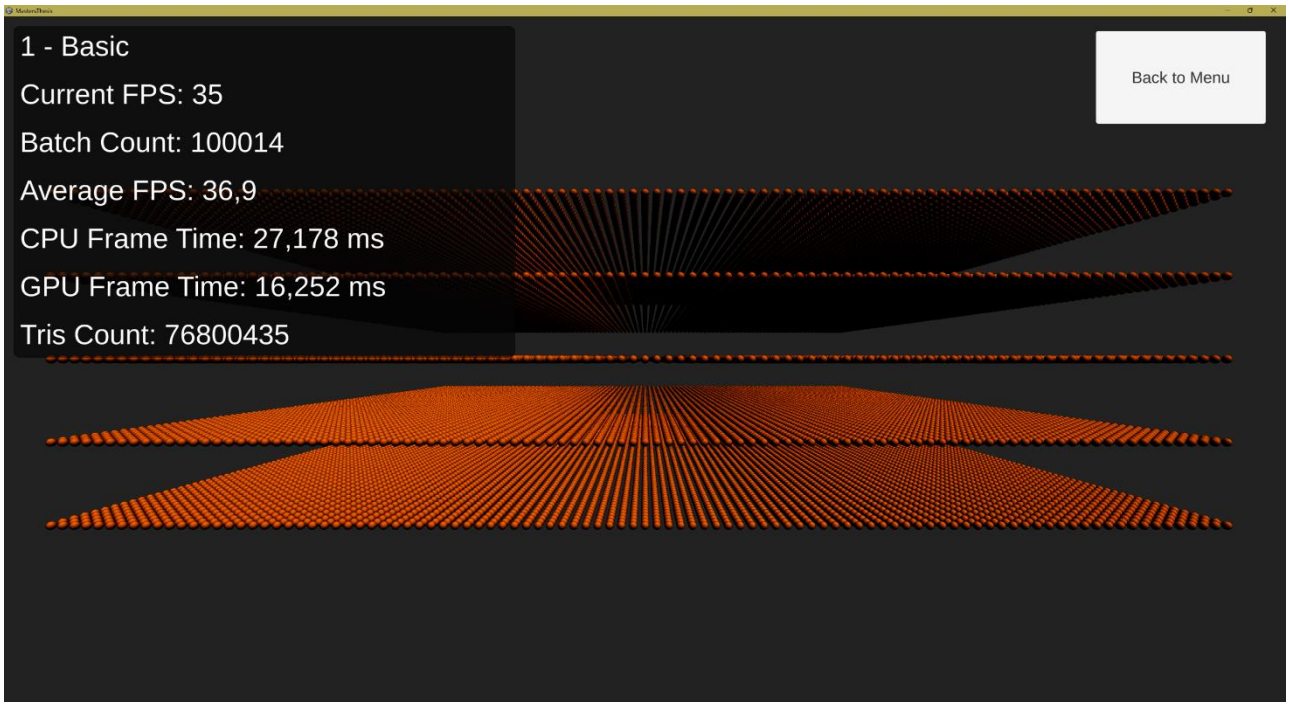


Рисунок 4.3 – експеримент базового сценарію №3

4.2. Спрощена модель (Light Mesh)

Спрощення геометрії кулі позитивно впливає на навантаження графічного процесора: середній GPU-час зменшується приблизно з 16 мс до 10 мс. Це означає, що обсяг роботи GPU скорочується орієнтовно на третину, що є хорошим результатом з погляду геометричної оптимізації.

Водночас FPS та CPU-час змінюються незначно (Avg FPS зростає з 37,3 до 43,5; CPU-час знижується лише до ≈ 23 мс). Це пов’язано з тим, що сцена й надалі залишається орієнтованою на CPU. Основний час витрачається на організацію рендерінгу великої кількості окремих об’єктів, а не на власне обробку полігонів на GPU.

Метод оптимізації	Batch Count	Avg FPS	Avg CPU Frame Time, мс	Avg GPU Frame Time, мс	Tris Count
Basic	100 011	37,3	26,870	16,294	76.8M
Light Mesh	100 011	43,5	22,989	10,054	8M

Таблиця 4.2 – порівняння базового методу з Light Mesh

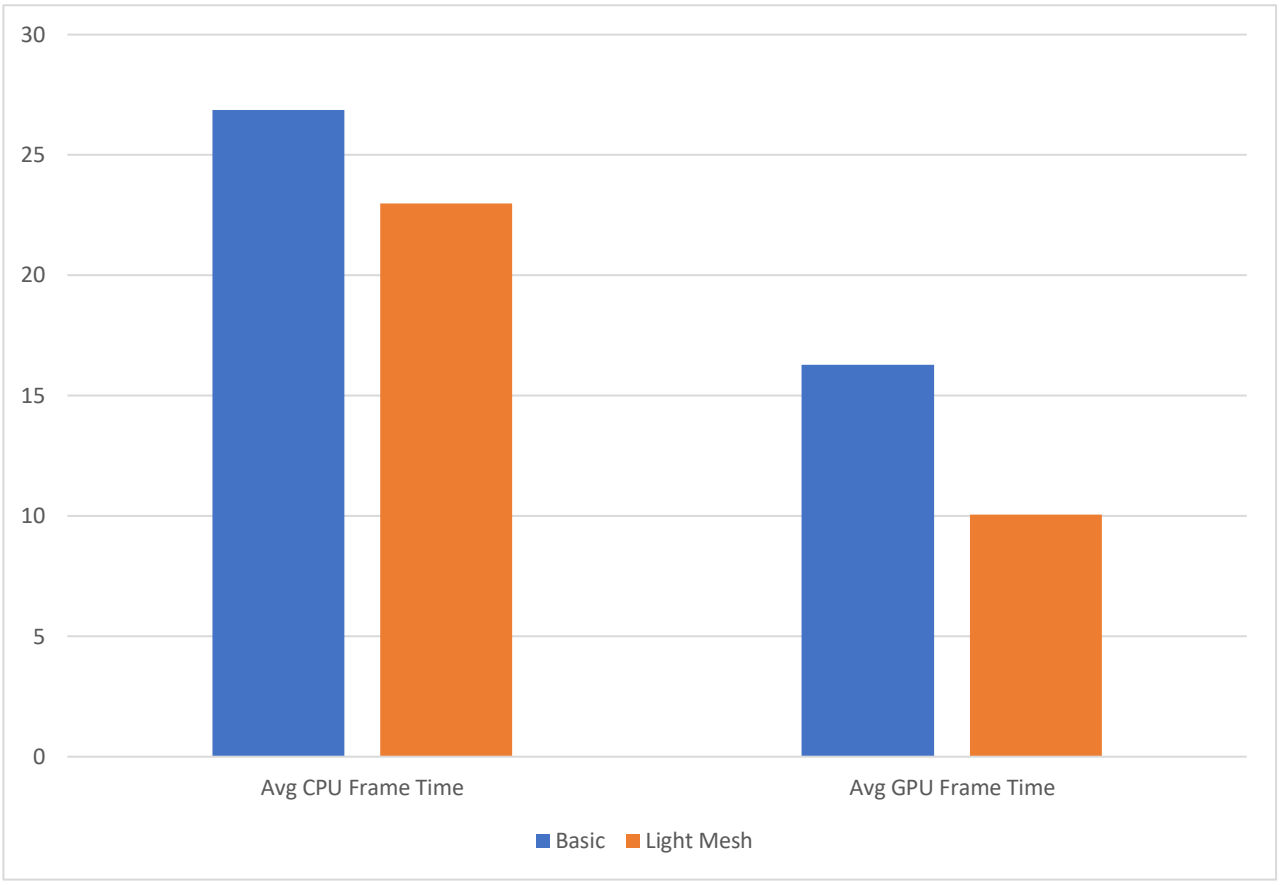


Рисунок 4.4 – діаграма порівняння Basic та Light Mesh сценаріїв

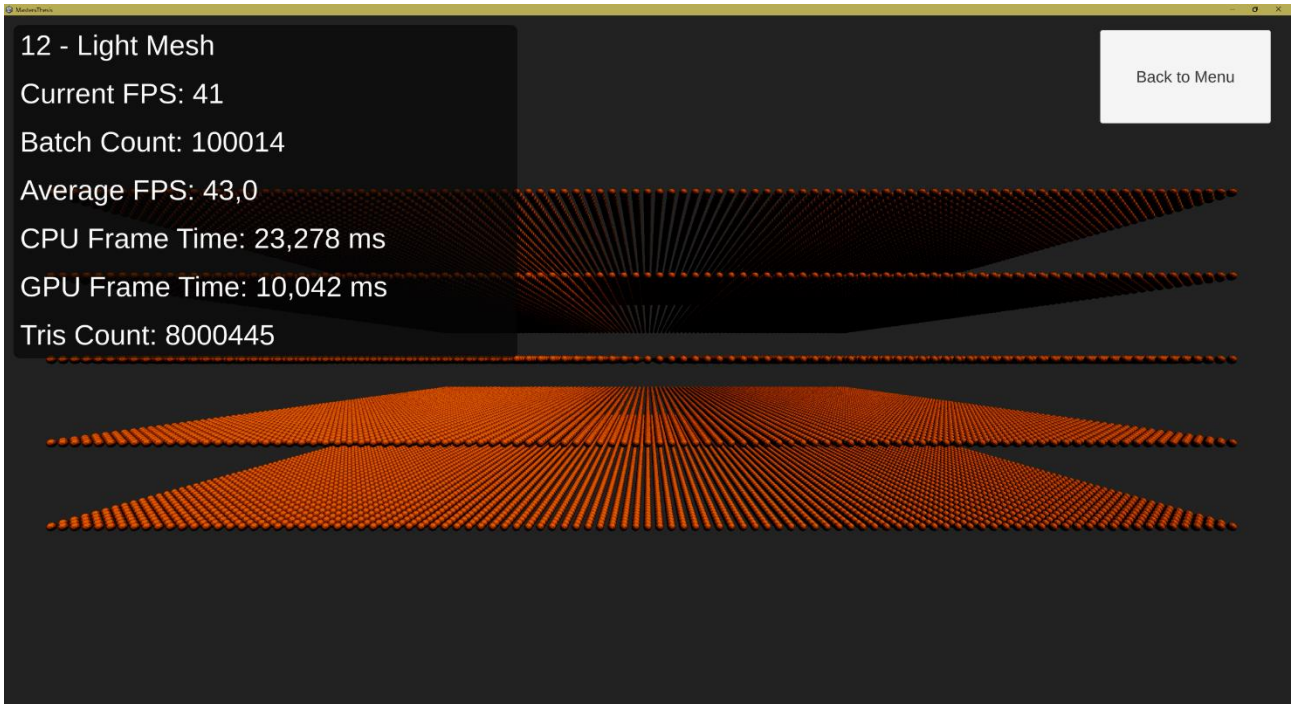


Рисунок 4.5 – експеримент сценарію з спрощеною моделлю №1

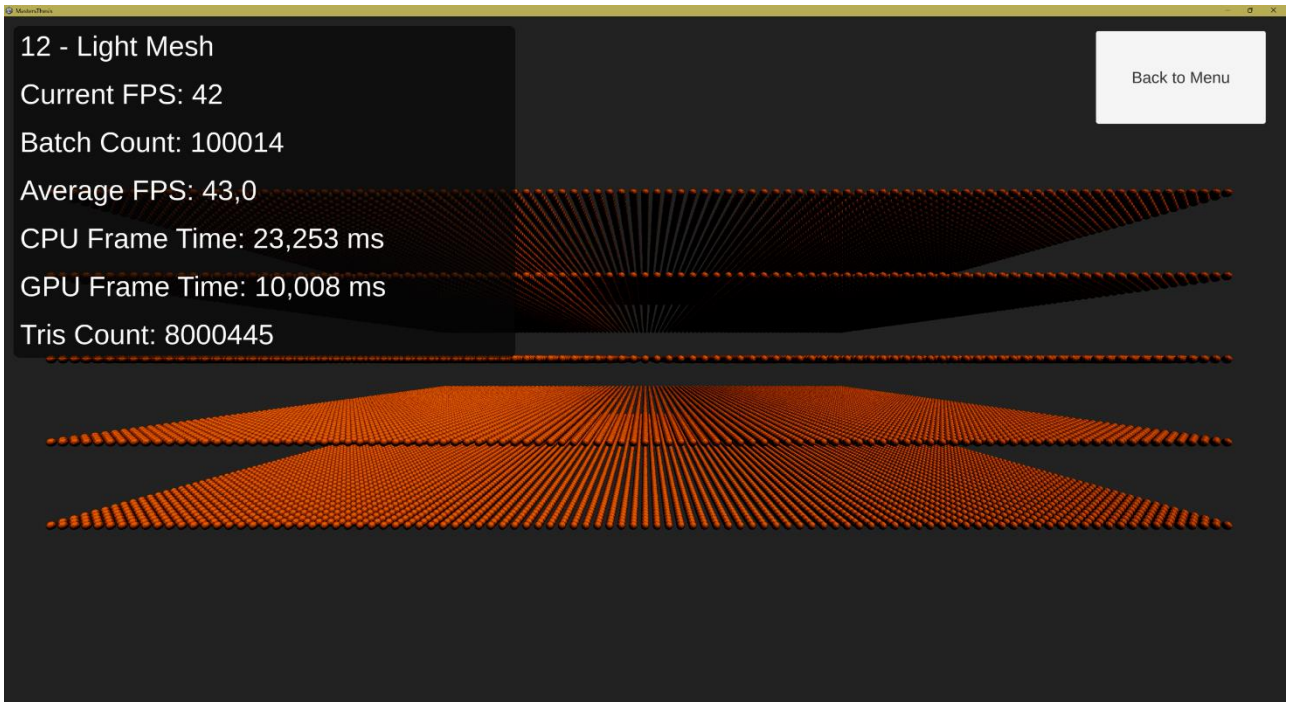


Рисунок 4.6 – експеримент сценарію з спрощеною моделлю №2

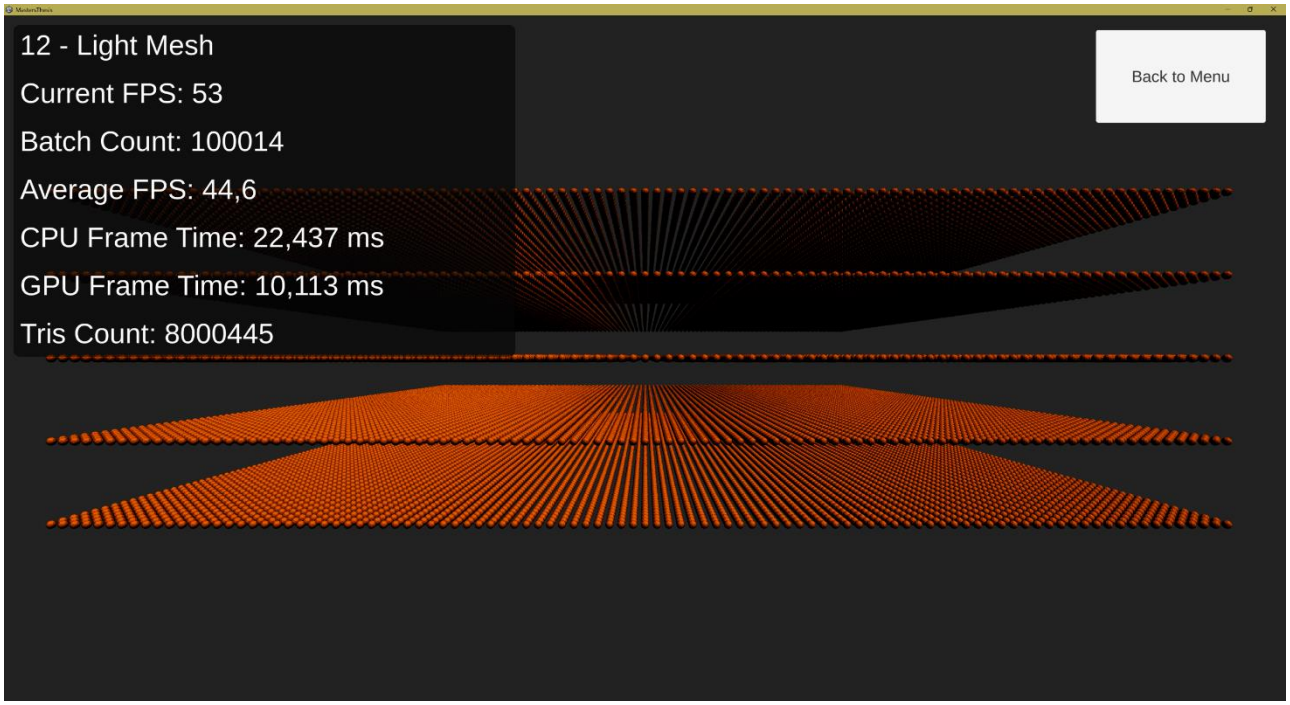


Рисунок 4.7 – експеримент сценарію з спрощеною моделлю №3

4.3. Спрощений матеріал (Light Material)

Перехід до спрощеного шейдера матеріалу дає невелике покращення як для GPU, так і для CPU. GPU-час зменшується з ≈ 16,3 мс до ≈ 14,23 мс, а CPU-час до ≈ 24 мс. Середній FPS зростає до 41,8 кадр/с.

Хоча приріст FPS не є радикальним, спостерігається загальне зниження обчислювальної вартості кожного кадру. Спрощення матеріалу зменшує кількість операцій у фрагментному шейдері, а також частково розвантажує CPU завдяки менш складній підготовці даних для рендерінгу. При цьому кількість батчів залишається високою, тому потенціал покращення продуктивності за рахунок цього методу обмежується саме CPU-вузьким місцем.

Метод оптимізації	Batch Count	Avg FPS	Avg CPU Frame Time, мс	Avg GPU Frame Time, мс	Tris Count
Basic	100 011	37,3	26,870	16,294	76.8M
Light Material	100 011	41,8	23,937	14,233	76.8M

Таблиця 4.3 – порівняння базового методу з Light Material

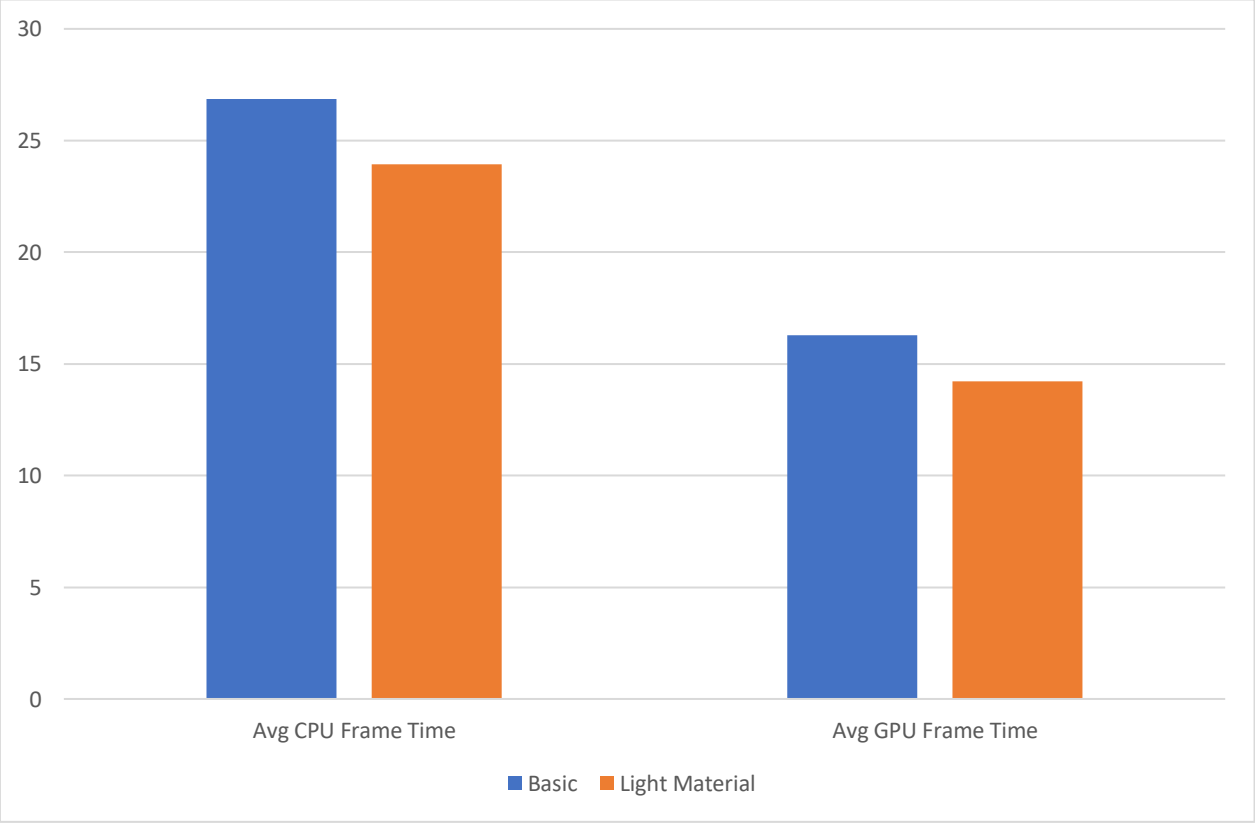


Рисунок 4.8 – діаграма порівняння Basic та Light Material сценаріїв

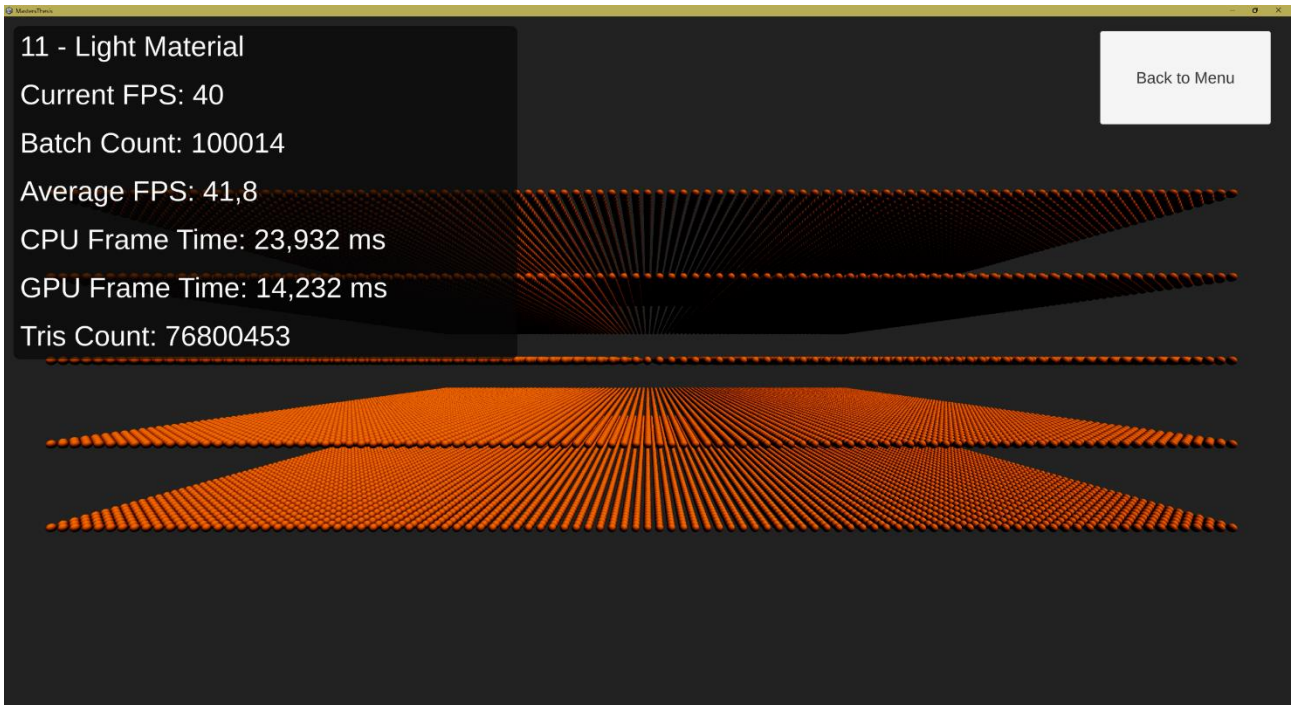


Рисунок 4.9 – експеримент сценарію з спрощеним матеріалом №1

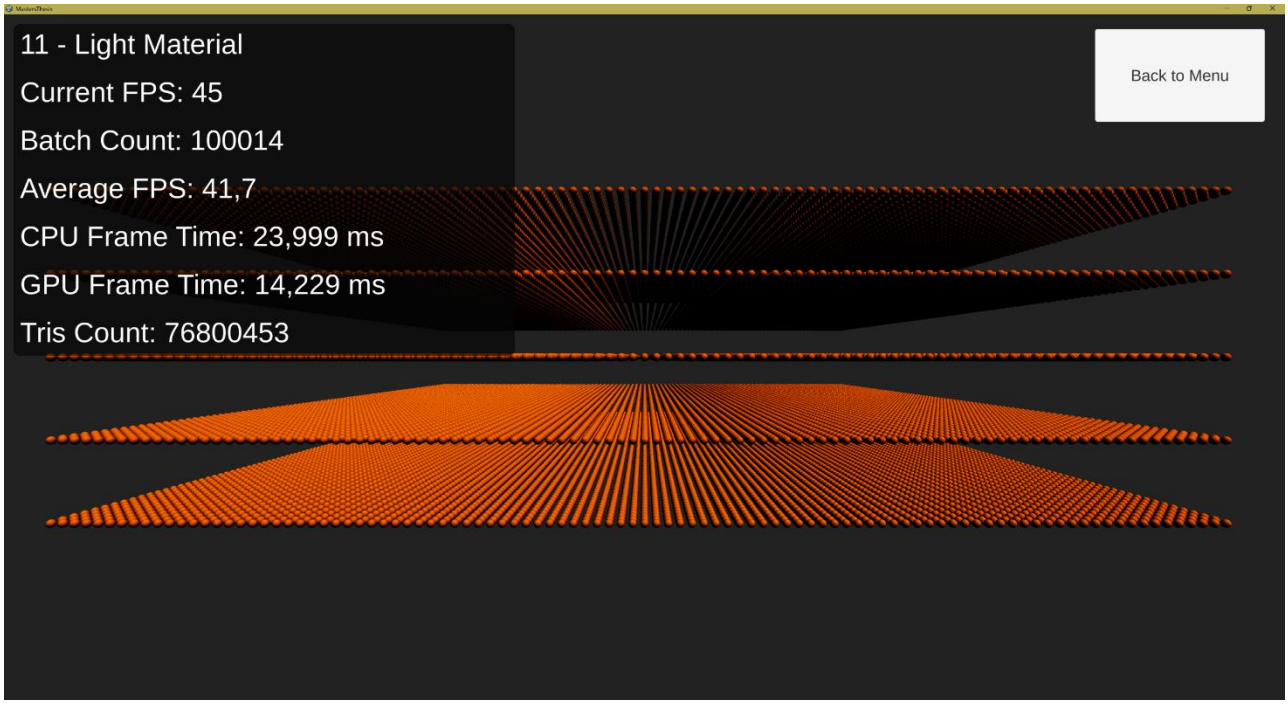


Рисунок 4.10 – експеримент сценарію з спрощеним матеріалом №2

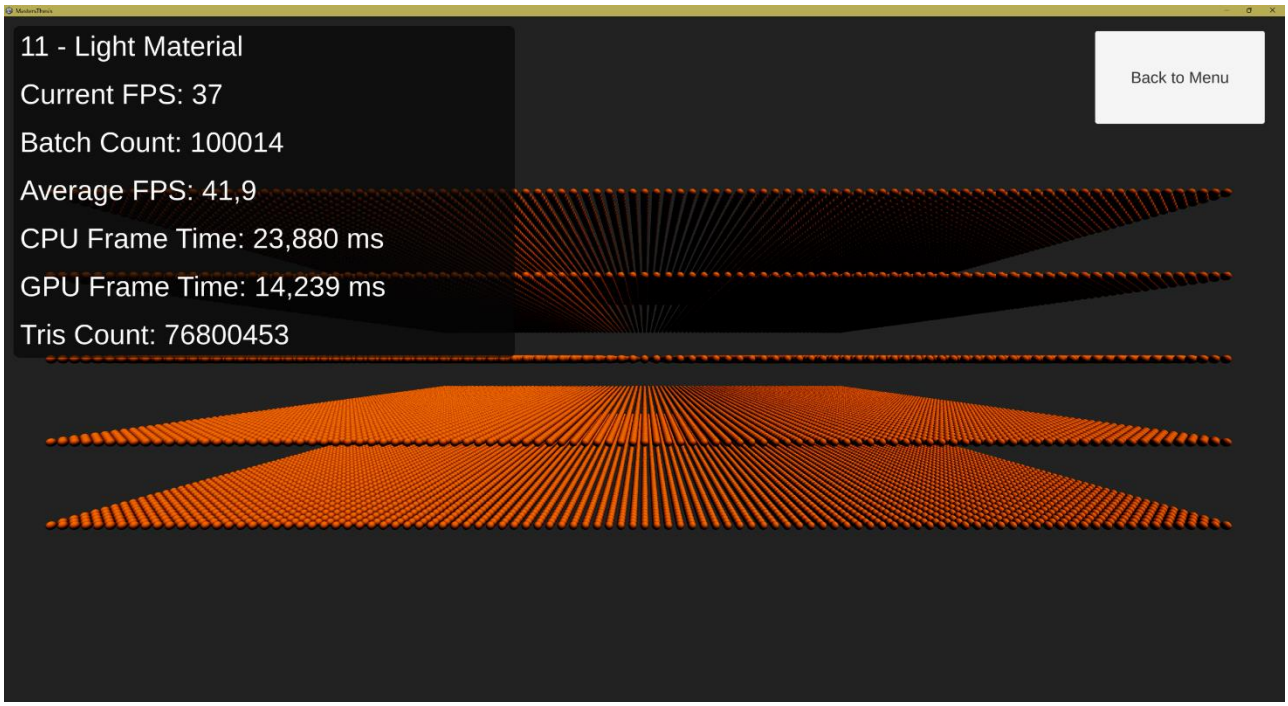


Рисунок 4.11 – експеримент сценарію з спрощеним матеріалом №3

4.4. Static Batching

Застосування статичного батчингу демонструє значно відчутніший ефект. Кількість батчів зменшується з ~100 000 до ~1 250, що різко скорочує кількість окремих draw call'ів. Це безпосередньо відображається на CPU-часі, який зменшується до ≈ 19,41 мс, а середній FPS зростає до 51,6 кадр/с.

GPU-час у цьому сценарії дещо збільшується (до ≈ 16,5 мс). Це можна пояснити тим, що великі збатчені об'єкти рідше відсікаються та займають більшу площу кадру, що збільшує реальний обсяг рендерінгу. Однак загальний результат є позитивним: за рахунок істотного зменшення кількості батчів сцена стає помітно ефективнішою з точки зору CPU, а приріст FPS підтверджує доцільність використання статичного батчингу для великих масивів статичних об'єктів.

Метод оптимізації	Batch Count	Avg FPS	Avg CPU Frame Time, мс	Avg GPU Frame Time, мс	Tris Count
Basic	100 011	37,3	26,870	16,294	76.8M
Static Batching	1 259	51,6	19,414	16,500	76.8M

Таблиця 4.4 – порівняння базового методу з Static Batching

Рисунок 4.12 – діаграма порівняння Basic та Static Batching сценаріїв

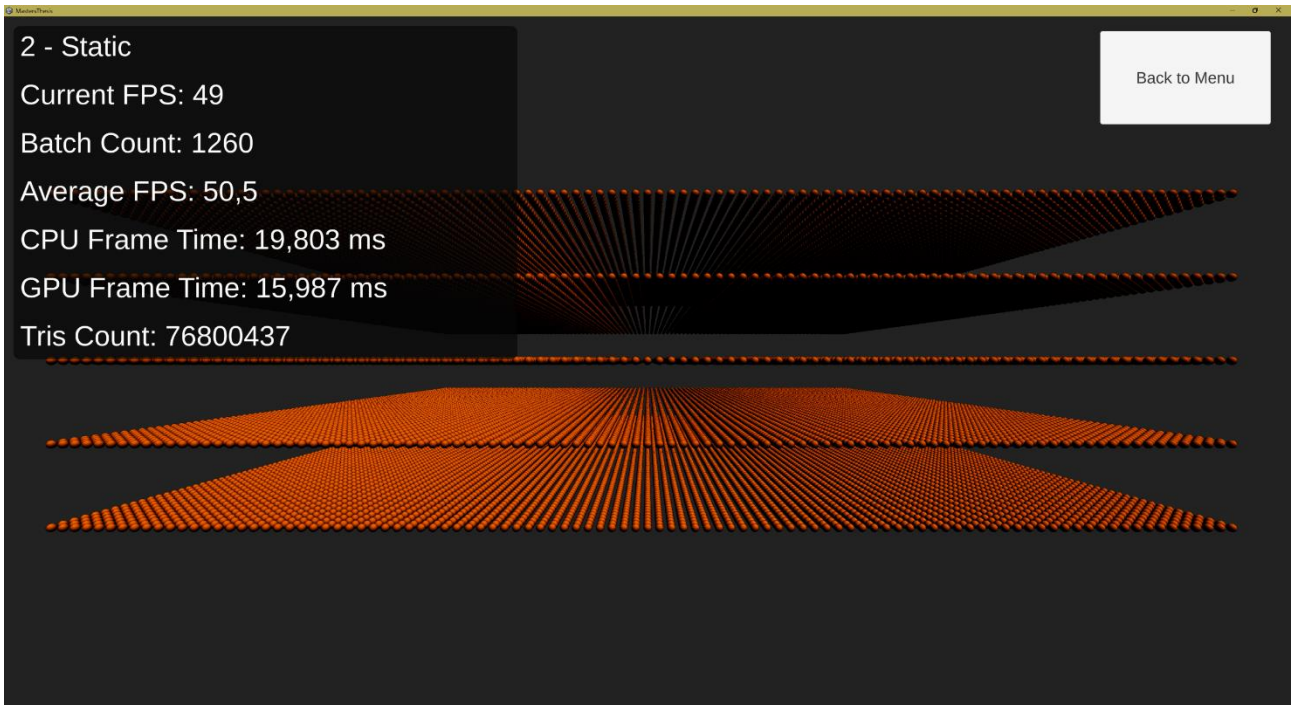


Рисунок 4.13 – експеримент сценарію з статичним батчингом №1

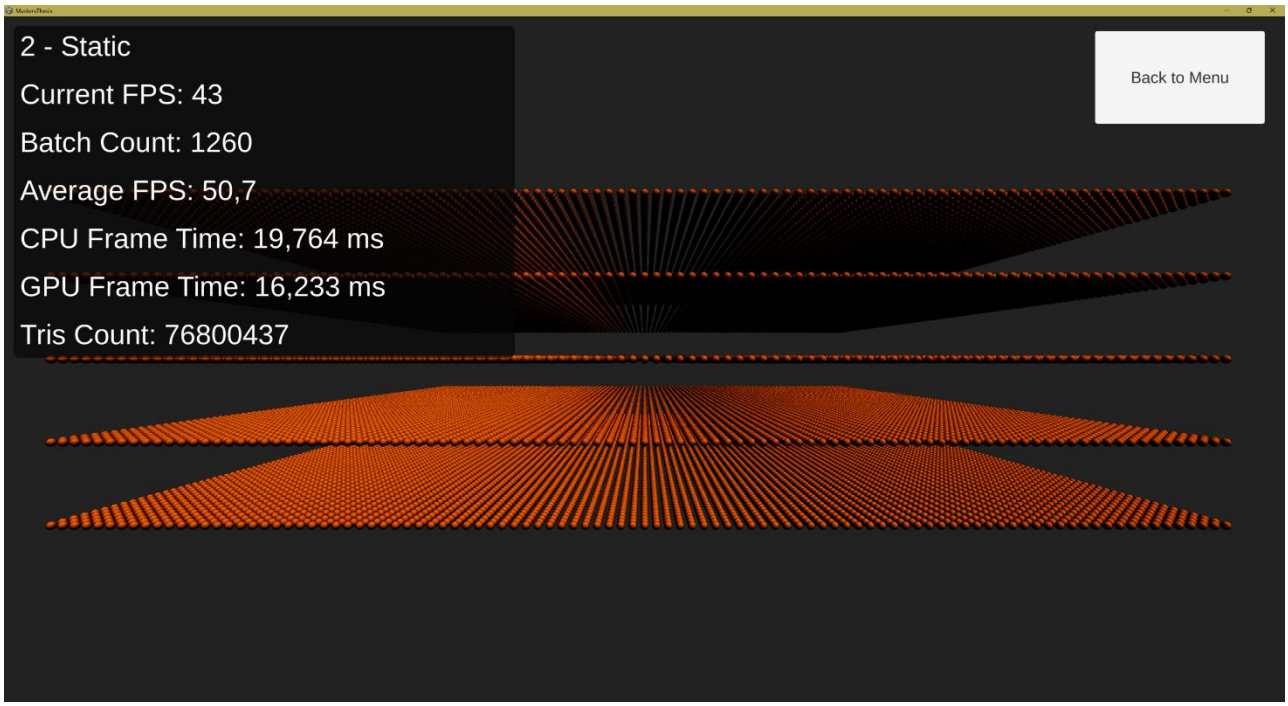


Рисунок 4.14 – експеримент сценарію з статичним батчингом №2

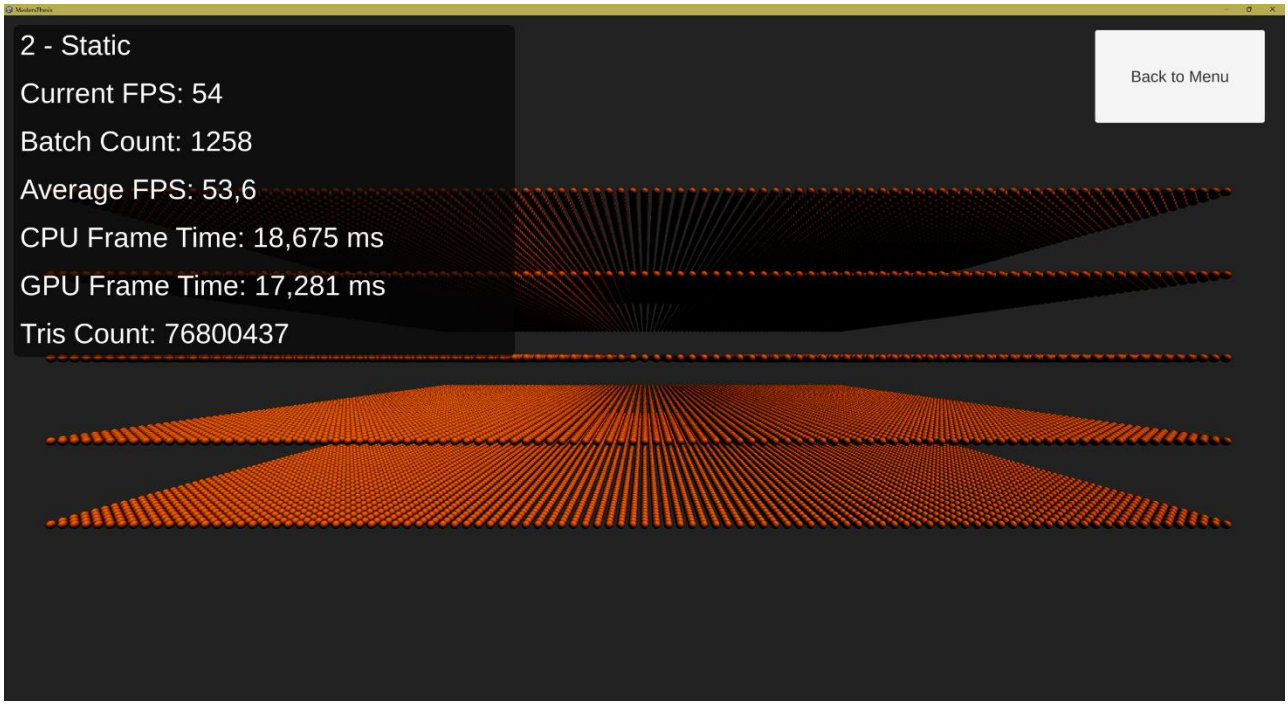


Рисунок 4.15 – експеримент сценарію з статичним батчингом №2

4.5. Рівні деталізації (LOD)

Застосування LOD призводить до подальшого покращення продуктивності. Середній FPS зростає до 53,3 кадр/с, Batch Count знижується до ~1 200, а GPU-час скорочується до ≈ 6,724 мс, тобто більш ніж удвічі порівняно з базовим сценарієм.

Значне зменшення GPU-часу свідчить про те, що механізм LOD ефективно зменшує кількість полігонів, які опрацьовує графічний процесор, CPU-час також покращується (≈ 18,765 мс), але не настільки суттєво, як GPU-час, тому сцена залишається переважно CPU-bound. У цілому LOD демонструє високу ефективність як метод зниження геометричної складності сцени за збереження прийнятної візуальної якості.

Метод оптимізації	Batch Count	Avg FPS	Avg CPU Frame Time, мс	Avg GPU Frame Time, мс	Tris Count
Basic	100 011	37,3	26,870	16,294	76.8M
LOD	1 189	53,3	18,765	6,724	9.1M

Таблиця 4.5 – порівняння базового методу з LOD

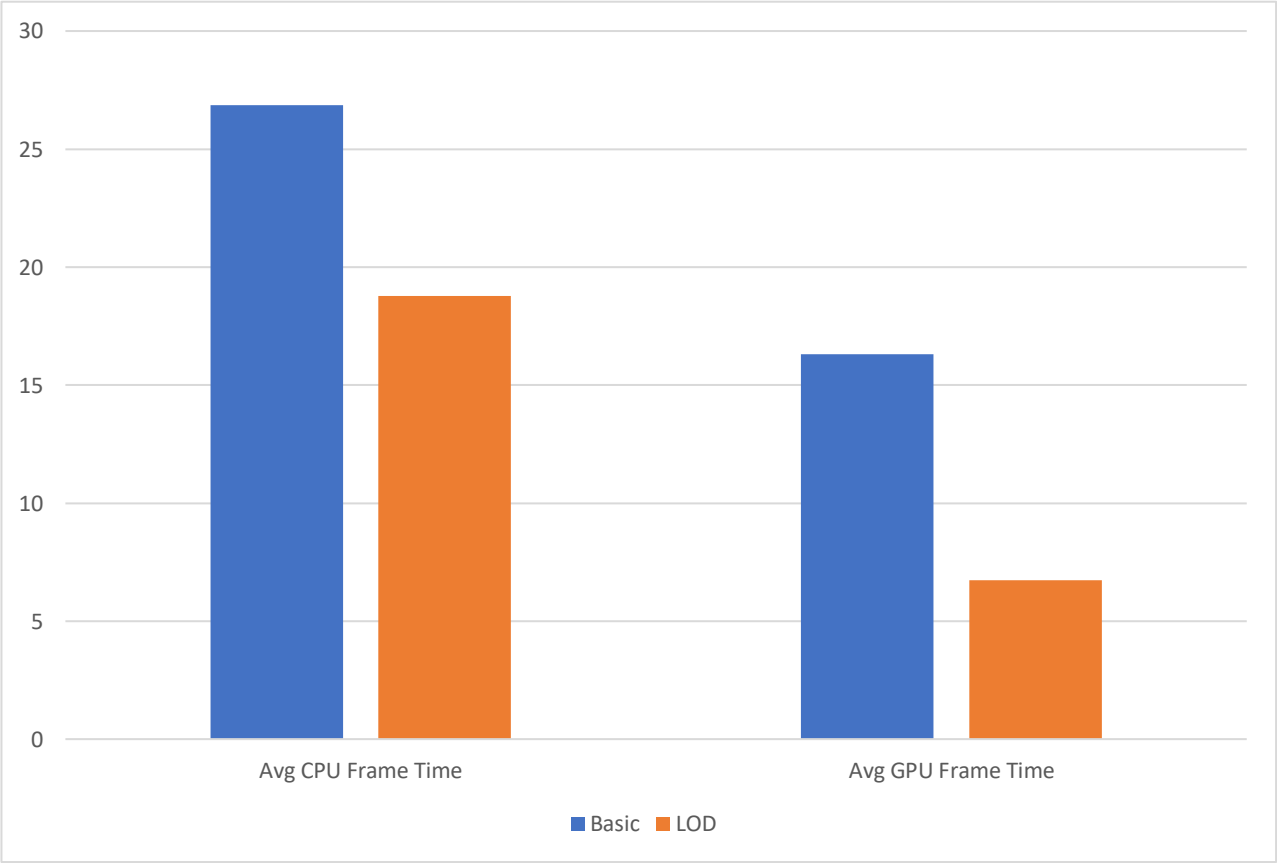


Рисунок 4.16 – діаграма порівняння Basic та LOD сценаріїв

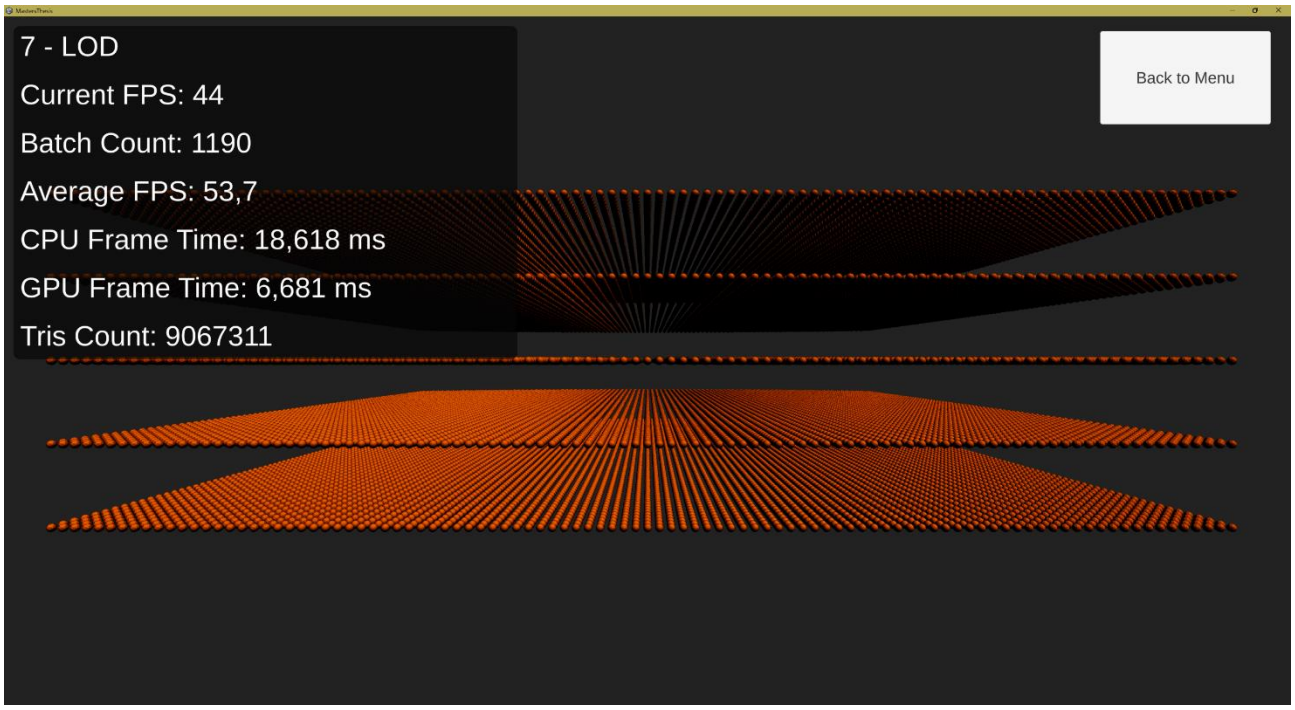


Рисунок 4.17 – експеримент сценарію з рівнями деталізації №1

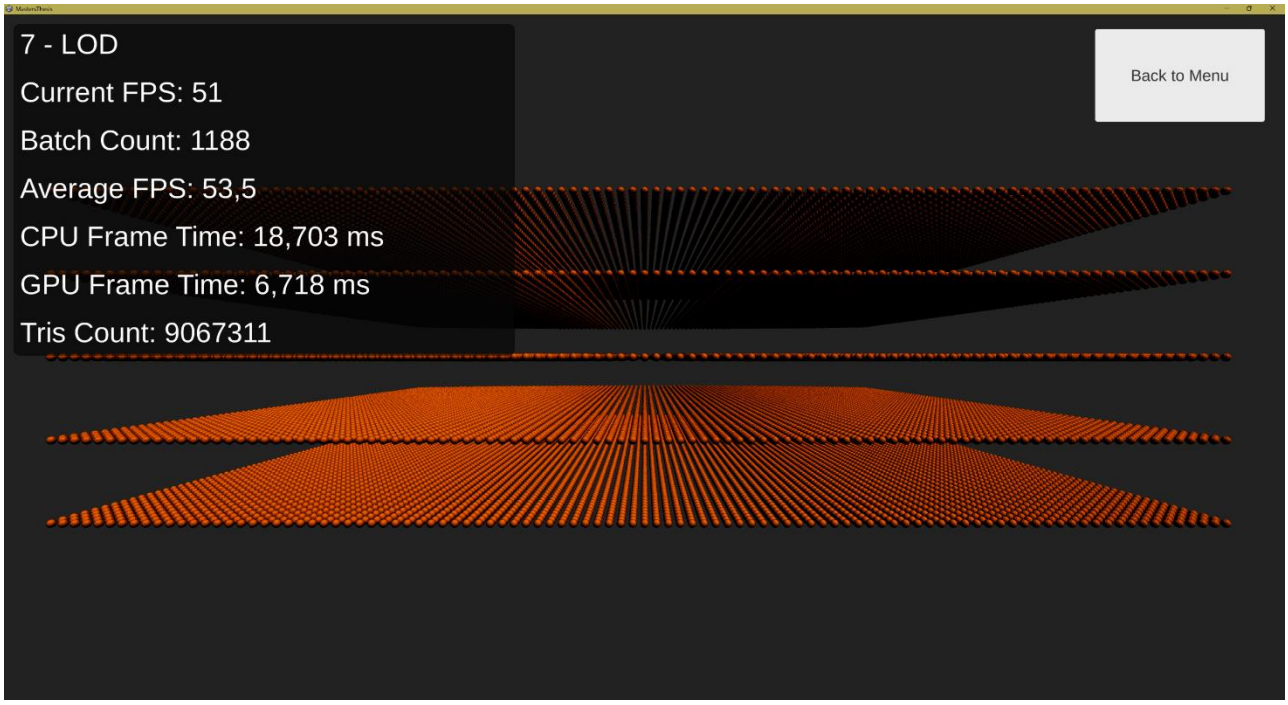


Рисунок 4.18 – експеримент сценарію з рівнями деталізації №2

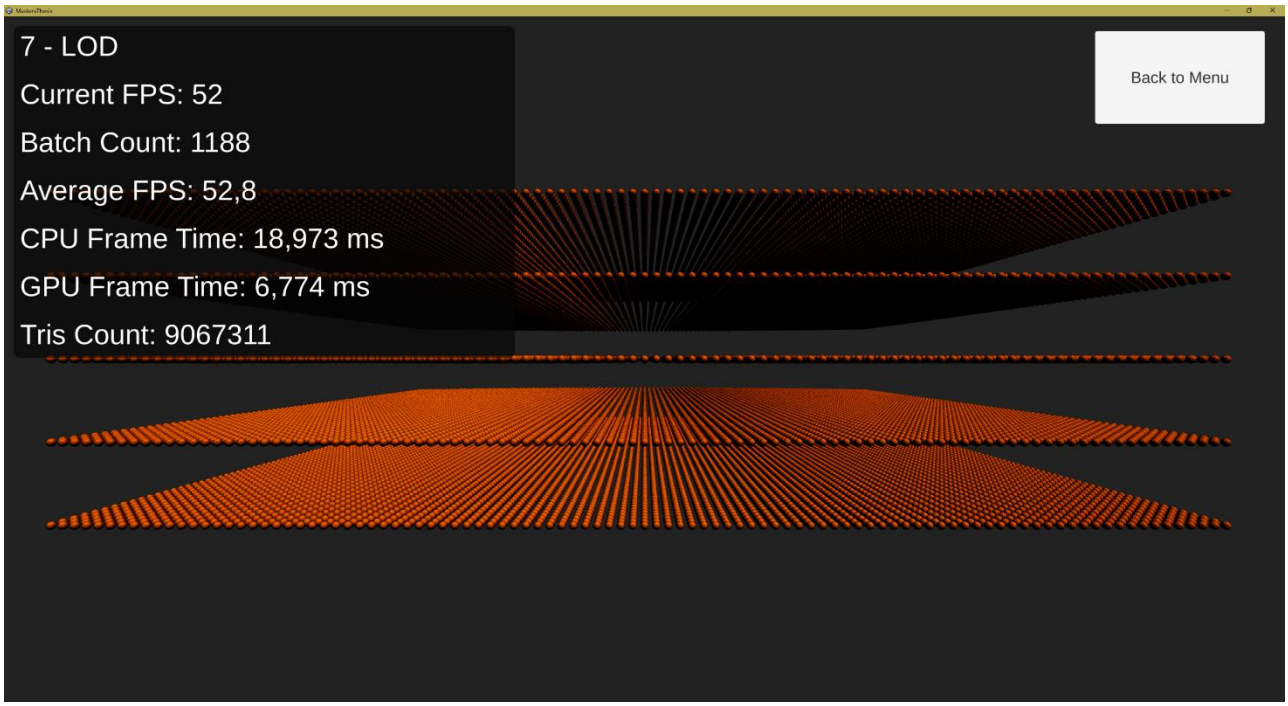


Рисунок 4.19 – експеримент сценарію з рівнями деталізації №3

4.6. Occlusion Culling

Сценарій з оклюзійним відсіканням (коли камера бачить орієнтовно половину об’єктів) забезпечує значний приріст продуктивності. Середній FPS зростає до 79,8 кадр/с, а Batch Count зменшується до 788. CPU-час скорочується до ≈ 12,551 мс, GPU-час до ≈ 11,157 мс.

Такий результат свідчить, що оклюзійне відсікання ефективно усуває з рендер-пайплайна об’єкти, які повністю закриваються іншими або знаходяться поза видимістю. Відповідно, менша кількість об’єктів потрапляє у батчі, що зменшує навантаження як на CPU (менше логіки підготовки рендеру), так і на GPU (менше полігонів та пікселів для обробки). Цей метод показує себе як один із найефективніших у разі, коли частина сцени не потрапляє в поле зору камери.

Метод оптимізації	Batch Count	Avg FPS	Avg CPU Frame Time, мс	Avg GPU Frame Time, мс	Tris Count
Basic	100 011	37,3	26,870	16,294	76.8M
Occlusion Culling	788	79,8	12,551	11,157	46M

Таблиця 4.6 – порівняння базового методу з Occlusion Culling

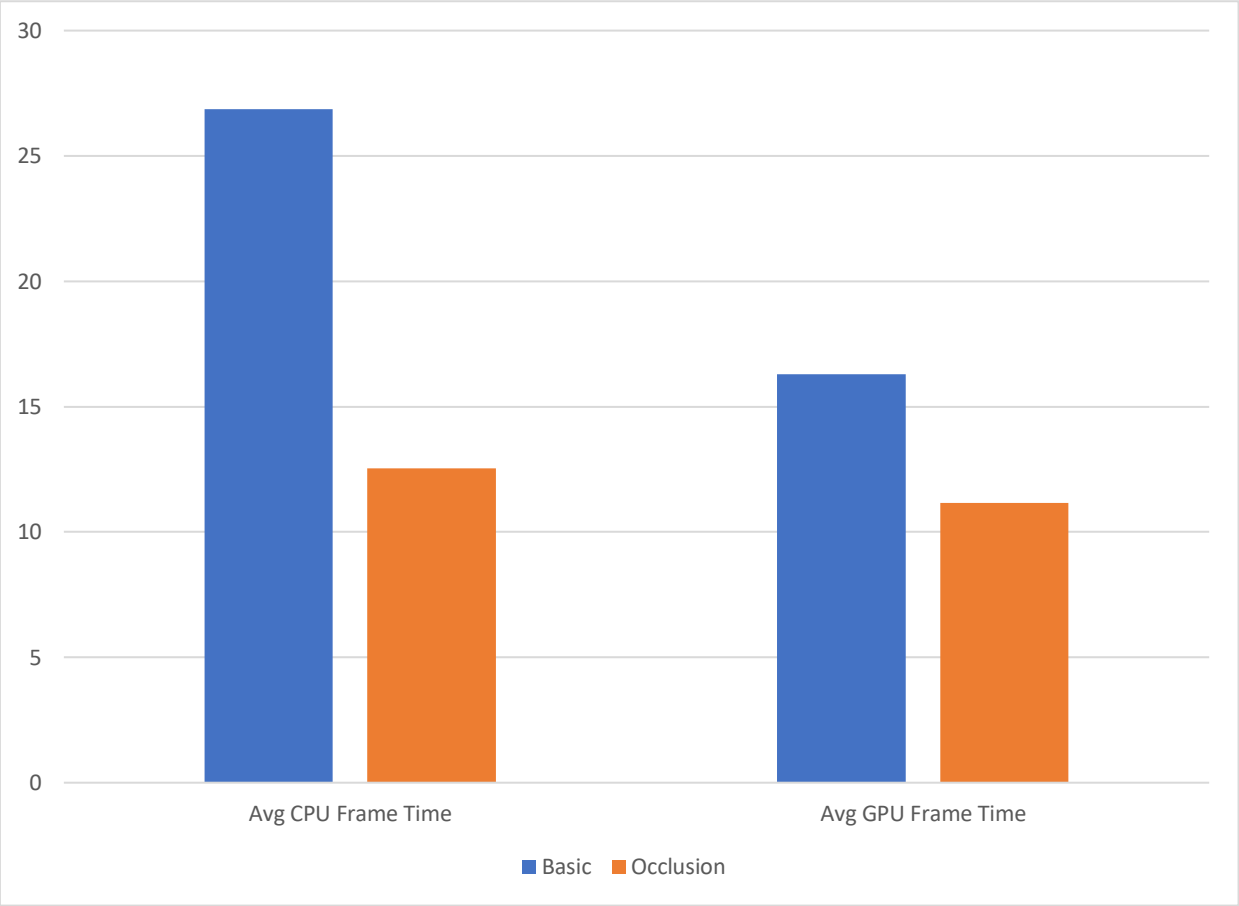


Рисунок 4.20 – діаграма порівняння Basic та Occlusion Culling сценаріїв

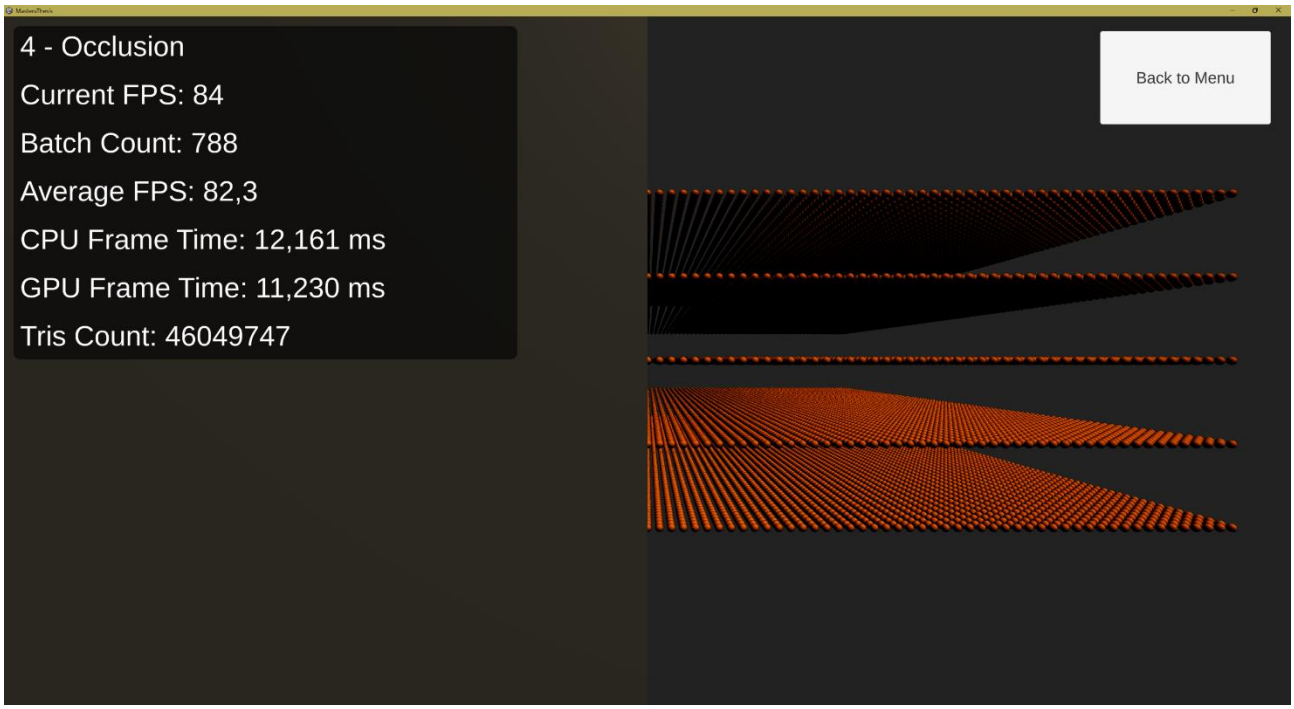


Рисунок 4.21 – експеримент сценарію з оклюзійним відсіканянм №1

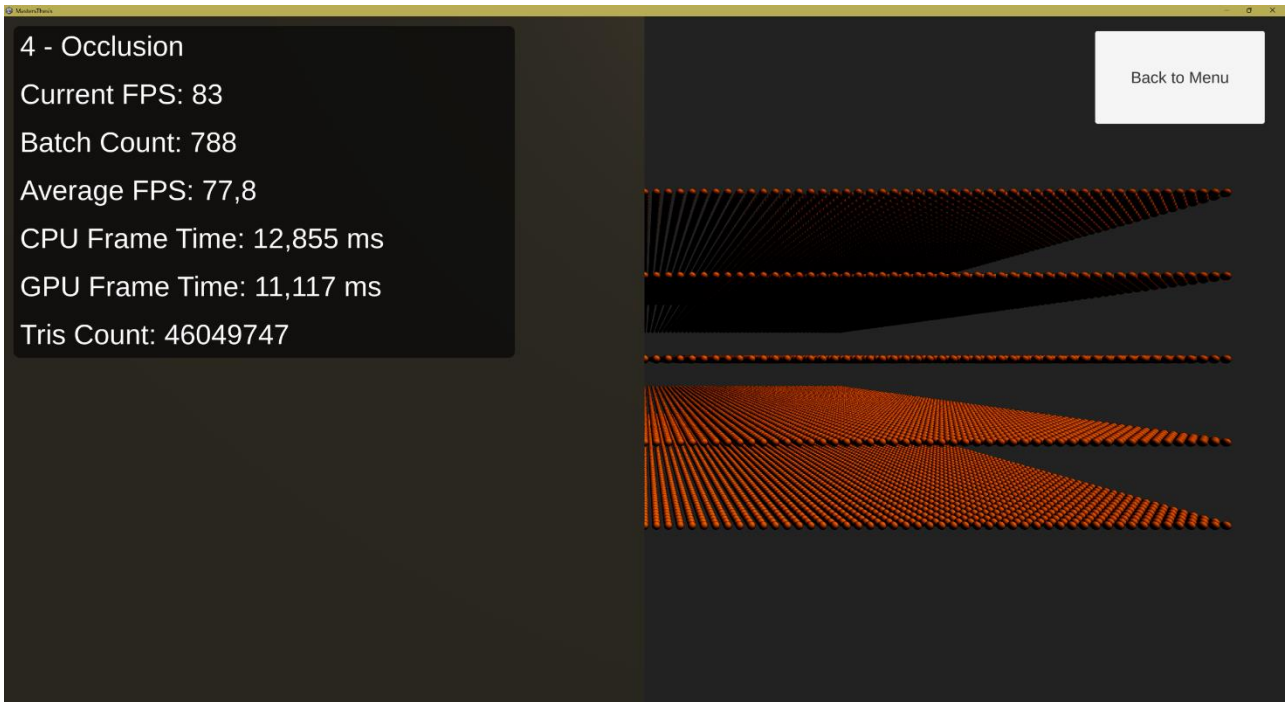


Рисунок 4.22 – експеримент сценарію з оклюзійним відсіканянм №2

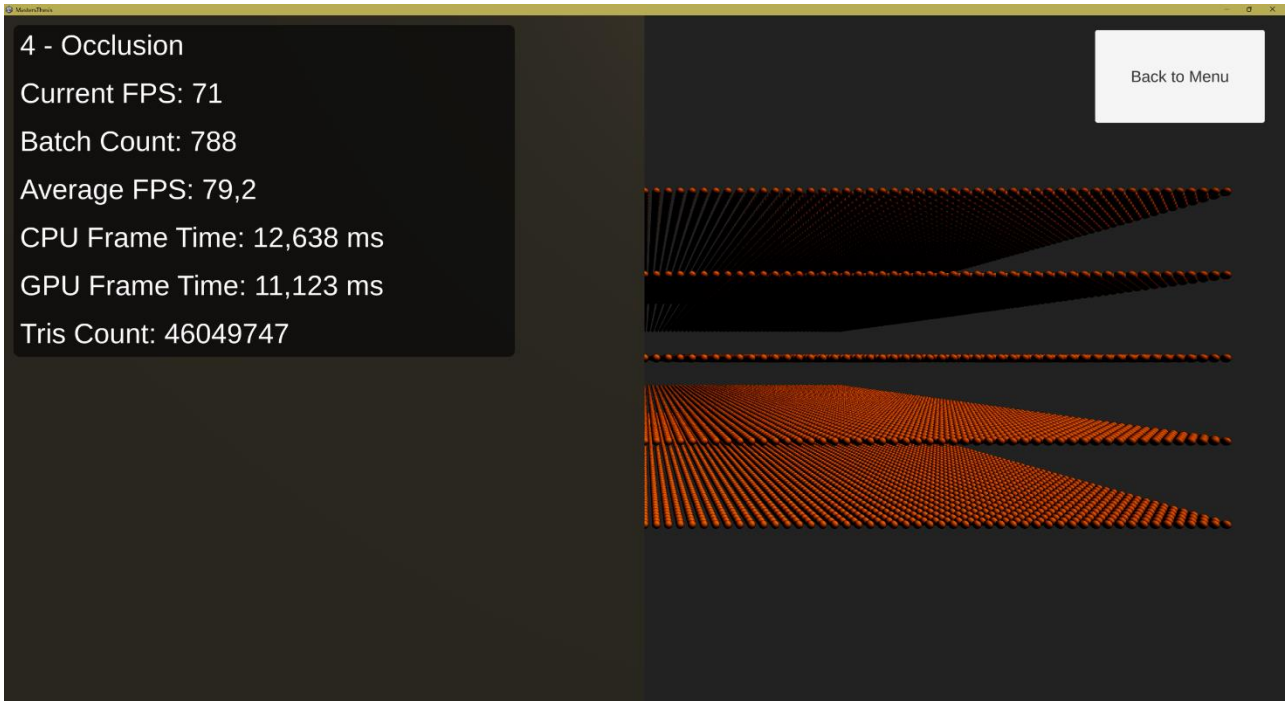


Рисунок 4.23 – експеримент сценарію з оклюзійним відсіканянм №3

4.7. GPU Instancing Indirect

Найкращі показники досягаються при використанні непрямого GPU-інстансингу. У цьому сценарії Batch Count зменшується до 15, що означає максимальну концентрацію рендерінгу в невеликій кількості draw call’ів. Середній FPS досягає 151,4 кадр/с, а CPU- та GPU-часи стають майже однаковими (≈ 6,61 мс та ≈ 6,58 мс відповідно).

Така конфігурація демонструє добре збалансований розподіл навантаження між CPU та GPU. Значна частина роботи з формування списку інстансів переноситься до GPU-керованого пайплайну, що мінімізує накладні витрати на CPU і водночас дозволяє графічному процесору ефективно обробляти велику кількість однотипних об’єктів у рамках небагатьох викликів рендерінгу. Отриманий результат підтверджує високу доцільність використання Indirect-інстансингу для сцен з дуже великою кількістю однакових мешів.

Метод оптимізації	Batch Count	Avg FPS	Avg CPU Frame Time, мс	Avg GPU Frame Time, мс	Tris Count
Basic	100 011	37,3	26,870	16,294	76.8M
GPU Instancing Indirect	15	151,4	6,609	6,587	467

Таблиця 4.7 – порівняння базового методу з GPU Instancing Indirect

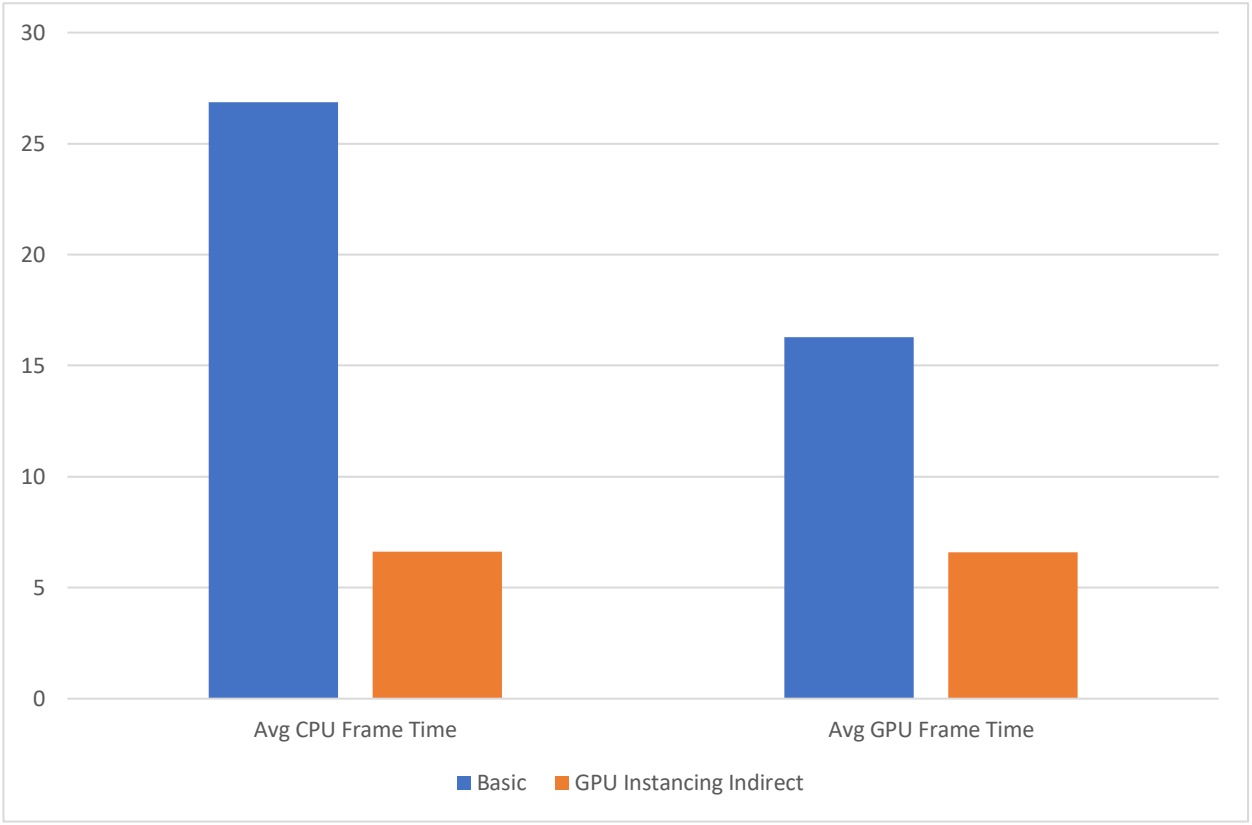


Рисунок 4.24 – діаграма порівняння Basic та GPU Instancing Indirect сценаріїв

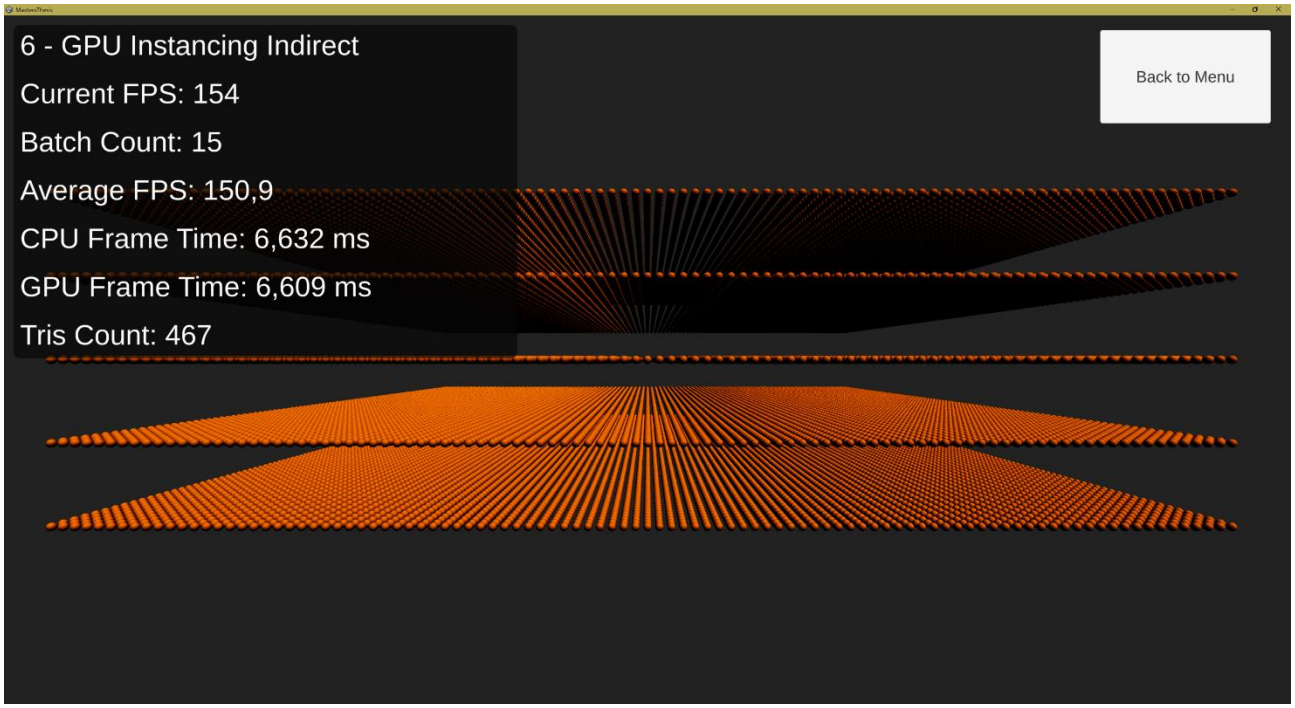


Рисунок 4.25 – експеримент сценарію з GPU Instancing Indirect №1

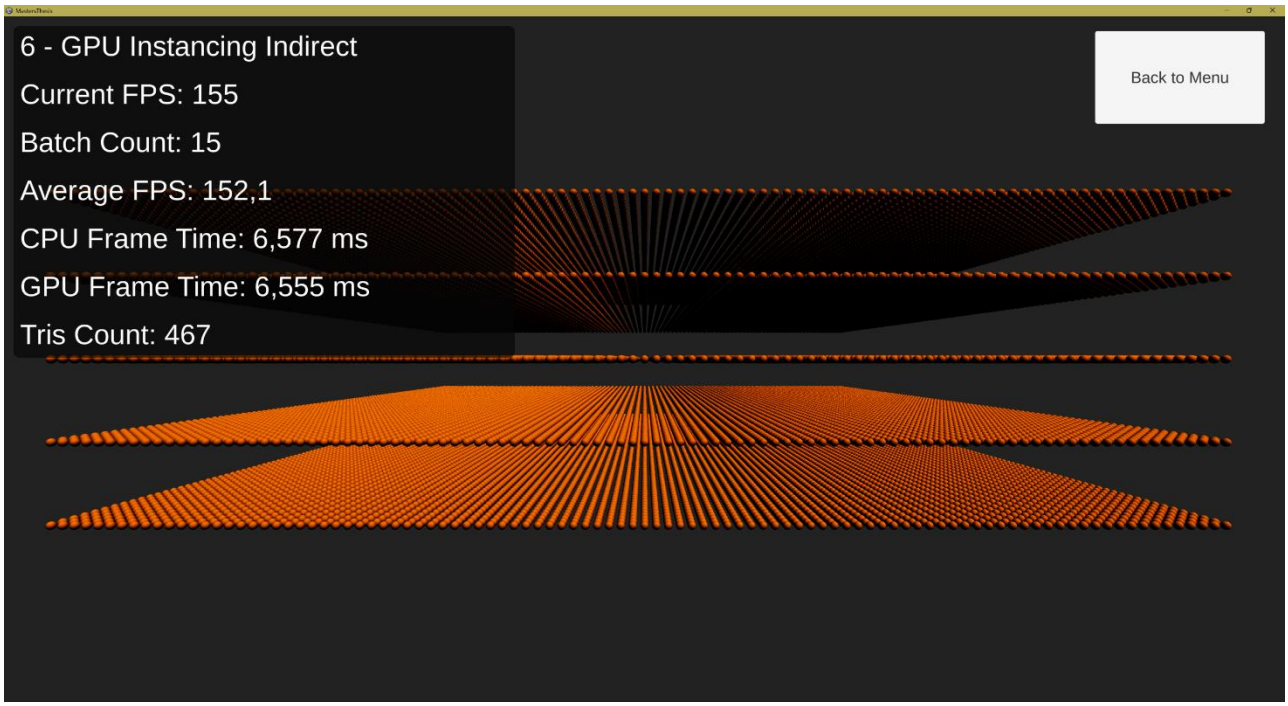


Рисунок 4.26 – експеримент сценарію з GPU Instancing Indirect №2

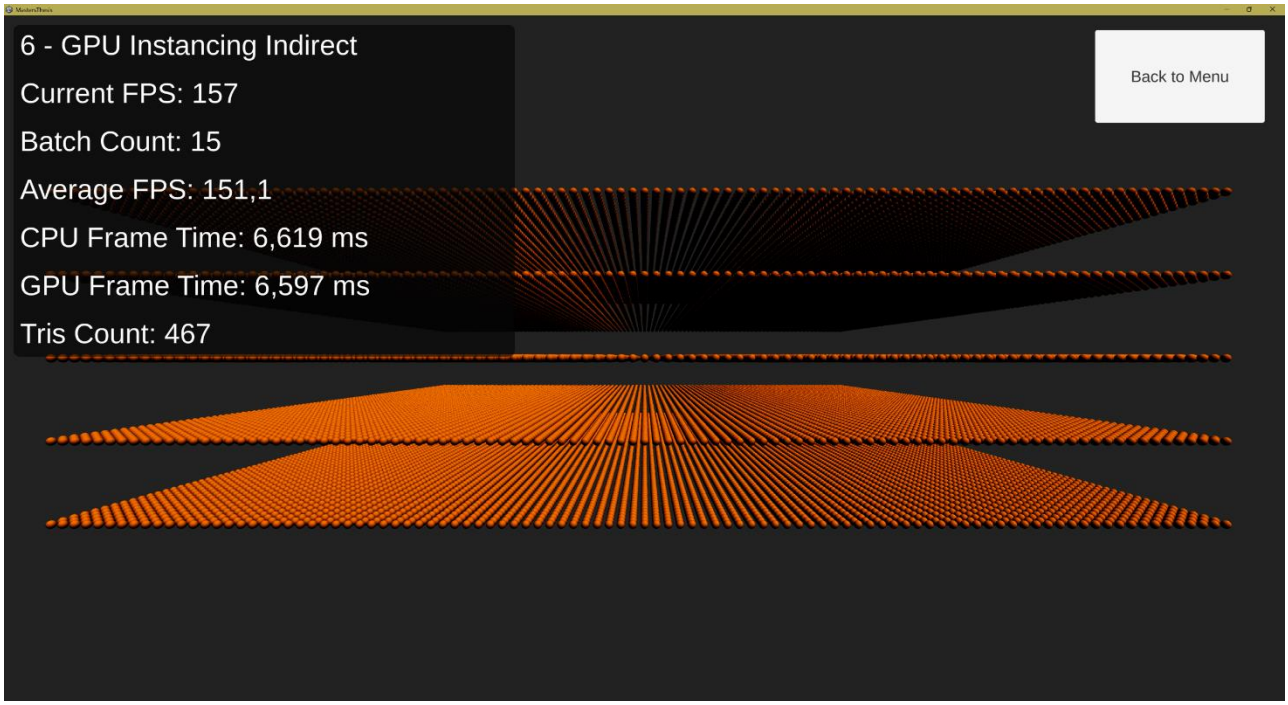


Рисунок 4.27 – експеримент сценарію з GPU Instancing Indirect №3

Висновки за розділом 4

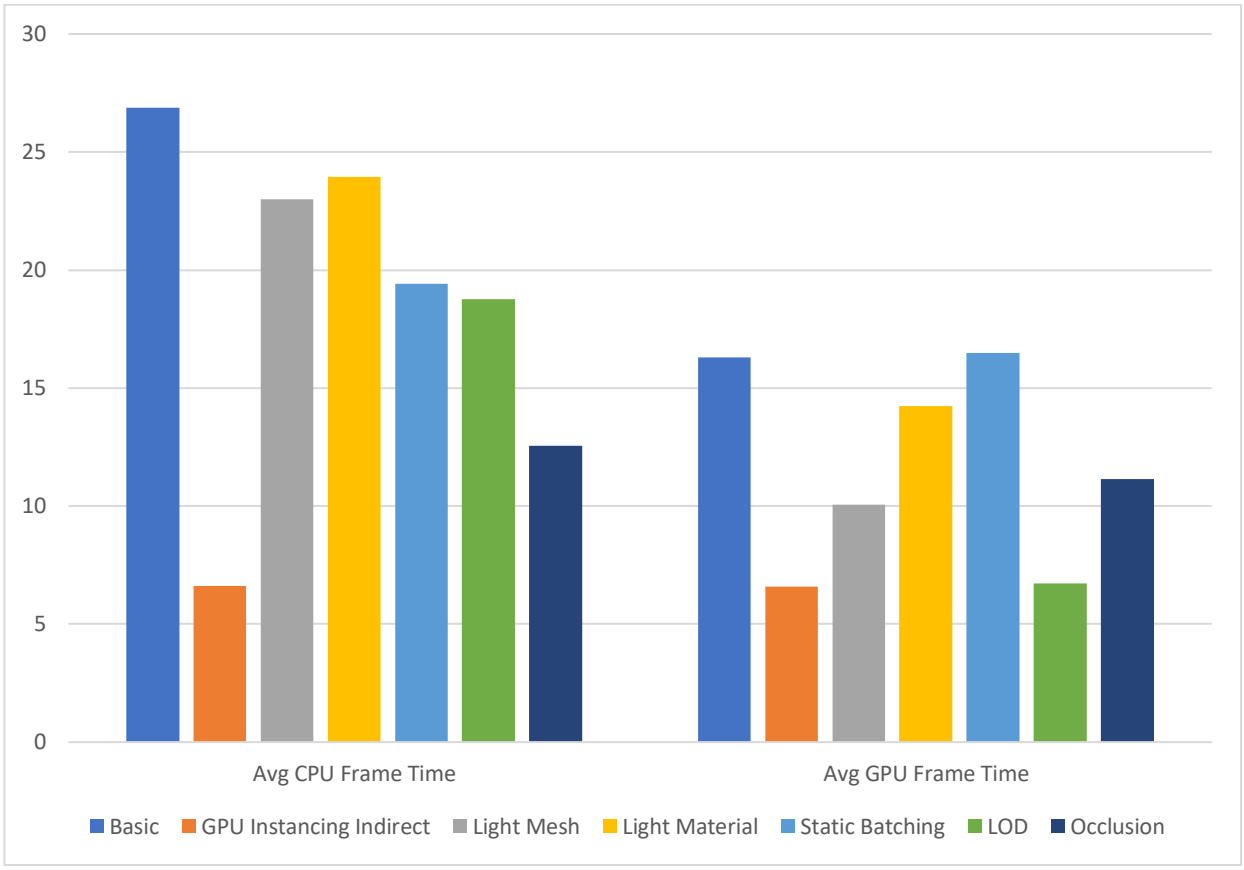


Рисунок 4.28 – діаграма порівняння всіх сценаріїв

У підсумку проведене дослідження показало, що на продуктивність рендерінгу великої кількості однотипних об’єктів у Unity ключовий вплив мають методи, які зменшують кількість викликів рендерінгу та обсяг реально відображуваної сцени. Статичний батчинг, LOD та Occlusion Culling продемонстрували суттєве зниження середнього часу кадру на центральному процесорі та помітне зростання частоти кадрів, тоді як найкращого результату вдалося досягти за рахунок використання GPU Instancing, що практично зрівняло навантаження між CPU та GPU і забезпечило найбільший приріст FPS. Це підтверджує доцільність перенесення частини роботи з формування батчів на GPU та активного використання сучасних можливостей рендер-пайплайна.

Разом з тим експерименти зі спрощенням геометрії та матеріалів показали, що навіть за відносно невеликій зміні FPS вони дають важливий позитивний ефект у вигляді помітного скорочення часу кадру на GPU. У поєднанні з батчингом, LOD та Occlusion Culling такі методи дозволяють балансувати навантаження між

CPU та GPU, зберігаючи прийнятну візуальну якість сцени. Отримані результати можуть бути використані як практичні рекомендації для вибору та комбінування методів оптимізації при розробці проектів, які містять велику кількість однотипних об’єктів і потребують стабільної роботи в режимі реального часу.

ЗАГАЛЬНИЙ ВИСНОВОК

У процесі виконання магістерської роботи було досліджено задачу оптимізації рендерінгу великої кількості однотипних об’єктів у середовищі Unity з використанням Universal Render Pipeline. Актуальність теми пов’язана з тим, що сучасні ігрові та інтерактивні застосунки часто працюють із насиченими сценами, де одночасно відображаються десятки тисяч об’єктів, а вимоги до стабільної частоти кадрів залишаються високими. Для таких сценаріїв важливо не лише правильно налаштувати рушій, а й обґрунтовано вибирати методи оптимізації, що дають відчутний ефект саме в умовах великої кількості повторюваних об’єктів.

У роботі було послідовно розв’язано низку завдань, а саме:

- проаналізовано теоретичні засади рендерінгу в реальному часі та основні поняття продуктивності (FPS, час кадру, batch count, tris count);
- розглянуто архітектуру рушія Unity, особливості Universal Render Pipeline та порівняно URP із HDRP у контексті задачі рендерінгу великої кількості однотипних об’єктів;
- сформульовано мету й постановку задачі дослідження, визначено набір метрик і єдину методику експериментів;
- розроблено спеціалізований інструмент використовуючи ігровий рушій Unity та C#, який дозволяє формувати сцену з 50 000 сфер, перемикати методи оптимізації та автоматично збирати показники продуктивності;
- проведено серію експериментів для різних методів оптимізації та виконано порівняльний аналіз отриманих результатів.

Створений програмний інструмент забезпечив відтворювані умови дослідження: сцена генерується за фіксованим алгоритмом, кількість і розташування об’єктів залишаються незмінними для всіх сценаріїв, а збір метрик виконується за єдиною схемою з розігрівом сцени та усередненням значень за визначений інтервал часу. Це дозволило точно порівняти різні методи між собою. Як вхідні дані розглядалися: параметри сцени, конфігурація URP,

вибраний метод оптимізації та апаратне середовище; як вихідні: середня частота кадрів, середній час кадру на CPU і GPU та кількість батчів.

Магістерська робота досягла поставленої мети, а саме було розроблено інструмент, проведено серію експериментів і виконано аналіз ефективності низки методів оптимізації рендерінгу в Unity. Отримані висновки можуть бути використані розробниками для вибору методів оптимізації в реальних проектах, де необхідно поєднати високу щільність об’єктів у сцені зі стабільною продуктивністю.

БІБЛІОГРАФІЧНИЙ СПИСОК

1. Unity Technologies. Unity User Manual (Documentation) [Електронний ресурс]. – Режим доступу: https://docs.unity3d.com/;
2. Captain GPU. Основи програмування шейдерів мовою GLSL [Електронний ресурс] // Medium. – Режим доступу: https://captaingpu.medium.com/основи-програмування-шейдерів-мовою-glsl-532560e22cee;
3. Unity Technologies. Optimizing draw calls [Електронний ресурс]. – Режим доступу: https://docs.unity3d.com/6000.3/Documentation/Manual/optimizing-draw-calls.html;
4. Real-Time Rendering [Електронний ресурс]. – Режим доступу: https://www.realtimerendering.com;
5. Kanand003. Draw calls [Електронний ресурс]. – Режим доступу: https://kanand003.github.io/Website/post/drawcalls/;
6. RainyRizzle. Advanced manual: Reduce draw calls [Електронний ресурс]. – Режим доступу: https://rainyrizzle.github.io/en/AdvancedManual/AD_ReduceDrawCalls.html;
7. Unity Technologies. Universal Render Pipeline (URP) introduction [Електронний ресурс]. – Режим доступу: https://docs.unity3d.com/Manual/urp/urp-introduction;
8. Unity Technologies. High Definition Render Pipeline (HDRP) documentation [Електронний ресурс]. – Режим доступу: https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@17.4/manual/index.html;
9. Unity Technologies. Configure URP for better performance [Електронний ресурс]. – Режим доступу: https://docs.unity.cn/Manual/urp/configure-for-better-performance.html;
10. Unity Technologies. LOD Group [Електронний ресурс]. – Режим доступу: https://docs.unity.cn/2023.2/Documentation/Manual/class-LODGroup.html;
11. Unity Technologies. Occlusion culling [Електронний ресурс]. – Режим

доступу: https://docs.unity.cn/2019.1/Documentation/Manual/OcclusionCulling.html;

12. Unity Technologies. Frame timing manager [Електронний ресурс]. – Режим доступу: https://docs.unity.cn/Manual/frame-timing-manager;
13. Unity Technologies. Optimizing Graphics in Unity [Електронний ресурс]. – Режим доступу: https://learn.unity.com/tutorial/optimizing-graphics-in-unity;
14. NVIDIA. GPU Gems. Chapter 28. Graphics Pipeline Performance [Електронний ресурс]. – Режим доступу: https://developer.nvidia.com/gpugems/gpugems/part-v-performance-and-practicalities/chapter-28-graphics-pipeline-performance;
15. Wloka, M. M. Optimizing the Graphics Pipeline (Eurographics 2004) [Електронний ресурс]. – Режим доступу: https://download.nvidia.com/developer/presentations/2004/Eurographics/EG_04_OptimizingGPUPipeline.pdf;
16. Microsoft. Pipelines and Shaders with Direct3D 12 [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/en-us/windows/win32/direct3d12/pipelines-and-shaders-with-directx-12.

ДОДАТОК А

Технічне завдання

ЗАТВЕРДЖУЮ
Проректор Українського державного
університету науки і технологій
Анатолій РАДКЕВИЧ

«ПРОГРАМА ДЛЯ ДОСЛІДЖЕННЯ МЕТОДІВ ОПТИМІЗАЦІЇ РЕНДЕРІНГУ ВЕЛИКОЇ КІЛЬКОСТІ ОБ'ЄКТІВ В UNITY»
Технічне завдання
44165850.1531-01-ЛЗ

Завідувач кафедри КІТ
_____Вадим ГОРЯЧКІН
Керівник розробки
_____Олександр ІВАНОВ
Виконавець
_____Олег СИДОРОВ
Нормоконтролер
_____Світлана ВОЛКОВА

2025

ЗАТВЕРДЖЕНО
44165850.1531-01

«ПРОГРАМА ДЛЯ ДОСЛІДЖЕННЯ МЕТОДІВ ОПТИМІЗАЦІЇ РЕНДЕРІНГУ ВЕЛИКОЇ КІЛЬКОСТІ ОБ'ЄКТІВ В UNITY»

Технічне завдання
44165850.1531-01
Листів 18

2025

ЗМІСТ

Вступ 3
1 Підстава для розробки 4
2 Призначення розробки 5
2.1 Функціональне призначення розробки 8
2.2 Експлуатаційне призначення розробки: 7
3 Вимоги до програми 9
3.1 Вимоги до функціональних характеристик 9
3.2 Вимоги до надійності 11
3.3 Вимоги експлуатації 11
3.4 Вимоги до складу та параметрів технічних засобів 12
3.5 Вимоги до інформаційної та програмної сумісності 13
4 Вимоги до програмної документації 14
5 Стадії та етапи розробки 15
6 Порядок прийняття 16
7 Технічно-економічні показники 17
8 БІБЛІОГРАФІЧНИЙ СПИСОК 18

ВСТУП

Сучасні відеоігри демонструють сталу тенденцію до зростання складності сцен, кількості об’єктів і вимог до якості графіки. В таких умовах питання оптимізації рендерінгу набуває особливої актуальності, оскільки саме відтворення графіки часто є вузьким місцем продуктивності.

Однією з найскладніших ситуацій для рушія є відображення великої кількості однотипних об’єктів, що одночасно присутні в сцені. Це створює навантаження як на процесор, який витрачає значні ресурси на підготовку команд рендерінгу, так і на графічний процесор, який змушений обробляти великий обсяг геометрії та пікселів.

Ігровий рушій Unity, а особливо його Universal Render Pipeline, надає розробникам широкий набір засобів для оптимізації рендерінгу. Однак на практиці постає питання не лише в наявності цих інструментів, а й в розумінні їх реального впливу на продуктивність в конкретних умовах.

Метою цієї магістерської роботи є експериментальне дослідження та порівняльний аналіз методів оптимізації рендерінгу великої кількості однотипних об’єктів в рушії Unity. В рамках роботи передбачається провести серію експериментів для різних методів оптимізації. Оцінювання ефективності методів здійснюється за допомогою таких показників, як середня частота кадрів, середній час кадру на центральному та графічному процесорах, кількість батчів та кількість трикутників.

Отримані результати можуть бути використані розробниками ігор під час розробки та проектування гри, планування цільових платформ та вимог до апаратного забезпечення.

1 ПІДСТАВА ДЛЯ РОЗРОБКИ

Основою для розробки є наказ ректора Українського державного університету науки і технології Радкевич А.В. «Про затвердження тем та призначення керівників дипломних проектів» №1401 ст від 02.10.2025 року.

Тема проекту: «Дослідження методів оптимізації рендерінгу великої кількості об'єктів в Unity».

Керівник дипломного проекту: Іванов О. П.

2 ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою розробки є створення програмного інструменту на базі рушія Unity для аналізу та порівняння методів оптимізації рендерінгу сцени з великою кількістю об’єктів. Такий інструмент дає змогу в єдиних умовах вимірювати та порівнювати ефективність різних методів оптимізації, а також оцінювати їх вплив на основні показники продуктивності.

Дослідження спрямоване на розв’язання таких основних завдань:

– отримання визначених метрик продуктивності визначених методів оптимізації;

– аналіз отриманих отриманих метрик продуктивності;

– формування практичних рекомендацій щодо застосування методів оптимізації для сцен з великою кількістю об’єктів.

Розроблюваний інструмент призначений для використання студентами, дослідниками та розробниками, які працюють із рушієм Unity та досліджують питання продуктивності рендерінгу. Він може слугувати як навчальним прикладом для вивчення впливу різних методів оптимізації на FPS, час кадру на CPU та GPU, так і базою для подальших експериментів у сфері оптимізації графіки.

2.1. Функціональне призначення розробки

Функціональне призначення розробки полягає у наданні засобів для вимірювання та аналізу продуктивності сцени в Unity з великою кількістю однотипних об’єктів за різних варіантів налаштування рендерінгу. Система виступає як програмний інструмент в якому можна змінювати методи оптимізації та оцінювати їхній вплив на основні метрики.

До основних функцій розробки належать:

– формування тестової сцени. Автоматизоване створення сцени з великою кількістю однотипних об’єктів з фіксованим розташуванням та незмінними параметрами камери й освітлення для всіх сценаріїв дослідження;

– керування методами оптимізації. Перемикання між різними варіантами рендерінгу (базовий сценарій, статичний батчинг, спрощені матеріали та моделі, LOD, оклюзійне відсікання, GPU instancing тощо) без зміни загальної структури сцени;

– збір і обробка метрик. Вимірювання середньої частоти кадрів, середнього часу кадру на CPU та GPU, кількості батчів і геометричної складності сцени;

– відображення результатів. Виведення зібраних метрик у зручній для аналізу формі безпосередньо під час експериментів;

Таким чином, розроблений інструмент дозволяє системно оцінити, як конкретні методи оптимізації рендерінгу впливають на різні показники продуктивності включно з часом на центральному та графічному процесорах.

2.2 Експлуатаційне призначення розробки:

Розроблений програмний інструмент призначений для практичного використання під час дослідження та оптимізації рендерінгу сцен з великою кількістю об’єктів в рушії Unity. Він може застосовуватися як в навчальному процесі, так і в якості прикладу для реальних проектів, де важливо забезпечити стабільну продуктивність за умов високої кількості об’єктів на сцені.

Основні напрями експлуатаційного призначення розробки:

- оцінка методів оптимізації рендерінгу. Інструмент дозволяє на практиці порівнювати різні методи оптимізації в єдиних умовах, аналізуючи їх вплив на частоту кадрів, час кадру на CPU і GPU, кількість батчів та трикутників;
- планування продуктивності ігрових сцен. Отримані результати можуть використовуватися розробниками для планування бюджету продуктивності: визначення допустимої кількості об’єктів, складності матеріалів та рівня деталізації моделей для цільових платформ;
- навчальні та демонстраційні цілі. Система може застосовуватися як наочний приклад в навчальних курсах з розробки ігор та комп’ютерної графіки, демонструючи, як зміна налаштувань рендерінгу й методів оптимізації відображається на ключових метриках продуктивності;
- прийняття технічних рішень. Інструмент дає змогу обґрунтовано обирати ті чи інші засоби оптимізації при проектуванні ігор. На основі експериментальних даних можна формувати внутрішні рекомендації команди розробки;

- база для подальших досліджень. Інструмент може бути розширений новими методами оптимізації, іншими типами об’єктів, додатковими рендер-пайплайнами або платформами, слугуючи відправною точкою для наступних досліджень у сфері оптимізації рендерінгу.

Експлуатаційне призначення розробки полягає в тому, щоб забезпечити практичне застосування результатів дослідження: від навчального використання та демонстрації впливу методів оптимізації на продуктивність до підтримки реальних рішень під час розробки й удосконалення ігрових проектів на Unity.

3 ВИМОГИ ДО ПРОГРАМИ

3.1 Вимоги до функціональних характеристик

Програмний засіб призначений для проведення експериментів з оптимізації рендерінгу сцени, що містить велику кількість об’єктів, у середовищі Unity з використанням Universal Render Pipeline. Основне функціональне завдання полягає в тому, щоб забезпечити формування однакових умов тестування для різних методів оптимізації та надати засоби для вимірювання й порівняння їхньої ефективності.

До основних функціональних можливостей програмного засобу належать:

– формування сцени. Автоматизоване створення тестової сцени, що містить 50 000 однотипних сфер, розташованих за заданим алгоритмом у вигляді сітки;

– фіксація параметрів середовища. Підтримка незмінних параметрів камери, освітлення, рендер-пайплайна URP та налаштувань якості для всіх сценаріїв дослідження;

– керування методами оптимізації. Можливість перемикання методів оптимізації рендерінгу шляхом вибору відповідної сцени;

– збір показників продуктивності. Автоматизований збір метрик після короткого розігріву сцени із фіксацією значень за заданий проміжок часу, що забезпечує отримання усереднених і відтворюваних результатів;

– відображення результатів. Виведення ключових метрик безпосередньо в інтерфейсі застосунку для оперативного аналізу та подальшого перенесення в таблиці чи діаграми;

Вхідні дані

До вхідних даних програмного інструменту належать:

Параметри сцени:

- кількість об’єктів;
- схема розташування об’єктів;
- положення та орієнтація камери;
- конфігурація освітлення.

Конфігурація рендерінгу:

- версія рушія Unity (Unity 6.0.49f1);
- використання Universal Render Pipeline як основного рендер-пайплайна;
- налаштування якості, тіней, роздільної здатності та постобробки.

Сценарій оптимізації:

- обраний метод оптимізації;
- параметри, специфічні для окремих методів (порогові відстані LOD, налаштування Occlusion Culling тощо).

Апаратне середовище:

- характеристики центрального та графічного процесорів;
- обсяг оперативної пам’яті;
- версія операційної системи та драйверів графічного адаптера.

Вихідні дані

До вихідних даних, що формуються в результаті роботи програмного інструменту, відносяться:

- середнє значення частоти кадрів (FPS);
- середній час кадру на центральному процесорі (CPU frame time);
- середній час кадру на графічному процесорі (GPU frame time);
- кількість батчів рендерінгу за один кадр (Batch Count);
- кількість трикутників, що беруть участь у рендерінгу.

Отримані вихідні дані використовуються для побудови порівняльних таблиць і графіків, а також для формування висновків щодо ефективності окремих методів оптимізації рендерінгу.

3.2 Вимоги до надійності

Програмний засіб має забезпечувати стабільну роботу під час виконання серії експериментів і коректне завершення вимірювань навіть за наявності окремих нестандартних ситуацій.

До основних вимог до надійності належать:

– відсутність аварійного завершення роботи під час послідовного запуску різних сценаріїв оптимізації та повторних вимірювань;

– збереження незмінності параметрів сцени та рендерінгу в межах однієї серії експериментів, щоб уникнути спотворення результатів;

– використання резервних копій проекту для запобігання втраті коду та налаштувань;

– обмеження небажаних дій користувача під час вимірювань.

3.3 Вимоги експлуатації

Експлуатація програмного засобу передбачає роботу користувача, який знайомий з основами Unity та базовими принципами рендерінгу в реальному часі. Інтерфейс і сценарій використання мають бути достатньо простими, щоб не відволікати від основної мети — проведення вимірювань і аналізу результатів.

Основні вимоги до експлуатації:

- Підготовка користувача. Користувач повинен мати базові навички роботи з Unity-редактором: вміти відкривати проект, запускати сцени, змінювати налаштування якості та збірки.
- Простота керування. Перемикання між сценаріями має здійснюватися через стандартні засоби, без необхідності вносити зміни до коду.

- Мінімальні додаткові налаштування. Для проведення експериментів користувачу не повинно бути потрібно вносити складні зміни в конфігурацію URP або проектні налаштування. Всі ключові параметри мають бути зафіксовані за замовчуванням.
- Зрозуміле відображення результатів. Під час експерименту інструмент має відображати поточні значення FPS та статус вимірювань, а після закінчення — підсумкові усереднені метрики.

3.4 Вимоги до складу та параметрів технічних засобів

Для коректної роботи програмного засобу та отримання репрезентативних результатів необхідно, щоб він запускався на персональному комп’ютері, який відповідає мінімальним технічним вимогам. Це дозволяє уникнути ситуацій, коли результати експериментів визначаються не стільки особливостями методів оптимізації, скільки надто слабким обладнанням.

Рекомендовано наступну конфігурацію:

- операційна система: Windows 10 або новіша версія;
- центральний процесор: не менше 4 фізичних ядер із тактовою частотою від 3,0 ГГц;
- оперативна пам’ять: від 16 ГБ для комфортної роботи з проектом, що містить велику кількість об’єктів;
- графічний адаптер: відеокарта з підтримкою сучасних графічних API та власною відео пам’яттю не менше 4 ГБ;
- монітор: роздільна здатність не нижче 1920×1080 для зручного перегляду інтерфейсу Unity та вікон профілювання;
- достатній обсяг вільного місця на диску для зберігання проекту, бібліотеки Unity та тимчасових файлів (не менше 20–30 ГБ).

За наявності більш потужного обладнання результати експериментів можуть демонструвати вищі значення FPS, однак відносні відмінності між методами оптимізації залишаться показовими.

3.5 Вимоги до інформаційної та програмної сумісності

Програмні продукти розробляється для всіх видів операційних систем сімейства “Windows” починаючи від версії 10 та наступні версії.

4 ВИМОГИ ДО ПРОГРАМНОЇ ДОКУМЕНТАЦІЇ

До складу документації мають входити:

- технічне завдання;
- текст програми;
- пояснювальна записка;
- керівництво користувача.

Вся документація програмних додатків повинна задовольняти вимоги до програмної документації.

5 СТАДІЇ ТА ЕТАПИ РОЗРОБКИ

Таблиця 5.1 – Стадії та етапи розробки

Стадія	Зміст	Строки виконання
Технічне завдання	Постановка задачі, збір інформації, вибір та обґрунтування критеріїв розробки. Попередній вибір методів рішення задач. Визначення вимог до технічних засобів. Узгодження і затвердження технічного завдання.	04.09.25 –15.09.25
Робочий проект	Програмування та відлагодження програми.	18.09.25 – 22.09.25
	Тестування програми	25.09.25 – 27.09.25
	Розробка, узгодження і затвердження програмної документації.	28.09.25 – 29.09.25

6 ПОРЯДОК ПРИЙНЯТТЯ

Контроль за виконанням роботи здійснює керівник розробки доц. Іванов О. П. Прийом здійснюється уповноваженою комісією.

7 ТЕХНІЧНО-ЕКОНОМІЧНІ ПОКАЗНИКИ

Показники та їх розрахунок не були описані у документах бо розробка програмного продукту несе в собі навчальний характер, а не комерційний.

8 БІБЛІОГРАФІЧНИЙ СПИСОК

1. Івченко, Ю.М. Основи стандартизації програмних систем: методичні вказівки до дипломного проектування та лабораторних робіт/уклад.: Ю.М. Івченко, В. І. Шинкаренко, В. Г. Івченко; Дніпропетр. нац. ун-т залізн. трансп. ім. акад. В. Лазаряна. – Д.: Вид-во Дніпропетр. нац. ун-ту залізн. трансп. ім. акад. В. Лазаряна, 2009. - 38 с

ДОДАТОК Б

Текст програми

ЗАТВЕРДЖУЮ
Проректор Українського державного
університету науки і технологій
Анатолій РАДКЕВИЧ

«ПРОГРАМА ДЛЯ ДОСЛІДЖЕННЯ МЕТОДІВ ОПТИМІЗАЦІЇ РЕНДЕРІНГУ ВЕЛИКОЇ КІЛЬКОСТІ ОБ'ЄКТІВ В UNITY»
Текст програми
44165850.01531–01 12 01

Завідувач кафедри КІТ
_____Вадим ГОРЯЧКІН
Керівник розробки
_____Олександр ІВАНОВ
Виконавець
_____Олег СИДОРОВ
Нормоконтролер
_____Світлана ВОЛКОВА

2025

ЗАТВЕРДЖЕНО

44165850.01531–01 12 01

ПРОГРАМА ДЛЯ ДОСЛІДЖЕННЯ МЕТОДІВ ОПТИМІЗАЦІЇ РЕНДЕРІНГУ ВЕЛИКОЇ КІЛЬКОСТІ ОБ'ЄКТІВ В UNITY

Текст програми
44165850.01531–01 12–01
Листів 11

2025

АНОТАЦІЯ

Документ 44165850.01531–01 12 01 «Програма для дослідження методів оптимізації рендерінгу великої кількості об'єктів в Unity» входить до складу програмної документації на програму, що аналізує результати методів оптимізації.

У даному документі представлений текст додатку. Додаток написаний на мові C# у програмному середовищі JetBrains Rider.

ЗМІСТ

1 Текст програми 3
1.1 Текст файлу Main 3
1.2 Текст файлу Level7 4
1.3 Текст файлу SceneTools 6
1.4 Текст файлу CubeInstanced 9

1 ТЕКСТ ПРОГРАМИ

1.1 Текст файлу Main

```
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;

namespace Project.Scripts
{
   /// <summary>
   /// Головне меню експерименту.
   /// На сцені є масив кнопок, кожна кнопка відповідає переходу на сцену
   /// з певним методом оптимізації (за індексом у Build Settings).
   /// </summary>
   public class MainMenu: MonoBehaviour
   {
      // Масив кнопок, які налаштовані в інспекторі (наприклад 7 кнопок для 7 експериментів).
      [SerializeField] private Button[] _buttons;

      private void Awake()
      {
         // Проходимо по всіх кнопках та або підписуємо подію кліку, або видаляємо зайві.
         for (var index = 0; index < _buttons.Length; index++)
         {
            var button = _buttons[index]; // тут зберігаємо посилання, щоб не звертатись до масиву кілька разів

            // Індекс сцени для завантаження.
            // +1 тому що сцена 0 — це Main Menu.
            var ind = index + 1;

            // Якщо кнопок більше, ніж сцен у Build Settings, зайві кнопки прибираємо.
            if (index >= SceneManager.sceneCountInBuildSettings)
            {
               // Видаляємо об’єкт кнопки зі сцени, щоб користувач не міг натиснути порожню кнопку.
               Destroy(button.gameObject);
            }
            else
            {
               // Підписуємо обробник натискання.
               // Використовується локальна змінна ind, щоб кожна кнопка завантажувала "свою" сцену.
               button.onClick.AddListener(() => OnButtonClicked(ind));
            }
         }
      }

      private void OnDestroy()
      {
         // При знищенні об’єкта відписуємось від усіх подій,
         // щоб уникати "висячих" лісенерів при перезавантаженні сцени або виході з неї.
         foreach (var button in _buttons)
         {
            button.onClick.RemoveAllListeners();
         }
      }

      /// <summary>
      /// Обробник кліку по кнопці.
      /// Завантажує сцену за індексом у Build Settings.
      /// </summary>
      private void OnButtonClicked(int index)
      {
         SceneManager.LoadScene(index);
      }
   }
}
```

1.2 Текст файлу Level7

```
using UnityEngine;
using UnityEngine.SceneManagement;

/// <summary>
/// Сцена експерименту з GPU Instancing Indirect.
/// Скрипт:
/// 1) формує статичну ґратку позицій для інстансів (через SceneTools.LoopPositions);
/// 2) завантажує позиції в ComputeBuffer(и), які читає шейдер;
/// 3) готує args-buffer для Graphics.DrawMeshInstancedIndirect;
/// 4) виконує один виклик DrawMeshInstancedIndirect щокадру.
/// </summary>
public class Level7 : MonoBehaviour
{
    [Header("Instance data")]
    [SerializeField] private Mesh _instanceMesh;           // Меш, який буде інстанситись (сфера/куб тощо)
    [SerializeField] private Material _instanceMaterial;   // Матеріал з шейдером, який читає position_buffer_1/2

    [Header("Draw settings")]
    // Bounds визначає область, у якій Unity вважає, що знаходяться інстанси.
    // Використовується для CPU frustum culling одного "пакету" інстансів.
    // Якщо Bounds занадто малий — частина інстансів може зникати. Якщо занадто великий — зайві проходи без користі.
    [SerializeField] private Vector3 _boundsCenter = new Vector3(49.5f, 17.5f, 49.5f);
    [SerializeField] private Vector3 _boundsSize   = new Vector3(110f, 50f, 110f);

    // Гаряча клавіша для швидкого перезапуску поточної сцени
    [SerializeField] private KeyCode _reloadKey = KeyCode.R;

    // Ідентифікатори властивостей шейдера.
    // Це швидше і надійніше, ніж постійно передавати рядок з назвою властивості.
    private static readonly int PositionBuffer1Id = Shader.PropertyToID("position_buffer_1");
    private static readonly int PositionBuffer2Id = Shader.PropertyToID("position_buffer_2");

    // Аргументи для DrawMeshInstancedIndirect:
    // [0] indexCountPerInstance  - кількість індексів в submesh (скільки індексів малювати на інстанс)
    // [1] instanceCount          - кількість інстансів
    // [2] startIndexLocation     - з якого індекса починати в індексному буфері
    // [3] baseVertexLocation     - base vertex для submesh
    // [4] startInstanceLocation  - офсет інстансів (зазвичай 0)
    private readonly uint[] _args = { 0, 0, 0, 0, 0 };

    // Два буфери позицій (start/end) — під твій шейдер, який інтерполює позиції
    private ComputeBuffer _positionBuffer1;
    private ComputeBuffer _positionBuffer2;

    // Indirect args buffer, який передається в DrawMeshInstancedIndirect
    private ComputeBuffer _argsBuffer;

    private void OnEnable()
    {
        // Якщо не виставлені посилання в інспекторі — нічого не робимо
        if (_instanceMesh == null || _instanceMaterial == null)
            return;

        // Створюємо argsBuffer (з індексами/кількістю інстансів)
        AllocateBuffers();

        // Генеруємо позиції інстансів та завантажуємо їх у GPU буфери
        BuildAndUploadPositions();

        // Заповнюємо argsBuffer на основі меша та кількості позицій
        UpdateArgsBuffer();

        // Підпис сцени для UI
        if (SceneTools.Instance != null)
            SceneTools.Instance.SetNameText("GPU Instancing Indirect (Static Grid)");
    }

    private void Update()
    {
        // Якщо буфери не створені — не намагаємось малювати
        if (_argsBuffer == null || _instanceMesh == null || _instanceMaterial == null)
```

```
        return;

    // Один виклик малює одразу всі інстанси (instanceCount береться з argsBuffer)
    Draw();

    // Перезапуск сцени для повторного тесту
    if (Input.GetKeyDown(_reloadKey))
        ReloadScene();
}

private void OnDisable()
{
    // Обов'язково звільняємо ComputeBuffer-и,
    // інакше буде витік пам'яті та/або помилки при повторному вході в Play Mode
    ReleaseBuffers();
}

private void AllocateBuffers()
{
    // argsBuffer містить один елемент (масив з 5 uint), тип IndirectArguments обов'язковий для indirect draw
    _argsBuffer = new ComputeBuffer(1, _args.Length * sizeof(uint), ComputeBufferType.IndirectArguments);
}

private void BuildAndUploadPositions()
{
    // Кількість інстансів визначена параметрами SceneTools (100 * 100 * 5)
    int count = SceneTools.GetCount;
    var positions = new Vector4[count];

    // Формуємо координати сітки. Тут p — координати в логічних одиницях сітки.
    // Y масштабується через DepthOffset, щоб "шари" по висоті мали потрібну відстань.
    SceneTools.LoopPositions((i, p) =>
    {
        positions[i] = new Vector4(p.x, p.y * SceneTools.DepthOffset, p.z, 0f);
    });

    // Пересоздаємо позиційні буфери під точну кількість інстансів
    _positionBuffer1?.Release();
    _positionBuffer2?.Release();

    // Vector4 = 4 float = 16 байт
    _positionBuffer1 = new ComputeBuffer(positions.Length, sizeof(float) * 4);
    _positionBuffer2 = new ComputeBuffer(positions.Length, sizeof(float) * 4);

    // Завантажуємо дані у GPU
    _positionBuffer1.SetData(positions);
    _positionBuffer2.SetData(positions);

    // Прив'язуємо буфери до матеріалу (шейдер читає position_buffer_1/2)
    _instanceMaterial.SetBuffer(PositionBuffer1Id, _positionBuffer1);
    _instanceMaterial.SetBuffer(PositionBuffer2Id, _positionBuffer2);

    // Параметр instanceCount у args: скільки інстансів малювати
    _args[1] = (uint)positions.Length;
}

private void UpdateArgsBuffer()
{
    // Беремо параметри submesh 0.
    // Якщо колись буде інший submesh, треба замінити індекс 0 на потрібний.
    _args[0] = _instanceMesh.GetIndexCount(0); // indexCountPerInstance
    _args[2] = _instanceMesh.GetIndexStart(0); // startIndexLocation
    _args[3] = _instanceMesh.GetBaseVertex(0); // baseVertexLocation
    _args[4] = 0;                              // startInstanceLocation

    // Завантажуємо args у GPU
    _argsBuffer.SetData(_args);
}

private void Draw()
{
    // Bounds задається в world space.
    // Unity використовує його як єдину область для кульінгу всієї групи інстансів.
    var bounds = new Bounds(_boundsCenter, _boundsSize);
```

```
        // Малюємо інстанси індиректом:
        // - кількість інстансів та параметри submesh беруться з argsBuffer,
        // - позиції інстансів беруться з ComputeBuffer-ів, які читає шейдер.
        Graphics.DrawMeshInstancedIndirect(_instanceMesh, 0, _instanceMaterial, bounds, _argsBuffer);
    }

    private static void ReloadScene()
    {
        var scene = SceneManager.GetActiveScene();
        SceneManager.LoadScene(scene.buildIndex);
    }

    private void ReleaseBuffers()
    {
        // Release() звільняє ресурс на GPU/в драйвері.
        // Важливо обнулити посилання, щоб уникнути повторного Release().
        _positionBuffer1?.Release(); _positionBuffer1 = null;
        _positionBuffer2?.Release(); _positionBuffer2 = null;
        _argsBuffer?.Release();      _argsBuffer = null;
    }
}
```

1.3 Текст файлу SceneTools

```
using System;
using System.Collections;
using TMPro;
using Unity.Mathematics;
using Unity.Profiling;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;

/// <summary>
/// Інструмент для експериментальних сцен:
/// 1) робить прогрів сцени;
/// 2) знімає Batch Count та Tris Count один раз після прогріву;
/// 3) збирає середні значення FPS, CPU Frame Time, GPU Frame Time за інтервал заміру;
/// 4) відображає статуси та таймери в UI.
/// </summary>
public class SceneTools : MonoBehaviour
{
    [Header("UI")]
    [SerializeField] private TMP_Text _sceneNameText;
    [SerializeField] private TMP_Text _batchCountText;
    [SerializeField] private TMP_Text _trisCountText;
    [SerializeField] private TMP_Text _realTimeFpsText;
    [SerializeField] private TMP_Text _avgFpsText;
    [SerializeField] private TMP_Text _cpuFrameTimeText;
    [SerializeField] private TMP_Text _gpuFrameTimeText;
    [SerializeField] private Button _menuButton;

    [Header("Measurement")]
    [Tooltip("Тривалість прогріву перед будь-якими замірами.")]
    [SerializeField] private float _warmupSeconds = 5f;

    [Tooltip("Тривалість інтервалу, протягом якого збираються FPS/CPU/GPU.")]
    [SerializeField] private float _sampleSeconds = 10f;


    public const float DepthOffset = 7f;
    public const float HeightScale = 2.5f;
    public const float BurstHeightScale = 1.5f;
    public const float NoiseScale = 0.2f;

    // Назви лічильників у Rendering Profiler module.
    private const string BatchesCounterName = "Batches Count";
    private const string TrianglesCounterName = "Triangles Count";

    // Єдині строки станів, щоб текст всюди був однаковим.
    private const string StatusWarmup = "Warming up";
    private const string StatusMeasuring = "Measuring";
    private const string ValueNA = "N/A";
```

```
// Параметри для генерації 50 000 об’єктів (100 * 100 * 5).
private const int SideLength = 100;
private const int Depth = 5;

// Публічні поля, які використовуються поза класом.
public static readonly quaternion RotGoal = quaternion.Euler(130, 50, 150);
public static Vector3 CubeScale { get; } = new(0.7f, 0.7f, 0.7f);
public static int GetCount => SideLength * SideLength * Depth;
public static SceneTools Instance;

// Рекордери для зчитування Batch/Tris з профайлера.
private ProfilerRecorder _batchesRecorder;
private ProfilerRecorder _trisRecorder;

private void Awake()
{
    // Простий singleton, щоб інші компоненти могли звертатись до SceneTools.Instance.
    Instance = this;

    // Не обмежуємо FPS, щоб сцена працювала у максимальному режимі платформи.
    Application.targetFrameRate = -1;

    // Кнопка повернення у головне меню (сцена 0).
    if (_menuButton != null)
        _menuButton.onClick.AddListener(() => SceneManager.LoadScene(0));
}

private IEnumerator Start()
{
    // Запускаємо рекордери для лічильників Rendering.
    _batchesRecorder = ProfilerRecorder.StartNew(ProfilerCategory.Render, BatchesCounterName);
    _trisRecorder = ProfilerRecorder.StartNew(ProfilerCategory.Render, TrianglesCounterName);

    // Показуємо назву активної сцени.
    if (_sceneNameText != null)
        _sceneNameText.text = SceneManager.GetActiveScene().name;

    // Пауза на 1 кадр, щоб UI встиг ініціалізуватися і не мигав.
    yield return null;

    // Один корутин, який робить весь цикл: прогрів -> знімок батчів/трис -> замір FPS/CPU/GPU -> результат.
    yield return RunMeasurement();
}

private void Update()
{
    // Поточний FPS (реальний час) — просто для індикатора користувачу.
    if (_realTimeFpsText == null) return;

    float dt = Time.deltaTime;
    float fps = dt > 0f ? 1f / dt : 0f;
    _realTimeFpsText.text = $"Current FPS: {fps:F0}";
}

private void OnDestroy()
{
    // Скидаємо singleton.
    Instance = null;

    // Зупиняємо всі корутини (якщо сцена змінюється або об’єкт знищується).
    StopAllCoroutines();

    // Коректно звільняємо ресурси рекордерів.
    if (_batchesRecorder.Valid) _batchesRecorder.Dispose();
    if (_trisRecorder.Valid) _trisRecorder.Dispose();

    // Прибираємо лісенери.
    if (_menuButton != null)
        _menuButton.onClick.RemoveAllListeners();
}

public void SetNameText(string testName)
{
    // Вивід номера сцени та назви тесту (методу оптимізації).
```

```
    if (_sceneNameText == null) return;
    _sceneNameText.text = $"{SceneManager.GetActiveScene().buildIndex + 1} - {testName}";
}

/// <summary>
/// Повний цикл збору метрик:
/// - прогрів (таймер у всіх полях),
/// - знімок Batch/Tris (один раз),
/// - замір FPS/CPU/GPU (таймер тільки у відповідних полях),
/// - фінальний вивід результатів.
/// </summary>
private IEnumerator RunMeasurement()
{
    // Захист від некоректних значень у інспекторі.
    if (_warmupSeconds < 0f) _warmupSeconds = 0f;
    if (_sampleSeconds <= 0f) _sampleSeconds = 0.01f;

    // Буфер для GetLatestTimings (беремо по одному останньому таймінгу).
    var timingBuf = new FrameTiming[1];

    // Під час прогріву показуємо таймер у всіх полях, включно з Batch/Tris.
    float warmupLeft = _warmupSeconds;
    while (warmupLeft > 0f)
    {
        SetWarmupStatus(warmupLeft);
        yield return null;
        warmupLeft -= Time.unscaledDeltaTime;
    }
    SetWarmupStatus(0f);

    // Даємо ще один кадр, щоб рекордери точно мали LastValue після завершеного кадру рендеру.
    yield return null;

    // Batch/Tris знімаємо один раз, бо в твоєму сценарії вони стабільні.
    string batchesText = (_batchesRecorder.Valid) ? _batchesRecorder.LastValue.ToString() : ValueNA;
    string trisText = (_trisRecorder.Valid) ? _trisRecorder.LastValue.ToString() : ValueNA;

    if (_batchCountText != null) _batchCountText.text = $"Batch Count: {batchesText}";
    if (_trisCountText != null) _trisCountText.text = $"Tris Count: {trisText}";

    float elapsed = 0f;

    // Для FPS достатньо рахувати кількість кадрів за інтервал.
    int fpsFrames = 0;

    // Для CPU/GPU підсумовуємо cpuFrameTime / gpuFrameTime і ділимо на кількість валідних семплів.
    double sumCpuMs = 0.0;
    double sumGpuMs = 0.0;
    int timingFrames = 0;

    while (elapsed < _sampleSeconds)
    {
        yield return null;

        float dt = Time.unscaledDeltaTime;
        elapsed += dt;

        // Таймер заміру показуємо тільки в полях FPS/CPU/GPU.
        float left = Mathf.Max(0f, _sampleSeconds - elapsed);
        SetMeasuringStatus(left);

        // FPS: один кадр = один приріст лічильника.
        fpsFrames++;

        // CPU/GPU: знімаємо FrameTiming.
        FrameTimingManager.CaptureFrameTimings();
        uint got = FrameTimingManager.GetLatestTimings(1, timingBuf);
        if (got > 0)
        {
            var ft = timingBuf[0];
            sumCpuMs += ft.cpuFrameTime;
            sumGpuMs += ft.gpuFrameTime;
            timingFrames++;
        }
    }
```

```
        // Середній FPS як frames / seconds.
        float avgFps = fpsFrames / Mathf.Max(_sampleSeconds, 0.000001f);

        // Якщо таймінги не зібрались (0 семплів), показуємо N/A.
        string avgCpuText = ValueNA;
        string avgGpuText = ValueNA;

        if (timingFrames > 0)
        {
            double avgCpu = sumCpuMs / timingFrames;
            double avgGpu = sumGpuMs / timingFrames;
            avgCpuText = $"{avgCpu:F3} ms";
            avgGpuText = $"{avgGpu:F3} ms";
        }

        // Фінальний вивід значень.
        if (_avgFpsText != null) _avgFpsText.text = $"Average FPS: {avgFps:F1}";
        if (_cpuFrameTimeText != null) _cpuFrameTimeText.text = $"CPU Frame Time: {avgCpuText}";
        if (_gpuFrameTimeText != null) _gpuFrameTimeText.text = $"GPU Frame Time: {avgGpuText}";
        // Batch/Tris уже виставлені після прогріву і більше не змінюються.
    }

    /// <summary>
    /// Формує статус прогріву та ставить його в усі поля метрик.
    /// </summary>
    private void SetWarmupStatus(float secondsLeft)
    {
        string status = $"{StatusWarmup}: {Mathf.Max(0f, secondsLeft):F1} s";

        if (_avgFpsText != null) _avgFpsText.text = $"Average FPS: {status}";
        if (_cpuFrameTimeText != null) _cpuFrameTimeText.text = $"CPU Frame Time: {status}";
        if (_gpuFrameTimeText != null) _gpuFrameTimeText.text = $"GPU Frame Time: {status}";
        if (_batchCountText != null) _batchCountText.text = $"Batch Count: {status}";
        if (_trisCountText != null) _trisCountText.text = $"Tris Count: {status}";
    }

    /// <summary>
    /// Формує статус заміру та ставить його тільки в поля FPS/CPU/GPU.
    /// Batch/Tris не чіпаємо під час заміру.
    /// </summary>
    private void SetMeasuringStatus(float secondsLeft)
    {
        string status = $"{StatusMeasuring}: {Mathf.Max(0f, secondsLeft):F1} s";

        if (_avgFpsText != null) _avgFpsText.text = $"Average FPS: {status}";
        if (_cpuFrameTimeText != null) _cpuFrameTimeText.text = $"CPU Frame Time: {status}";
        if (_gpuFrameTimeText != null) _gpuFrameTimeText.text = $"GPU Frame Time: {status}";
    }

    /// <summary>
    /// Допоміжний метод для обходу всіх позицій ґратки.
    /// Використовується при розстановці/генерації об’єктів у сцені.
    /// </summary>
    public static void LoopPositions(Action<int, Vector3> action)
    {
        int i = 0;
        for (int y = 0; y < Depth; y++)
        for (int x = 0; x < SideLength; x++)
        for (int z = 0; z < SideLength; z++)
            action(i++, new Vector3(x, y, z));
    }
}
```

1.4 Текст файлу CubeInstanced

```
Shader "Test/CubeInstanced"
{
    Properties
    {
        // Базовий колір інстансів (помножується на освітлення)
        _FarColor ("Color", Color) = (.2, .2, .2, 1)
```

```
    // Масштаб геометрії кожного інстанса.
    // Значення за замовчуванням = 0.7
    _InstanceScale ("Instance Scale", Range(0.1, 2.0)) = 0.7
}

SubShader
{
    Tags
    {
        // Непрозорий рендер у URP
        "RenderType"="Opaque"
        "RenderPipeline"="UniversalRenderPipeline"
    }

    Pass
    {
        HLSLPROGRAM
        #pragma vertex vert
        #pragma fragment frag
        #pragma multi_compile_instancing

        #include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl"

        // Властивості матеріалу (з Properties)
        float4 _FarColor;
        float  _InstanceScale;

        // Буфери з позиціями інстансів.
        // start.xyz / end.xyz — світові координати інстанса
        // start.w — зсув фази для анімації (використовується у sin)
        StructuredBuffer<float4> position_buffer_1;
        StructuredBuffer<float4> position_buffer_2;

        // Вхідні дані вершини з меша
        struct Attributes
        {
            float3 normalOS   : NORMAL;
            float4 positionOS : POSITION;
        };

        // Дані, які передаємо у фрагментний шейдер
        struct Varyings
        {
            float4 positionCS : SV_POSITION;
            half3  normalWS   : TEXCOORD0;
        };

        Varyings vert(Attributes v, uint instance_id : SV_InstanceID)
        {
            // Беремо позиції інстанса зі StructuredBuffer
            float4 start = position_buffer_1[instance_id];
            float4 end   = position_buffer_2[instance_id];

            // Плавна анімація між start та end у діапазоні [0..1]
            float t = (sin(_Time.y + start.w) + 1.0) * 0.5;

            // Масштабуємо геометрію інстанса (локальні вершини меша).
            // Це зменшує/збільшує розмір об’єкта, не змінюючи координати самої сітки розстановки.
            float3 local = v.positionOS.xyz * _InstanceScale;

            // Позиція вершини у world space (позиція інстанса + локальна вершина)
            float3 worldStart = start.xyz + local;
            float3 worldEnd   = end.xyz   + local;
            float3 posWS      = lerp(worldStart, worldEnd, t);

            Varyings o;
            // Переводимо world space у clip space
            o.positionCS = TransformWorldToHClip(posWS);

            // Переводимо нормаль у world space та нормалізуємо.
            // Це ключова частина, щоб освітлення не виглядало як самосвітіння.
            o.normalWS = normalize(TransformObjectToWorldNormal(v.normalOS));

            return o;
        }
```

```
        half4 frag(Varyings i) : SV_Target
        {
            // Отримуємо головне світло сцени (Directional Light у URP)
            Light light = GetMainLight();
            // Дифузне освітлення за Ламбертом (без емісії)
            half3 diffuse = LightingLambert(light.color, light.direction, i.normalWS);

            // Фінальний колір: базовий колір * освітлення
            half3 col = saturate(_FarColor.rgb * diffuse);
            return half4(col, 1.0);
        }
        ENDHLSL
    }
  }
}
```

ДОДАТОК В

Керівництво користувача

ЗАТВЕРДЖУЮ
Проректор Українського державного
університету науки і технологій
Анатолій РАДКЕВИЧ

«ПРОГРАМА ДЛЯ ДОСЛІДЖЕННЯ МЕТОДІВ ОПТИМІЗАЦІЇ РЕНДЕРІНГУ ВЕЛИКОЇ КІЛЬКОСТІ ОБ'ЄКТІВ В UNITY»

Керівництво користувача
44165850.01531– 01 І3 01

Завідувач кафедри КІТ
_____Вадим ГОРЯЧКІН
Керівник розробки
_____Олександр ІВАНОВ
Виконавець
_____Олег СИДОРОВ
Нормоконтролер
_____Світлана ВОЛКОВА

2025

ЗАТВЕРДЖЕНО
44165850.01531-01 І3 01

«ПРОГРАМА ДЛЯ ДОСЛІДЖЕННЯ МЕТОДІВ ОПТИМІЗАЦІЇ РЕНДЕРІНГУ ВЕЛИКОЇ КІЛЬКОСТІ ОБ'ЄКТІВ В UNITY»

Керівництво користувача
44165850.01531– 01 І3 01

Листів 9

2025

АНОТАЦІЯ

Документ 1116130.01531– 01 І3 01 «Дослідження методів оптимізації рендерінгу великої кількості об'єктів в Unity. Керівництво користувача».

Програми написані на мові C#, з використанням рушія Unity, в програмному середовищі JetBrains Rider.

ЗМІСТ

1 Введення 4
2 Призначення та умови застосування 5
3 Підготовка до роботи 6
4 Опис операцій 7
5 Аварійні ситуації 8
6 Рекомендації щодо застосування 9

1 ВВЕДЕННЯ

Програмний засіб «Інструмент дослідження методів оптимізації рендерінгу великої кількості об’єктів у Unity» призначений для експериментального вивчення та порівняльного аналізу ефективності методів оптимізації під час відображення сцени з великою кількістю однотипних об’єктів. Основною метою розробки є створення інструментарію, що дозволяє в контрольованих умовах запускати експерименти для різних підходів оптимізації та отримувати вимірювані показники продуктивності, які характеризують навантаження на процесор і графічний процесор (FPS, час кадру на CPU/GPU), а також параметри рендерінгу (кількість батчів і кількість трикутників).

Програмний засіб орієнтований на студентів, дослідників і розробників, які виконують аналіз продуктивності Unity-проектів і потребують наочних, відтворюваних результатів для оцінки впливу оптимізації. У межах дослідження передбачено випробування типових методів оптимізації рендерінгу, зокрема підходів до зменшення кількості операцій відрисовки (батчинг/інстанціювання) та зниження зайвих витрат рендерінгу (наприклад, occlusion culling).

Інтерфейс програми реалізовано у вигляді меню з кнопками вибору методу, що забезпечує швидкий перехід до відповідної сцени експерименту без введення додаткових параметрів. На сцені експерименту відображаються зібрані метрики продуктивності та передбачено кнопку повернення до головного меню для повторного запуску досліду з іншим методом. Така організація взаємодії є простою у використанні, не потребує спеціальної підготовки користувача та забезпечує зручне повторення вимірювань у межах одного запуску застосунку.

2 ПРИЗНАЧЕННЯ ТА УМОВИ ЗАСТОСУВАННЯ

Функціональним призначенням програмного засобу є проведення експериментального аналізу ефективності методів оптимізації рендерінгу сцени з великою кількістю однотипних об’єктів в середовищі Unity. Додаток забезпечує вибір методу оптимізації через головне меню та запуск відповідного експерименту на окремій сцені, після чого виконується збір і відображення метрик продуктивності. Отримані значення можуть бути використані для порівняння впливу методу відносно базового сценарію та подальшого формування висновків щодо доцільності застосування кожного підходу.

Експлуатаційне призначення додатку полягає у наданні інструментарію для повторюваного тестування і порівняльної оцінки методів оптимізації в однакових умовах рендерінгу. Це дозволяє отримувати відтворювані результати та приймати обґрунтовані рішення щодо вибору методів оптимізації залежно від характеру обмеження продуктивності.

Для забезпечення сталого функціонування програми та коректності порівняння результатів рекомендується дотримуватися таких умов:

- використання актуальних та офіційно підтримуваних драйверів відеокарти;
- фіксація однакових налаштувань якості та роздільної здатності під час усіх запусків експерименту;
- мінімізація фонового навантаження системи під час вимірювань.

Додаток розрахований на використання на пристроях, що відповідають базовим вимогам до запуску застосунків Unity на настільних платформах:

- 64-бітна ОС Windows 10/11;
- відеоадаптер із підтримкою графічного API рівня DirectX 10/11/12 (або еквівалентного для відповідної ОС);
- оперативна пам’ять: для стабільної роботи середовища Unity рекомендовано не менше 8 ГБ.

3 ПІДГОТОВКА ДО РОБОТИ

Попередня установка:

- інсталяція середовища розробки Unity Hub та Unity Editor. Для роботи з проектом необхідно встановити Unity Hub, який використовується для керування версіями Unity Editor та проектами. Процедура інсталяції Unity Hub виконується через офіційний інсталятор Unity;
- далі через Unity Hub необхідно встановити Unity Editor потрібної версії.

Отримання проекту:

- клонування репозиторію. Вихідний код проекту отримується з репозиторію та завантажується на локальний диск командою:

  git clone *https://github.com/OtjuQQ/MastersThesis*

- додавання проекту в Unity Hub та відкриття. У Unity Hub проект додається через функцію відкриття/додавання існуючого проекту, після чого виконується його запуск у встановленій версії Unity Editor.

Запуск проекту в Unity Editor

- після відкриття проекту необхідно відкрити сцену Main Menu;
- запуск у режимі тестування виконується стандартною кнопкою Play в Unity Editor;
- у головному меню користувач обирає метод оптимізації, після чого здійснюється перехід на відповідну сцену експерименту, де відображаються метрики та доступна кнопка повернення до меню.

Збірка та запуск виконуваного файлу (за потреби)

- для створення автономної збірки застосунку використовується вікно Build Settings (*File → Build Settings*), де обирається цільова платформа і виконується Build або Build And Run.
- після збірки Unity формує виконуваний файл (.exe);
- запуск автономної збірки з обраного каталогу.

4 ОПИС ОПЕРАЦІЙ

Після запуску програмного засобу користувач отримує доступ до елементів інтерфейсу, наведених на рис. 1 та рис. 2.

У головному меню реалізовано меню вибору методу оптимізації у вигляді семи кнопок. Кожна кнопка відповідає окремому методу оптимізації та виконує перехід на відповідну сцену експерименту після натискання.

На екрані експерименту відображається тестова сцена та інформаційна панель з результатами вимірювань. Панель містить текстові поля, у яких відображаються ключові метрики продуктивності та параметри рендерінгу. Результати виводяться на екран у текстовому вигляді.

Для керування навігацією передбачено кнопку “Back to Menu”, яка забезпечує повернення користувача до головного.

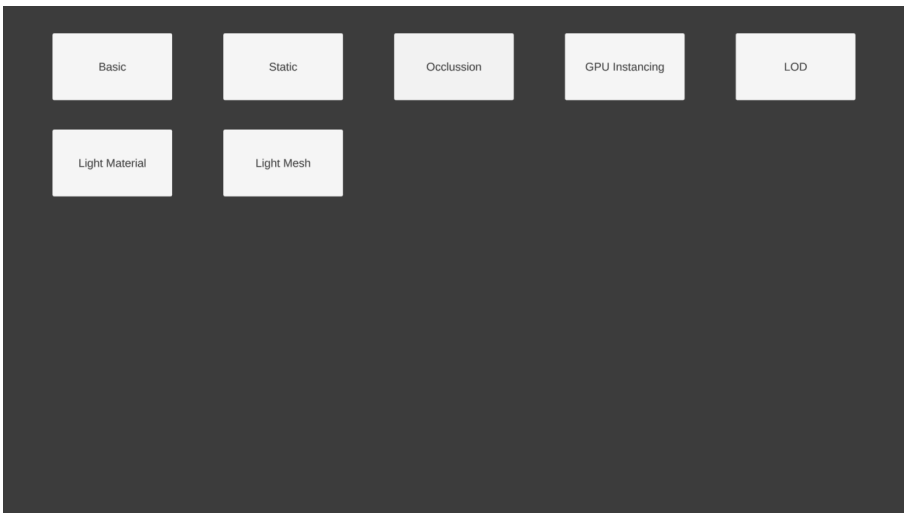


Рисунок 1 – інтерфейс головного меню

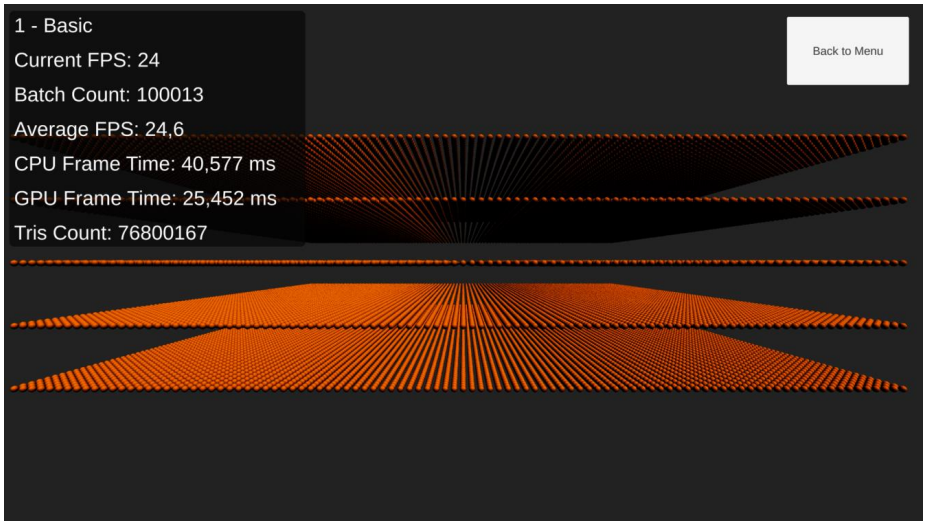


Рисунок 2 – інтерфейс експерименту

5 АВАРІЙНІ СИТУАЦІЇ

Якщо програмний засіб під час роботи буде поводити некоректно чи із помилкою, необхідно перезапустити додаток для продовження роботи.

6 РЕКОМЕНДАЦІЇ ЩОДО ЗАСТОСУВАННЯ

Після розпакування програмного засобу на пристрій необхідно запустити виконуваний файл .exe, який знаходиться в каталозі збірки.

Після запуску додатку відкривається головний екран, який містить меню з **7** кнопками для вибору методу оптимізації. Натискання на відповідну кнопку ініціює завантаження сцени з реалізацією обраного методу.

Після переходу на сцену експерименту відображається тестова сцена з великою кількістю об’єктів та інформаційна панель з метриками продуктивності. На сцені експерименту також присутня кнопка повернення до головного меню, що дозволяє повторно виконати експеримент для іншого методу оптимізації без перезапуску застосунку.

Завершення роботи програмного засобу виконується стандартними засобами операційної системи: закриттям вікна застосунку або комбінацією Alt + F4.

ДОДАТОК Д

Тези

ЗАТВЕРДЖУЮ
Проректор Українського державного
університету науки і технологій
Анатолій РАДКЕВИЧ

«ПРОГРАМА ДЛЯ ДОСЛІДЖЕННЯ МЕТОДІВ ОПТИМІЗАЦІЇ РЕНДЕРІНГУ ВЕЛИКОЇ КІЛЬКОСТІ ОБ'ЄКТІВ В UNITY»
Тези
44165850.01531– 90–99–ЛЗ

Завідувач кафедри КІТ
_____Вадим ГОРЯЧКІН
Керівник розробки
_____Олександр ІВАНОВ
Виконавець
_____Олег СИДОРОВ
Нормоконтролер
_____Світлана ВОЛКОВА

2025

ЗАТВЕРДЖЕНО
44165850.01531– 90–99

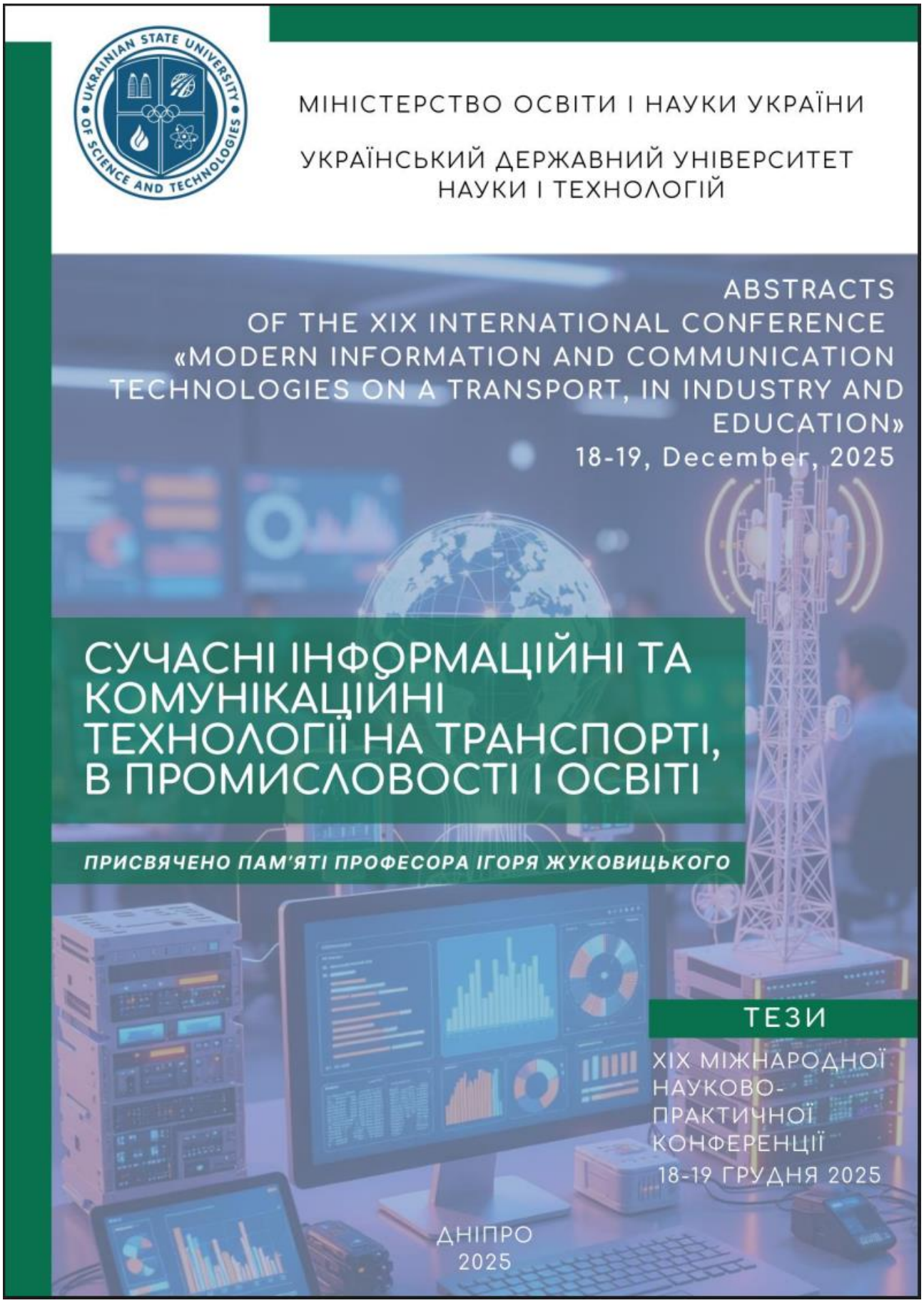
«ПРОГРАМА ДЛЯ ДОСЛІДЖЕННЯ МЕТОДІВ ОПТИМІЗАЦІЇ РЕНДЕРІНГУ ВЕЛИКОЇ КІЛЬКОСТІ ОБ'ЄКТІВ В UNITY»

Тези

44165850.01531– 90–99

Листів 6

2025


МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
УКРАЇНСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ НАУКИ І ТЕХНОЛОГІЙ
ABSTRACTS
OF THE XIX INTERNATIONAL CONFERENCE
«MODERN INFORMATION AND COMMUNICATION TECHNOLOGIES ON A TRANSPORT, IN INDUSTRY AND EDUCATION»
18-19, December, 2025
СУЧАСНІ ІНФОРМАЦІЙНІ ТА КОМУНІКАЦІЙНІ ТЕХНОЛОГІЇ НА ТРАНСПОРТІ, В ПРОМИСЛОВОСТІ І ОСВІТІ
ПРИСВЯЧЕНО ПАМ'ЯТІ ПРОФЕСОРА ІГОРЯ ЖУКОВИЦЬКОГО
ТЕЗИ
XIX МІЖНАРОДНОЇ НАУКОВО-ПРАКТИЧНОЇ КОНФЕРЕНЦІЇ
18-19 ГРУДНЯ 2025
ДНІПРО
2025

Міністерство освіти і науки України

Український державний університет науки і технологій

ТЕЗИ
XIX Міжнародної науково-практичної конференції
«СУЧАСНІ ІНФОРМАЦІЙНІ ТА КОМУНІКАЦІЙНІ ТЕХНОЛОГІЇ НА ТРАНСПОРТІ, В ПРОМИСЛОВОСТІ І ОСВІТІ»
Присвячено пам'яті Ігоря ЖУКОВИЦЬКОГО

ABSTRACTS
of the XIX International Conference
«MODERN INFORMATION AND COMMUNICATION TECHNOLOGIES ON A TRANSPORT, IN INDUSTRY AND EDUCATION»
Dedicated to the memory of Igor ZHUKOVYTSKY

18.12.2025 – 19.12.2025

Дніпро
2025

УДК 658.512.2:681.3.06

Сучасні інформаційні та комунікаційні технології на транспорті, в промисловості і освіті: Тези XIX Міжнародної науково-практичної конференції (Дніпро, 18-19 грудня 2025 р.). – Д.: УДУНТ, 2025. – 172 с.

У збірнику представлені тези доповідей XIX Міжнародної науково-практичної конференції «Сучасні інформаційні та комунікаційні технології на транспорті, в промисловості і освіті», яка відбулася 18-19 грудня 2025 року в Українському державному університеті науки та технологій в онлайн форматі. Конференцію присвячено пам'яті Ігоря ЖУКОВИЦЬКОГО, доктора технічних наук, професора кафедри електронних обчислювальних машин (УДУНТ, м. Дніпро). Розглянуто результати теоретичних і експериментальних досліджень, а також проблемні питання функціонування та перспективи розвитку інформаційних технологій транспорту, промисловості й освіти.

Збірник призначений для науково-технічних працівників залізниць, підприємств транспорту, викладачів вищих навчальних закладів, докторантів, аспірантів і студентів.

Адреса редакційної колегії:
49010, м. Дніпро, вул. Лазаряна, 2, УДУНТ

Тези доповідей друкуються мовою оригіналу в редакції авторів.

Дослідження методів оптимізації рендерінгу великої кількості об'єктів в Unity

Сидоров О.В., Іванов О.П., Український державний університет науки і технологій, Україна

Сучасні відеоігри характеризуються наявністю великої кількості однотипних об'єктів, що призводить до значного навантаження на апаратні ресурси системи. У таких умовах особливої актуальності набувають методи оптимізації рендерінгу, адже вони дозволяють зменшити витрати обчислювальних ресурсів та забезпечити ефективне використання графічного процесора. Головною задачею оптимізації рендерінгу є забезпечення стабільної високої частоти кадрів та плавної роботи застосунку навіть при великій кількості об'єктів. Дослідження присвячене аналізу ефективності методів оптимізації рендерінгу великої кількості об'єктів в Unity. Метою роботи є визначення найбільш дієвих підходів для підвищення продуктивності рендерінгу.

Для досягнення поставленої мети були використані наступні методи оптимізації: Static Batching, LOD, спрощення матеріалів і моделей, Occlusion Culling, GPU Instancing. Експеримент відбувався в ігровому рушії Unity з використанням мови C#. Для оцінювання ефективності кожного методу оптимізації вимірювалися ключові показники продуктивності: частота кадрів (FPS), час формування кадру на центральному (CPU) та графічному (GPU) процесорах, кількість батчів (Batches) і кількість відображуваних трикутників (Tris). Збір і аналіз цих метрик здійснювалися за допомогою спеціального інструменту, розробленого в рамках роботи.

Одним із досліджуваних методів був Occlusion Culling. Його суть полягає у відсіченні об'єктів, які фізично присутні у сцені, але повністю закриті іншими елементами та не потрапляють у поле зору камери. Завдяки цьому графічний процесор не витрачає ресурси на рендеринг невидимих об'єктів, що суттєво зменшує кількість трикутників для обробки та підвищує загальну продуктивність. У практичних умовах застосування Occlusion Culling дозволяє досягти помітного приросту FPS у сценах із великою кількістю статичних або складних моделей.

Результати експериментів показали, що використання визначених методів оптимізації суттєво підвищує продуктивність рендерінгу сцени з великою кількістю об'єктів. Найбільший ефект продемонструвало використання GPU Instancing. Ці методи дозволили збільшити FPS та зменшити час формування кадру на CPU і GPU, а також скоротити кількість батчів і виведених трикутників. Таким чином, впровадження відповідних методів оптимізації дає змогу ефективно рендерити великі масиви об'єктів без втрати продуктивності. Отримані результати мають практичне значення для розробників ігор та 3D-додатків на рушії Unity, оскільки допомагають обрати оптимальні стратегії для забезпечення високої продуктивності рендерінгу у складних сценах.

Важливим висновком дослідження є необхідність балансування навантаження між CPU та GPU. Різні методи оптимізації впливають на різні частини графічного конвеєра: одні знижують навантаження на CPU скорочуючи кількість викликів рендерінгу, тоді як інші методи полегшують роботу GPU, зменшуючи обсяг відображеної геометрії і складність шейдерів. Найбільший приріст продуктивності досягається у випадку поєднання таких підходів, що дозволяє вирівняти використання ресурсів CPU і GPU та запобігає ситуації, коли один з них стає вузьким місцем. Ефективність цього комплексного підходу підтверджено експериментально, що робить його головним для підтримання стабільно високої частоти кадрів у сценах Unity з великою кількістю об'єктів.