Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження методів оптимізації OpenGL та Vulkan
(тема)

Виконав:
студент 2 курсу, групи ІПЗм-22-2

Цапко Б.В.
(прізвище, ініціали)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)
Тип програми освітньо-наукова

Керівник доц. каф. ПІ Чуприна А.С.
(посада, прізвище, ініціали)

Допускається до захисту
Зав. кафедри _____ Дудар З.В.
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерних наук
Кафедра програмної інженерії
Рівень вищої освіти другий (магістерський)
Спеціальність 121 – Інженерія програмного забезпечення
Тип програми освітньо-наукова програма
Освітня програма Інженерія програмного забезпечення
(шифр і назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___» _____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Цапку Богдану Вікторовичу
(прізвище, ім’я, по батькові)

1. Тема роботи «Дослідження методів оптимізації OpenGL та Vulkan»
Затверджена наказом по університету від 29.03.2024р. № 250 Ст
2. Термін подання студентом роботи до екзаменаційної комісії 17.06.2024
3. Вихідні дані до роботи характеристика графічних інтерфейсів OpenGL та Vulkan, характеристика DirectX 11 та 12 для порівняння з наведеними API, порівняльна характеристика менш популярних інтерфейсів для графічного моделювання, опис та реалізація методів і способів оптимізації для підвищення продуктивності
4. Перелік питань, що потрібно опрацювати в роботі
вступ, аналіз предметної галузі та актуальність роботи, постановка задачі, порівняльна характеристика OpenGL та Vulkan з іншими API, аналіз реалізації та використання методів оптимізації, технічні обмеження, тестування розроблених середовищ, аналіз результатів, висновки

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз предметної галузі та постановка задачі	29.02.2024 – 10.03.2024	*виконано*
2	Аналіз інших API і оцінка їх можливостей	11.03.2024 – 24.03.2024	*виконано*
3	Аналіз методів і способів для оптимізації рендерингу	25.03.2024 – 06.04.2024	*виконано*
4	Планування експериментів	07.04.2024 – 11.04.2024	*виконано*
5	Програмна реалізація API OpenGL та Vulkan	12.04.2024 – 26.04.2024	*виконано*
6	Експериментальні дослідження	27.04.2024 – 30.04.2024	*виконано*
7	Аналіз результатів експериментальних досліджень та розробка рекомендацій	01.05.2024 – 04.05.2024	*виконано*
8	Написання та оформлення тез доповіді	05.05.2024 – 15.05.2024	*виконано*
9	Підготовка пояснювальної записки	16.05.2024 – 27.05.2024	*виконано*
10	Підготовка презентації та доповіді	28.05.2024 – 31.05.2024	*виконано*
11	Нормоконтроль	05.06.2024	*виконано*
12	Рецензування	10.06.2024	*виконано*
13	Занесення диплома в електронний архів	13.06.2024	*виконано*
14	Попередній захист	15.06.2024	*виконано*
15	Допуск до захисту у зав. кафедри	16.06.2024	*виконано*

Дата видачі завдання 29 лютого 2024р.

Студент _____ Цапко Б.В.
(підпис)

Керівник роботи _____ доц. каф. ПІ Чуприна А.С.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка містить 85 стор., 55 рис., 6 табл., 20 джерел, 5 додатків.

ВИСОКОРІВНЕВИЙ, ГРАФІКА, НИЗЬКОРІВНЕВИЙ, ОПТИМІЗАЦІЯ, ПРОДУКТИВНІСТЬ, РЕНДЕРІНГ, API, DIRECTX, OPENGL, VULKAN.

Об’єктом дослідження є методи оптимізації графічного моделювання для API OpenGL та Vulkan.

Метою роботи є дослідження методів оптимізації графічного моделювання для API OpenGL та Vulkan для дослідження порівняльної характеристики продуктивності та реалізації схожих методів в обох інтерфейсах.

Методами дослідження є створення відповідного середовища для тестування подібних чи однакових функцій, методів чи способів для виявлення кращої оптимізації за певними критеріями оцінювання з можливістю аналізу цих результатів та створенню порівняльної характеристики.

У результаті роботи було проведено аналіз методів графічного моделювання для API OpenGL та Vulkan, їхню реалізацію та способи відтворення в програмі для їх використання, розроблено порівняльну характеристику схожих методів та їх реалізації, розроблено висновки на основі отриманих результатів.

HIGH-LEVEL, GRAPHICS, LOW-LEVEL, OPTIMIZATION, PERFORMANCE, RENDERING, API, DIRECTX, OPENGL, VULKAN.

The object of research is methods of optimizing graphic modeling for OpenGL and Vulkan APIs.

The aim of the work is to study the optimization methods of graphic modeling for the OpenGL and Vulkan APIs to study the comparative performance characteristics and the implementation of similar methods in both interfaces.

Research methods are the creation of a suitable environment for testing similar or identical functions, methods or ways to identify better optimization according to certain evaluation criteria with the possibility of analyzing these results and creating a comparative characteristic.

As a result of the work, an analysis of graphic modeling methods for API OpenGL and Vulkan, their implementation and methods of reproduction in the program for their use was carried out, a comparative characteristic of similar methods and their implementation was developed, conclusions were drawn based on the results obtained.

Заява щодо самостійного виконання кваліфікаційної роботи та можливості її публікації в електронному архіві відкритого доступу ElArKhNURE.

Я, Цапко Богдан Вікторович, студент гр. ІПЗм-22-2, здобувач вищої освіти на другому (магістерському) рівні кафедри «Програмна інженерія», заявляю: моя кваліфікаційна робота на тему «Дослідження методів оптимізації OpenGL та Vulkan», що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElArKhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений(на) з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

ЗМІСТ

ВСТУП

Для моделювання будь-якої графічної сцени потрібно потужне графічне обладнання, а також API для її розробки. Саме вибір останнього та підхід до його використання грає вирішальну роль у відтворенні графіки, тобто рендерингу графічного обладнання. І відповідає за це саме розробник програми, в якої він хоче використати свої вміння розроблювати графіку.

На сьогодні є безліч графічних API, за допомогою яких можна це зробити, але, як часто буває, прогрес не стоїть на місці, графічні розробники та інженери посилено працюють над вдосконаленням продуктивності та ефективності графічних програм, тому не дивно, що через деякий час один API може переростати в інший, тобто модернізуватися та покращуватися.

Одними із таких прикладів є інтерфейси OpenGL та Vulkan, перший з яких вже поступається другому в кількості можливостей та функціоналу, який можна використати в другому. Та насправді так і є, бо розробниками API Vulkan так і замислювалися, що він стане логічним продовженням OpenGL.

Отже, мета даної дипломної роботи полягає в тому, щоб сприяти подальшому розвитку графічного моделювання, роблячи акцент на оптимізації для OpenGL та Vulkan. Отримані результати стануть цінним ресурсом для розробників графічних інтерфейсів та будуть сприяти подальшій еволюції технологій рендерингу комп'ютерної графіки.

У результаті дипломної роботи було вивчено та порівняно різні підходи до оптимізації графічного моделювання, такі як для визначених методів рендерингу та обробки графічних об'єктів. Методи, такі як використання шейдерів, оптимізація текстур, були ретельно розглянуті та порівняні з точки зору продуктивності та способу їх реалізації в API OpenGL та Vulkan.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Аналіз предметної галузі дослідження

Сьогодні розробники обирають різноманітні інтерфейси програмування застосунків (API) для моделювання графічних моделей чи об’єктів. Велика роль відводиться саме розробнику, бо він пише код, реалізує логіку в програмі та оптимізує її певними функціями або способи, які пропонує API. А якщо вони мають різний, але схожий функціонал, то одне і теж моделювання на різних інтерфейсах може дати різну статистику від рендерингу, тобто швидкість моделювання однієї сцени (кадру) в секунду, зайнятість відеопам’яті тощо.

Таким чином правильно обрати найкращий API, який дає максимальну продуктивність. Але для досягнення цієї мети потрібно докласти більше зусиль, бо такі інтерфейси моделювання комп’ютерної графіки хоч і дають більше продуктивності, але потребують від розробника більше зусиль та розуміння програмного коду, бо вони стають більш низькорівневими.

А що робити розробникам, які вже мають готовий продукт, розроблений за допомогою певного інтерфейсу і хочуть обрати новий через те, що він має кращу оптимізацію? Чи варте це цього, якщо їхній продукт отримай тільки невеличкі покращенні продуктивності при великих витрат час та грошей? А чи потрібно брати важкий для розуміння API для моделювання невеликих сцен або коли просто рендериться звичайне вікно з кнопками? Це питання лежить тільки на плечах розробника(ів), бо тільки вони визначають розміри свого проекту та чи потрібна там висока оптимізація чи ні.

Одним із таких прикладів є два інтерфейси, перший з яких модернізувався в другий, тобто є логічним продовженням першого: OpenGL та Vulkan.

OpenGL (Open Graphics Library) – це кросплатформенний відкритий стандарт для програмування графічних застосунків. Він надає набір функцій для взаємодії з графічним прискорювачем та відтворенням 2D- і 3D-графіки. Випущений 30 червня 1992 року. OpenGL використовується для розробки різноманітних графічних додатків, включаючи відеоігри, симуляції, комп'ютерну графіку, віртуальну реальність і багато іншого. OpenGL є старішою технологією, тоді як

Vulkan є новішою та має на меті забезпечити вищу продуктивність і більше контролю над GPU (графічний процесор).

Vulkan – це кросплатформенний відкритий стандарт для низькорівневого програмування графіки та обчислень. Розроблений групою Khronos Group, Vulkan призначений для надання більшого контролю над графічним прискорювачем, порівняно з попереднім стандартом OpenGL. Випущений 16 лютого 2016 року.

Обидва API мають власний набір плюсів і мінусів, і вибір між ними залежить від конкретних потреб проекту [1].

Розглянемо плюси OpenGL:

– широка підтримка. OpenGL є одним із найстаріших і найпоширеніших графічних API, що означає, що він має широку підтримку від постачальників обладнання та програмного забезпечення. Це полегшує розробку кросплатформних програм, оскільки OpenGL доступний на різних платформах, включаючи Windows, Linux і macOS;
– простота використання. OpenGL має відносно простий і зрозумілий API, що полегшує розробникам початок роботи та навчання. Це може допомогти скоротити час розробки та підвищити продуктивність, особливо для менших проектів або прототипів;
– велика спільнота. Завдяки широкому використанню OpenGL має велику та активну спільноту розробників і користувачів, які можуть надати підтримку та поділитися своїм досвідом. Це може бути цінним ресурсом для розробників, які шукають допомоги чи натхнення.

Тепер розглянемо мінуси OpenGL:

– продуктивність. Хоча OpenGL підходить для багатьох програм, він не такий швидкий і ефективний, як більш сучасні графічні API, такі як Vulkan. Це може бути обмеженням для вимогливих додатків, таких як ігри високого класу, де продуктивність має вирішальне значення;
– обмежений контроль. OpenGL забезпечує менший контроль над графічним процесором, що може бути недоліком для більш складних програм. Наприклад, OpenGL не надає явного контролю над конвеєром

GPU, що може обмежити можливість оптимізації продуктивності та зменшення навантаження на CPU.

Тепер перейдемо до більш низькорівневого API – Vulkan. Розглянемо його плюси:

– висока продуктивність. Vulkan розроблений для високопродуктивних програм і забезпечує більше контролю над конвеєром GPU. Це дозволяє оптимізувати продуктивність і зменшити навантаження на процесор, що є критичним для вимогливих додатків, таких як ігри високого класу [2];
– підтримка різних платформ. Як і OpenGL, Vulkan підтримує широкий спектр платформ, включаючи Windows, Linux і macOS. Це робить можливим розробку кросплатформних програм із стабільною продуктивністю;
– точне керування. Vulkan забезпечує точне керування конвеєром GPU, що дозволяє розробникам оптимізувати продуктивність і зменшити навантаження на процесор. Це дає змогу досягти вищої продуктивності та меншої затримки порівняно з OpenGL.

Щодо мінусів в Vulkan маємо:

– крута крива навчання. Хоча Vulkan забезпечує більше контролю та продуктивності, вона також має крутішу криву навчання порівняно з OpenGL. Це може ускладнити початок і збільшити час розробки, особливо для невеликих проектів або прототипів [3];
– обмежена спільнота. Хоча спільнота розробників і користувачів Vulkan зростає, вона все ще відносно невелика порівняно зі спільнотою OpenGL. Це може ускладнити пошук підтримки та ресурсів, особливо для розробників, які тільки починають.

Отже, Vulkan є більш низькорівневим та ефективним графічним API порівняно з OpenGL. Він забезпечує більший контроль розробникам, оптимізований доступ до апаратного забезпечення, підтримку мультипотоковості, що дуже важливо на сьогоднішній час, та обчислювальних завдань, що робить його

сприятливим вибором для сучасних графічних застосунків, особливо в ігровій індустрії та області віртуальної реальності [4].

А це не означає, що OpenGL більше нікому не потрібен і його повністю замінив Vulkan? Насправді ні, OpenGL не мертвий. Хоча це старіша технологія, вона все ще широко використовується в багатьох програмах і має велику й активну спільноту розробників і користувачів. Насправді багато галузей, наприклад автомобільна, аерокосмічна та наукова візуалізація, все ще покладаються на OpenGL для своїх графічних потреб.

OpenGL розвивався протягом багатьох років, щоб не відставати від прогресу графічного обладнання та програмного забезпечення. Остання версія OpenGL, OpenGL 4.6, надає широкий спектр функцій для сучасних графічних програм і підтримує найновіше апаратне забезпечення та операційні системи.

У той час як нові графічні API, такі як Vulkan, були представлені для забезпечення вищої продуктивності та більшого контролю над графічним процесором, OpenGL продовжує залишатися популярним вибором для багатьох програм завдяки простоті використання, широкій підтримці та великій спільноті.

Таким чином, хоча OpenGL не може бути найновішою чи найпередовішою технологією, вона все ще дуже жива та здорова, і, ймовірно, продовжуватиме використовуватися в багатьох програмах у доступному для огляду майбутньому.

Також, потрібно відмітити про окрему версію OpenGL, яка не буде розглядатися для дослідження, але варта уваги, бо спеціалізована спеціально для вбудованих систем – OpenGL ES (OpenGL for Embedded Systems). Вона представляє собою легковагу версію стандарту OpenGL, спеціально призначену, як вже було сказано, для вбудованих систем, зокрема мобільних платформ. Цей графічний інтерфейс визначає обмежений набір функцій порівняно з повноцінним OpenGL, що спрощує взаємодію з графічним апаратним забезпеченням на ресурсообмежених пристроях. OpenGL ES широко використовується на мобільних платформах, таких як Android та iOS, і служить стандартом для реалізації графічних можливостей мобільних додатків та ігор. Основною його особливістю є низький рівень деталізації, спрощений інтерфейс для взаємодії з графічним

апаратним забезпеченням, що робить його оптимальним для ресурсообмежених пристроїв. Цей стандарт широко підтримується і використовується багатьма розробниками для створення графічних додатків та ігор для різноманітних мобільних пристроїв, що дозволяє створювати універсальні та ефективні додатки для ринку мобільних технологій. Але сьогодні багато компаній відмовляються від нього на користь інших API, наприклад, того ж Vulkan [5].

Тому було б дуже круто мати порівняльну характеристику результатів продуктивності графічного моделювання та реалізації методів та способів оптимізації для підвищення продуктивності від двох схожих API.

1.2 Актуальність роботи

На тлі постійної еволюції графічних технологій, моделювань та підходів до їхньої реалізації, у світі, де рендерінг стає все більше важливим компонентом різноманітних програмних застосунків, питання продуктивності та вибору відповідного графічного API стає все більш і більш актуальним.

Якщо розглядати проблему з одного боку, ми спостерігаємо за вражаючим розвитком API, таких як Vulkan, які пропонують високий рівень ефективності та контролю для розробників графічних програм. А з іншого боку, традиційні технології, такі як OpenGL, продовжують існувати та вдосконалюватися, забезпечуючи простоту використання та велику спільноту користувачів.

Актуальність роботи полягає в тому, щоб дослідити, які з цих API краще відповідають потребам різних сфер, враховуючи їхню продуктивність та зручність використання. Розробка порівняльної характеристики та аналіз методів для виявлення оптимізації відкриє нові можливості для розробників, допомагаючи їм зробити обґрунтований вибір у виборі графічного API залежно від конкретних завдань та вимог їхнього проекту або допоможе тим розробникам, які мають готовий продукт, розроблений за допомогою інтерфейсу OpenGL, чи обирати їм Vulkan для підвищення продуктивності програми чи ні.

Отже, дана робота є важливим кроком у розумінні і актуалізації області графічного моделювання на прикладі API OpenGL та Vulkan, надаючи цінний

внесок у вибір оптимального інструментарію для розробників у сучасному світі комп'ютерної графіки.

1.3 Постановка задачі

Метою даної дипломної роботи є дослідження методів оптимізації графічного моделювання для API OpenGL та Vulkan для дослідження порівняльної характеристики продуктивності та реалізації схожих методів в обох API, тобто дослідити акцент на виборі між двома важливими графічними інтерфейсами – OpenGL та Vulkan. Це особливо актуально, оскільки останній вже проявляє переваги в кількості можливостей та функціоналу порівняно з першим.

Постановка задачі ґрунтується на важливості вибору оптимального графічного API для розробки програм з високою продуктивністю графічного моделювання. Вона включає перед собою завдання дослідити та порівняти різні підходи до оптимізації графічного моделювання, зокрема методи рендерингу та обробки графічних об'єктів. Серед розглянутих методів важливе місце займають використання шейдерів, оптимізація текстур та буферів.

Для інтерфейсу OpenGL буде розглядатися:

– шейдери в OpenGL. Використання шейдерів у OpenGL є ключовим аспектом оптимізації. Дослідження включає в себе аналіз різних видів шейдерів, таких як вершинний, фрагментний та геометричний, та їхній вплив на продуктивність;
– оптимізація текстур в OpenGL. Вивчення методів оптимізації роботи з текстурами, таких як міп-мапи [6], асинхронна передача;
– робота з буферами в OpenGL. Аналіз оптимальних підходів до роботи з буферами даних, включаючи Vertex Buffer Objects (VBO) та Uniform Buffer Objects (UBO), та їх вплив на швидкодію програми.

Аналогічно буде проведено і для інтерфейсу Vulkan:

– низькорівневий доступ та мультипотоковість. Дослідження можливостей Vulkan забезпечити низькорівневий доступ до апаратного забезпечення,

що дозволяє розробникам більший контроль над оптимізацією та ефективністю;

– оптимізація роботи з текстурами в Vulkan. Аналіз підходів до оптимізації текстур в Vulkan, таких як робота з Image Views та Sampler-ами, а також використання додаткових функцій, доступних у цьому API;
– робота з буферами в Vulkan. Дослідження використання буферів, зокрема Uniform та Vertex буферів, та їхній вплив на продуктивність графічних програм.

Критерії оцінки продуктивності для виявлення кращого оптимізованого методу рендерингу такі:

– завантаженість графічного процесора. Відображає, наскільки ефективно метод рендерингу використовує можливості GPU. Надмірне завантаження може призвести до перегріву та тротлінгу;
– завантаженість відеопам’яті. Вказує на обсяг використовуваної відеопам'яті для текстур і буферів. Перевантаження VRAM може викликати затримки через переміщення даних між відеопам’яттю і оперативною пам’яттю;
– завантаженість центрального процесора. Оцінює навантаження на CPU під час рендерингу. Високе завантаження CPU може сповільнювати інші процеси в системі;
– завантаженість оперативної пам’яті. Показує, скільки оперативної пам'яті використовується для рендерингу;
– кількість кадрів за секунду. Швидкість обробки графічної інформації. Високе значення забезпечує плавність графіки, тоді як низьке може свідчити про вузькі місця в процесі рендерингу.

Отримані результати дослідження будуть важливим ресурсом для розробників графічних інтерфейсів, допомагаючи їм робити оптимальні вибори при розробці програм з використанням OpenGL та Vulkan. Крім того, робота сприятиме подальшому розвитку технологій рендерингу комп'ютерної графіки та оптимізації їх продуктивності.

2 АНАЛІЗ ІСНУЮЧИХ АНАЛОГІВ ТА МЕТОДІВ ОПТИМІЗАЦІЇ

2.1 Порівняння API OpenGL та DirectX

Якщо розглянути часові випуски OpenGL та його аналоги на момент використання після випуску, то твердо можна сказати, що його головним аналогом є DirectX. Будемо розглядати версію DirectX 11, бо минулі хоч і можна теж порівнювати, але, скоріше за всього, саме 11 була аналогом OpenGL по графічному функціоналу, який досить схожий в обох версіях. А наступну, тобто 12 версію, з її можливостями краще порівнювати з API Vulkan, бо якщо розглянути останній як логічне продовження OpenGL та як оновлення, то вже 12 буде схожа по функціоналу як Vulkan, але тут буде більше розбіжностей, про які ми поговоримо пізніше.

По суті, і DirectX, і OpenGL використовують один і той самий графічний конвеєр протягом багатьох років. Обидва API використовують точки, які називаються вершинами, для створення складних графічних примітивів. Кожна вершина зберігає власні координати та іншу важливу інформацію. Але кожна бібліотека обробляє вершини по-своєму, наприклад, ви знайдете змішування вершин у DirectX, але не в OpenGL, кілька операційних систем є в OpenGL [7].

Вони можуть виконувати подібні функції, але ви помітите різницю між OpenGL і DirectX 11, якщо попрацюєте з ними. Тому будемо розглядати їх за окремими пунктами, щоб більш помітити різницю.

Область застосування API. Важливо зазначити, що хоча обидва вони є графічними API, OpenGL і DirectX мають різні сфери застосування. OpenGL – це чистокровний графічний API, і це означає, що він не працює ні з чим іншим, крім 2D і 3D-графіки. А DirectX – це набір графічних, аудіо, мережевих та інших апаратних API, які працюють разом. Щоб доповнити цю різницю, використовують інші бібліотеки, наприклад, OpenAL, яка спеціалізується на використанні аудіо. Але далі будемо ігнорувати цю різницю між двома API, зосереджуючись на графічних елементах кожного інструменту.

Платформи. OpenGL і DirectX представляють два різні підходи до графічного програмування, і вибір між ними залежить від ряду факторів. OpenGL – це

відкритий та безкоштовний API, розроблений Khronos Group, що надає кросплатформенну підтримку, охоплюючи різні операційні системи, включаючи Linux, Nintendo Switch і Playstation 5. З іншого боку, DirectX, розроблений Microsoft, спеціалізується на платформах Windows і Xbox, що робить його зручним для розробки ігор та додатків для цих систем. Важливо враховувати ці відмінності для платформ, обираючи між ними в залежності від конкретних потреб та цільової аудиторії розробки.

Апаратні ресурси. Вони грають ключову роль в графічному програмуванні, і в цьому контексті різниця між DirectX і OpenGL також відображається. DirectX дозволяє розробникам ефективно керувати апаратним забезпеченням, розподіляючи ресурси за потребою. Це дає більше контролю та гнучкості щодо оптимізації використання апаратних можливостей. З іншого боку, OpenGL теж взаємодіє з апаратним забезпеченням, але він бере на себе частину управління ресурсами, зменшуючи турботи розробника про деякі аспекти управління. Це може бути вигідним в тому випадку, якщо розробнику бажано зосередитися на інших аспектах програмування без глибокого втручання в керування ресурсами. Таким чином, вибір між ними в даному контексті залежить від конкретних потреб та уподобань розробника.

Документація. Документація важлива для ефективної розробки ігор, і в порівнянні між DirectX і OpenGL, обидва API надають доступ до детальної і регулярно оновлюваної документації. DirectX має обширну документацію, яка охоплює всі аспекти використання цього API. Крім того, Microsoft надає широкі можливості для підтримки розробників, включаючи онлайн-ресурси та форуми спільноти. З іншого боку, OpenGL, яке керується некомерційною організацією Khronos Group, також має докладну документацію, яка описує різні аспекти використання цього API. Вона прагне підтримувати спільноту розробників, і вони регулярно оновлюють свою документацію, враховуючи нові можливості та вдосконалення. Тому обидва ці API є динамічними і активно підтримуються, що забезпечує розробникам актуальну та зрозумілу інформацію для успішного використання їх функціоналу.

Сумісність з рушієм. Сумісність з рушієм є важливим аспектом для багатьох розробників ігор, оскільки більшість з них використовують рушії, такі як Unity та Unreal Engine, для створення своїх проектів. Ці рушії визначають стандарти та можливості для розробки ігор, і обидва підтримують як OpenGL, так і DirectX. Unity, як один із найпопулярніших рушіїв, дозволяє розробникам вибирати між різними графічними API, включаючи як OpenGL, так і DirectX. Це надає розробникам можливість адаптувати свої ігри під конкретні потреби та обмеження платформи. Unreal Engine, ще один впливовий рушій, також підтримує обидва графічні API. Це робить Unreal Engine гнучким та адаптованим до потреб розробників, які мають можливість вибрати оптимальне API залежно від конкретних вимог їх проекту. Таким чином, розробники можуть вибирати між OpenGL і DirectX в залежності від їхніх уподобань та вимог проекту, забезпечуючи широкі можливості для роботи на різних платформах.

Простота використання. Вона є важливим аспектом для розробників ігор, і тут можна врахувати ряд ключових моментів. Спочатку слід відзначити, що деякі експерти стверджують, що робота з OpenGL є легшою, ніж з DirectX. Це пояснюється тим, що OpenGL спрощує завдання розробки, обмежуючись лише графікою, що робить його більш доступним і простим для розуміння розробникам. Також, важливим фактором є те, що OpenGL є безкоштовним API, яким керує некомерційна організація Khronos Group. Як вже було сказано, це робить його більш універсальним і підходить для розробників, які працюють на різних платформах. Проте, важливо відзначити, що простота використання графічного API не така вирішальна, як раніше, завдяки поширенню рушіїв, таких як Unity та Unreal Engine. Ці рушії забезпечують абстракцію від графічних API, дозволяючи розробникам просто змінювати використовуване API з легкістю. Отже, якщо розробник працює з високорівневими інструментами, різниця в простоті використання між OpenGL і DirectX може стати менш важливою.

Тести продуктивності. Є поширена думка, що OpenGL працює краще, ніж DirectX, але часто останній API досягає кращих результатів в продуктивності. Також, потрібно врахувати, що це залежить від багатьох факторів, а часто

використовувані для порівняння тести, які можна знайти у вільному доступі, порівнюють невеликі проекти, тому зі збільшенням розміру (складності) цей фактор може змінитися.

Тож, підбиваючи підсумки, вибір між OpenGL та DirectX залежить від конкретних потреб проекту та власних вподобань розробника програми. OpenGL є більш мультиплатформеним та може бути сприятливим для простих застосувань, тоді як DirectX, зокрема, застосовується для Windows і Xbox і надає більше гнучкості та контролю, що може бути корисним для великих та більш складних проектів. Але, якби там не було, обидва до сих пір API залишаються актуальними з можливістю використання в різних галузях розробки ігор та графічних додатків.

2.2 Порівняння API Vulkan та DirectX

Як вже було згадано, для порівняння з API Vulkan буде розглядатися DirectX 12, бо він має схожий функціонал та можливості.

Vulkan і DirectX – API і пропонують два різні рішення для однієї проблеми: гарантувати, що програмне забезпечення, наприклад відеоігри, може спілкуватися з апаратним забезпеченням вашого ПК, і навпаки, найефективнішим способом.

Так само, як і з OpenGL, будемо розглядати порівняння по певним пунктам, щоб краще оцінити їх можливості [8].

Платформа. Платформна підтримка є важливим аспектом вибору між графічними API, такими як Vulkan і DirectX. Vulkan є платформонезалежним і розробленим для крос-платформеного використання, підтримуючи різні операційні системи, включаючи Windows, Linux та Android. Його універсальність дозволяє розробникам створювати програми для різних пристроїв. Натомість DirectX головним чином спрямований на платформи Microsoft, такі як Windows і Xbox, що робить його менш універсальним для крос-платформеного розроблення. Розробники повинні враховувати цей фактор при виборі API в залежності від платформи свого проекту.

Рівень абстракції. Він є важливим критерієм при порівнянні Vulkan і DirectX. Vulkan, як API низькорівневого рівня, надає розробникам великий контроль над

графічним процесором і апаратним забезпеченням. Це робить його потужним інструментом для оптимізації продуктивності, але вимагає більше коду і ресурсів. З іншого боку, DirectX надає вищий рівень абстракції, що полегшує розробку, але при цьому обмежує контроль над окремими аспектами графічного процесу [9]. Вибір між низькорівневим і високорівневим підходами залежить від потреб конкретного проекту та вмінь розробників. Звичайно вискорівневий підхід буде потрібен для невеличких проектів та програм, а низькорівневий – для проектів, які серйозно залежать від системи, на якій вони будуть запущені та використані.

Простота використання. Вона має свої певні відмінності. DirectX, розроблене Microsoft, зазвичай вважається менш складним у використанні, особливо для розробників, які спеціалізуються на платформі Windows. Це API надає вищий рівень абстракції та забезпечує зручний інтерфейс для розробки графічних застосунків. Натомість Vulkan, створене групою Khronos, славиться своєю низькорівневістю та більшою кількістю контролю, що може здаватися складнішим для вивчення та використання, особливо для початківців. Проте ця складність дозволяє розробникам отримати більший контроль над апаратними ресурсами та оптимізацією продуктивності. Таким чином, обираючи між цими API, розробникам слід враховувати свій рівень досвіду, конкретні потреби проекту та вподобання в щодо рівня контролю над системою.

Документація. Вона відіграє важливу роль у розробці програм, і вона є ключовим аспектом при виборі між API Vulkan та DirectX. Обидва API, випущені Khronos Group та Microsoft відповідно, надають високоякісну та детальну документацію, яка стає важливим ресурсом для розробників. DirectX включає офіційні документи, приклади коду, туторіали та широку спільноту розробників, яка може допомогти вирішити питання та поділитися досвідом. З іншого боку, Vulkan також має докладну специфікацію та документацію, яка дозволяє розробникам краще розуміти його низькорівневі можливості. Обидва API регулярно оновлюють свою документацію, щоб відображати нові функції та зміни. Розробники можуть користуватися цією детальною інформацією для ефективної

розробки, усунення неполадок та вдосконалення продуктивності своїх графічних застосунків.

Мультипотоковість. Вона відіграє важливу роль у підвищенні продуктивності графічних додатків. Інтерфейс Vulkan активно підтримує мультипотоковість та надає розробникам більший контроль над паралельним виконанням завдань. Обробка команд може відбуватися асинхронно, дозволяючи використовувати багатоядерні процесори для виконання різних завдань, таких як рендеринг та обчислення, паралельно [10]. DirectX також підтримує мультипотоковість, але у випадку DirectX 12 була внесена значна робота для поліпшення підтримки багатоядерних систем. Додаткові можливості багатозадачності були додані для кращого розподілу завдань між потоками та використання ресурсів апаратного забезпечення.

Екосистема інструментів. Інструментарій інтерфейсу відіграє важливу роль у виборі між API Vulkan та DirectX для розробників графічних застосунків. DirectX надає широкий спектр інтегрованих інструментів, зокрема Visual Studio та DirectX Graphics Tools, що полегшує розробку та налагодження графічних програм. У той час як Vulkan не має такого рівня інтеграції з інструментами, які надає Microsoft, він компенсує це вищою крос-платформеністю, що робить його привабливим вибором для розробників, які працюють на різних операційних системах та платформах.

Отже, можемо зробити висновок, що вибір між API Vulkan та DirectX залежить від конкретних потреб розробника та особливостей проекту. DirectX надає високий рівень абстракції та інтегровані інструменти для зручної розробки на платформі Windows, але обмежує крос-платформеність. З іншого боку, Vulkan вигідний для більшої контрольованості, ефективності та крос-платформеності, але вимагає більше ручного управління. Для розробників, які прагнуть високої продуктивності та контролю над апаратними ресурсами та враховують крос-платформенність, Vulkan може бути більш привабливим вибором. У той час як DirectX залишається важливим для розробників, які спеціалізуються на Windows та Xbox і використовують інтегровані інструменти для зручної розробки. У кожного

API є свої переваги, і вибір залежить від конкретного контексту та потреб розробника.

2.3 Порівняння API OpenGL та Vulkan з іншими API

Окрім популярного DirectX, існують й інші API, які є досить популярними серед користувачів: наприклад, Metal, WebGL та інші. Ми розглянемо лише перші дві інтерфейси.

Metal – це графічний API, призначений для платформ Apple, який забезпечує високу продуктивність і розширені можливості для розробки на macOS та iOS. Він надає розробникам чудовий рівень контролю і може найкращим чином використовувати апаратні ресурси пристроїв Apple. Однак основним обмеженням Metal є те, що він орієнтований лише на пристрої Apple та їхні операційні системи. Для порівняння, OpenGL і Vulkan є універсальними, крос-платформними рішеннями, що надають широкі можливості для розробки графіки на різних платформах, включаючи Windows, Linux та інші операційні системи [11].

WebGL та OpenGL/Vulkan є графічними API, але в різних контекстах використання. OpenGL і Vulkan призначені для нативної графіки на комп'ютерах та інших пристроях, тоді як WebGL – це JavaScript API для відображення 3D і 2D графіки у веб-браузері без необхідності встановлення додаткових плагінів. WebGL базується на OpenGL ES, спеціальній версії OpenGL для мобільних платформ, і призначений для реалізації графічних можливостей у веб-середовищі, забезпечуючи високу сумісність і доступність для розробників в Інтернеті [12].

OpenGL та Vulkan використовуються в ігровій індустрії та в декстопних графічних додатках. З точки зору продуктивності та ефективності, Vulkan виділяється завдяки високому рівню продуктивності та низькорівневим функціям, в той час як OpenGL та інші веб-інтерфейси призначені для менш ресурсоємних завдань і простіші у використанні. Завдяки широкому спектру застосувань OpenGL використовується в різних галузях, включаючи аерокосмічну та наукову візуалізацію, тоді як Vulkan надають перевагу ігровій та графічній індустріям. На рівні специфікацій та екосистеми OpenGL має широку підтримку та активну

спільноту розробників, в той час як Vulkan має на меті надати розробникам більше контролю, з такими передовими функціями.

WebGL призначений для веб-графіки, що запускається у веб-браузері, спрощуючи інтеграцію графіки у веб-середовище [13]. Вибір цього інтерфейсу залежить від конкретних потреб, платформ і вимог до продуктивності проекту. OpenGL залишається придатним для багатьох додатків, Vulkan підходить для високопродуктивних графічних додатків, тоді як WebGL спрощують розробку графіки у веб-середовищі.

2.4 Аналіз методів для оптимізації рендерингу

Розглянемо та проаналізуємо методи оптимізації графічних моделей для тестування продуктивності системи при реалізації конкретних сцен в API OpenGL та Vulkan [14].

Використання шейдерів, що представлені невеликими програми, що виконуються на графічному процесорі для обробки графіки та візуальних ефектів, ми будемо використовувати 3 основних типів шейдерів: вершинний, фрагментний і геометричний. В OpenGL використовується мова GLSL (OpenGL Shading Language), тоді як в Vulkan можна використовувати будь-яку мову, бо він має функціонал, який переводить цю мову в код SPIR-V, який є проміжним представленням шейдерів, бо він є продуктивнішим і ефективнішим, оскільки забезпечує кращу оптимізацію, переносимість та швидкість компіляції [15].

Оптимізація текстур в обох інтерфейсах за допомогою міп-мапів є фундаментальною хімічною основою стратегій оптимізації. Це дозволяє зменшити обсяг пам'яті та підвищити швидкість передачі, при цьому не втративши якість текстури.

OpenGL та Vulkan використовують схожі концепції, такі як VBO (Vertex Buffer Object) та UBO (Uniform Buffer Object), для ефективної передачі та використання буферизованих даних. Це дозволяє оптимально обробляти графічні об'єкти. VBO використовується для зберігання вершинних даних, таких як позиції, нормалі та текстурні координати, тоді як UBO зберігає такі дані, які є спільними

для багатьох шейдерів, наприклад, матриці трансформацій. Крім цього, обидва API підтримують використання індексованих буферів (IBO або EBO) для ефективного управління вершинами через індекси, а також інстансинг, який дозволяє рендерити багато копій об'єкта з різними трансформаціями з мінімальними витратами обчислювальних ресурсів [16].

Vulkan забезпечує низькорівневий доступ і гарантує ефективне виконання команд та апаратних операцій. Крім того, він використовує можливості багатопоточності для розпаралелювання обчислень і підвищення ефективності.

У Vulkan використання представлень зображень і зразків гарантує оптимальний доступ до текстур і їх використання. Використання специфічних функцій Vulkan спрямоване на підвищення продуктивності при роботі зі структурами. У OpenGL текстури обробляються через шейдерні програми, що дозволяє виконувати складні графічні обчислення. На відміну від OpenGL, який використовує монолітний підхід для рендерингу, Vulkan використовує графічні слої, що забезпечує більш гнучке і контрольоване управління рендерингом. Це дозволяє Vulkan досягати вищої продуктивності за рахунок більш точного контролю над апаратними ресурсами.

Фреймбуфери в OpenGL і Vulkan дозволяють виконувати рендеринг у текстури або інші буфери, що дає змогу реалізовувати різні постобробні ефекти, такі як розмиття, тіні та відображення [17]. Це досягається шляхом створення власного фреймбуфера, який замінює стандартний фреймбуфер екрану. Після рендерингу в цей буфер, згенеровану текстуру можна використовувати для подальших обчислень або відображення.

Інстансинг дозволяє рендерити багато копій одного й того ж об'єкта з різними трансформаціями та параметрами, використовуючи один виклик рендерингу. Це дуже ефективно для рендерингу великих сцен з великою кількістю однакових або схожих об'єктів, наприклад, дерев у лісі або будівель у місті. Інстансинг зменшує навантаження на CPU і дозволяє ефективно використовувати ресурси GPU.

Ці оптимізації та приклади коду важливі для досягнення оптимальної продуктивності та ефективності графічних додатків у відповідному API.

3 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

3.1 Концепція спільних налаштувань

Для того, щоб чітко провести дослідження спільних методів на наявність кращих показників оптимізації, потрібно встановити спільні параметри для рендерингу, використовувати однакові об’єкти, текстури тощо. Єдиною різницею повинно бути тільки спосіб реалізації та написання коду для різних інтерфейсів.

Почнемо з першого параметру, який ми повинні врахувати в обох випадках – це використання шейдерів. Так як команди для використання в двох інтерфейсах різні, хоча мають спільний код, якщо не враховувати, що він компілюється по-різному, то вони повинні мати максимально спільні параметри, які можна змінювати, вершинні рівні тощо, але головна увага приділяється функції main(), яка є головною та в якій знаходиться вся логіка певного шейдера. Так як ми пишемо код на мові GLSL для обох інтерфейсів, то різниця буде тільки в способі отримання певних даних для використання в шейдері (див. рис. 3.1, 3.2).

```
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoords;

out vec3 FragPos;
out vec3 Normal;
out vec2 TexCoords;

layout (std140) uniform Matrices
{
    mat4 projection;
    mat4 view;
};
uniform mat4 model;
uniform mat3 normModel;

void main()
{
    FragPos = vec3(model * vec4(aPos, 1.0));
    Normal = normModel * aNormal;
    TexCoords = vec2(aTexCoords.x, 1 - aTexCoords.y);

    gl_Position = projection * view * vec4(FragPos, 1.0);
}
```

Рисунок 3.1 – Вершинний шейдер освітлення для OpenGL (рисунок виконаний самостійно)

```
layout(binding = 0) uniform UniformBufferObject {
    mat4 view;
    mat4 proj;
} ubo;

layout(push_constant) uniform PushConstants {
    mat4 model;
    vec3 viewPos;
} pushConstants;

layout(location = 0) in vec3 inPosition;
layout(location = 1) in vec3 inNormal;
layout(location = 2) in vec2 inTexCoord;

layout(location = 0) out vec3 FragPos;
layout(location = 1) out vec3 Normal;
layout(location = 2) out vec2 TexCoords;

void main() {
    FragPos = vec3(pushConstants.model * vec4(inPosition, 1.0));
    Normal = mat3(transpose(inverse(pushConstants.model))) * inNormal;
    TexCoords = vec2(inTexCoord.x, 1 - inTexCoord.y);

    gl_Position = ubo.proj * ubo.view * vec4(FragPos, 1.0);
}
```

Рисунок 3.2 – Вершинний шейдер освітлення для Vulkan (рисунок виконаний самостійно)

Наступним параметром є згладжування. Ми будемо використовувати згладжування MSAA, яке легко налаштовується і в OpenGL, і в Vulkan.

Відсікання нелицьових гранів також повинно бути присутнім, бо в практичному використанні всі програми використовують його, бо воно дає легку можливість зекономити ресурси.

Через те, що відтворення об’єктів в середовищі повинно бути максимально однаковим, ми повинні також враховувати і середовище, в якому воно буде виконуватися. Мова йде саме про Windows, тому для його використання будемо використовувати бібліотеку GLFW.

Розширення екрану, кількість об’єктів для інстансованого рендерингу, вертикальна синхронізація, тести глибини та змішування кольорів також повинно бути однаковим для обох створених програм. А дані об’єктів повинні бути вже завантажені в програмі перед рендерингом, незважаючи на те, що буде створюватися. Так ми зможемо чітко оцінити завантаженість системи з різними буферами, і різницею додатків будуть тільки відповідні команди інтерфейсів.

Також потрібно додати можливість переміщуватися, щоб можна було оглядати об’єкт та перевіряти якість рендерингу.

3.2 Спосіб проведення дослідження

Отримання результатів буде проводитися за допомогою програми MSI Afterburner та RivaTunerStatisticsServer [18]. Вони надають можливість легкого налаштування та отримання даних з процесора, відеокарти, оперативної пам’яті тощо, тому це ідеальні додатки для проведення нашого дослідження (див. рис. 3.3).

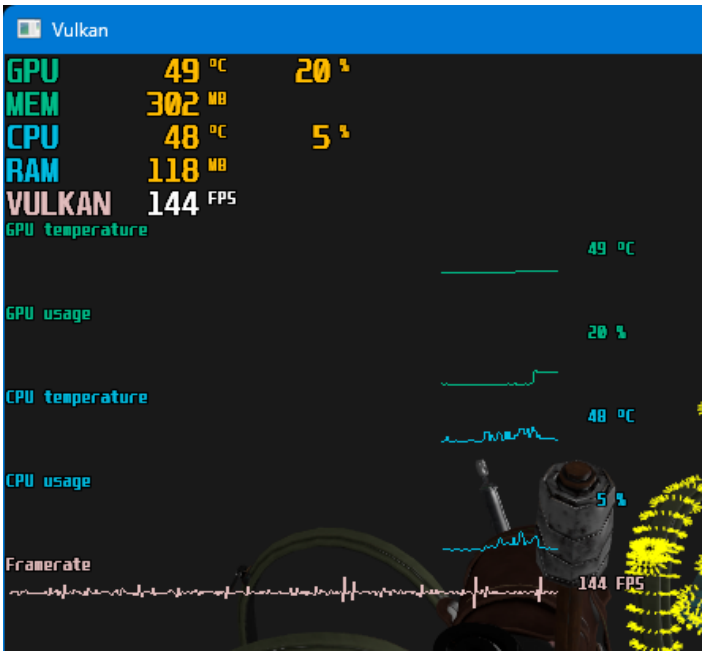

Рисунок 3.3 – Приклад використання RivaTunerStatisticsServer (рисунок виконаний самостійно)

Базовим предметом для тестування стане рюкзак для виживання [19], а для інстансованого рендерингу – планета Марс [20].

Для тестування шейдерів ми створимо декілька, один з яких симулює освітлення за моделлю Блінна-Фонга, два інших потрібні для тестування геометричного шейдера, які звичайно відрізняються між собою, інші потрібні для інстансованого рендерингу чи для інших потреб з мінімальним навантаженням.

Відмітимо, що будемо використовувати OpenGL 4.6, GLSL 4.5, Vulkan 1.3.275. Конфігурація системи, на якій розроблялась програма та виконувались тести, містить процесор Intel i7-12650H, оперативну пам’ять DDR5 4800МГц 16 ГБ, відеокарту NVIDIA RTX 4060 Laptop, операційну систему Windows 11.

4 ОПИС ЕКСПЕРЕМЕНТАЛЬНИХ ДОСЛІДЖЕНЬ

4.1 Підготовка до проведення дослідження

Після встановлення основних вимог до додатків, а також критеріїв оцінки для виявлення кращою оптимізації, можемо перейти до замірів цих критеріїв. Але перед цим визначимо ще 2 додаткових параметри для дослідження та які віртуальні сцени будемо відтворювати, щоб отримати потрібні нам показники для кожного з досліджуваних частин системи.

Додатковими параметрами є загальна температура для центрального процесора (не для кожного ядра) та для відеокарти. Ці параметри залежать від їх навантаження та системи охолодження, тому на кожній системі вони можуть відрізнятися. Але так як ми будемо проводити тестування на одній системі, а отже система охолодження буде єдиною для всіх тестів, тому параметри будуть залежати від навантаження на ці пристрої. Але насправді зі збільшенням температури часто система охолодження починає працювати сильніше, щоб температура не піднімалася вище заданого числа, тому точно визначити, що критерій температури буде залежати тільки від навантаження дуже важко, якщо не виключати систему охолодження або встановити для неї сталу частоту її охолодження. Але так як ці параметри є додатковими, вони не грають визначну роль в нашому дослідженні, тому вони існують тільки для кращого розуміння результатів тестів. Отже, ми можемо знехтувати невеликими коливаннями температури та будемо приділяти увагу тільки значним показникам різниці між ними.

Віртуальні сцени – це сцени з певною кількістю об’єктів та відповідними налаштуваннями програми, наприклад, згладжуванням, та шейдерами. Тому щоб провести наше дослідженням ми розіб’ємо наші досліджувані параметри для певних об’єктів і будемо відтворювати тільки ті, які нам потрібні. Тобто кожна сцена буде містити тільки ті шейдери, налаштування та предмети, які потрібні, щоб отримати дані для порівняння оптимізації для двох інтерфейсів і не більше, а також, як вже було сказано раніше, порівнювані сцени повинні бути однаковими.

Ми створили 10 різних сцен, але тестів буде 20 для кожного інтерфейсу, бо кожна сцена буде тестуватися з вертикальної синхронізацією (обмеження в 144

кадра за секунду) та без неї. Це потрібно, щоб перевірити, як система реагує на максимальне навантаження від команд інтерфейсу.

Отже, сцени, які будуть потрібні для проведення нашого дослідження такі:

– шейдер освітлення, 2 предмета, згладжування 8x;
– геометричний шейдер, 1 предмет, згладжування 8x;
– інший геометричний шейдер, 1 предмет, згладжування 8x;
– комбінація з 2 геометричних шейдерів, 2 предмета, згладжування 8x;
– комбінація з 1 шейдера освітлення та 2 геометричних, 4 предмета, згладжування 8x;
– шейдер для виведення 3000 планет, згладжування 8x;
– інстансований шейдер для виведення 3000 планет, згладжування 8x;
– комбінація з 1 шейдера освітлення, 2 геометричних, 1 інстансованого, 3000 планет і 4 рюкзака, згладжування 8x;
– комбінація з 1 шейдера освітлення, 2 геометричних, 1 інстансованого, 3000 планет і 4 рюкзака, без згладжування;
– пусте середовище.

Але потрібно провести ще 12 тестів для кожного інтерфейсу. Це тести з комбінацією із 1 шейдера освітлення, 2 геометричних, 1 інстансованого та зі згладжуванням 8x. Але відрізнятися вони будуть від попереднього тим, що ми будемо захвачувати екраном всі об’єкти на сцені, половину планет та рюкзаків і залишимо тільки останні на екрані. Це потрібно, щоб перевірити, як інтерфейси справляються з об’єктами, яких не видно на екрані, але вони існують у віртуальному середовищі.

Для того, щоб правильно розподілити отриманні дані, нам потрібно створити таблицю і навести коротке пояснення до назв стовпців: VSync – вертикальна синхронізація, GPU T – температура на графічному процесорі в °C, GPU – завантаженість графічного процесора в %, GPU M – завантаженість відеопам’яті в MB, CPU T – температура на центральному процесорі в °C, CPU – завантаженість центрального процесора в %, RAM – завантаженість оперативної пам’яті в MB, FPS – кількість кадрів за секунду.

4.2 Проведення експериментів для інтерфейсу OpenGL

Почнемо з першого експерименту – шейдер освітлення зі згладжуванням 8x, включеною та вимкненою вертикальною синхронізацією (див. рис. 4.1, 4.2).

Рисунок 4.1 – Шейдер освітлення з вертикальною синхронізацією для OpenGL (рисунок виконаний самостійно)

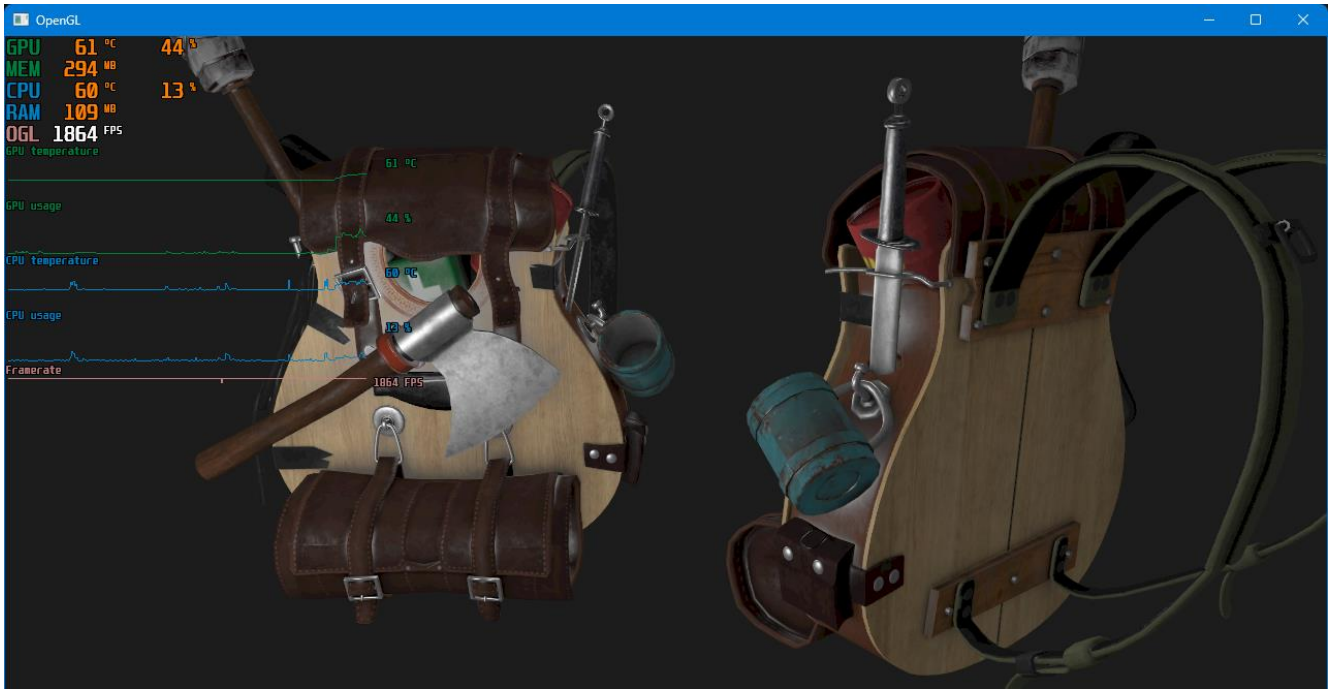

Рисунок 4.2 – Шейдер освітлення без вертикальної синхронізації для OpenGL (рисунок виконаний самостійно)

Наступним експериментом є геометричний шейдер зі згладжуванням 8x, включеною та вимкненою вертикальною синхронізацією (див. рис. 4.3, 4.4).

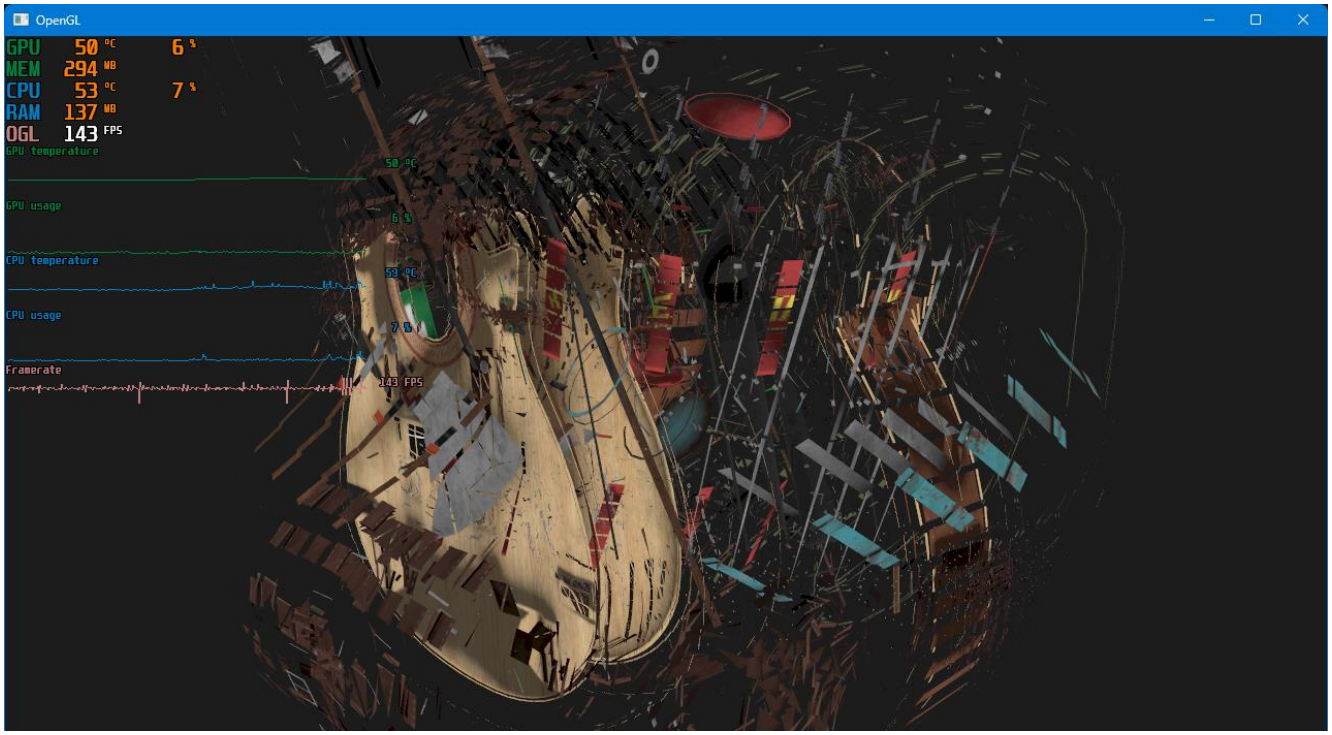

Рисунок 4.3 – Геометричний шейдер з вертикальною синхронізацією для OpenGL (рисунок виконаний самостійно)

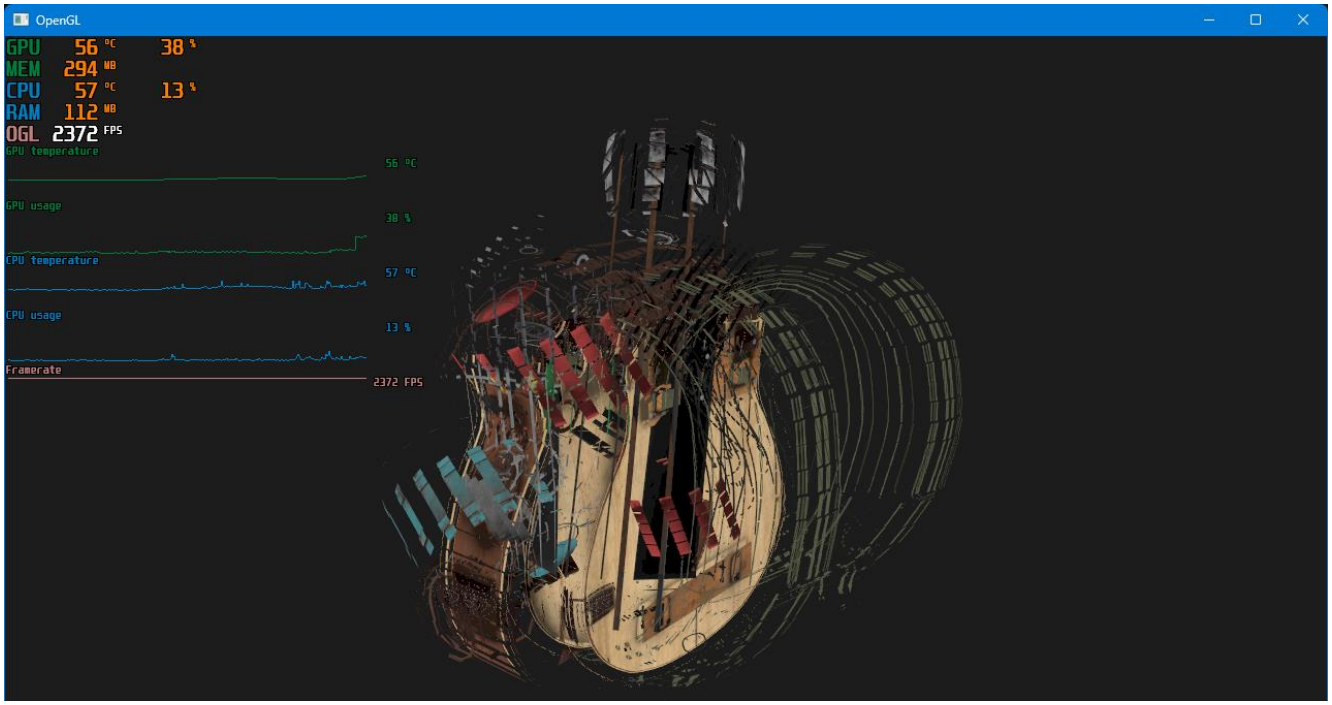

Рисунок 4.4 – Геометричний шейдер без вертикальної синхронізації для OpenGL (рисунок виконаний самостійно)

Далі йде інший геометричний шейдер зі згладжуванням 8x, включеною та вимкненою вертикальною синхронізацією (див. рис. 4.5, 4.6).

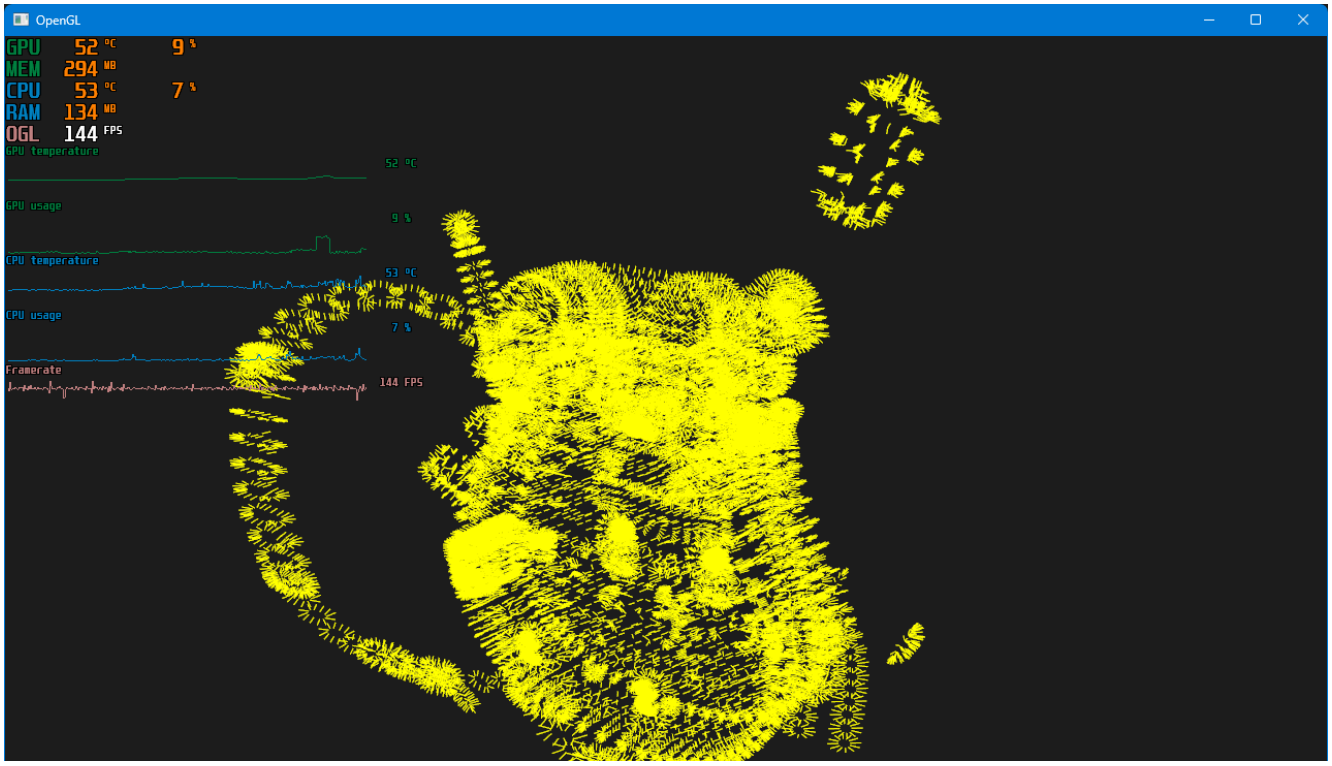

Рисунок 4.5 – Інший геометричний шейдер з вертикальною синхронізацією для OpenGL (рисунок виконаний самостійно)

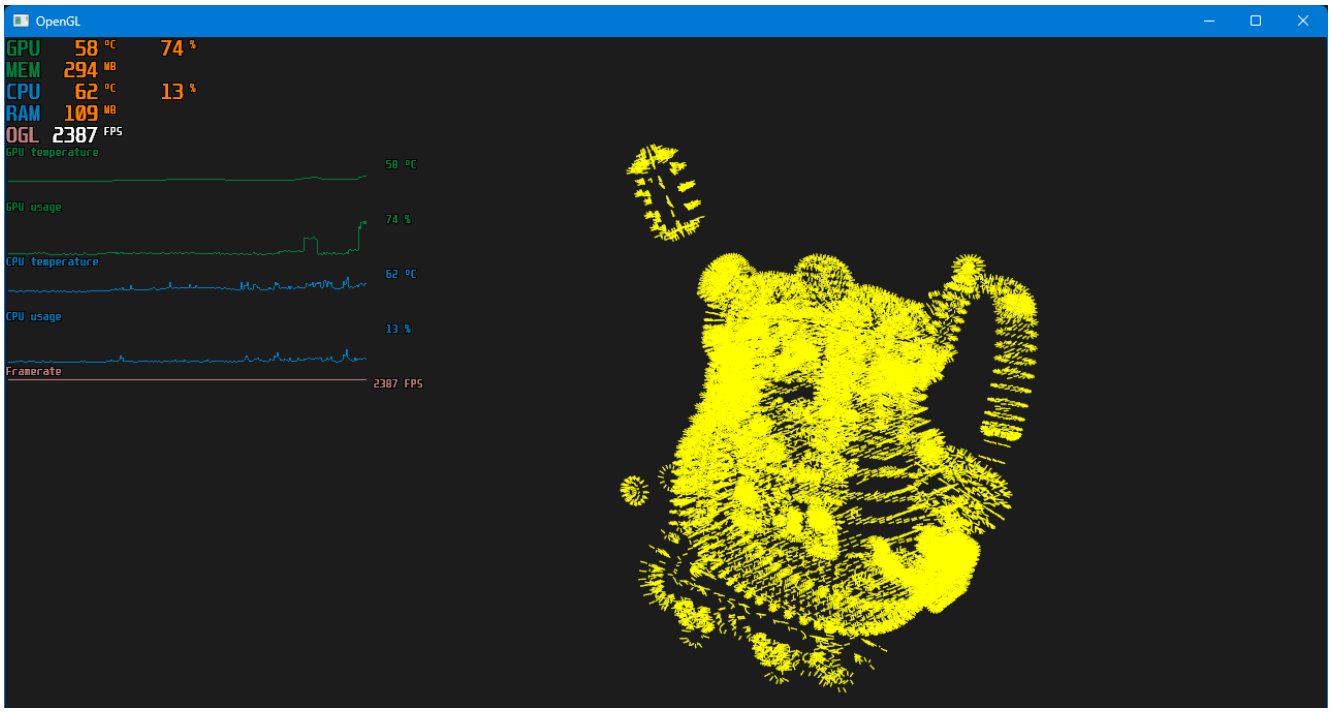

Рисунок 4.6 – Інший геометричний шейдер без вертикальної синхронізації для OpenGL (рисунок виконаний самостійно)

Перейдемо до експерименту з комбінації із першого та другого геометричного шейдерів зі згладжуванням 8x, включеною та вимкненою вертикальною синхронізацією (див. рис. 4.7, 4.8).

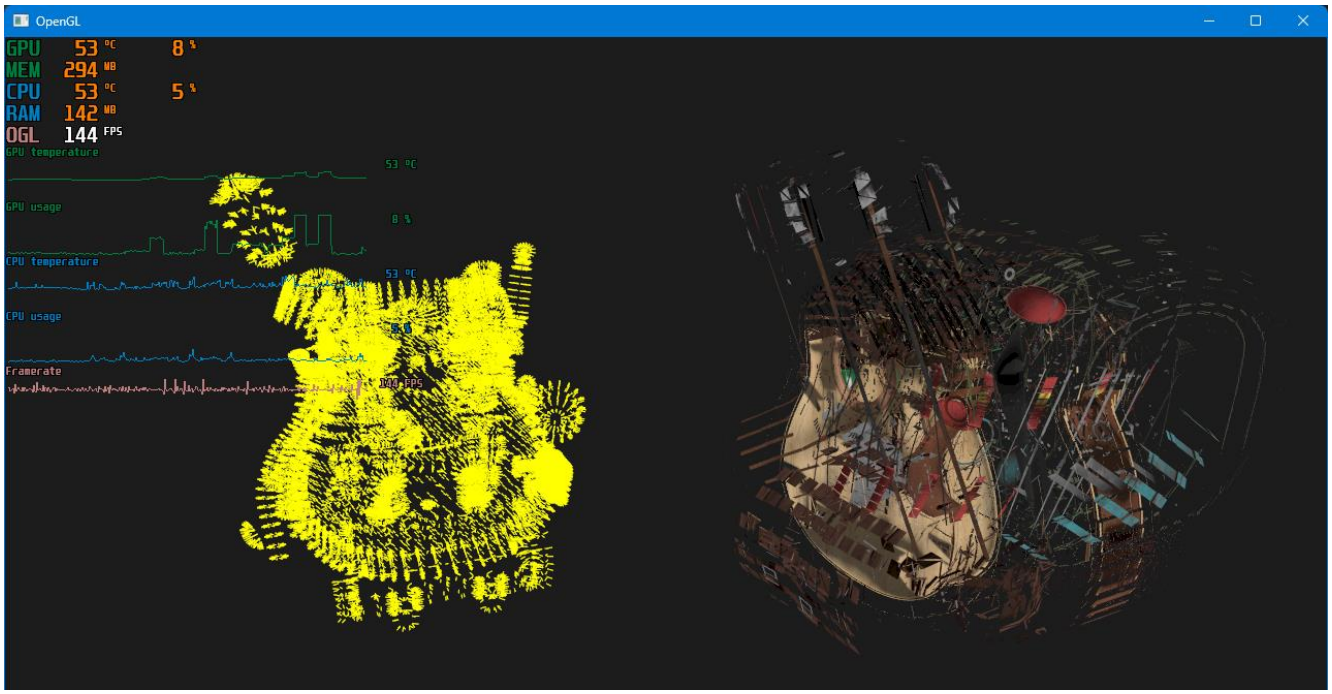

Рисунок 4.7 – Комбінація із першого та другого геометричного шейдерів з вертикальною синхронізацією для OpenGL (рисунок виконаний самостійно)

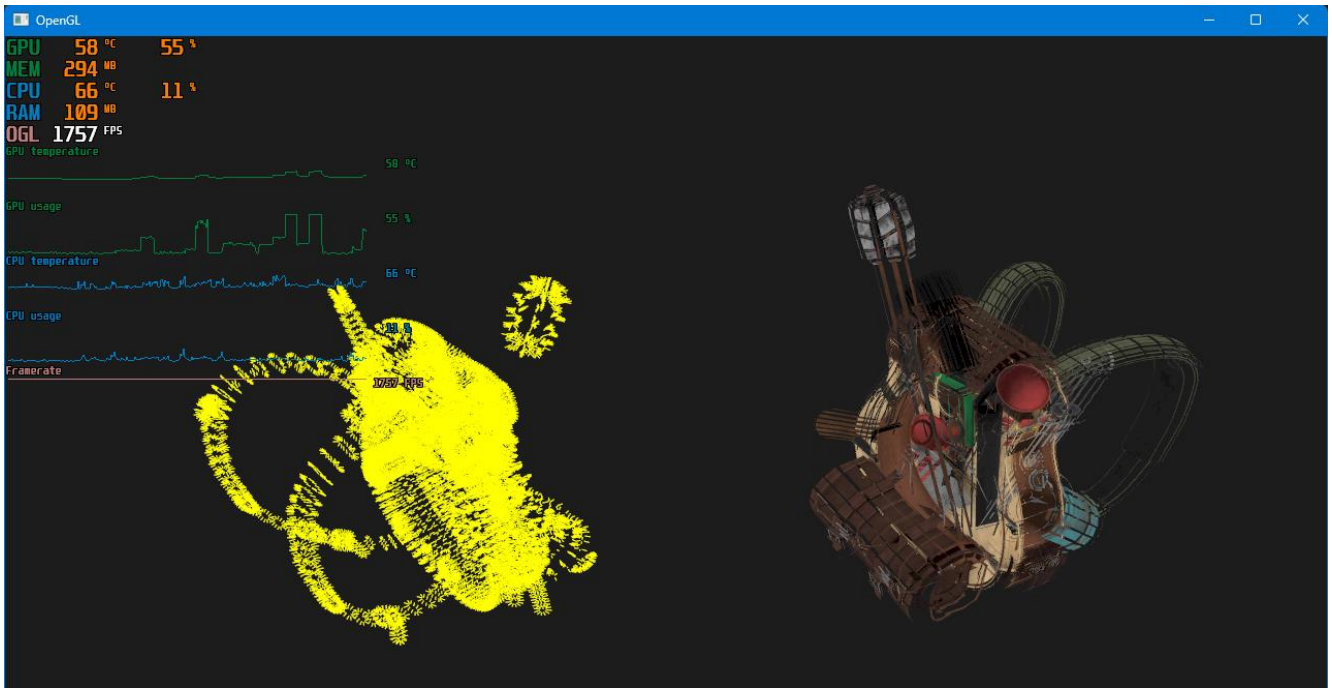

Рисунок 4.8 – Комбінація із першого та другого геометричного шейдерів без вертикальної синхронізації для OpenGL (рисунок виконаний самостійно)

Наступним є комбінація із шейдера освітлення та двох геометричних шейдерів зі згладжуванням 8x, включеною та вимкненою вертикальною синхронізацією (див. рис. 4.9, 4.10).

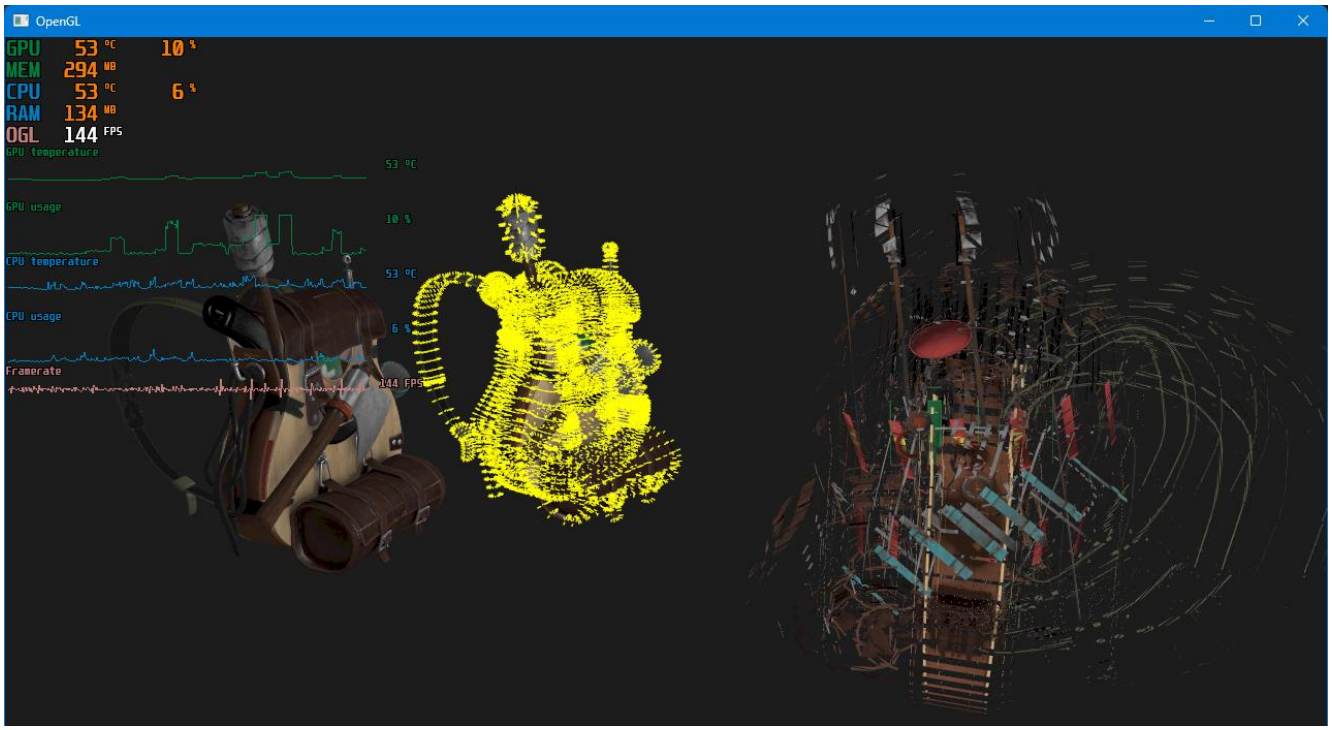

Рисунок 4.9 – Комбінація із шейдера освітлення та двох геометричних шейдерів з вертикальною синхронізацією для OpenGL (рисунок виконаний самостійно)

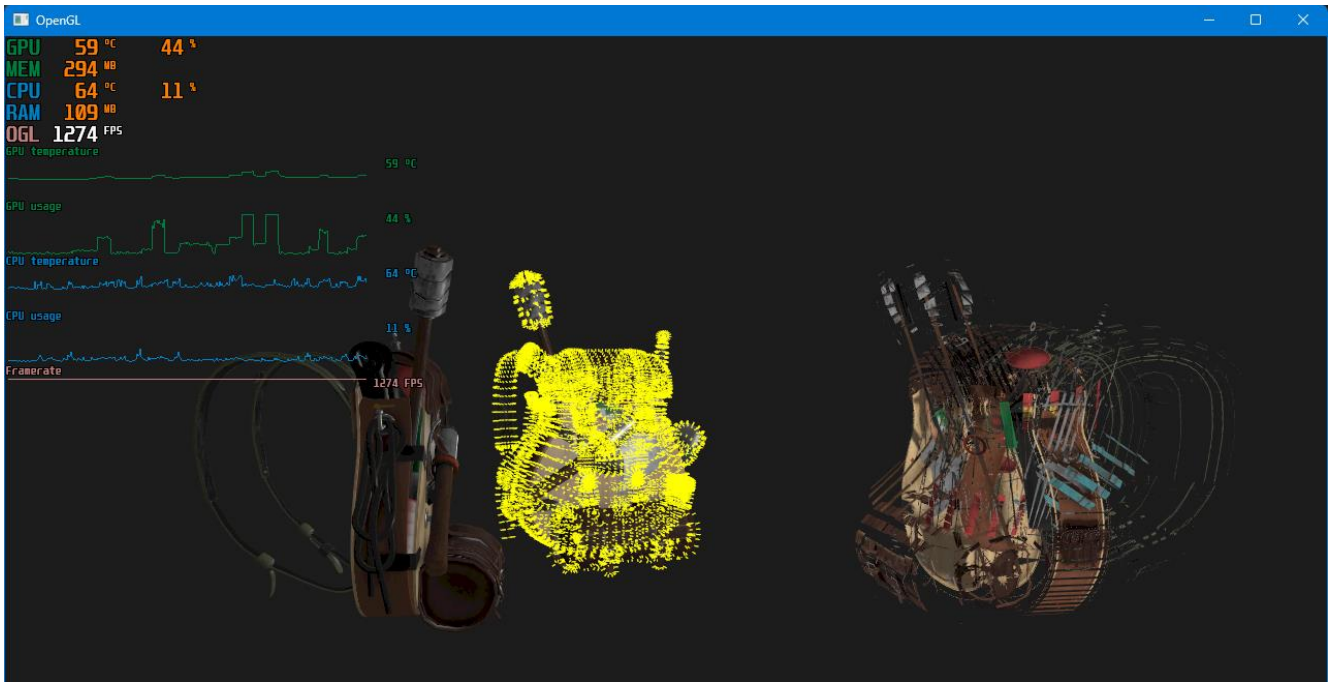

Рисунок 4.10 – Комбінація із шейдера освітлення та двох геометричних шейдерів без вертикальної синхронізації для OpenGL (рисунок виконаний самостійно)

Далі йде звичайний шейдер для виведення 3000 планет окремими командами, тобто команда для рендерингу однієї планети визивається 3000 разів. Тестування проводиться зі згладжуванням 8x, включеною та вимкненою вертикальною синхронізацією (див. рис. 4.11, 4.12).

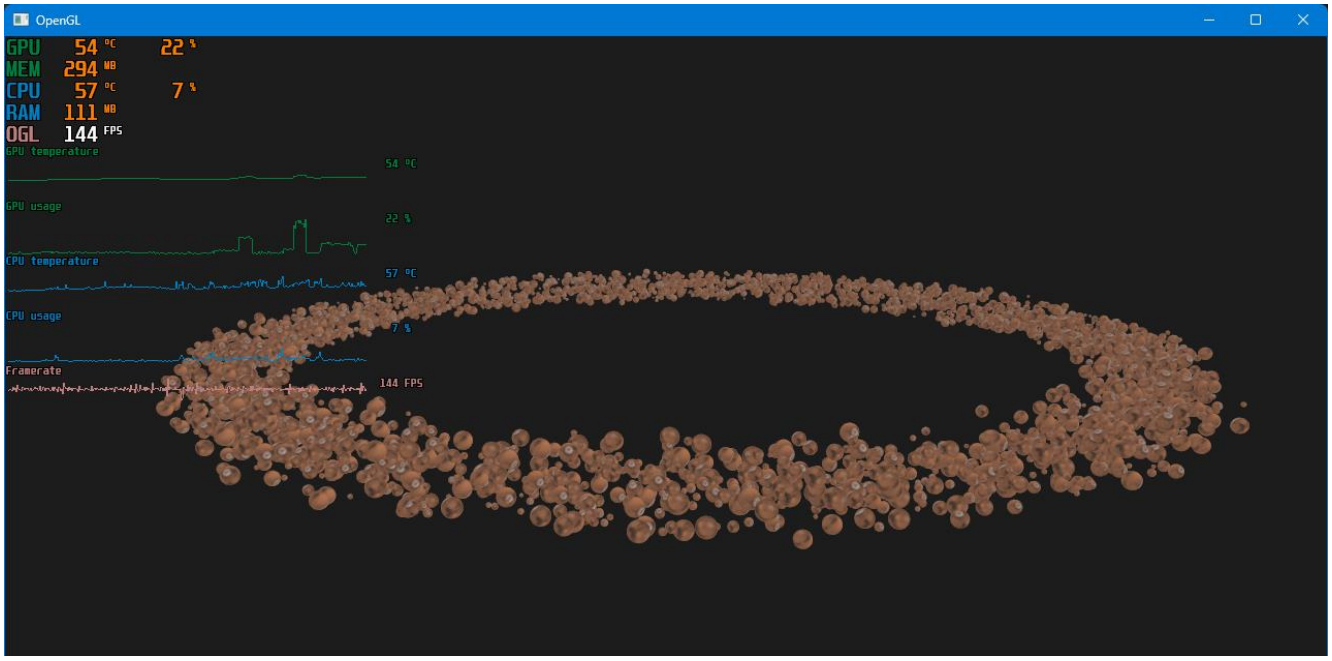

Рисунок 4.11 – Рендеринг 3000 планет з вертикальною синхронізацією для OpenGL (рисунок виконаний самостійно)

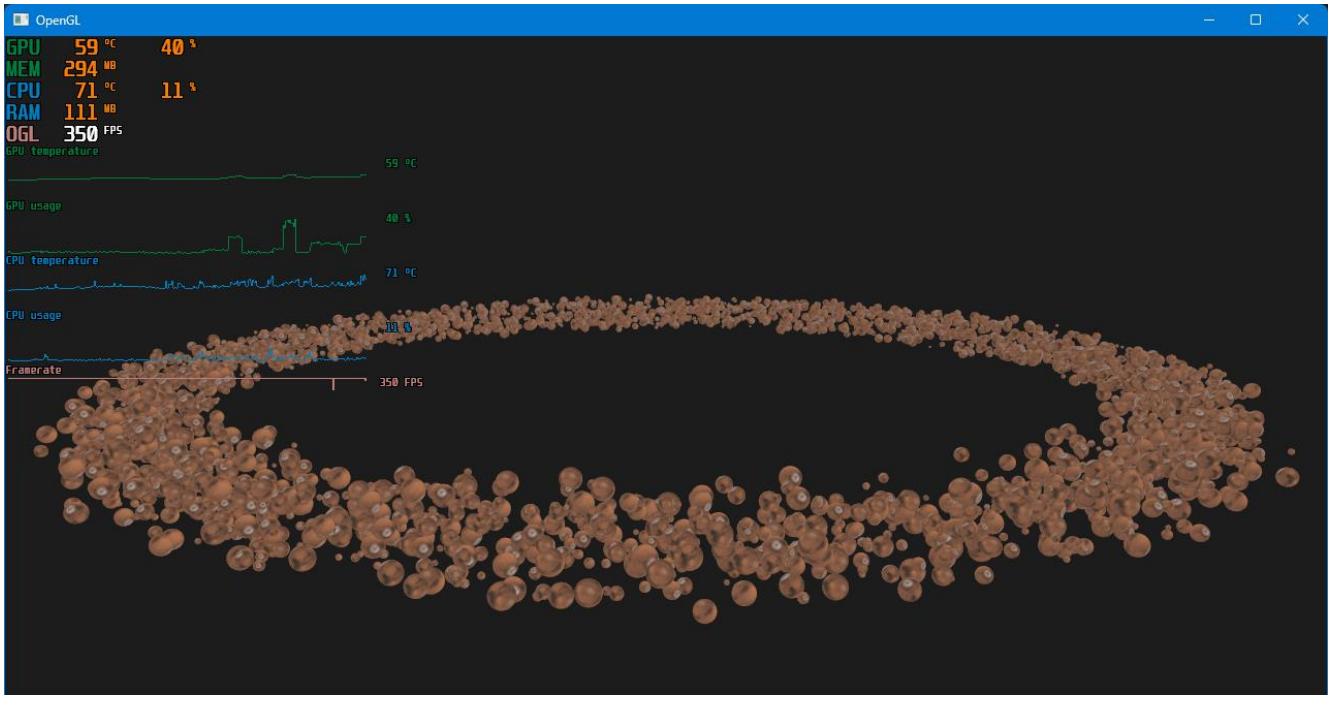

Рисунок 4.12 – Рендеринг 3000 планет без вертикальної синхронізації для OpenGL (рисунок виконаний самостійно)

А тепер використовуємо шейдер для інстансованого рендерингу 3000 планет, тобто однією командою. Тестування проводиться зі згладжуванням 8x, включеною та вимкненою вертикальною синхронізацією (див. рис. 4.13, 4.14).

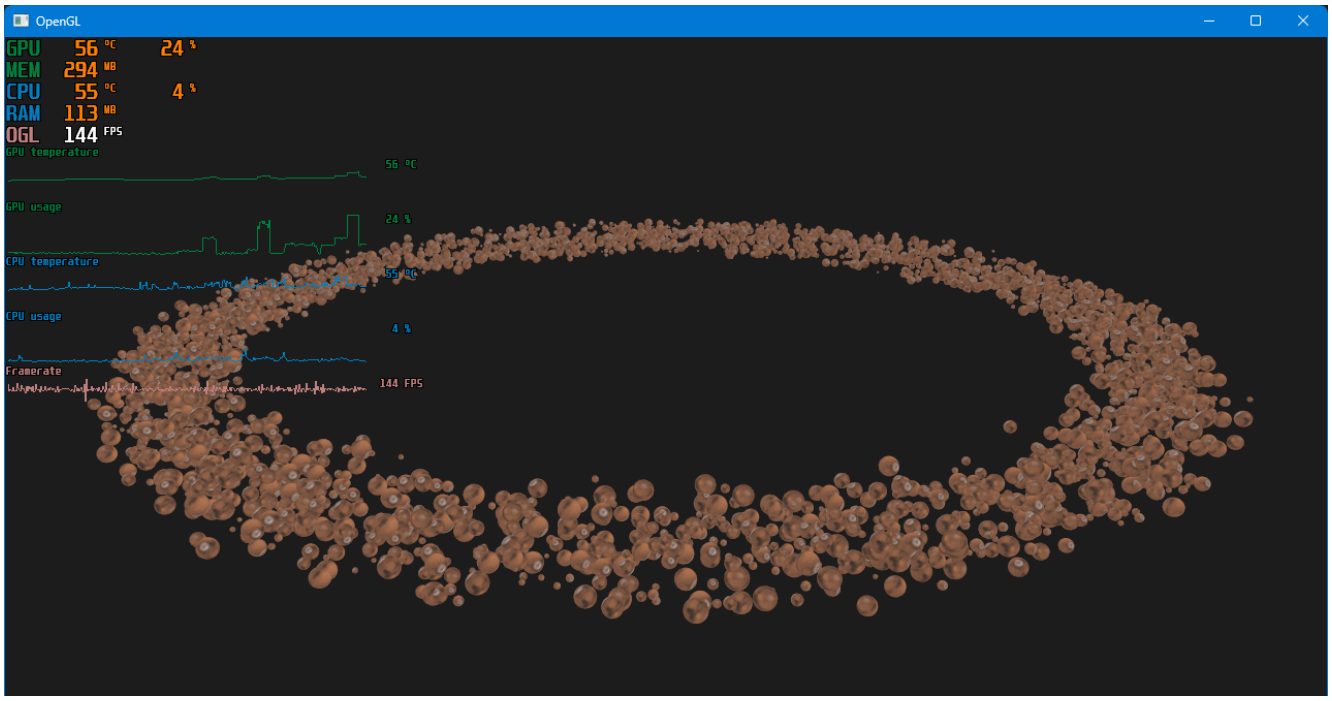

Рисунок 4.13 – Інстансований рендеринг 3000 планет з вертикальною синхронізацією для OpenGL (рисунок виконаний самостійно)

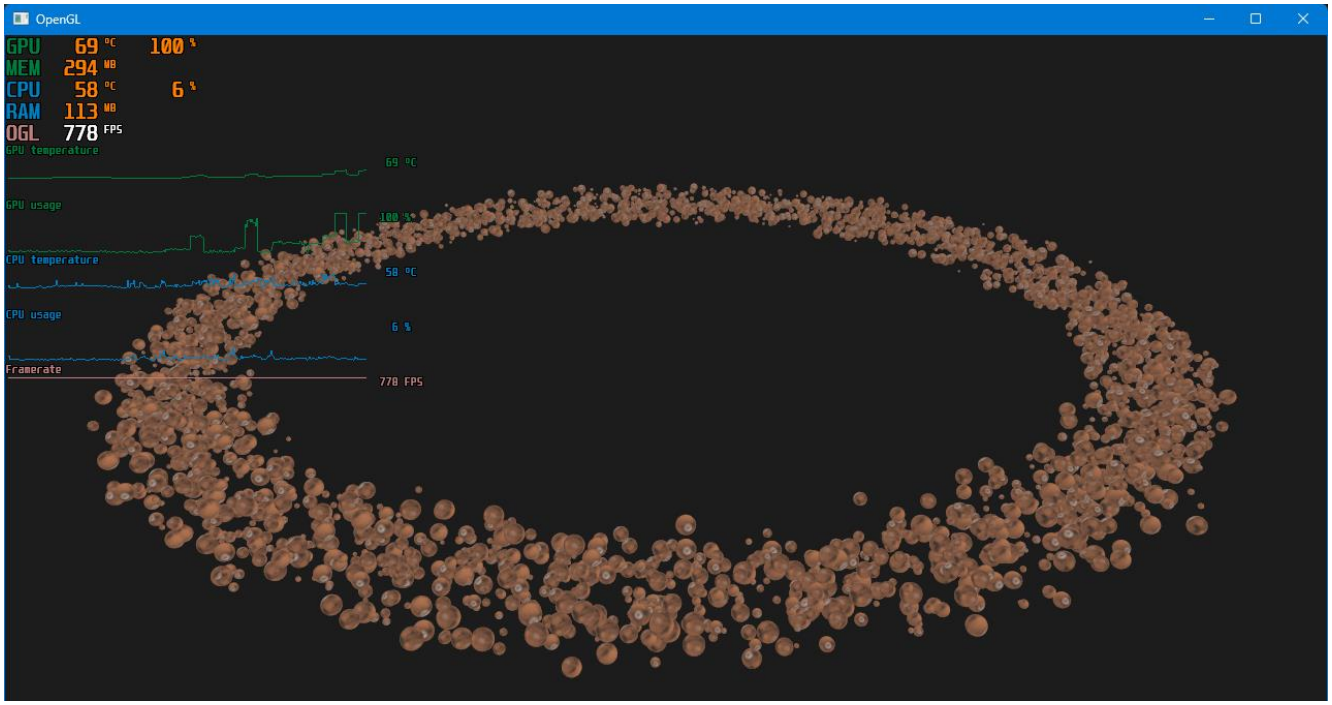

Рисунок 4.14 – Інстансований рендеринг 3000 планет без вертикальної синхронізації для OpenGL (рисунок виконаний самостійно)

Використаємо всі наші шейдери (окрім звичайного для виведення предметів використовуємо інстансований) для створення 3000 планет і 4 рюкзака зі згладжуванням 8x, включеною та вимкненою вертикальною синхронізацією (див. рис. 4.15, 4.16).

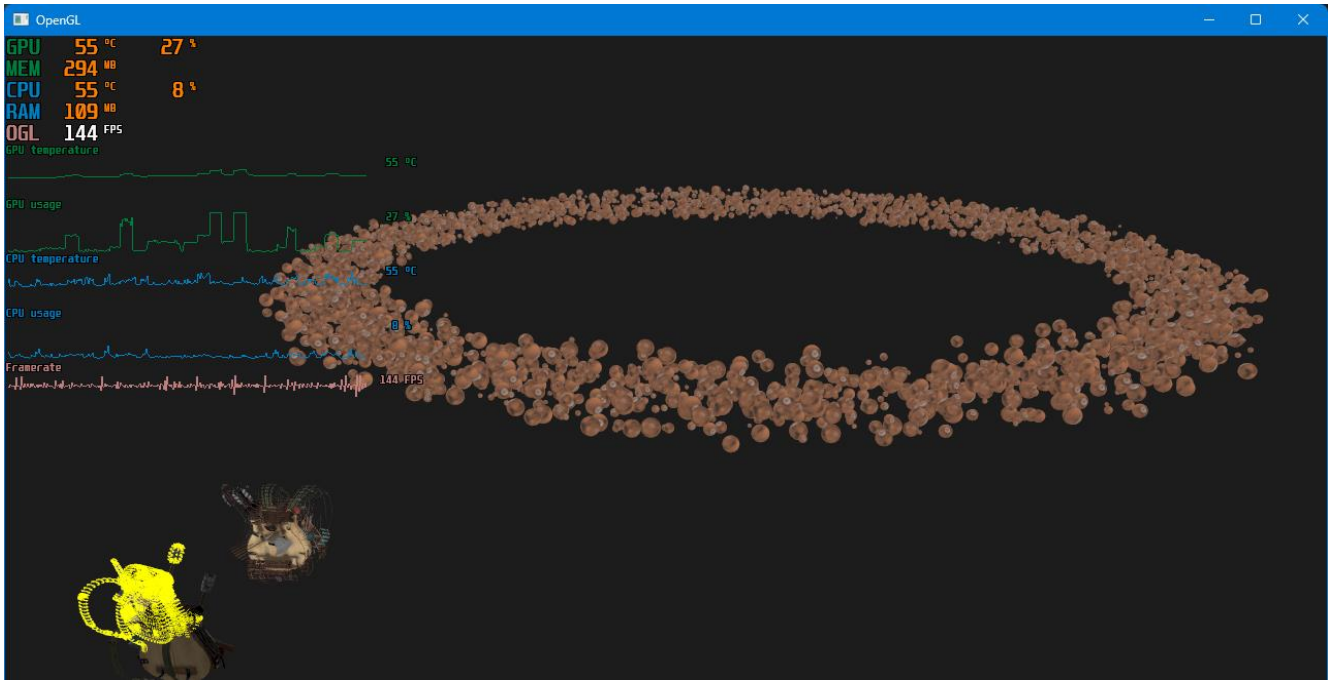

Рисунок 4.15 – Всі наші об’єкти з вертикальною синхронізацією для OpenGL (рисунок виконаний самостійно)

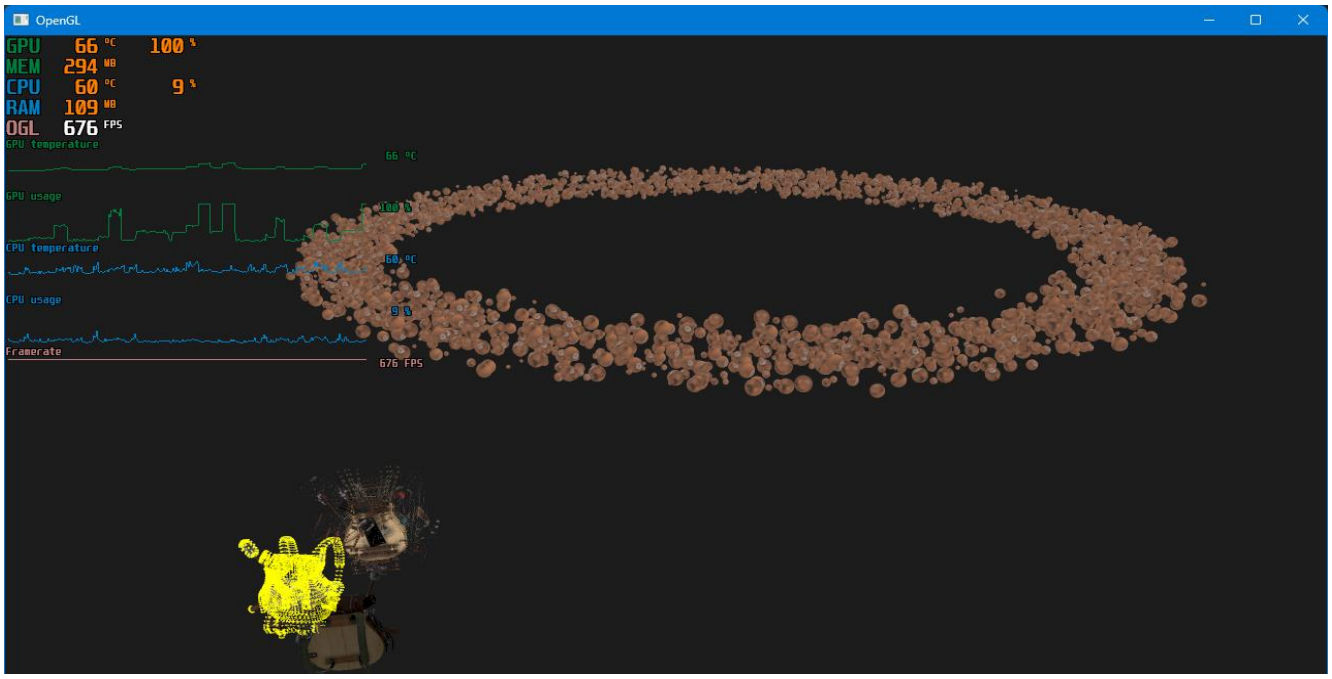

Рисунок 4.16 – Всі наші об’єкти без вертикальної синхронізації для OpenGL (рисунок виконаний самостійно)

Зараз використовуємо попередню сцену, але тепер без згладжування, з включеною та вимкненою вертикальною синхронізацією (див. рис. 4.17, 4.18).

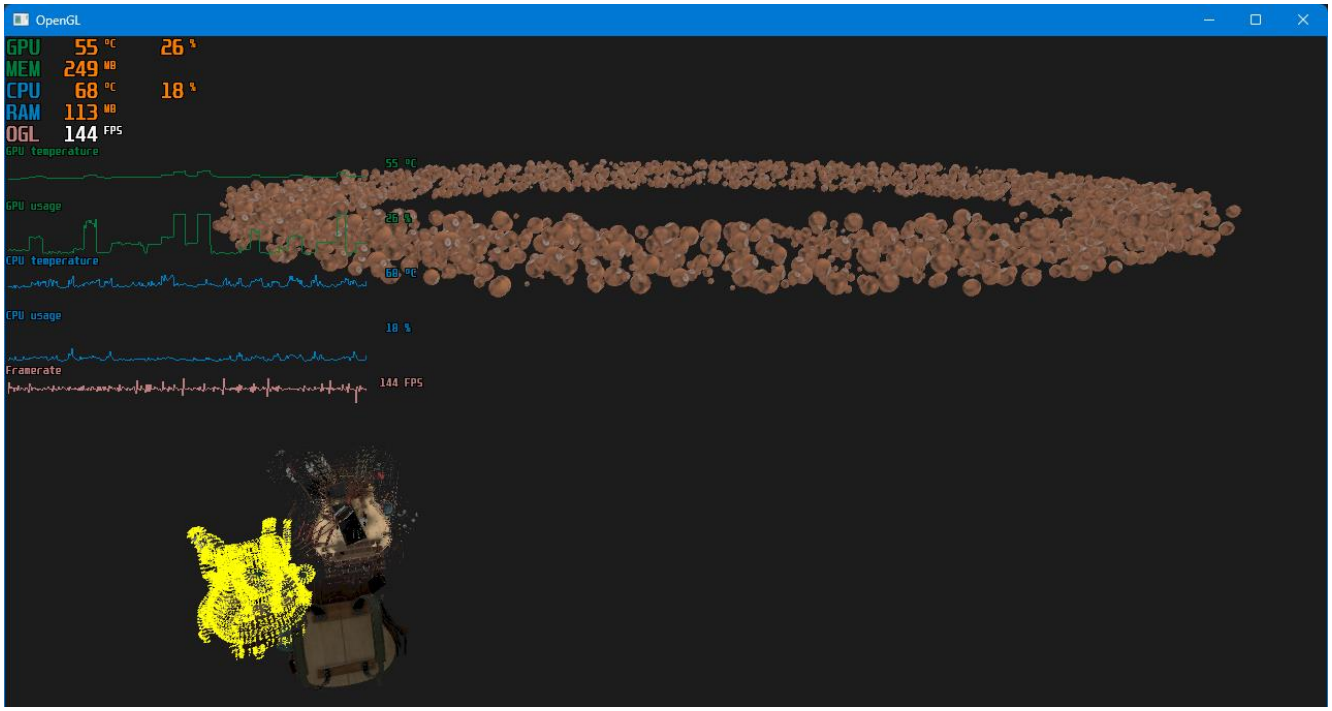

Рисунок 4.17 – Рендеринг без згладжування з вертикальною синхронізацією для OpenGL (рисунок виконаний самостійно)

Рисунок 4.18 – Рендеринг без згладжування і вертикальної синхронізації для OpenGL (рисунок виконаний самостійно)

Останньою сценою є відсутність будь-яких предметів, з включеною та вимкненою вертикальною синхронізацією (див. рис. 4.19, 4.20).

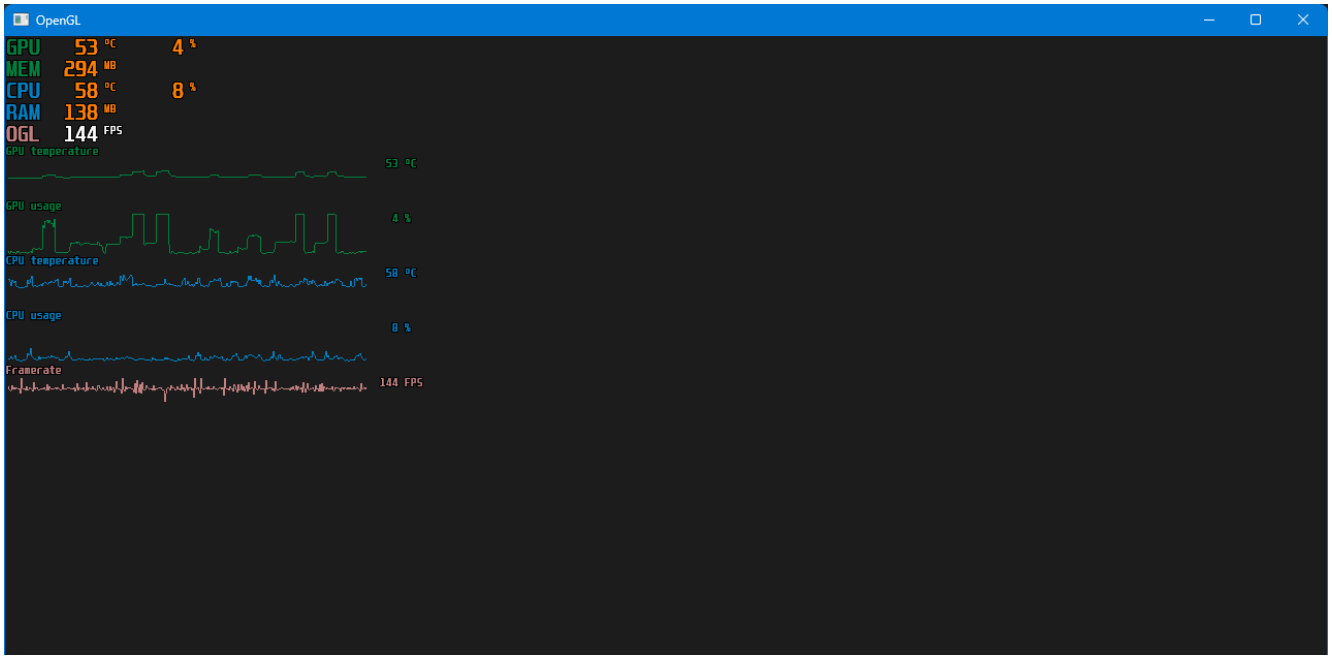

Рисунок 4.19 – Сцена без предметів з вертикальною синхронізацією для OpenGL (рисунок виконаний самостійно)

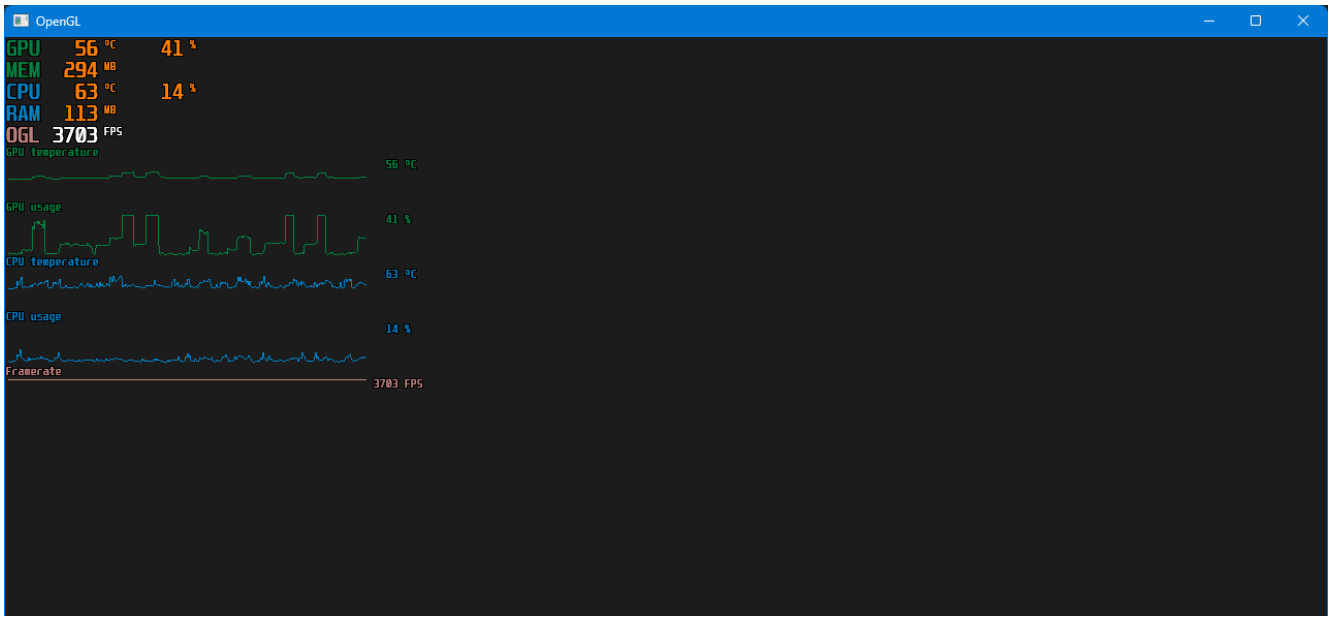

Рисунок 4.20 – Сцена без предметів без вертикальної синхронізації для OpenGL (рисунок виконаний самостійно)

Після завершення всіх тестів, на яких ми фіксували отримані дані, розраховуємо середнє значення кожного критерія та заносимо їх в таблицю 4.1.

Таблиця 4.1 – Результати для API OpenGL (таблиця виконана самостійно)

	GPU T	GPU	GPU M	CPU T	CPU	RAM	FPS
Шейдер освітл. з VSync	48	5	294	48	5	125	144
Шейдер освітл. без VSync	55	36	294	64	12	125	1800
Геом. шейдер з VSync	49	6	294	51	7	129	144
Геом. шейдер без VSync	54	40	294	65	10	129	2160
Інший геом. шейдер з VSync	51	8	294	53	7	109	144
Інший геом. шейдер без VSync	57	75	294	64	12	109	2280
Комбінація із геом. шейдерів з VSync	51	9	294	51	5	109	144
Комбінація із геом. шейдерів без VSync	58	60	294	51	13	109	1800
Геом. шейдери та освітлення з VSync	51	10	294	53	5	109	144
Геом. шейдери та освітлення без VSync	59	45	294	71	11	109	1230
Рендер. планет з VSync	52	23	294	58	8	114	144
Рендер. планет без VSync	59	39	294	73	11	114	345
Інстансований рендер. планет з VSync	54	25	294	54	5	109	144
Інстансований рендер. планет без VSync	68	100	294	60	5	109	777
Всі предмети з VSync	54	27	294	54	5	109	144
Всі предмети без VSync	68	100	294	65	8	109	673

Кінець таблиці 4.1

	GPU T	GPU	GPU M	CPU T	CPU	RAM	FPS
Всі об’єкти без згладж., з VSync	55	26	249	55	5	109	144
Всі об’єкти без згладж. і VSync	67	100	249	66	8	109	703
Порожня сцена з VSync	52	4	294	53	5	110	144
Порожня сцена без VSync	55	40	294	66	13	110	3600

Після занесення всіх даних до таблиці ми провели дослідження для інтерфейсу OpenGL, але ці дані нічого корисного без порівняння з даними іншого інтерфейсу не дають, тому потрібно провести дослідження для інтерфейсу Vulkan.

4.3 Проведення експериментів для інтерфейсу Vulkan

Дослідження для інтерфейсу Vulkan нічим не відрізняється від OpenGL, окрім того, що вони виконуються на різних додатках. Тому ми коротко виконаємо кожний тест, який був проведений і для OpenGL. (див. рис. 4.21-4.40).

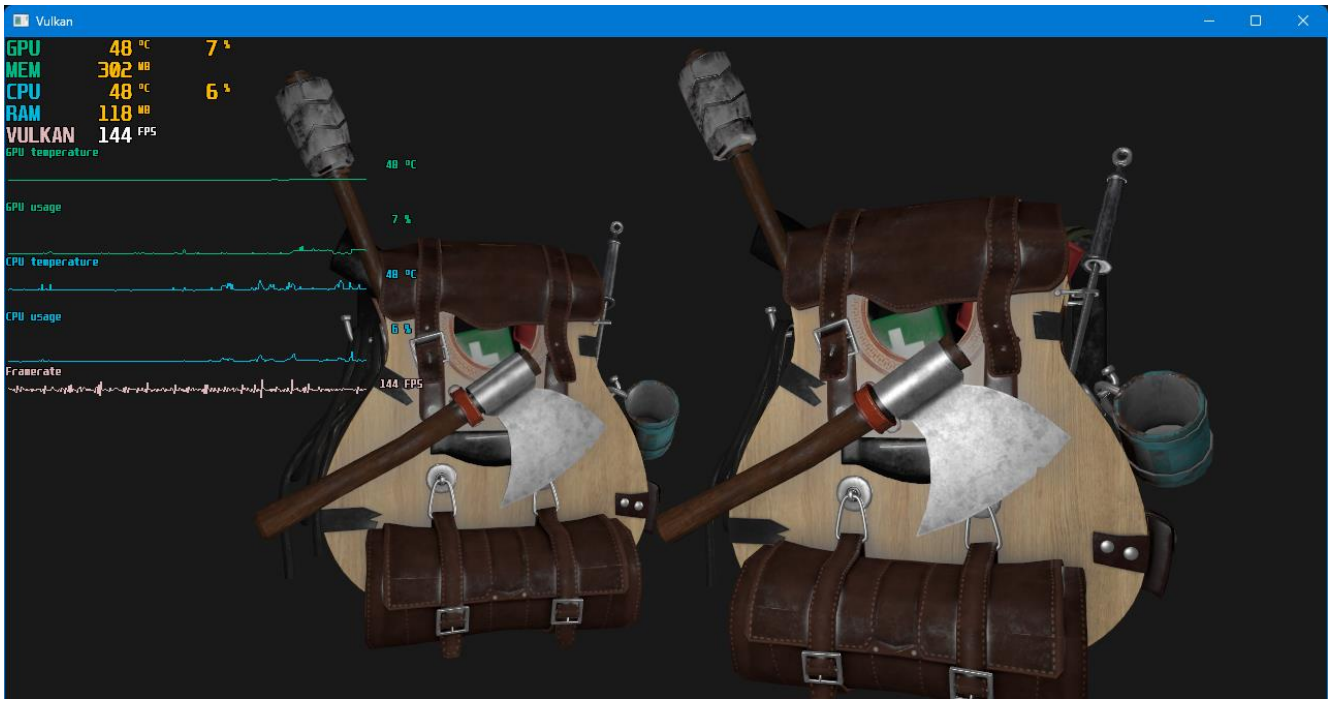

Рисунок 4.21 – Шейдер освітлення з вертикальною синхронізацією для Vulkan (рисунок виконаний самостійно)

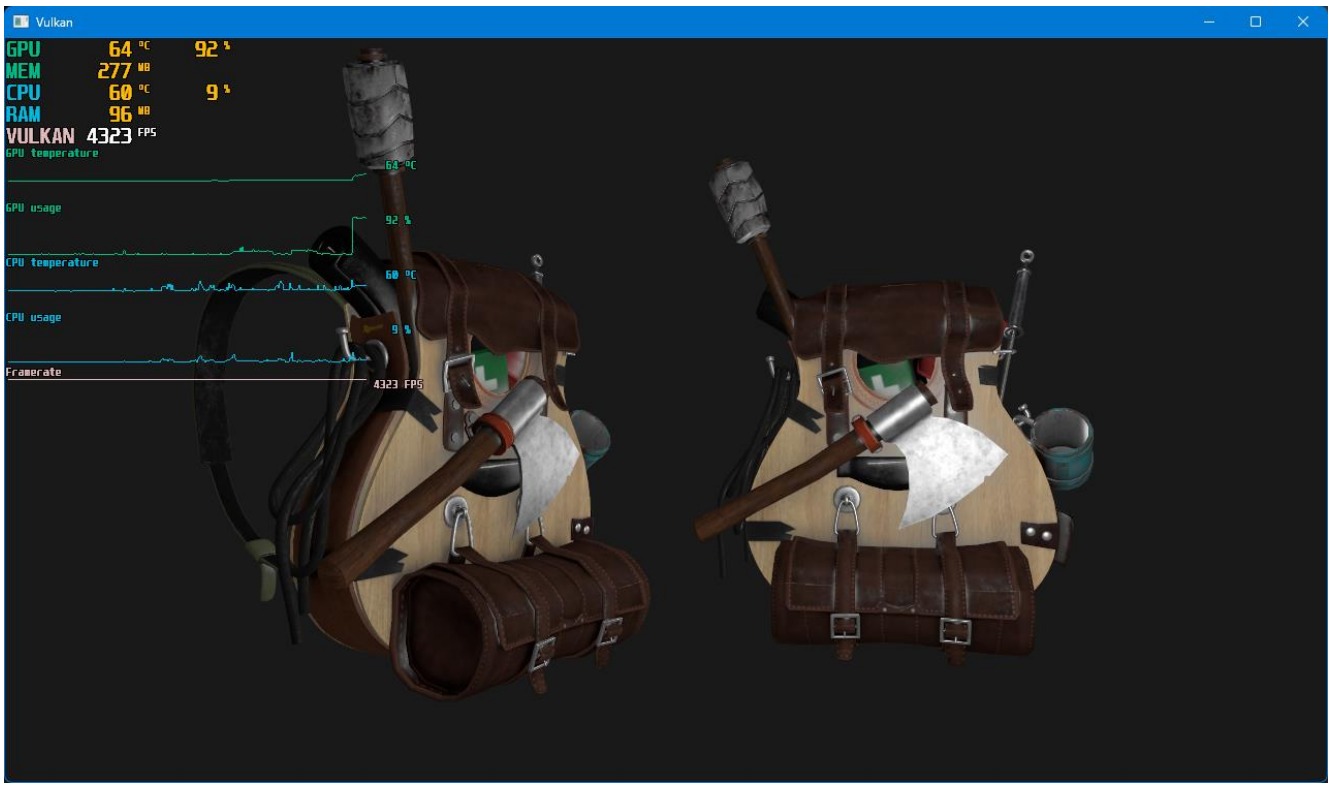

Рисунок 4.22 – Шейдер освітлення без вертикальної синхронізації для Vulkan (рисунок виконаний самостійно)

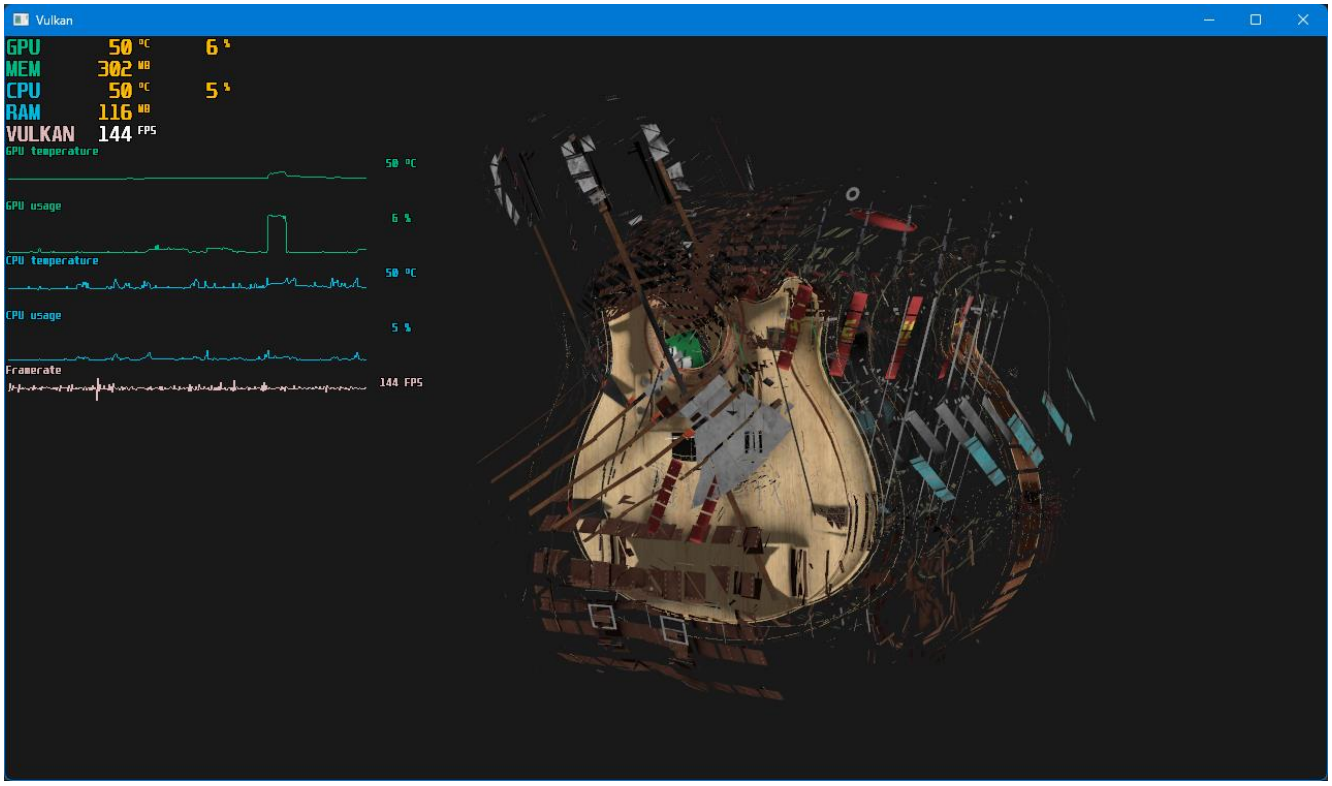

Рисунок 4.23 – Геометричний шейдер з вертикальною синхронізацією для Vulkan (рисунок виконаний самостійно)

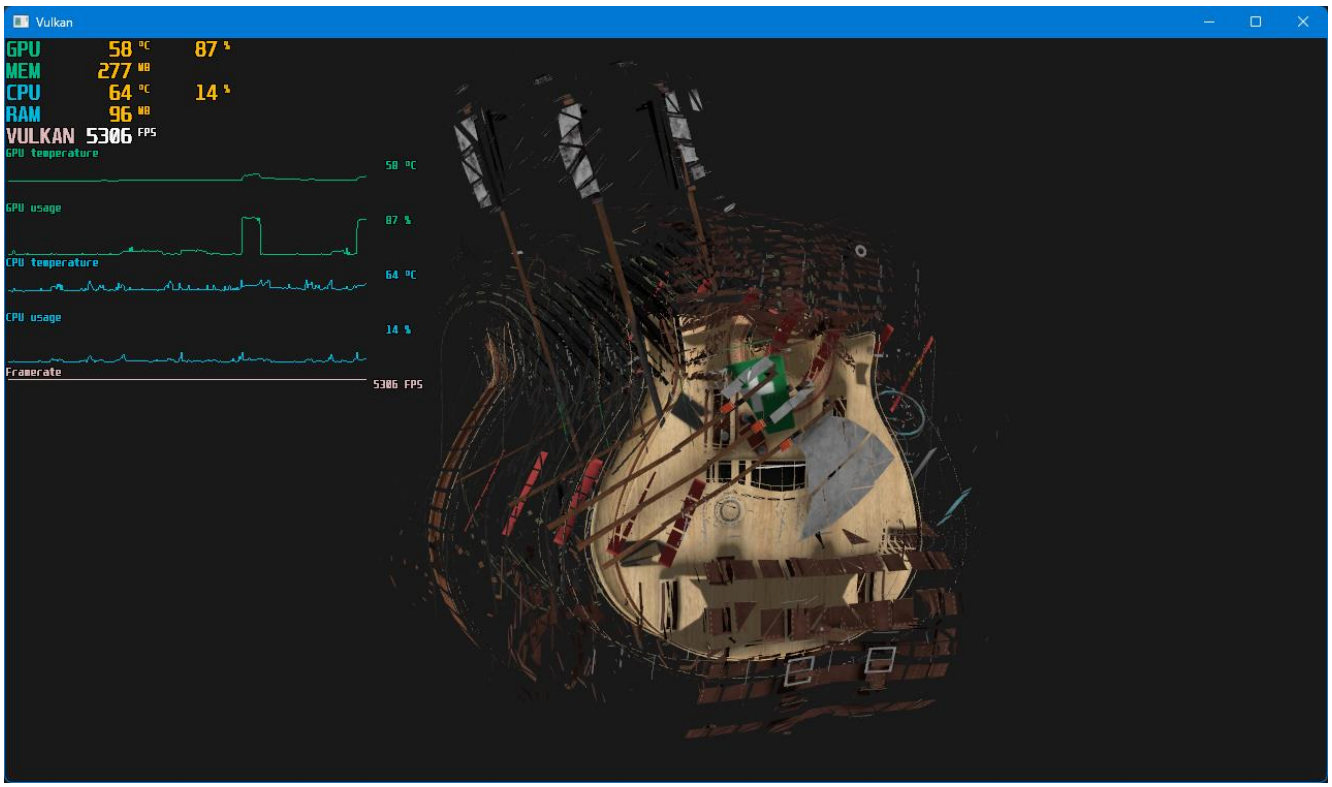

Рисунок 4.24 – Геометричний шейдер без вертикальної синхронізації для Vulkan (рисунок виконаний самостійно)

Рисунок 4.25 – Інший геометричний шейдер з вертикальною синхронізацією для Vulkan (рисунок виконаний самостійно)

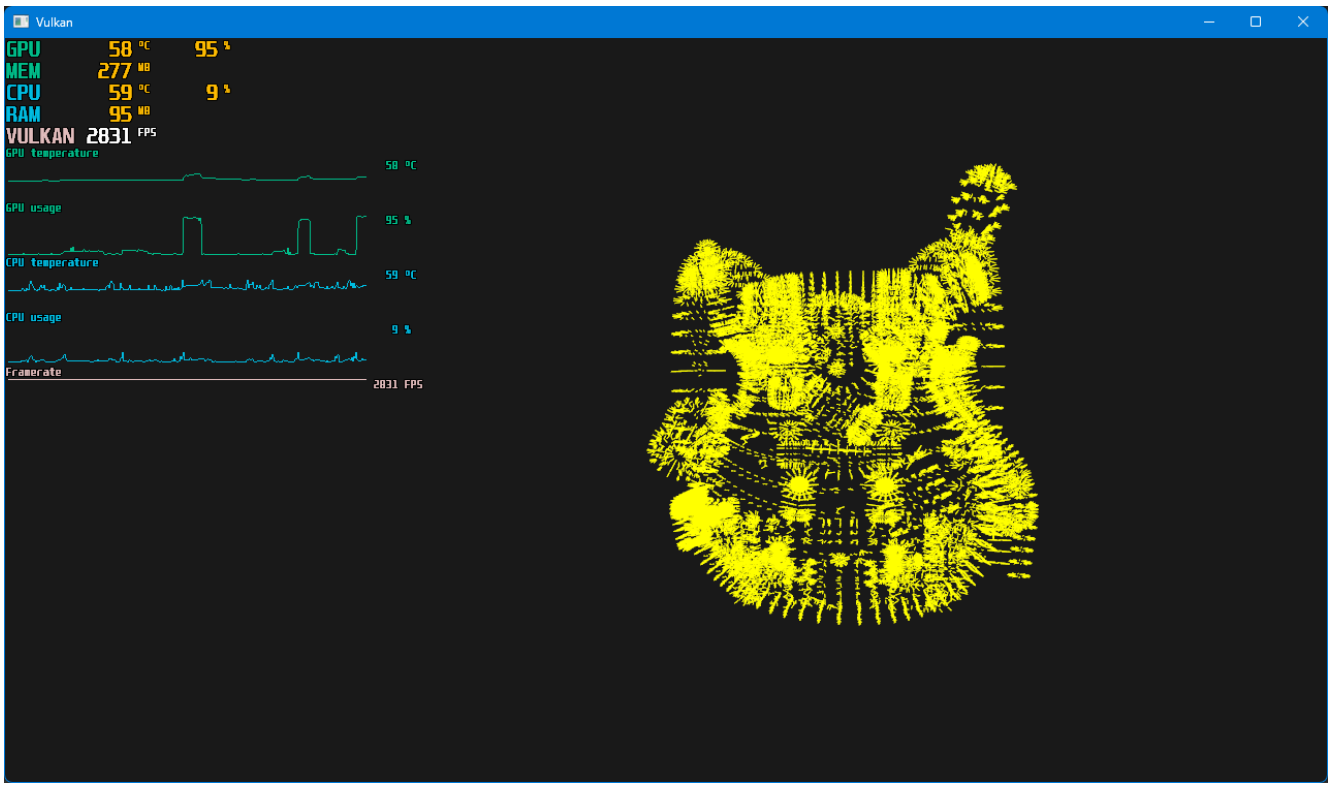

Рисунок 4.26 – Інший геометричний шейдер без вертикальної синхронізації для Vulkan (рисунок виконаний самостійно)

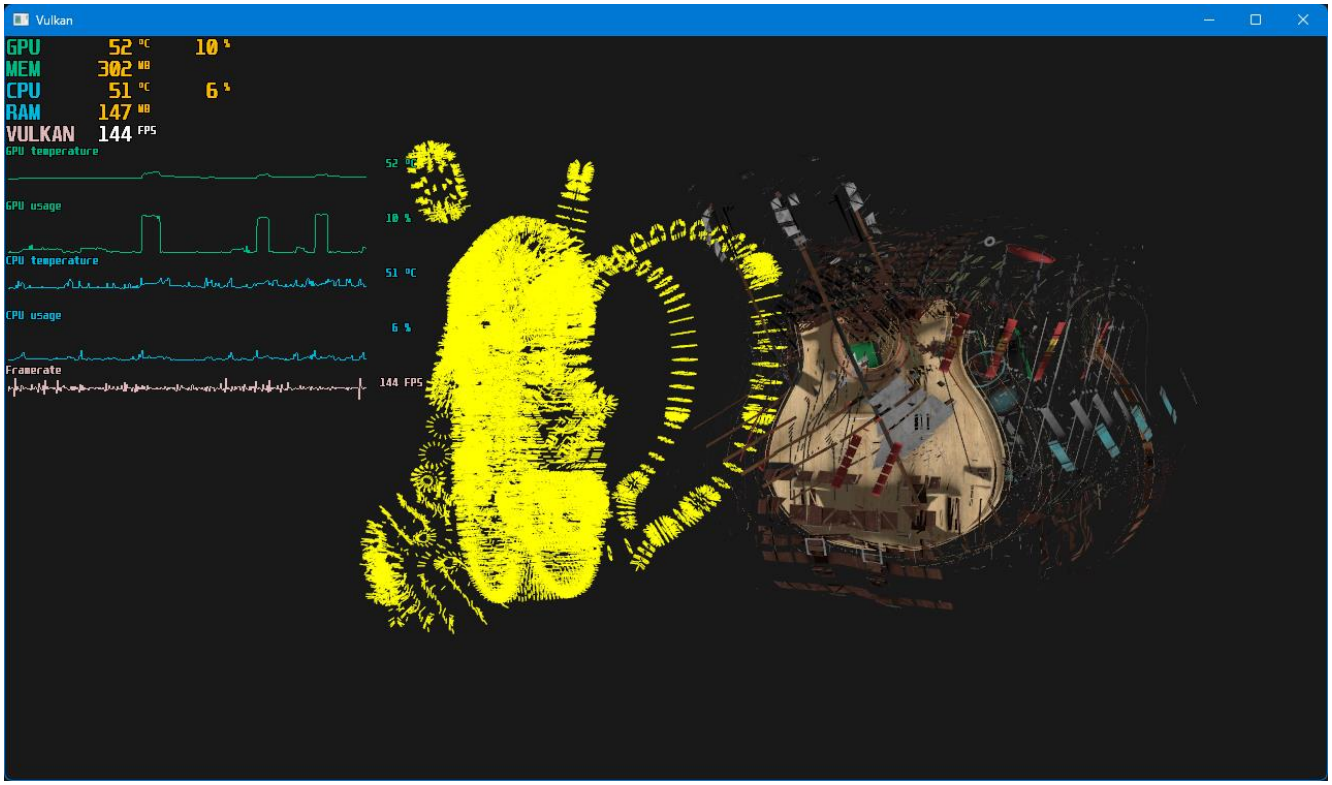

Рисунок 4.27 – Комбінація із першого та другого геометричного шейдерів з вертикальною синхронізацією для Vulkan (рисунок виконаний самостійно)

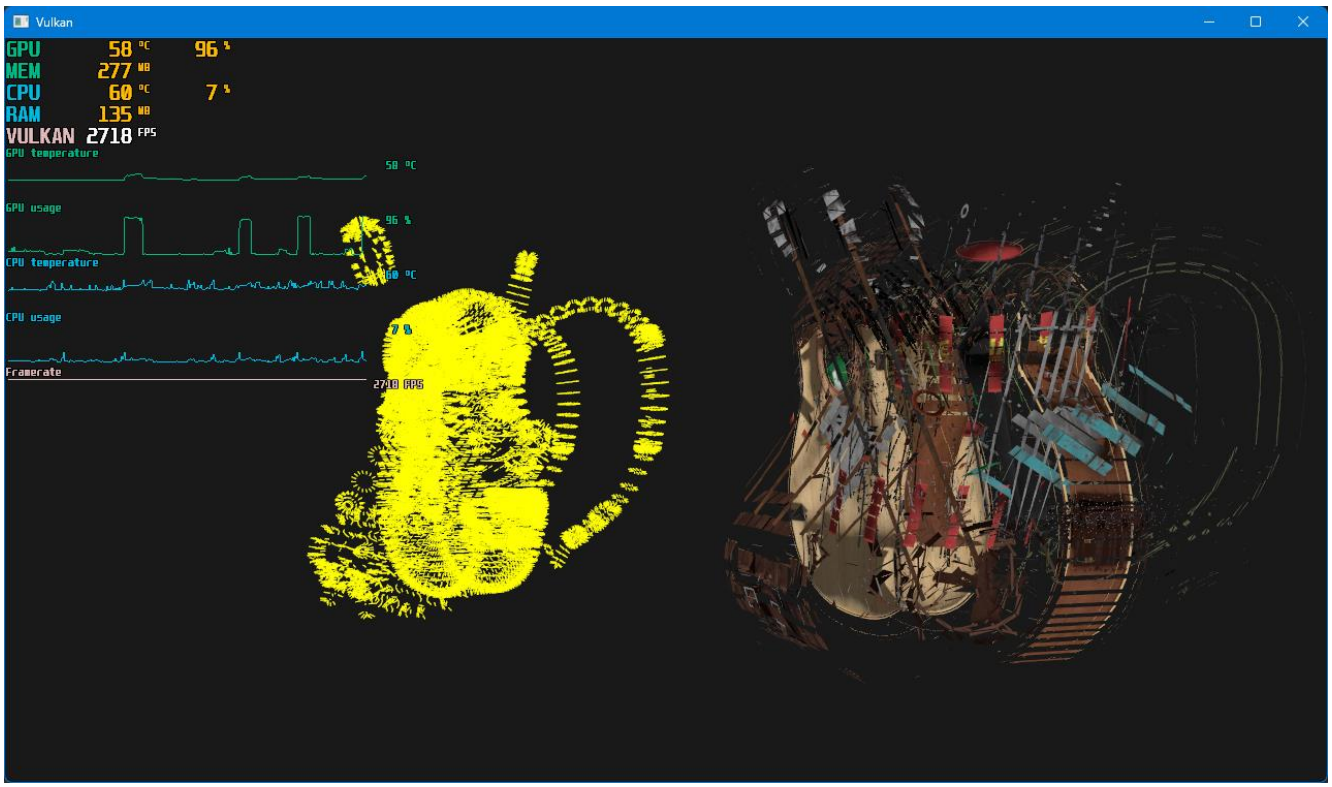

Рисунок 4.28 – Комбінація із першого та другого геометричного шейдерів без вертикальної синхронізації для Vulkan (рисунок виконаний самостійно)

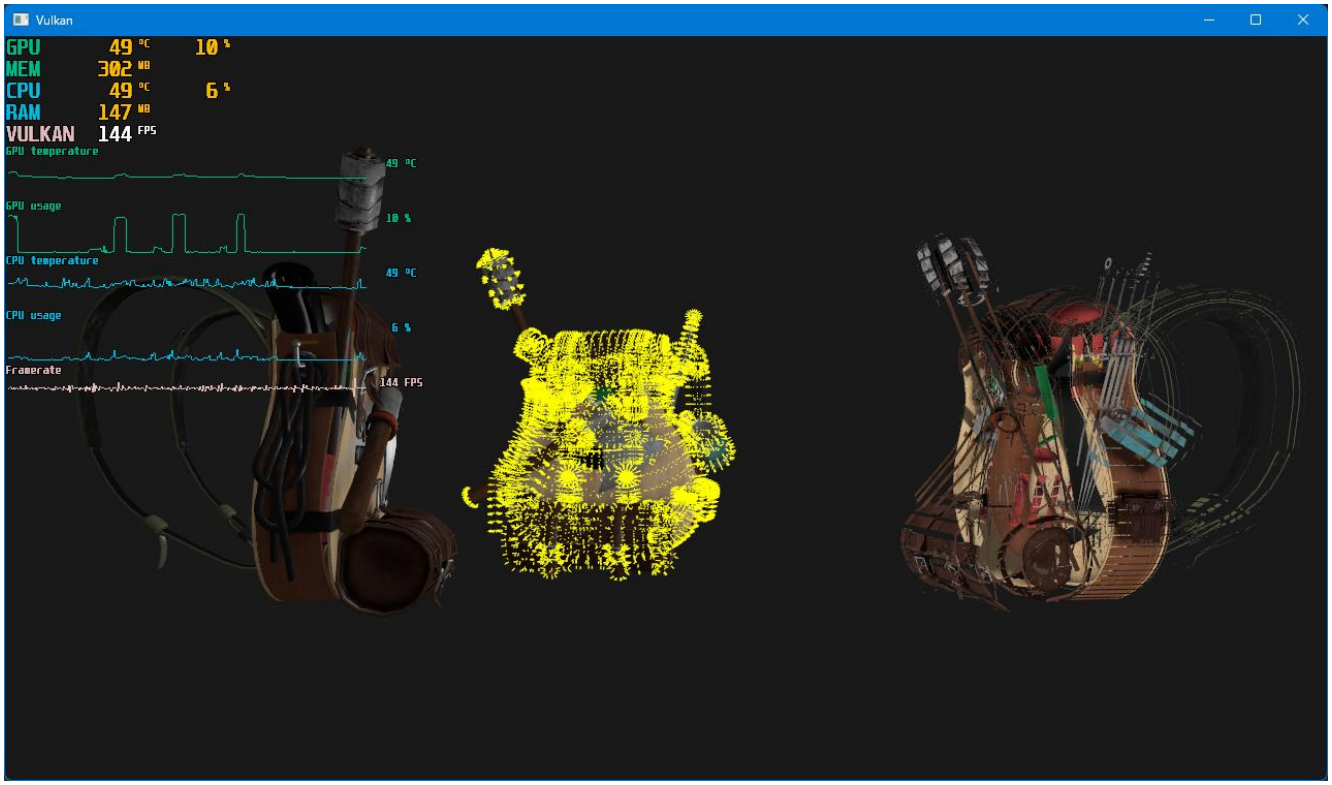

Рисунок 4.29 – Комбінація із шейдера освітлення та двох геометричних шейдерів з вертикальною синхронізацією для Vulkan (рисунок виконаний самостійно)

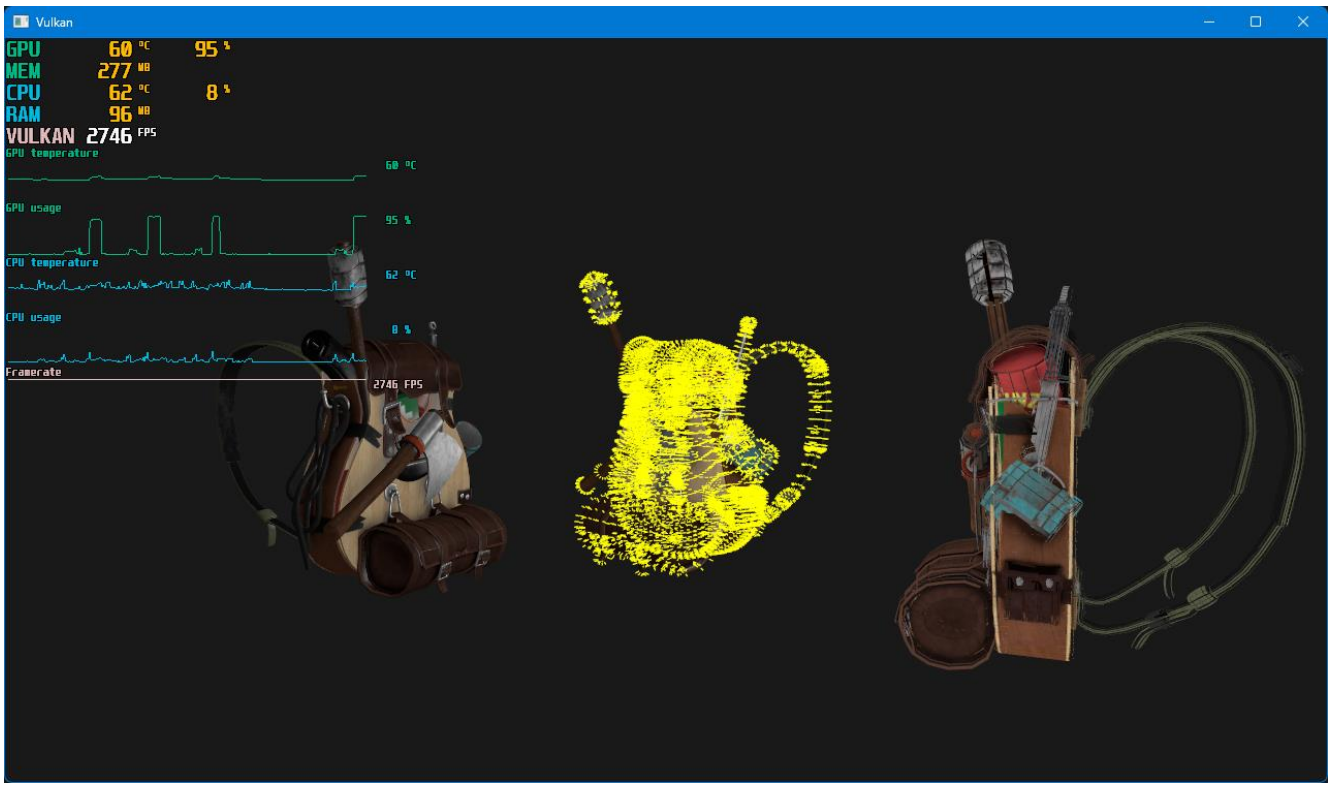

Рисунок 4.30 – Комбінація із шейдера освітлення та двох геометричних шейдерів без вертикальної синхронізації для Vulkan (рисунок виконаний самостійно)

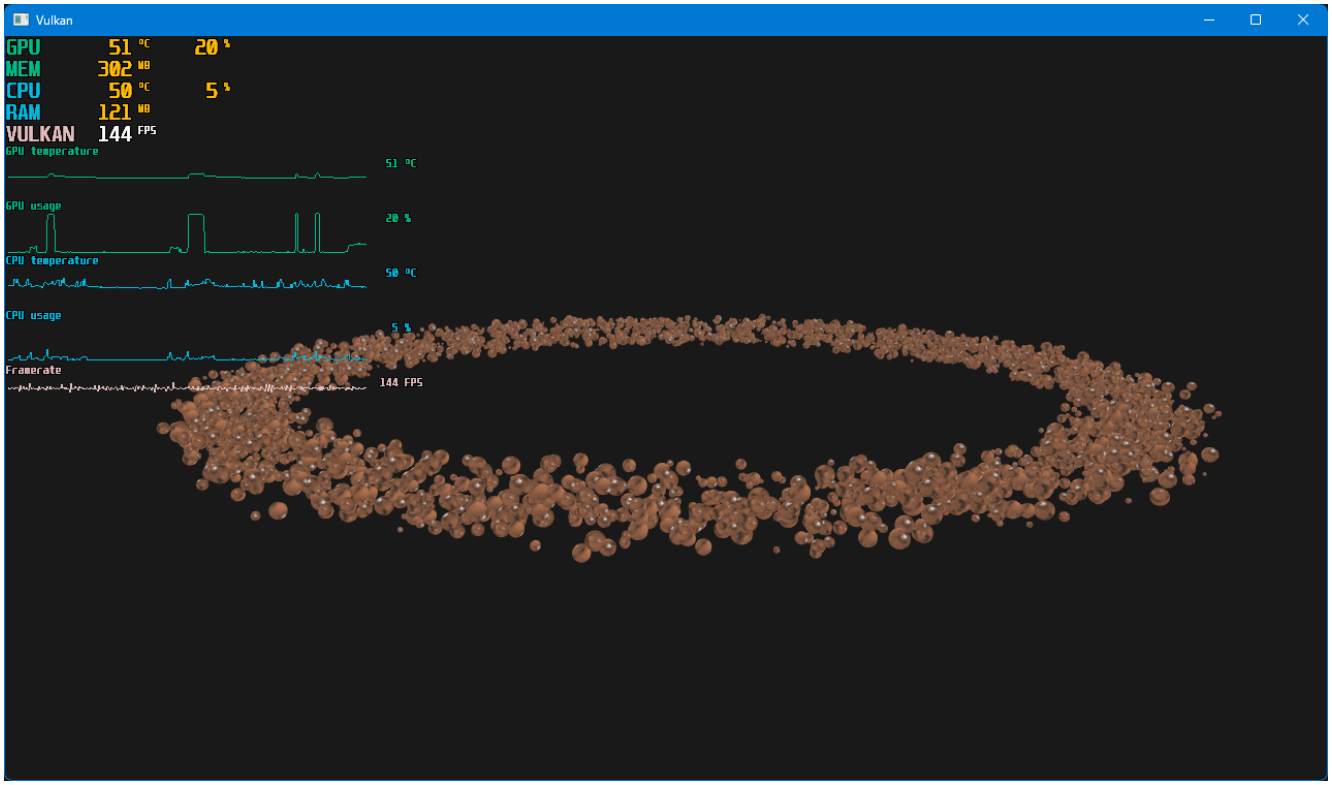

Рисунок 4.31 – Рендеринг 3000 планет з вертикальною синхронізацією для Vulkan (рисунок виконаний самостійно)

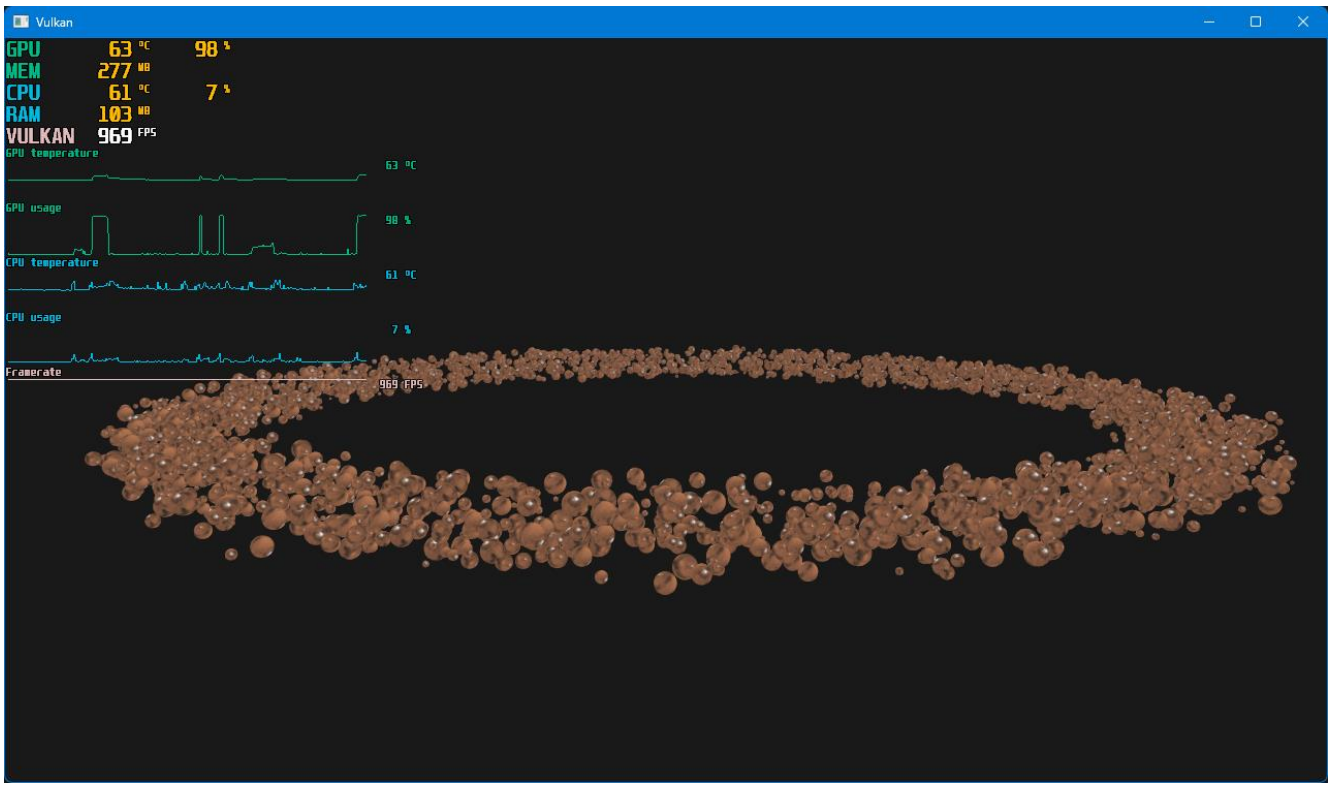

Рисунок 4.32 – Рендеринг 3000 планет без вертикальної синхронізації для Vulkan (рисунок виконаний самостійно)

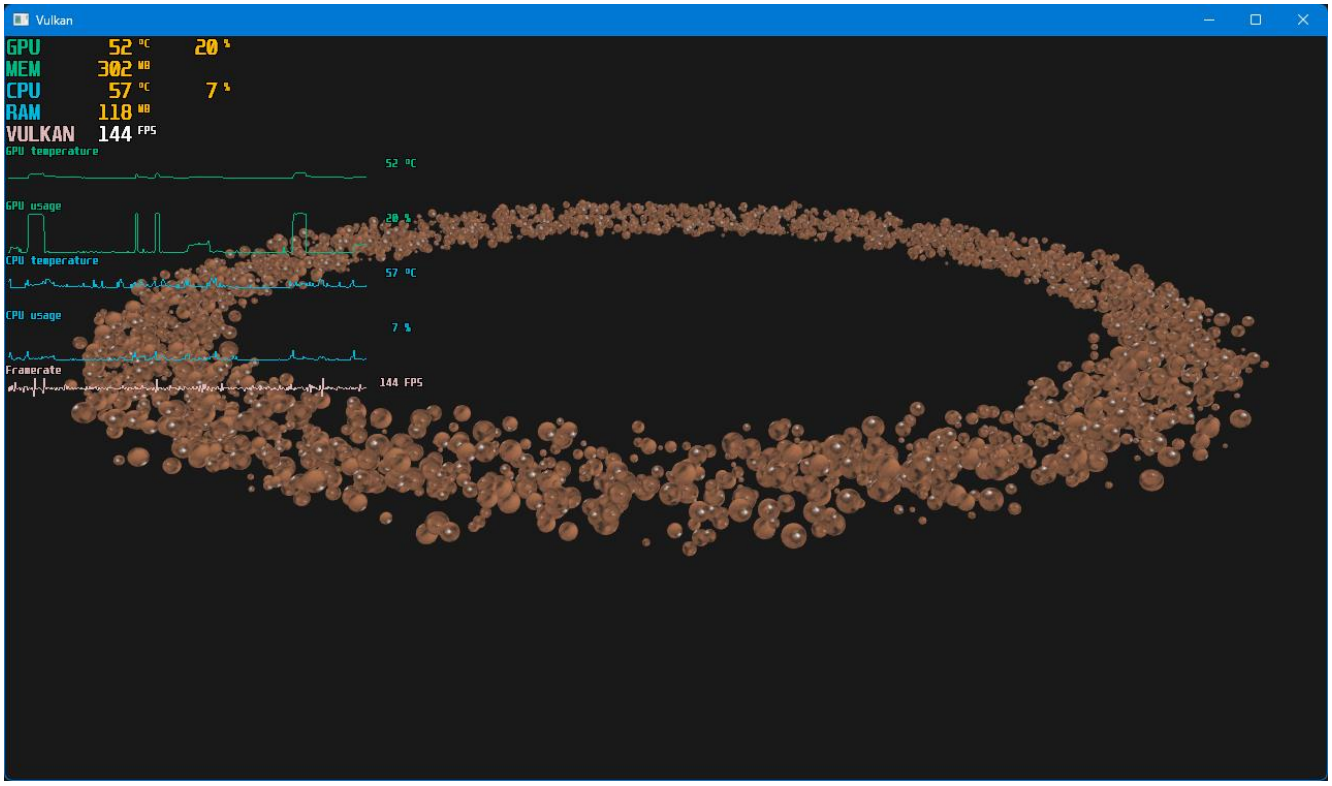

Рисунок 4.33 – Інстансований рендеринг 3000 планет з вертикальною синхронізацією для Vulkan (рисунок виконаний самостійно)

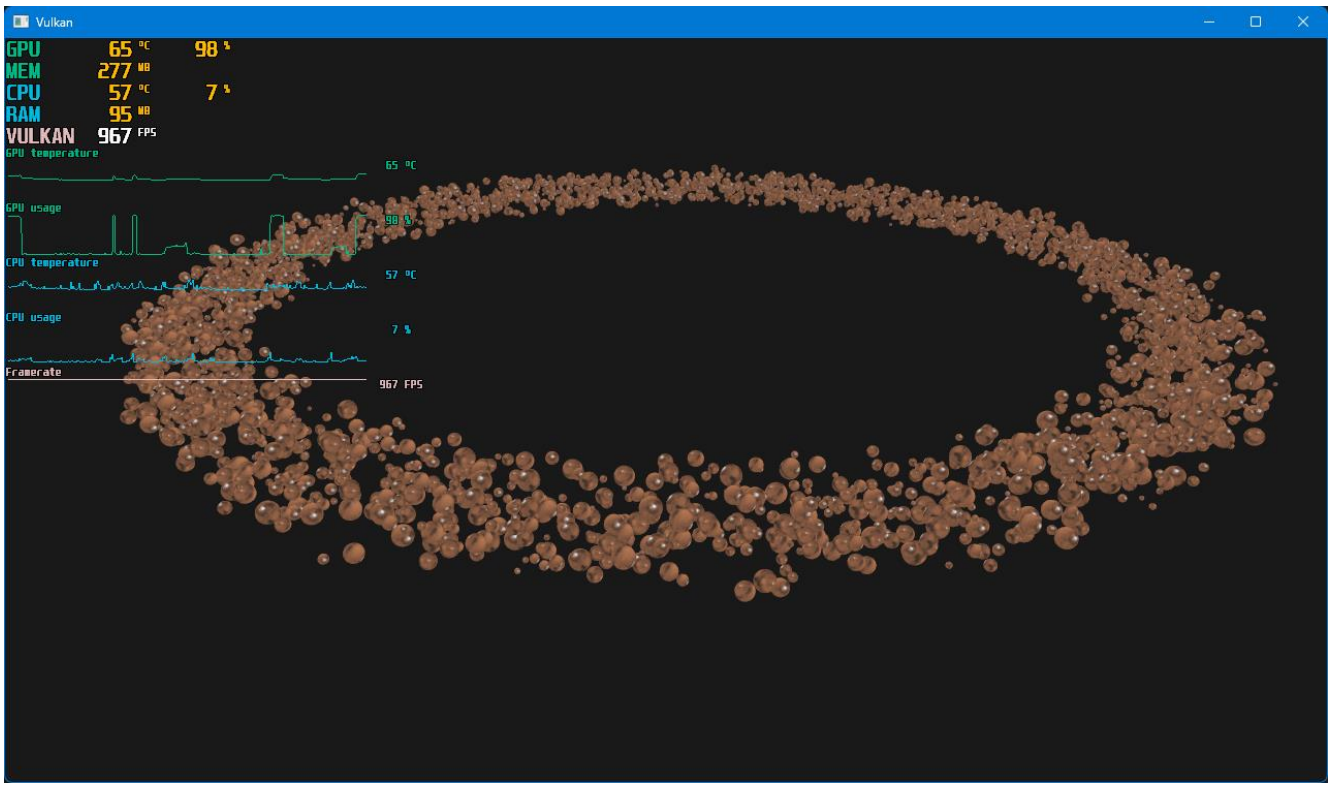

Рисунок 4.34 – Інстансований рендеринг 3000 планет без вертикальної синхронізації для Vulkan (рисунок виконаний самостійно)

Рисунок 4.35 – Всі наші об’єкти з вертикальною синхронізацією для Vulkan (рисунок виконаний самостійно)

Рисунок 4.36 – Всі наші об’єкти без вертикальної синхронізації для Vulkan (рисунок виконаний самостійно)

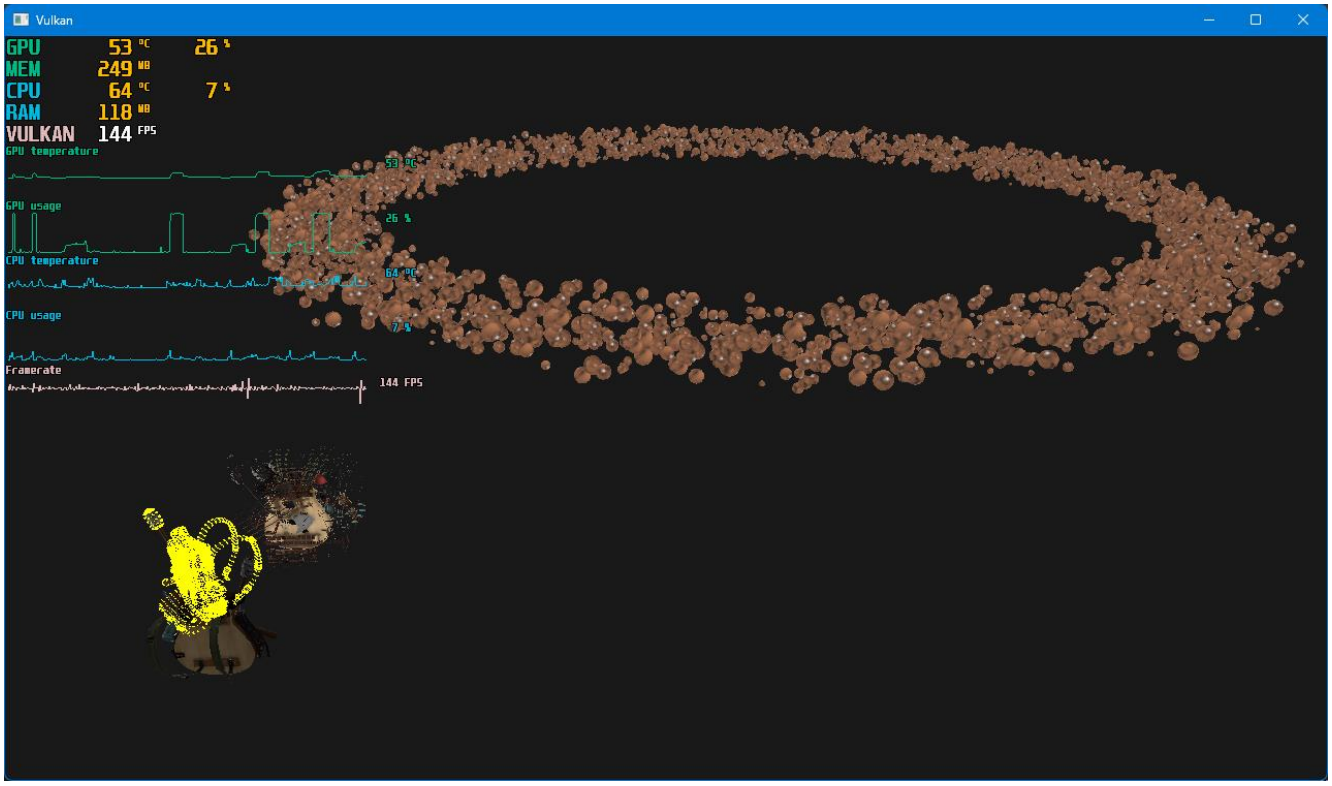

Рисунок 4.37 – Рендеринг без згладжування з вертикальною синхронізацією для Vulkan (рисунок виконаний самостійно)

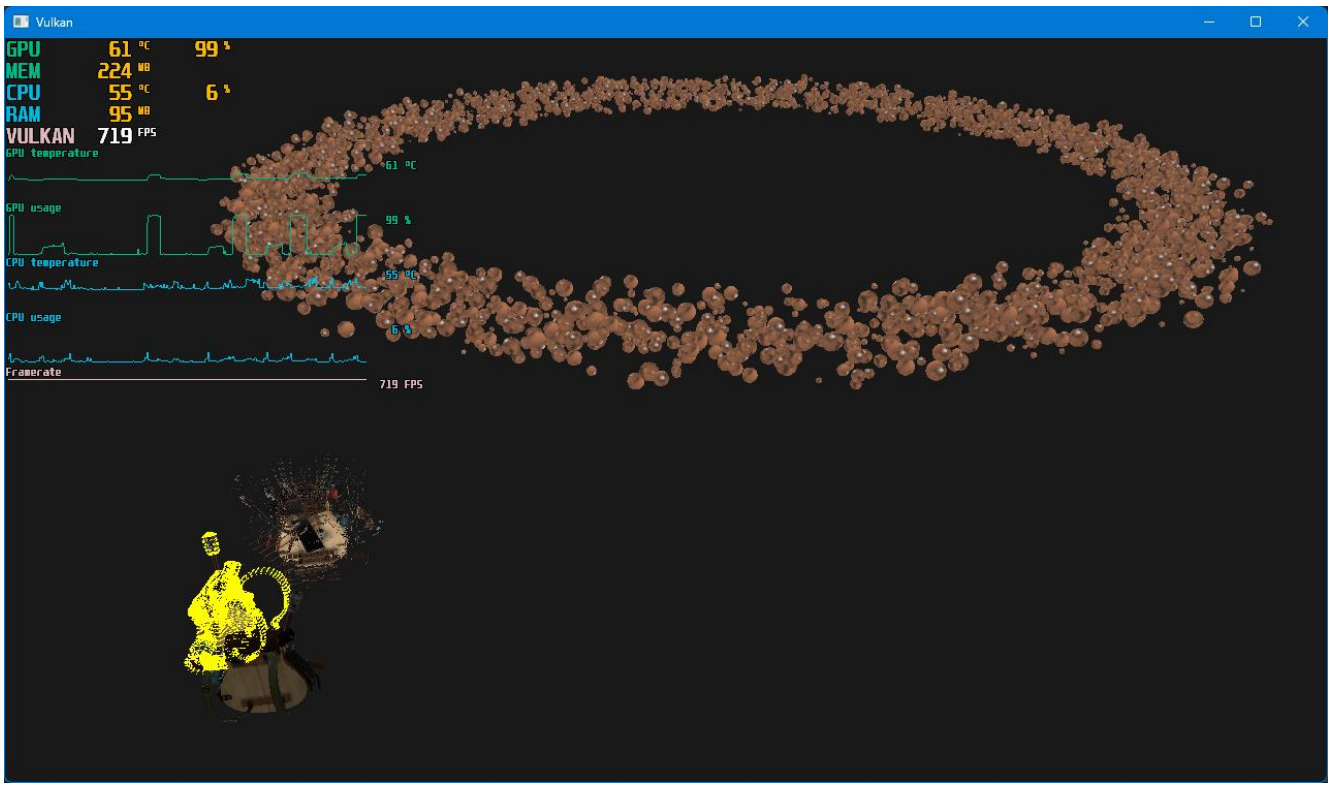

Рисунок 4.38 – Рендеринг без згладжування і вертикальної синхронізації для Vulkan (рисунок виконаний самостійно)

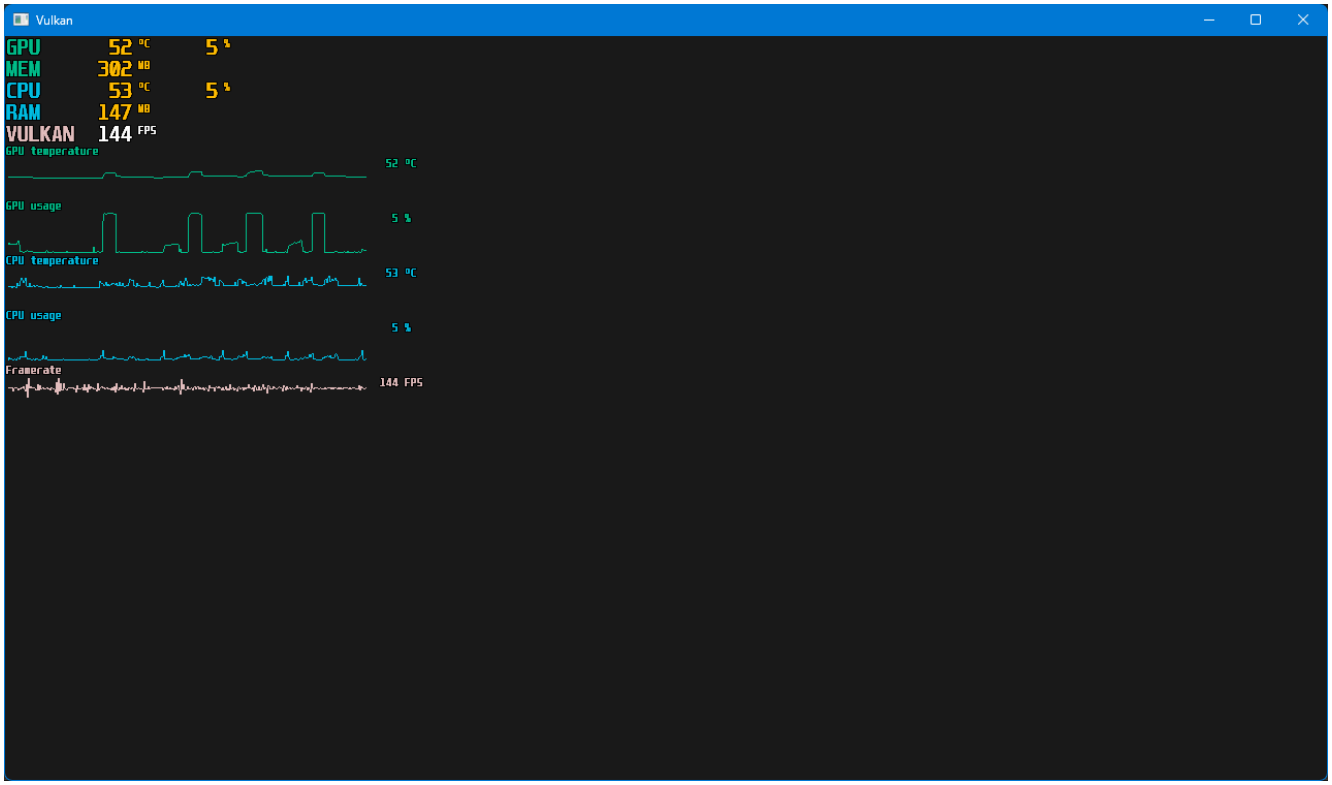

Рисунок 4.39 – Сцена без предметів з вертикальною синхронізацією для Vulkan (рисунок виконаний самостійно)

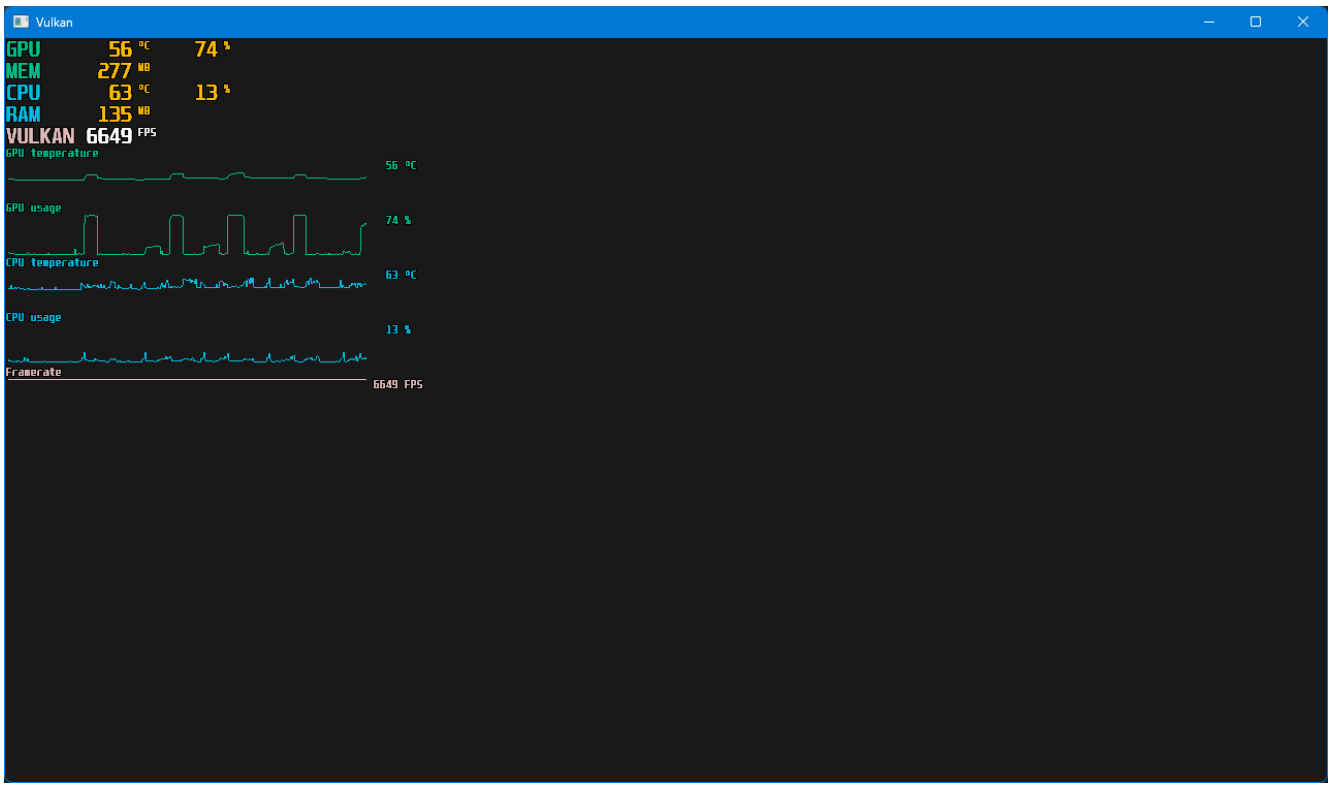

Рисунок 4.40 – Сцена без предметів без вертикальної синхронізації для Vulkan (рисунок виконаний самостійно)

Так само, після завершення всіх тестів розраховуємо середнє значення кожного критерія та заносимо їх в таблицю 4.2.

Таблиця 4.2 – Результати для API Vulkan (таблиця виконана самостійно)

	GPU T	GPU	GPU M	CPU T	CPU	RAM	FPS
Шейдер освітл. з VSync	50	6	302	50	5	126	144
Шейдер освітл. без VSync	62	92	277	62	10	96	4700
Геом. шейдер з VSync	49	5	302	50	5	116	144
Геом. шейдер без VSync	58	88	277	62	11	97	5350
Інший геом. шейдер з VSync	50	8	302	50	6	113	144
Інший геом. шейдер без VSync	57	95	277	58	9	97	3000

Кінець таблиці 4.2

	GPU T	GPU	GPU M	CPU T	CPU	RAM	FPS
Комбінація із геом. шейдерів з VSync	50	8	302	51	5	119	144
Комбінація із геом. шейдерів без VSync	58	95	277	60	8	95	3200
Геом. шейдери та освітлення з VSync	48	8	302	48	5	119	144
Геом. шейдери та освітлення без VSync	60	95	277	60	8	97	2900
Рендер. планет з VSync	50	20	302	50	5	121	144
Рендер. планет без VSync	66	98	277	57	7	97	970
Інстансований рендер. планет з VSync	51	18	302	50	5	115	144
Інстансований рендер. планет без VSync	65	98	277	60	7	95	970
Всі предмети з VSync	48	23	302	48	5	117	144
Всі предмети без VSync	62	99	277	54	8	102	830
Всі об’єкти без згладж., з VSync	51	26	249	51	5	118	144
Всі об’єкти без згладж. і VSync	59	99	224	53	5	95	710
Порожня сцена з VSync	48	5	302	50	5	119	144
Порожня сцена без VSync	52	75	277	60	11	96	5900

Після занесення всіх даних до таблиці ми провели дослідження для інтерфейсу Vulkan і можемо переходити до аналізу. Але спочатку проведемо тести для виявлення, як інтерфейси справляються з об’єктами, яких не видно на екрані.

4.4 Проведення експериментів з рендерингом поза екраном

Проведемо останні 12 тестів для рендерингу об’єктів поза екраном для виявлення продуктивності на сцені для всіх наших об’єктів (див. рис. 4.41-4.52).

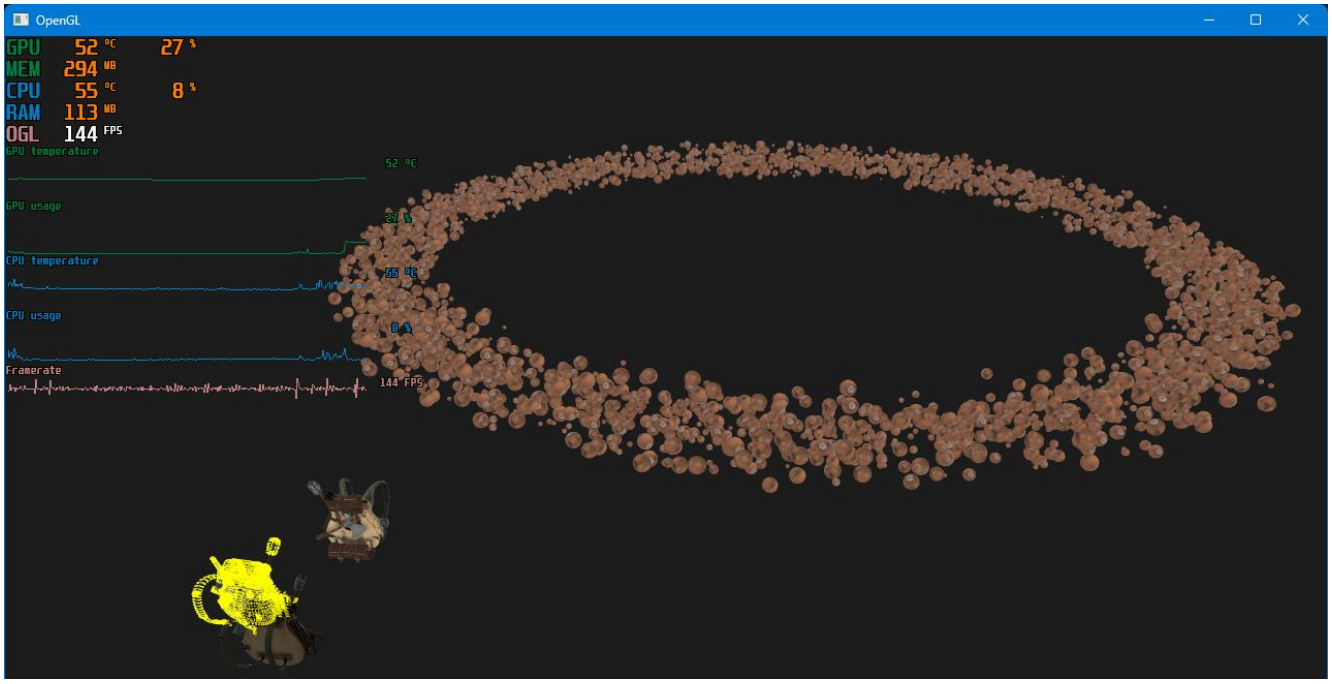

Рисунок 4.41 – Рендеринг всіх об’єктів з вертикальною синхронізацією для OpenGL (рисунок виконаний самостійно)

Рисунок 4.42 – Рендеринг всіх об’єктів без вертикальної синхронізації для OpenGL (рисунок виконаний самостійно)

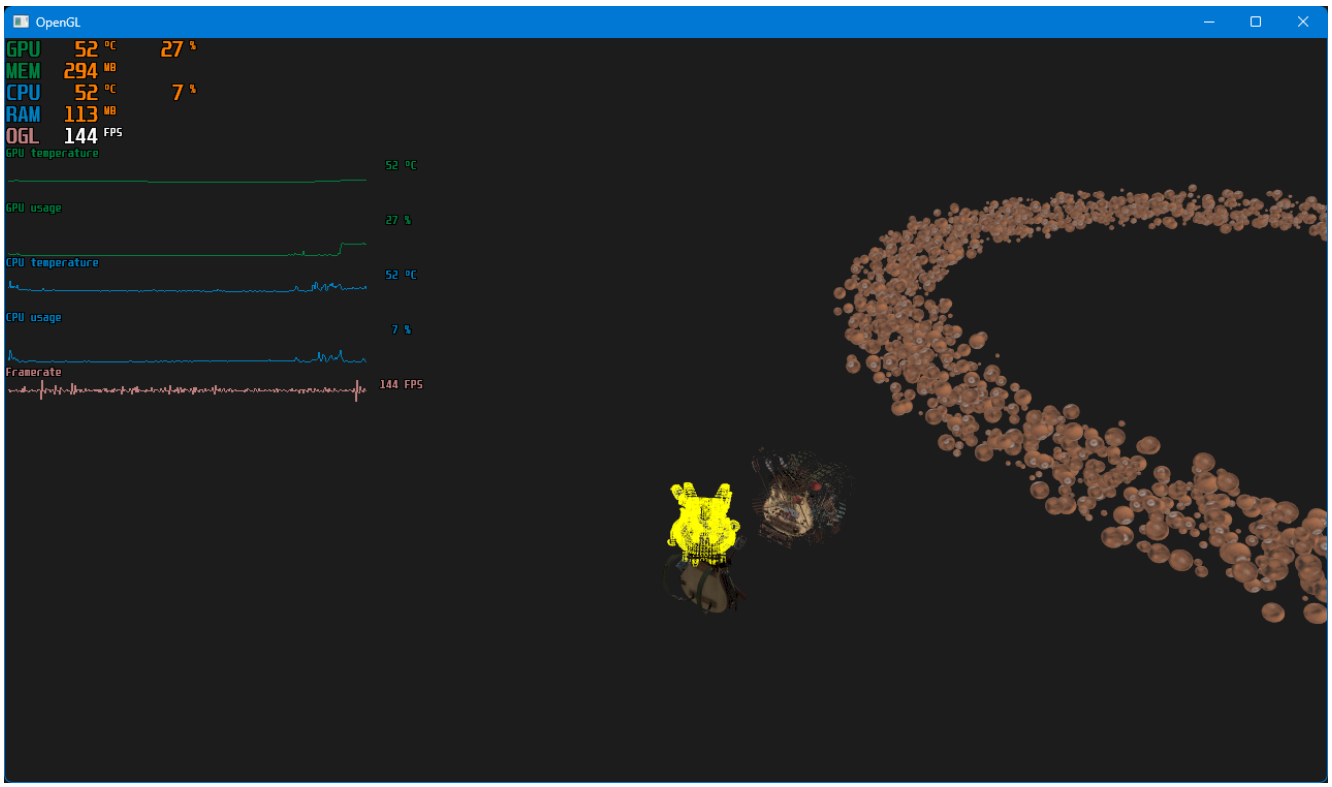

Рисунок 4.43 – Рендеринг половини планет з вертикальною синхронізацією для OpenGL (рисунок виконаний самостійно)

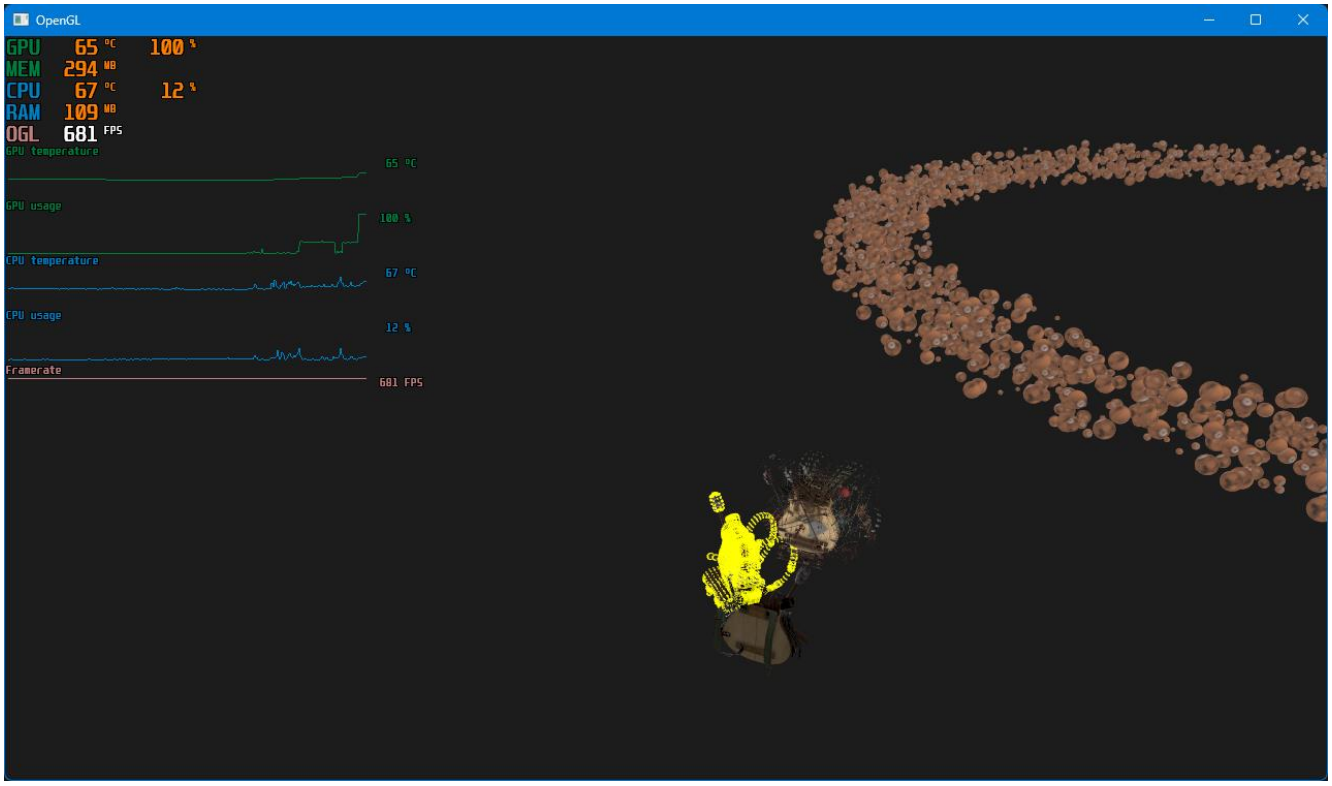

Рисунок 4.44 – Рендеринг половини планет без вертикальної синхронізації для OpenGL (рисунок виконаний самостійно)

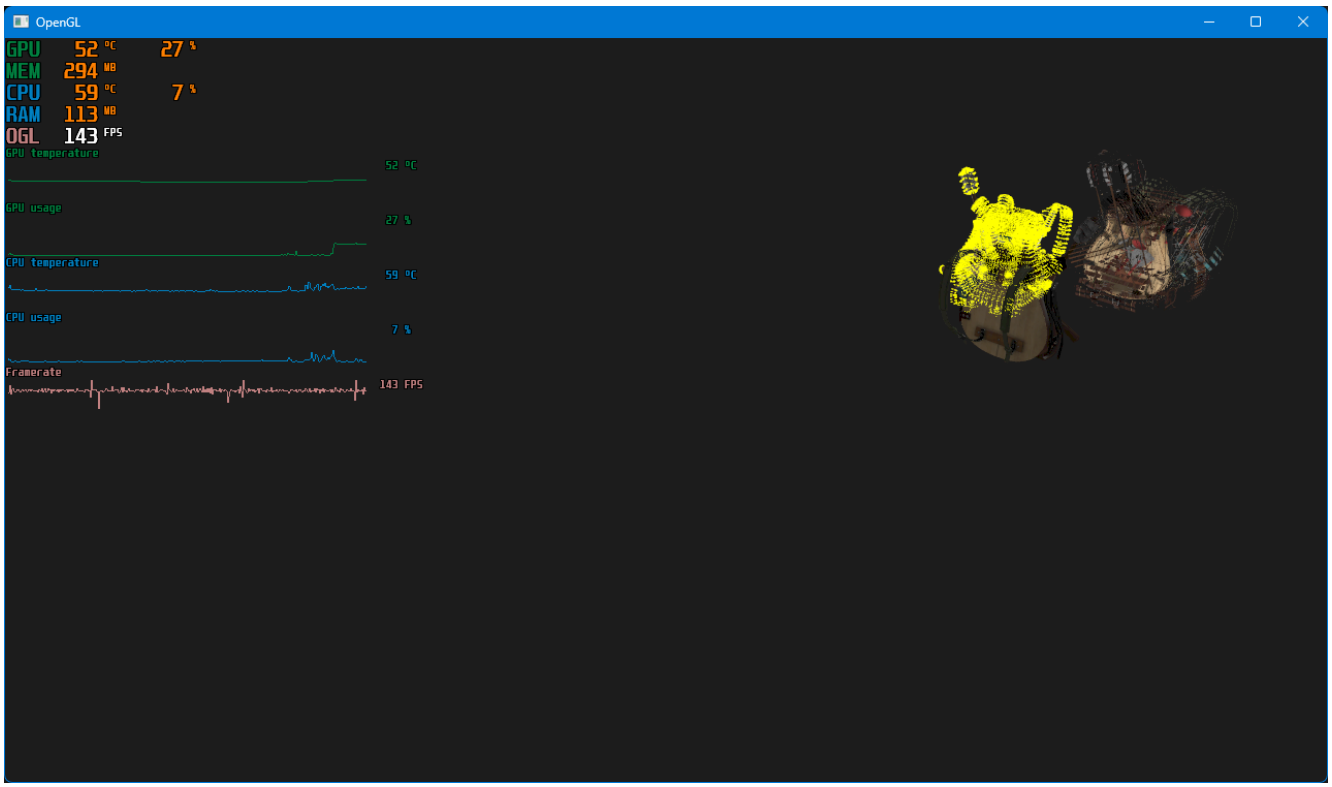

Рисунок 4.45 – Рендеринг без планет з вертикальною синхронізацією для OpenGL (рисунок виконаний самостійно)

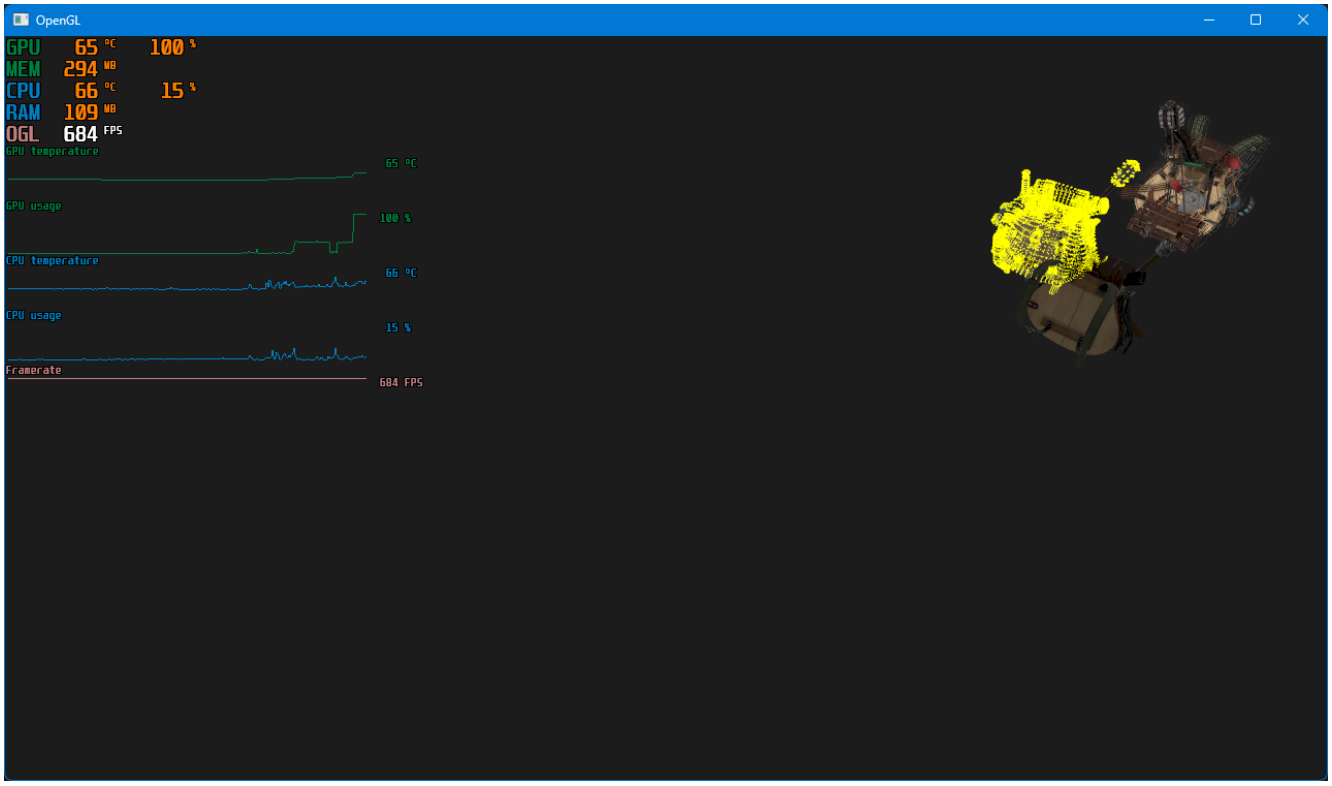

Рисунок 4.46 – Рендеринг без планет і вертикальної синхронізації для OpenGL (рисунок виконаний самостійно)

Рисунок 4.47– Рендеринг всіх об’єктів з вертикальною синхронізацією для Vulkan (рисунок виконаний самостійно)

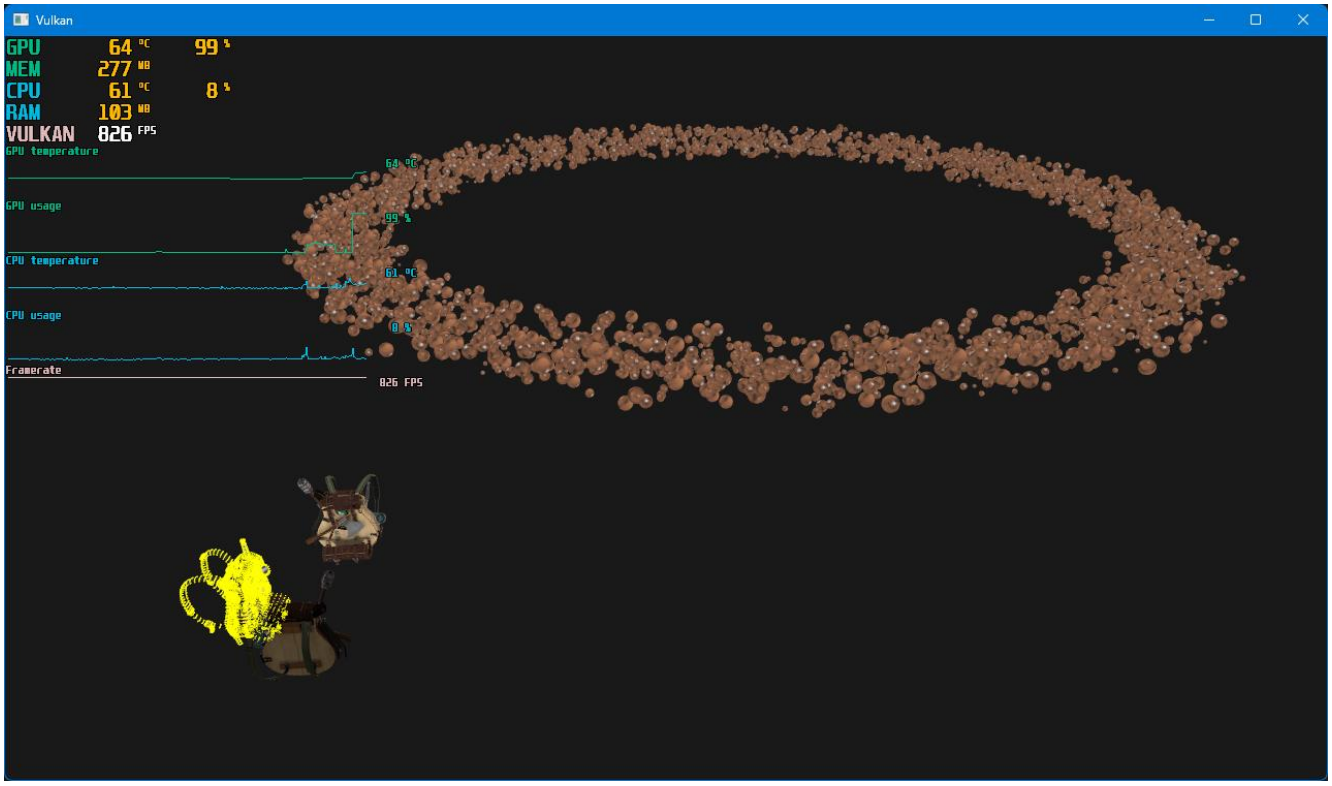

Рисунок 4.48 – Рендеринг всіх об’єктів без вертикальної синхронізації для Vulkan (рисунок виконаний самостійно)

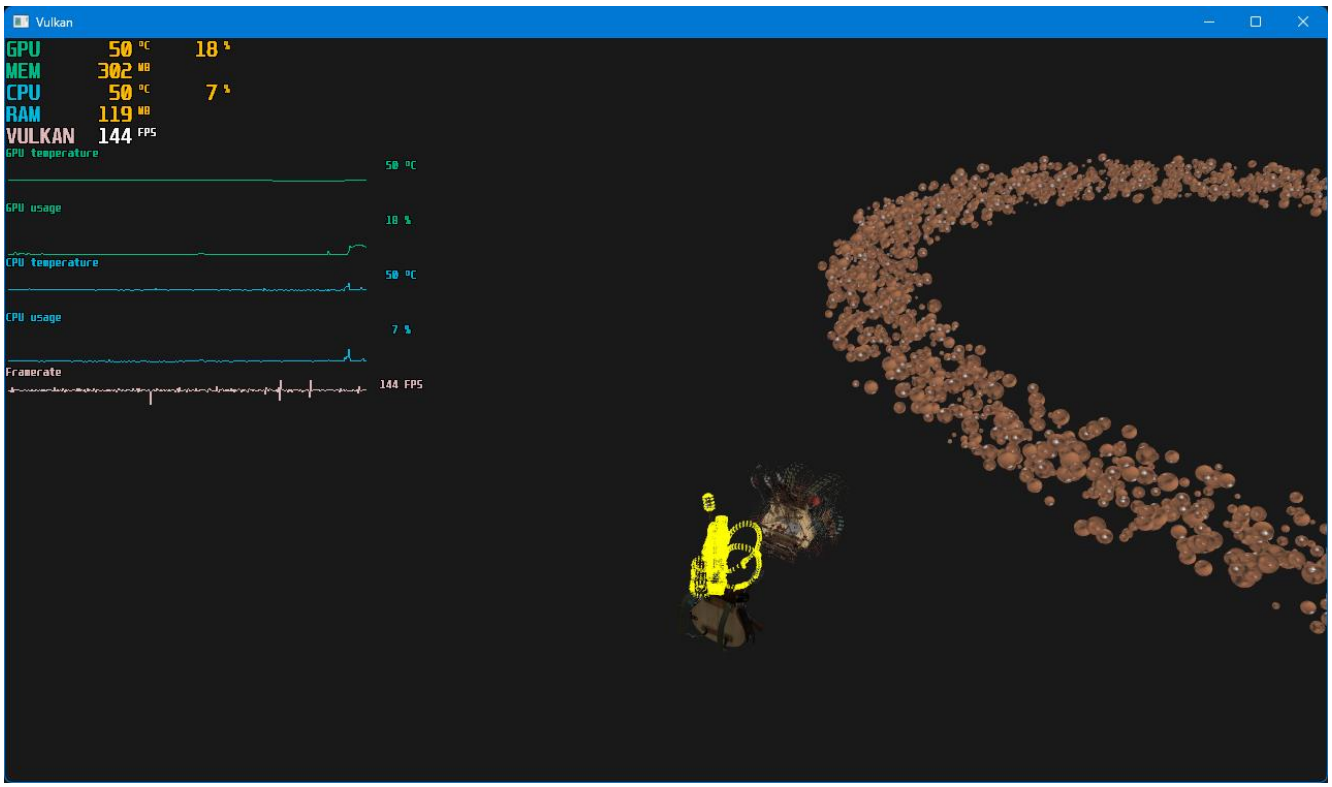

Рисунок 4.49 – Рендеринг половини планет з вертикальною синхронізацією для Vulkan (рисунок виконаний самостійно)

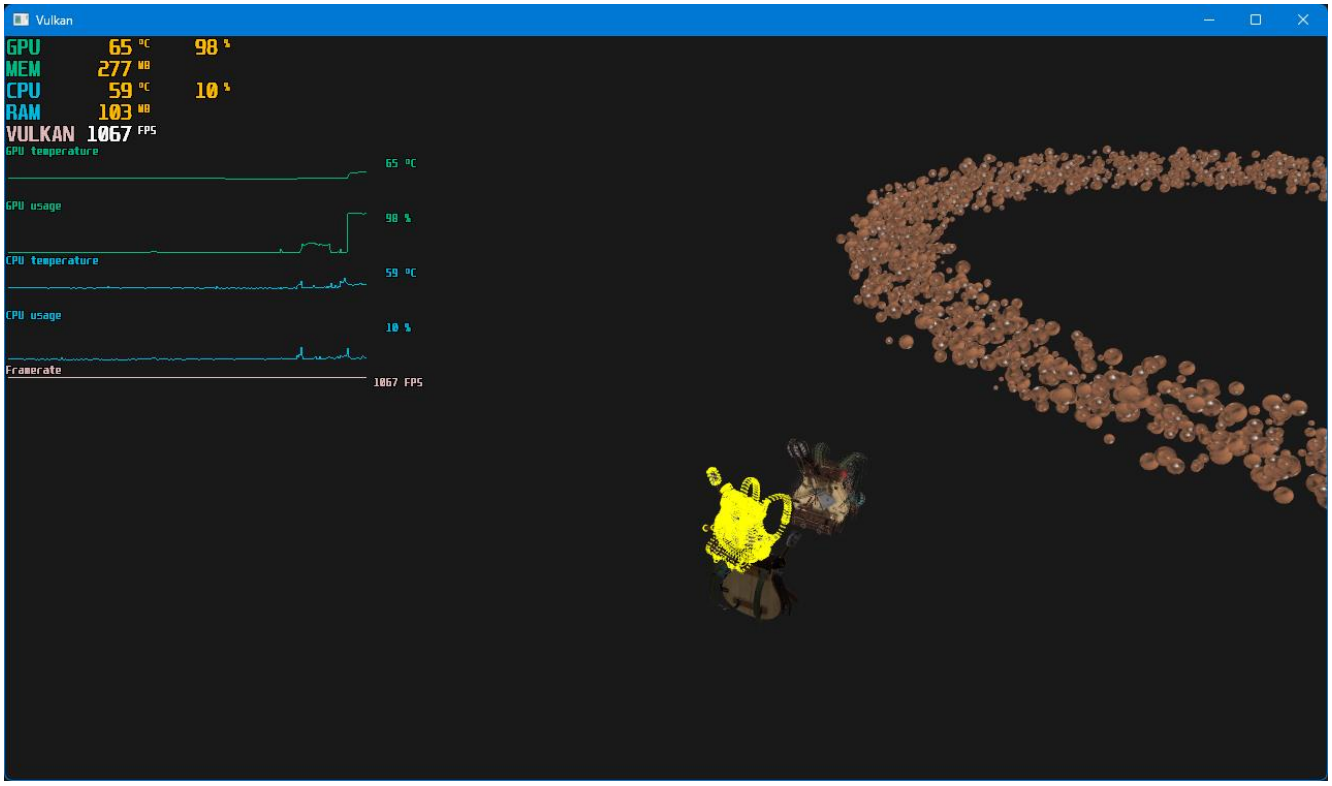

Рисунок 4.50 – Рендеринг половини планет без вертикальної синхронізації для Vulkan (рисунок виконаний самостійно)

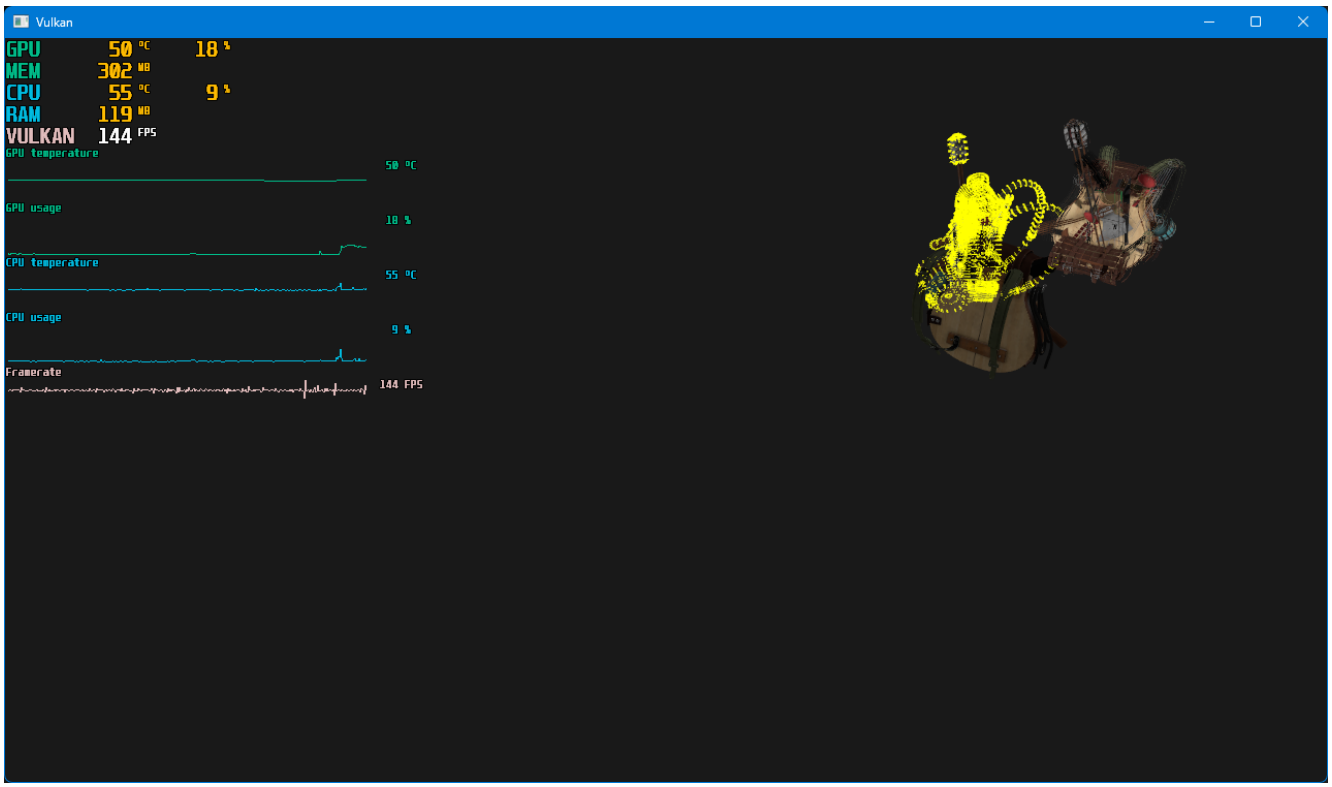

Рисунок 4.51 – Рендеринг без планет з вертикальною синхронізацією для Vulkan (рисунок виконаний самостійно)

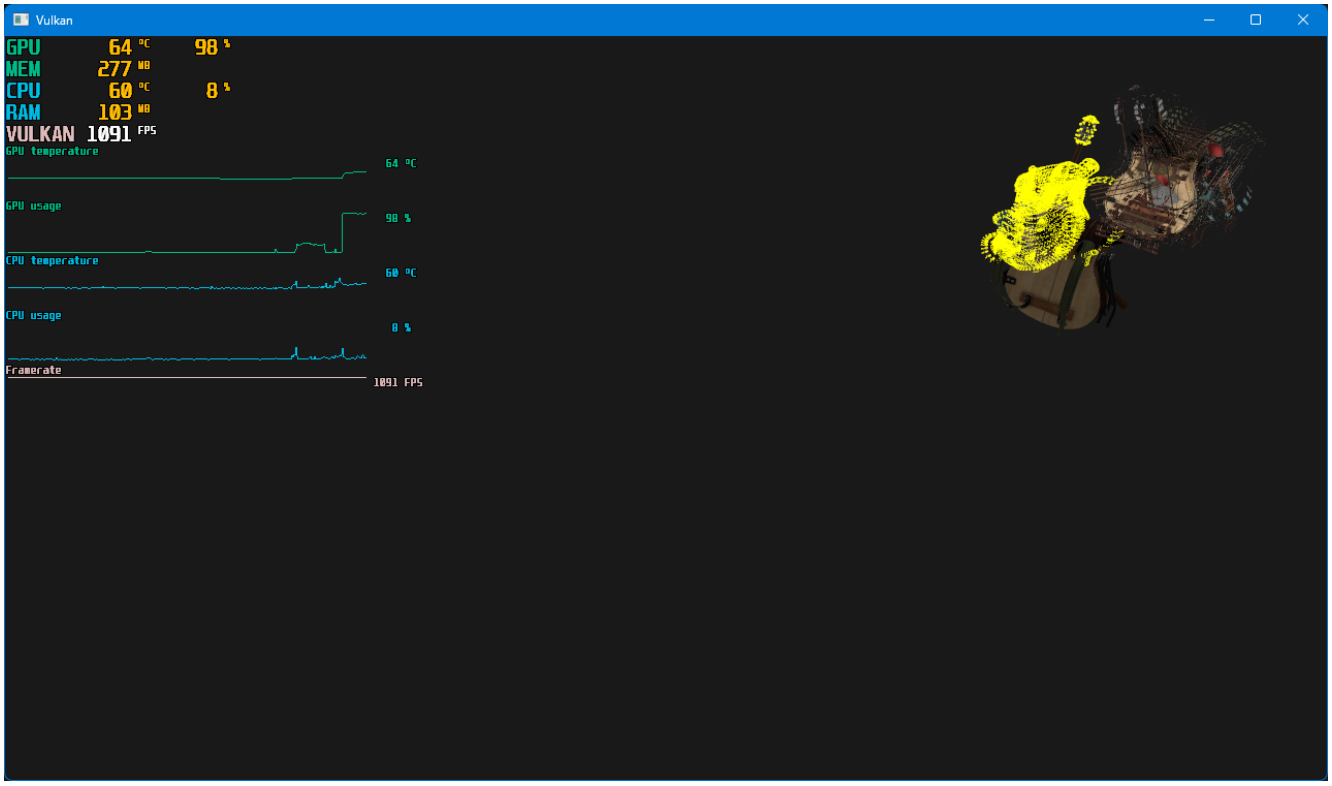

Рисунок 4.52 – Рендеринг без планет і вертикальної синхронізації для Vulkan (рисунок виконаний самостійно)

Після завершення спеціальних тестів розраховуємо середнє значення кожного критерія та заносимо їх в таблицю 4.3 для OpenGL та в таблицю 4.4 для Vulkan.

Таблиця 4.3 – Результати рендерингу поза екраном для OpenGL (таблиця виконана самостійно)

	GPU T	GPU	GPU M	CPU T	CPU	RAM	FPS
Всі об’єкти з VSync	54	27	294	53	4	109	144
Половина планет з VSync	54	26	294	55	5	109	144
Тільки рюкзаки з VSync	54	26	294	53	6	109	144
Всі об’єкти без VSync	66	100	294	66	7	114	680
Половина планет без VSync	66	100	294	66	8	114	680
Тільки рюкзаки без VSync	65	100	294	70	11	114	680

Таблиця 4.4 – Результати рендерингу поза екраном для Vulkan (таблиця виконана самостійно)

	GPU T	GPU	GPU M	CPU T	CPU	RAM	FPS
Всі об’єкти з VSync	51	23	302	50	6	109	144
Половина планет з VSync	50	19	302	50	7	101	144
Тільки рюкзаки з VSync	50	18	302	52	8	100	144
Всі об’єкти без VSync	65	99	277	57	6	103	800
Половина планет без VSync	66	98	277	70	6	102	1050
Тільки рюкзаки без VSync	64	98	277	69	9	101	1100

Після занесення всіх основних та спеціальних даних можемо перейти до аналізу кожної сцени з та без вертикальної синхронізації для обох видів тестів.

4.5 Аналіз отриманих даних

Почнемо аналіз 20 основних тестів. Але перед цим створимо таблицю 4.5, щоб чітко бачити різницю між отриманими даними з двох інтерфейсів. Тому дані API OpenGL віднімалися від даних API Vulkan, щоб також бачити різницю між сучасним і старішим інтерфейсами.

Таблиця 4.5 – Різниця результатів дослідження для перших 20 тестів (таблиця виконана самостійно)

	GPU T	GPU	GPU M	CPU T	CPU	RAM	FPS
Шейдер освітл. з VSync	2	1	8	2	0	1	0
Шейдер освітл. без VSync	7	56	-17	-2	-2	-29	2900
Геом. шейдер з VSync	0	-1	8	-1	-2	-13	0
Геом. шейдер без VSync	4	48	-17	-3	1	-32	3190
Інший геом. шейдер з VSync	-1	0	8	-3	-1	4	0
Інший геом. шейдер без VSync	0	20	-17	-6	-3	-12	720
Комбінація із геом. шейдерів з VSync	-1	-1	8	0	0	10	0
Комбінація із геом. шейдерів без VSync	0	35	-17	9	-5	-14	1400
Геом. шейдери та освітлення з VSync	-3	-2	8	-5	0	10	0
Геом. шейдери та освітлення без VSync	1	50	-17	-11	-3	-12	1670

Кінець таблиці 4.5

	GPU T	GPU	GPU M	CPU T	CPU	RAM	FPS
Рендер. планет з VSync	-2	-3	8	-8	-3	7	0
Рендер. планет без VSync	7	59	-17	-16	-4	-17	625
Інстансований рендер. планет з VSync	-3	-7	8	-4	0	6	0
Інстансований рендер. планет без VSync	-3	-2	-17	0	2	-14	193
Всі предмети з VSync	-6	-4	8	-6	0	8	0
Всі предмети без VSync	-6	-1	-17	-11	0	-7	157
Всі об’єкти без згладж., з VSync	-4	0	0	-4	0	9	0
Всі об’єкти без згладж. і VSync	-8	-1	-25	-13	-3	-14	7
Порожня сцена з VSync	-4	1	8	-3	0	9	0
Порожня сцена без VSync	-3	35	-17	-6	-2	-14	2300

Відразу відмітимо, що невеликою різницею ми будемо нехтувати або не брати до уваги взагалі, а температура буде додатковим параметром оцінки, але не вирішальним. Розглядати будемо кожну сцену, починаючи з першої.

Перша сцена представлена двома рюкзаками з шейдером, який симулює освітлення за моделлю Блінна-Фонга. З вертикальною синхронізацією бачимо майже однакові параметри, крім відеомпа’яті, де в Vulkan трохи більше займає. Можливо, це пов’язано з додаванням якихось додаткових параметрів від самого інтерфейсу чи від нас самих, тобто розробників. Але все одно такий показник зовсім не є критичним, тому забуваємо про нього. А ось без вертикальної синхронізації бачимо високу різницю між показниками GPU. Коли в Vulkan воно досягає більше 90%, то в OpenGL тримається на відмітці в 36%, що показує, що

OpenGL погано справляється з максимальним навантаженням без обмежень на відеокарту, бо набагато гірше виконується паралельність дій, тобто їх асинхронність виконання. Звідси і високий показник FPS у Vulkan, який перевищує аж на 2900 кадрів (в 2,6 разів). А ще до цього можна додати різну мову шейдерів, де SPIR-V працює швидше, тому це тем може впливати на такий показник. Також можна відмітити різницю відеопамяті та оперативної пам’яті. Скоріше за всього, вони пов’язані з буфером кадрів для виводу на екран. А так як не потрібно зберігати кадри, які ще зберігаються з семплінгом в 8 одиниць, бо вони відразу посилаються на екран, пам’ять може зменшуватися. Це ми неодноразово побачимо далі. А температура на відоепроцесорі піднялась через підвищення рівня завантаженості на нього.

Наступною сценою є перший геометричний шейдер. Тут відрізняються майже ті самі показники. FPS майже в 2,5 рази більший у Vulkan.

Розглянемо сцену з другим геометричним шейдером. Тут знову майже те саме, окрім CPU та GPU. Центральний процесор має меншу завантаженість у Vulkan, ні в OpenGL, звідси і температура його менше, а GPU працює на показнику в 95%. Центральний процесор може працювати менше, якщо, наприклад, він посилає весь рендеринг однією командою, або більше, якщо він постійно взаємодіє з GPU. Тому показник, що він завантажений менше не свідчить про те, що маємо менше кадрів за секунду, а навпаки – більше (на 720), тим паче із вищим показником роботи відеопроцесора.

Далі у нас комбінація геометричних шейдерів. Аналогічно як і з попередньою сценою, але тут при вимкненій вертикальній синхронізації у Vulkan FPS майже в 1,8 більший, ніж в інтерфейсі OpenGL, що свідчить про те що перший справляється з різними геометричними шейдерами краще, ніж другий. Але температура процесора більше, навіть при нижчому показнику його завантаженості. Це може бути пов’язано із робою всіх ядер процесора, навіть енергоефективних, тому середнє значення виходить менше, хоча працювати він може більше. Можливо, через це і показник FPS такий великий, що процесор краще працює.

Наступною сценою є комбінація всіх шейдерів для рюкзаків. Аналогічна ситуація, як і з другим геометричним шейдером, окрім того, що кадрів за секунду на 1670 більше (в 2,35 рази) у Vulkan, і завантаженість GPU на 50%.

Перейдемо до сцени з рендерингом 3000 планет. Така сама ситуація, як і з попередньою сценою: FPS з вертикальною синхронізацією більший на 625 кадрів (в 2,8 рази), але температура GPU більша на 7 одиниць. Це може свідчить про її високу завантаженість, що так і є, бо вона більша на 59% (98% у Vulkan).

Для інстансованого рендерингу аналогічна ситуація, але дані суттєво відрізняються. Якщо при звичайному рендерингу ми дуже сильно вигравали, то тут тільки на 193 кадри в секунду або в 1,25 рази, а завантаженість графічного процесора чи центрального процесора приблизно така сама. А якщо поглянути на дані тільки Vulkan, то можемо помітити, що вони майже не відрізняються. Це свідчить про те, що інстансинг для Vulkan не дає виграшу в продуктивності. Це може бути пов’язано з тим, що всі команди на графічному процесорі працюють асинхронно, особливо шейдери, також потрібно не забувати, що команди на звичайний рендеринг всі давалися послідовно, не було команди для рендерингу іншого об’єкта, крім планет, тому в реальному проекті, де велика кількість об’єктів, шейдерів, механік тощо можуть бути зовсім інші дані, тому в нашому випадку виграш від інстансингу в продуктивності ми не отримали, але виграв інтерфейс OpenGL, який має на 430 кадрів більше, ніж без інстансингу, і завантаженість процесора стала меншою, що так і повинно бути через зменшення 3000 команд до однієї, а графічного процесора – більша, через те, що він відразу рендерить ці 3000 планет, а не чекаю від процесора команди на візуалізацію. Швидкість між процесором і відеокартою дуже низька, тому розробникам радять менше разів звертатися до них з невеликою кількістю команд, а більше – з великим об’ємом, щоб компенсувати низьку швидкість, як ми це довели з інстансингом.

Наступною є сцена із всіма нашими об’єктами. Дані не сильно відрізняються, але відмітимо температура процесора стала меншою на 11 одиниць, що може свідчити про меншу його роботу, а графічний процесора став працювати на 6%

менше без втрати в продуктивності, а навпаки – кількість кадрів в секунду на 157 більше у Vulkan (830), ніж в OpenGL (673).

Якщо вимкнути згладжування, то отримаємо, що кількість кадрів за секунду майже не змінилися на обох інтерфейсах, але температура відеопроцесора та центрального процесора суттєво менша, а кількість відеопам'яті теж, через менший об'єм текстур, хоча не дуже сильно.

Остання сцена не має ні шейдерів, ні об'єктів, тому тут перевіряється максимально допустимий FPS. В API Vulkan він становить 5900 кадрів за секунду, а в OpenGL – 3600 кадрів. Це пов'язано з тим, що OpenGL не завантажує сучасну відеокарту на максимум, бо вона працює на 40%, коли в Vulkan – 75%.

Після аналізу всіх головних тестів перейдемо до тестів рендерингу об'єктів поза екраном. Для цього також створимо додаткову таблицю 4.6, щоб краще бачити різницю між Vulkan та OpenGL.

Таблиця 4.6 – Різниця результатів рендерингу для об'єктів поза екраном (таблиця виконана самостійно)

	GPU T	GPU	GPU M	CPU T	CPU	RAM	FPS
Всі об'єкти з VSync	-3	-4	8	-3	2	0	0
Половина планет з VSync	-4	-7	8	-5	2	-8	0
Тільки рюкзаки з VSync	-4	-8	8	-1	2	-9	0
Всі об'єкти без VSync	-1	-1	-17	-9	-1	-11	120
Половина планет без VSync	0	-2	-17	4	-2	-12	370
Тільки рюкзаки без VSync	-1	-2	-17	-1	-2	-13	420

Якщо подивимося на тести інтерфейсу OpenGL, то побачимо, що незважаючи на те, що знаходиться поза екраном, дані майже не змінюються як з вертикальною синхронізацією, як і без неї, окрім процесора, де пропорційно зі зменшенням

об’єктів збільшується його завантаженість. А ось для Vulkan ці дані вже інші: чим менше предметів знаходиться в межах екрану, тим менше навантаження на графічний процесор і тим більше кількість кадрів в секунду, але також і більше завантаженість центрального процесора, як і для OpenGL. Останні дані, при цьому, змінюються по-різному: для OpenGL найменшим є 4% з вертикальною синхронізацією, а найбільший 11% без неї, а ось для Vulkan найменшим є 6% як з так і без вертикальної синхронізації, а найбільший 9% без неї. Але температура для останнього інтерфейсу менша, ніж для OpenGL, що може з деякою ймовірністю свідчити про меншу його завантаженість, але стверджувати чітко не будемо, що так воно і є, тому будемо акцентувати погляд тільки на основні дані.

Отже, оцінивши кожну сцену, кожний тест, порівнявши отримані дані, виділимо найбільшу продуктивність інтерфейсів під час конкретної сцени:

- інтерфейс Vulkan при роботі шейдера освітлення показав завантаженість GPU на 92% та FPS у 4700 кадрів без вертикальної синхронізації, що відповідно на 56% та 2900 кадрів більше, ніж OpenGL;
- інтерфейс Vulkan при роботі першого геометричного шейдера показав завантаженість GPU на 88% та FPS у 5350 кадрів без вертикальної синхронізації, що відповідно на 48% та 3190 кадрів більше, ніж OpenGL;
- інтерфейс Vulkan при роботі комбінації геометричних шейдерів показав завантаженість GPU на 95%, CPU на 8% та FPS у 3000 кадрів без вертикальної синхронізації, що відповідно більше на 35% для GPU та 1400 кадрів, менше на 5% для CPU, ніж OpenGL;
- інтерфейс Vulkan при роботі шейдерів для рюкзаків показав завантаженість GPU на 95% та FPS у 2900 кадрів без вертикальної синхронізації, що відповідно на 50% та 1670 кадрів більше, ніж OpenGL;
- інтерфейс Vulkan при роботі звичайного шейдера для 3000 планет показав завантаженість GPU на 98%, CPU на 7% та FPS у 970 кадрів без вертикальної синхронізації, що відповідно більше на 59% для GPU та 625 кадрів, менше на 4% для CPU, ніж OpenGL;

– інтерфейс Vulkan при захваті екраном менше об’єктів, ніж є у віртуальному просторі, показав зменшену завантаженість GPU з вертикальною синхронізацією, більший FPS на 120, 370 та 420 кадрів відповідно від всіх планет до повного їх зникнення порівняно з OpenGL.

Отже, можемо зробити висновок, що інтерфейс Vulkan краще працює з різними шейдерами, навантажує відеокарту на максимум через свою асинхронність дій, виводить багато об’єктів, які подаються однією командою, набагато швидше, збільшує кількість кадрів в секунду, коли об’єкти або їх частини не присутні на екрані.

Скоріше за все це пов’язано з тим, що API Vulkan краще працює з сучасними відеокартами, шейдерами, бо представлені кодом, який виконується швидше, та краще працює з паралельністю дій, тобто асинхронністю, звідси і завантаження графічного процесора на максимум, та сама завантаженість відеокарти, як і в OpenGL, тільки при цьому вона рендерить більше кадрів в секунду, а також краще працює з об’єктами поза екраном.

Можна сказати, що OpenGL краще підходить для розробників невеликих десктопних програм, 2D-ігор або простих (якщо оцінювати на теперішній час) 3D-ігор через швидкість написання коду та достатньою на сучасний час кількістю кадрів в секунду, якщо оцінювати, що ми тестували в 144 кадрів, коли для ігрових консолей часто ставлять 60 кадрів, рідше 30. API Vulkan краще підходить вже для серйозних проектів, тобто сильно навантажених різними графічними можливостями, через те, що для написання коду потрібно більше досвіду та навичок, ніж для OpenGL, але розробники отримують високу оптимізацію графічної сцени. Якщо коротко, то OpenGL економить час розробки, а Vulkan оптимізує графічні сцени.

ВИСНОВКИ

Під час виконання роботи було проаналізовано сферу графічного моделювання на прикладі графічних інтерфейсів OpenGL та Vulkan, порівнюючи їх схожість, подібні функції, методи графічного моделювання в API OpenGL та Vulkan, їх реалізацію та те, як їх можна відтворити в програмі. Було проаналізовано характеристики цих інтерфейсів для кращого розуміння їхнього функціоналу та можливостей.

Було виявили кілька проблем, з якими стикаються розробники програмного забезпечення при використанні візуалізації, таких як продуктивність та простота використання, пов'язані з використанням різних API. Тому були сформовані порівняльні характеристики, щоб виявити найкращі методи за своєю швидкодією, та критерії аналізу їх оптимізації для підвищення продуктивності графічних інтерфейсів OpenGL і Vulkan та дослідження взаємозв'язків між ними.

Для їх використання були розроблені спеціальні тести для цих інтерфейсів, враховуючи реалізацію OpenGL та Vulkan, взаємозв'язок між методами. З'ясування переваг та обмежень кожного з цих інтерфейсів допомогло визначити оптимальний підхід до графічного моделювання в конкретних випадках використання.

Були визначені методи обох інтерфейсів, їх сильні та слабкі сторони. Це дозволило краще зрозуміти, які методи можуть бути найбільш ефективними в конкретних сценаріях розробки графічних додатків.

Результат роботи можна розглядати як досягнення мети аналізу методів для підвищення оптимізації, що надаються графічними інтерфейсами OpenGL та Vulkan. Також дає можливість розробникам виявити потребу переходу готових продуктів з інтерфейсу OpenGL до Vulkan або потребу обрати певний в залежності від швидкості розробки або продуктивності візуалізації програми. Це відкриває шлях для подальших досліджень та розвитку у цій сфері з метою покращення продуктивності та ефективності графічних додатків.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. OpenGL vs Vulkan - That One Game Dev. URL: https://thatonegamedev.com/cpp/opengl-vs-vulkan/ (дата звернення: 26.03.2024).

2. Vulkan vs. DirectX 12: Which Should You Choose? URL: https://www.howtogeek.com/884042/vulkan-vs-directx-12/ (дата звернення: 26.03.2024).

3. Transitioning from OpenGL to Vulkan | NVIDIA Developer. URL: https://developer.nvidia.com/transitioning-opengl-vulkan (дата звернення: 26.03.2024).

4. OpenGL vs. Vulkan: What are the Key Differences? - History-Computer. URL: https://history-computer.com/opengl-vs-vulkan-what-are-the-key-differences/ (дата звернення: 26.03.2024).

5. Unity - Manual: Mipmaps introduction. Unity - Manual: Unity User Manual 2022.3 (LTS). URL: https://docs.unity3d.com/Manual/texture-mipmaps-introduction.html (дата звернення: 01.06.2024).

6. vulkan_best_practice_for_mobile_developers | Vulkan best practice for mobile developers. URL: https://arm-software.github.io/vulkan_best_practice_for_mobile_developers/samples/vulkan_basics.html (дата звернення: 26.03.2024).

7. OpenGL vs DirectX | Key Differences between OpenGL vs DirectX. URL: https://www.educba.com/opengl-vs-directx/ (дата звернення: 26.03.2024).

8. DirectX vs Vulkan | Top Differences of DirectX vs Vulkan. URL: https://www.educba.com/directx-vs-vulkan/ (дата звернення: 26.03.2024).

9. Vlasenko, L.A., Rutkas, A.G. On a differential game in a system described by an implicit differential-operator equation // Diff. Equations 2015 №51, pp. 798–807.

10. Vlasenko, L.A., Rutkas, A.G., Chikrii, A.A. On a Differential Game in a Stochastic System Proc. of the Steklov Inst. of Mathematics, 2020, 309, c. S185-S198.

11. A Comparison of Modern Graphics APIs. URL: https://alain.xyz/blog/comparison-of-modern-graphics-apis. (дата звернення: 26.03.2024).

12. The story of WebGPU — The successor to WebGL | by Eytan Manor | Medium. URL: https://eytanmanor.medium.com/the-story-of-webgpu-the-successor-to-webgl-bf5f74bc036a (дата звернення: 26.03.2024).

13. What is Web Graphics? – GeeksforGeeks. URL: https://www.geeksforgeeks.org/what-is-web-graphics/ (дата звернення: 01.06.2024).

14. Learning Vulkan. URL: https://subscription.packtpub.com/book/game-development/9781786469809/1/ch01lvl1sec8/vulkan-versus-opengl (дата звернення: 26.03.2024).

15. Selecting a Shading Language - OpenGL Wiki. The Khronos Group Inc. URL: https://www.khronos.org/opengl/wiki/Selecting_a_Shading_Language (дата звернення: 01.06.2024).

16. Vlasenko L.A., Rutkas A.G., Chikrii, A.O. Functional-Differential Games with Nonatomic Difference Operator// Ukrainian Mathematical Journal. 2022. №74(2), pp. 186–202.

17. Create the Framebuffers - LunarXchange. URL: https://vulkan.lunarg.com/doc/view/latest/windows/tutorial/html/12-init_frame_buffers.html (дата звернення: 01.06.2024).

18. RivaTuner (RTSS) homepage. www.guru3d.com. URL: https://www.guru3d.com/page/rivatuner-rtss-homepage/ (дата звернення: 01.06.2024).

19. Survival Guitar Backpack - Download Free 3D model by Berk Gedik (@berkgedik). Sketchfab. URL: https://sketchfab.com/3d-models/survival-guitar-backpack-799f8c4511f84fab8c3f12887f7e6b36 (дата звернення: 01.06.2024).

20. Mars - Download Free 3D model by Akshat (@shooter24994). Sketchfab. URL: https://sketchfab.com/3d-models/mars-9c7bbc64d8c74acfa9ec344c0fc10e1a (дата звернення: 01.06.2024).