Тернопільський національний педагогічний університет
імені Володимира Гнатюка

Фізико-математичний факультет
Кафедра інформатики та методики її навчання

Кваліфікаційна робота
на тему:
«Оптимізація рендерингу великих сцен через адаптивне керування пам’яттю GPU»

Спеціальність:
122 Комп'ютерні науки (Інженерія ігрових проектів)

Студента 4 курсу групи ІІП-46
Бухта Роман Тарасович

Керівник:
Вовкодав Олександр Валерійович

Робота захищена з оцінкою:
Національна шкала:_____
Кількість балів:____ Оцінка: ECTS___

Тернопіль 2026

Вступ

Сучасні ігрові, симуляційні та візуалізаційні застосунки оперують сценами з величезною кількістю графічних ресурсів – тисячами текстур, моделей, матеріалів та шейдерів. Обсяг даних, які потрібно зберігати у відеопам'яті GPU під час рендерингу, постійно зростає, тоді як фізичний обсяг VRAM навіть на сучасних відеокартах залишається обмеженим. Це призводить до типових проблем: мікрофризи, просідання частоти кадрів, додатковий обмін даними між оперативною пам'яттю та VRAM, а у крайніх випадках – повна неможливість запуску сцени. Особливо гостро ця проблема стоїть на конфігураціях середнього та нижнього цінового сегмента, де користувач фактично змушений вибирати між якістю зображення і плавністю рендерингу.

Класичні методи оптимізації – mipmapping, LOD-системи, стиснення текстур – вирішують лише частину задачі. Вони зменшують обсяг ресурсів, але не керують їхньою резидентністю в реальному часі. Сучасні підходи, такі як streaming, virtual texturing та sparse residency, демонструють, що ефективніше керувати не самим ресурсом, а тим, які його частини потрібно завантажити в VRAM саме зараз. Однак для дослідницьких та невеликих проєктів ці механізми часто є надмірно складними, нестачово прозорими або тісно пов'язаними зі специфікою конкретного API. Тому актуальною залишається задача побудови простішого, але працездатного механізму адаптивного керування відеопам'яттю, який можна було б реалізувати у рамках типового проєкту і виміряти його ефективність експериментально.

Актуальність теми обумовлена постійним зростанням обсягу графічних ресурсів у сучасних 3D-застосунках і обмеженим обсягом VRAM у пристроях кінцевих користувачів, що робить задачу ефективного керування пам'яттю GPU однією з ключових у напрямку оптимізації рендерингу.

Метою роботи є дослідження принципів адаптивного керування пам'яттю GPU та розробка прототипу системи, яка дозволяє зменшити споживання відеопам'яті при рендерингу великих сцен без істотного зниження продуктивності.

Для досягнення поставленої мети необхідно розв'язати такі завдання:

проаналізувати існуючі підходи до завантаження та керування графічними ресурсами у сучасних рушіях і графічних API;

визначити критерії пріоритету ресурсів та сформулювати вимоги до системи адаптивного керування пам'яттю GPU;

спроєктувати модульну архітектуру системи, що поєднує визначення видимості, пріоритетизацію, асинхронне завантаження, prefetching та політику вивантаження ресурсів;
реалізувати прототип системи у середовищі Unity 6.3 LTS з рендер-пайплайном HDRP;
розробити методику експериментального порівняння адаптивного та статичного режимів роботи;
провести вимірювання та порівняльний аналіз отриманих результатів за показниками використання VRAM, кількості резидентних ресурсів і часу рендерингу кадру.

Об’єктом дослідження є процес рендерингу великих 3D-сцен у сучасних рушіях реального часу.

Предметом дослідження є методи та засоби адаптивного керування пам’яттю GPU під час рендерингу великих сцен.

Методи дослідження: аналіз науково-технічної літератури з комп’ютерної графіки та оптимізації рендерингу; методи об’єктно-орієнтованого проєктування програмних систем; експериментальне моделювання у середовищі Unity HDRP; статистична обробка результатів вимірювання у середовищі Python з використанням бібліотек pandas, numpy та matplotlib.

Наукова новизна роботи полягає у побудові інтегрованої архітектури адаптивного менеджера пам’яті GPU, яка об’єднує визначення видимості через фрустум камери, обчислення інтегрального пріоритету ресурсу за зваженою формулою screen coverage, distance та recency, а також гібридну політику вивантаження, що поєднує LRU за часом бездіяльності з евікцією за пріоритетом у момент перевищення ліміту VRAM.

Практичне значення одержаних результатів полягає у тому, що розроблений прототип може бути використаний як основа для оптимізації реальних проєктів на Unity, які працюють зі сценами великого обсягу. Експериментально показано, що адаптивна система знижує фактичне використання відеопам’яті більш ніж у два рази без істотного впливу на середню частоту кадрів, що дозволяє запускати такі сцени на конфігураціях з меншим обсягом VRAM.

Структура роботи. Кваліфікаційна робота складається зі вступу, трьох розділів, висновків та списку використаних джерел. У першому розділі розглянуто проблему рендерингу великих сцен і проаналізовано існуючі підходи до оптимізації. У другому розділі сформульовано задачу, описано

вхідні дані для прийняття рішень та спроєктовано загальну архітектуру системи. У третьому розділі описано реалізований прототип, методику експериментів і наведено результати порівняння адаптивного та статичного режимів.

Розділ 1. Теоретичні основи оптимізації рендерингу великих сцен через адаптивне керування пам'яттю GPU

1.1 Проблема рендеренгу великих сцен у сучасних ігрових та візуалізаційних системах

Рендеринг великих сцен є однією з найскладніших задач сучасної комп’ютерної графіки, оскільки він вимагає одночасної обробки великої кількості текстур, 3D-моделей, шейдерів, матеріалів освітлення та інших графічних ресурсів. У сучасних ігрових проєктах, симуляторах, системах віртуальної реальності, архітектурної візуалізації та інтерактивних тренажерах сцена може містити тисячі об’єктів, а обсяг текстур і геометрії легко перевищує доступний обсяг відеопам’яті. Це призводить до необхідності оптимізації не лише самого процесу відображення, ай способу зберігання, завантаження та вивантаження ресурсів у пам’ять графічного адаптера [1].

Основна проблема полягає в тому, що відеопам’ять GPU є обмеженим ресурсом. Навіть на сучасних дискретних відеокартах обсяг VRAM не завжди достатній для одночасного розміщення всіх даних, необхідних для відображення великої сцени у високій якості. Якщо рушій намагається завантажити забагато текстур або моделей одночасно, це може призвести до перевищення ліміту пам’яті. У результаті система починає переміщувати дані між оперативною пам’яттю та VRAM, що значно повільніше, ніж прямий доступ до локальної пам’яті GPU. Саме це є однією з причин падіння продуктивності, появи мікрофрізів і відчутних затримок у кадрі.

У великих сценах проблема проявляється особливо гостро під час руху камери або переміщення гравця між різними ділянками рівня. У цей момент рушій повинен швидко визначити, які ресурси стануть потрібними найближчим часом, а які можна тимчасово прибрати з відеопам’яті. Якщо зввантаження виконується із запізненням, гравець бачить затримку в появі текстур, “розмиті” поверхні, різке просідання частоти кадрів або короткочасне зависання зображення. Такі ефекти особливо помітні в іграх з відкритим світом, де сцена постійно змінюється, а також у VR-додатках, де будь-яка затримка швидко погіршує користувацький досвід.

Ще однією важливою причиною складності є те, що різні ресурси мають різний рівень пріоритету. Наприклад, текстури об’єктів, які знаходяться прямо перед камерою, мають бути завантажені в першу чергу, тоді як ресурси для далеких або поки що невидимих ділянок сцени можуть бути відкладені. Аналогічно, об’єкти, що займають більшу площу на екрані, мають більший вплив на сприйняття, ніж дрібні або далекі елементи. Тому просте

завантаження “всього і одразу” не є ефективним. Потрібна система, яка буде адаптуватися до поточного стану сцени, поведінки камери та доступного обсягу пам’яті.

Традиційний підхід до рендерингу полягає в тому, що всі ресурси сцени готуються заздалегідь і завантажуються в пам’ять у повному обсязі. Така схема може бути прийнятною для невеликих рівнів або простих 2D-застосунків , але вона стає неефективною в умовах великих сцен. По-перше, значно зростає стартовий час завантаження гри, рівня чи локації. По-друге, обсяг використаної пам’яті може перевищувати можливості конкретного комп’ютера. По-третє, навіть якщо сцена формально вміщується у VRAM, відсутність механізму пріоритетного керування призводить до нераціонального використання ресурсів. У результаті система витрачає пам’ять на об’єкти, які не беруть участі в кадрі, тоді як критично важливі дані можуть завантажуватися із затримкою.

Проблема стає ще складнішою, якщо врахувати різноманітність апаратних конфігурацій. Одні користувачі мають потужні відеокарти з великою кількістю VRAM, інші працюють на середньому або навіть слабкому обладнанні. Це означає, що універсальна стратегія завантаження ресурсів не завжди ефективна. Система повинна вміти адаптуватися до конкретної конфігурації: на потужних ПК можна зберігати більше ресурсів у пам’яті, а на слабших - частіше виконувати вивантаження, зменшувати деталізацію або підвантажувати тільки найбільш потрібні тайли текстур. Таким чином, адаптивне керування пам’яттю GPU є не просто способом оптимізації, а необхідною умовою стабільної роботи велеких сцен на різному обладнанні.

Окремо слід підкреслити, що візуальна якість сцени безпосередньо залежить від того, наскільки швидко та коректно відбувається обмін даними між системною пам’яттю, накопичувачем і відеопам’яттю. Якщо передача даних організована неефективно, виникають додаткові затримки, які не завжди можна компенсувати навіть високою потужністю GPU. Особливо це помітно при використанні великих текстур високої роздільності, складних матеріалів та великої кількості окремих об’єктів у кадрі. Тому в сучасних системах дедалі частіше застосовують підходи на кшталт стрімінгу ресурсів, LOD, virtual texturing та spare residency, які дозволяють завантажувати тільки ті дані, які справді потрібні в конкретний момент [1].

Таким чином, проблема рендерингу великих сцен полягає не лише у великому навантаженні на графічний процесор, а й у необхідності інтелектуального управління ресурсами. Без адаптивного підходу виникають

перевитрати пам’яті, нерівномірне завантаження системи, затримки відображення, падіння FPS і погіршення загального враження від застосунку. Саме тому дослідження та реалізація механізмів адаптивного керування пам’яттю GPU є актуальним завданням, яке має як теоретичне, так і практичне значення для комп’ютерної графіки та інженерії ігрових проєктів.

1.2 Основні підходи до завантаження графічних ресурсів

У класичному підході до будови сцен усі графічні ресурси завантажуються заздалегідь і зберігаються у пам’яті протягом усього часу роботи сцени. Такий спосіб є простим у реалізації, але він погано масштабується на великі обсяги даних. Його основним недоліком є неефективне використання VRAM: ресурси, які насправді не використовуються у конкретний момент, все одно займають пам’ять і можуть витісняти більш важливі елементи. Для невеликих застосунків це допустимо, але для відкритих світів, великих карт або детальних віртуальних середовищ такого підходу вже недостатньо.

Наступним етапом розвитку стали методи стрімінгу ресурсів, коли дані підвантажуються не одразу, а поступово, залежно від того, що саме потрібно відобразити в поточному чи найближчому кадрі. Такий підхід дозволяє розвантажити пам’ять і зменшити стартовий час сцени. Його особливо активно застосовують для текстур, моделей та ландшафтів, які можна розбивати на частини й завантажувати поетапно. У практиці рендерингу саме стрімінг став базовою концепцією для багатьох сучасних рушіїв [1].

Одним із найпоширеніших рішень є mipmapping, тобто використання набору зображень різної роздільності для однієї текстури. Принцип полягає в тому, що для далеких об’єктів доцільно використовувати менш деталізовані версії текстур, а для близьких – повну якісь. Це зменшує навантаження на пам’ять і пришвидшує доступ до даних. Проте mipmapping сам по собі не вирішує проблему великої кількості ресурсів, оскільки він лише оптимізує якість відображення окремої текстури, а не керує її наявністю у пам’яті. Тому mipmapping зазвичай використовується як частина ширшої системи оптимізації [2].

Ще одним важливим підходом є LOD (Level oj Detail), тобто рівні деталізації для 3D-моделей. Суть методу полягає в тому, що об’єкти які знаходяться далі від камери, можуть відображатися у спрощеному вигляді з меншою кількістю полігонів. Це дозволяє зменшити навантаження не лише на GPU, а й на систему завантаження ресурсів. Однак LOD не роз’язує проблему повністю, оскільки навіть спрощені моделі й текстури все одно потребують все одно

потребують місця у VRAM. Отже, LOD є корисним інструментом оптимізації, але не змінює адаптивне керування пам’яттю [3].

Сучасні графічні системи також використовують virtual textueing. У цьому підході текстура розбивається на дрібні частини – тайли, а в пам’яті зберігаються лише ті тайли, які справді потрібні для відображення поточного кадру. Саме такий прицип реалізується у Streaming Virtual Texturing в Unity, де текстури діляться на тайли та підвантажуться асинхронно, а також у механізмах sparse residency у Vulkan, які дозволяють керувати резидентністю окремих частин зображення або підресурсів. Unity прямо вказує, що Streaming Virtual Texturing розбиває текстури на тайли, а документація Vulkan описує spare residency як механізм управління окремими облястями зображення на рівні sparse image bloks; при цьому для Unity SVT підтримка залежить від графічного API, зокрема DirectX11, DirectX12, Metal і Vulkan [4; 5; 6].

На цьому рисунку 1.1 видно, як одна велика текстура умовно ділитися на інші блоки, частина яких завантажена в пам’ять, а частина – ні. Це допоможе візуально пояснити принцип економії VRAM.

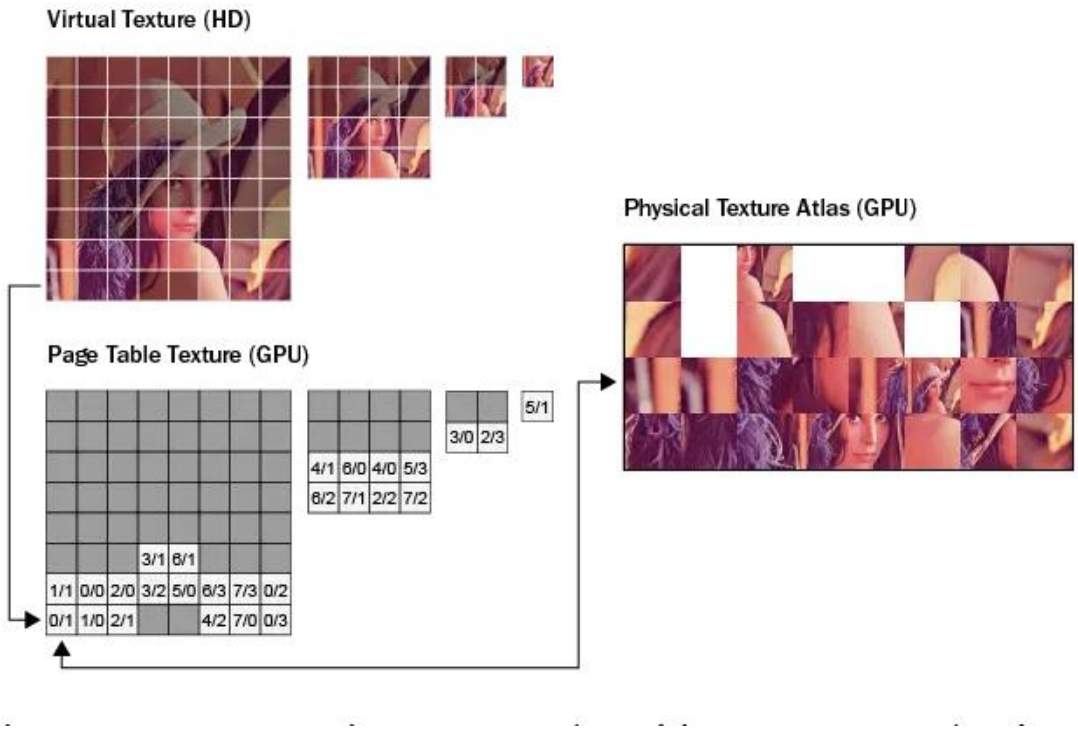

Рисунок 1.1 Структура тайлової текстури

Джерело: [7]

Окремо слід зазначити, що багато підходів до стрімінгу спираються на політику кешування та заміщення даних. Найчастіше застосовуються варіації LRU (Least Recently Used) або близькі до неї евристики, де найменш актуальні ресурси витісняються першими. Такий підхід добре працює тоді, коли

поведінка камери та гравця має більш-менш передбачуваний характер. Проте для великих сцен цього недостатньо, адже деякі ресурси можуть бути нещодавно використаними, але вже не потрібними найближчим часом, тоді як інші – поки що не завантажені, але скоро стануть критично важливими. Через це у практичних системах часто поєднують LRU із пріоритетами, передбаченням руху камери та асинхронним prefetching [1].

Отже, аналіз існуючих підходів показує, що окремі методи – статичне завантаження, streaming, mipmapping, LOD і virtual texturing – вирішують лише частину задачі. Найкращий результат досягається тоді коли вони працюють разом і доповнюються механізмом адаптивного досліджувати не один ізольований метод, а комплексну систему оптимізації.

1.3 Принципи адаптивного керування відеопам'яттю

Адаптивне керування відеопам'яттю GPU – це підхід, за якого система не просто зберігає графічні ресурси, а динамічно вирішує, які з них повинні залишатися в пам'яті, які слід підвантажити, а які можна тимчасово вивантажити без втрати якості сприйняття. Такий механізм базується на постійному аналізі поточного стану сцени, положення камери, видимості об'єктів і доступного обсягу відеопам'яті. У результаті система прагне зберегти баланс між якістю зображення й швидкодією рендерингу.

Ключовим поняттям тут є резидентність ресурсу. Ресурс вважається резидентним, якщо він знаходиться у відеопам'яті та може бути використаний GPU без додаткового завантаження. Якщо ресурс не резидентний, система повинна або чакати його завантаження, або тимчасово використовувати запасний варіант, наприклад нижчу деталізацію. У сучасних технологіях на кшталт Vulkan sparse residency резидентність може керуватися на рівні частин зображення або підресурсів, що робить цей механізм особливо придатним для великих сцен. Unity SVT також працює за принципом поділу текстури на тайли, які завантажуються в міру потреби [6; 4].

Для побудови ефективної системи необхідно визначити правила пріоритету. Найбільш очевидними критеріями є видимість об'єкта у фруструмі камери, відстань до нього, розмір на екрані, важливість для геймплею та передбачуваний напрямок руху гравця. Наприклад, якщо камера рухається вперед по коридору, то система може завчасно підвантажувати текстури й геометрію тих ділянок, які стануть видимими через одну-дві секунди. Таким чином, адаптивність полягає не лише в реакції на дефіцит пам'яті, а й у прогнозуванні майбутніх потреб сцени.

Ще одним важливим елементом є eviction policy, тобто політика вивантаження ресурсів. Коли VRAM наближається до межі, система повинна звільняти місце для нових даних. Простий варіант – вивантажувати найменш використовувані ресурси, але практична система може враховувати також частоту звернень, тип ресурсу, його розмір, вартість повторного завантаження та ймовірність повторного використання. Саме тому адаптивний менеджер пам'яті часто складається з кількох шарів: окремо відстежується активний набір ресурсів, які слід отримати наперед.

Важливою складовою є prefetching, тобто попереднє завантаження ресурів до того, як вони стануть необхідними. Цей механізм допомагає зменшити затримки при різкій зміні сцени, але потребує обережного налаштування. Якщо ж prefetching занадто слабкий, користувач знову зіткнеться з підвантаженням "на льоту". Отже, ефективність механізму полягає в балансі між завчасною підготовкою та раціональним споживанням ресурсів.

На практиці адаптивна система повинна працювати у зв'язці з асинхронним I/O. Це означає, що зчитування даних із диска, їх декодування і передача до GPU не повинні блокувати основний рендеринговий потік. Для цього обробка ресурсів виконується в окремих потоках або за допомогою спеціальних черг завдань. Саме асинхронність дозволяє зменшити ймовірність "просідання" кадру в той момент, коли сцена потребує нового набору текстур. У технічному сенсі це один із найважливіших аспектів усієї роботи, який можна побачити на Рисунку 1.2 схему потоків даних між диском, оперативною пам'яттю та VRAM.

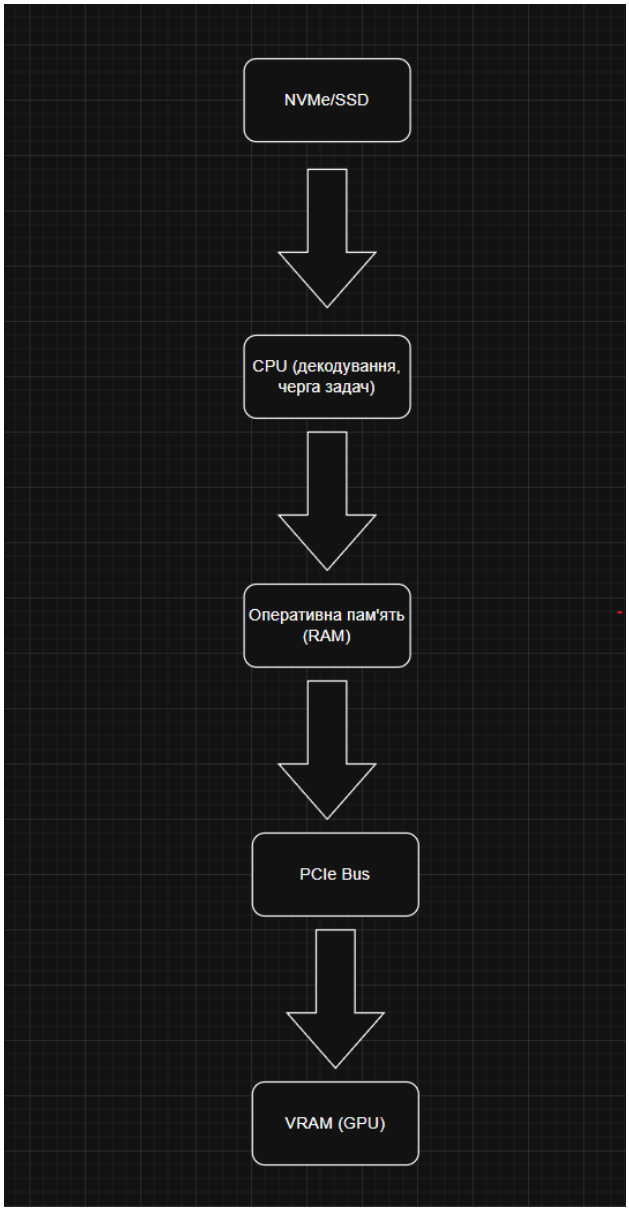

Рисунок 1.2 Схема потоків даних

Таким чином, принципи адаптивного керування відеопам’яттю ґрунтується на резидентності ресурсів, пріоритетизації, заміщенні, попередньому завантаженні та асинхронній обробці даних. У сукупності вони створюють механізм, який дозволяє ефективніше використовувати обмежену VRAM і знижувати ризик фризів у великих сценах. Для даної роботи саме ця логіка є основою практичної частини.

1.4 Інструменти та технології для реалізації системи

Вибір інструментів для реалізації системи адаптивного керування пам’яттю GPU залежить від того, наскільки низькорівневою має бути розробка. У нашому випадку доцільно використати ігровий рушій із вбудованими механізмами стрімінгу, або побудувати більш технічний прототип на основі графічного API. Наприклад, Unity дає змогу реалізувати прототип значно

швидше, а Vulkan дозволяє детальніше контролювати резидентність ресурсів, хоча й потребує суттєво більшоого обсягу коду. У документації Vulkan sparse residency прямо описується механізм роботи з частинами зображень та підресурсами на рівні sparse image bloks, що робить його природною базою для дослідження таких задач. Unity SVT, у свою чергу розбиває текстури на тайли й підтримує роботу через кілька графічних API, включаючи Vulkan [6; 4].
У таблиці доцільно порівняти такі варіанти:

Стек	Складність	Швидкість прототипування	Контроль пам'яті	Профілювання	Придат-ність
Unity + C#	Низька	Висока	Середній	Вбудоване	Висока
Unreal Engine	Середня	Середня	Середній	Вбудоване	Висока
C++ + Vulkan	Висока	Низька	Повний	Nsight	Середня
C++ + DirectX12	Висока	Низька	Повний	PIX/ Nsight	Середня

Для кожного варіанта варто вказати рівень складності, швидкість протипування, ступінь контролю пам'яті, зручність профілювання та придатність для реалізації.
Для контролю продуктивності доцільно використовувати інструменти профілювання, які дозволяють аналізувати не лише середній FPS, а й розподіл часу кадру. Це важливо, тому що середній FPS не завжди показує реальну плавність: користувач зазвичай сильніше відчуває рідкі, але великі стрибки. У практиці тестування ігрових застосунків рекомендовано записувати frame-time дані у CSV і будувати графіки розподілу кадрів, а також використовувати 1% і 0.1% low як показники "хвостів" продуктивності. Саме на це орієнтуються сучасні підходи до аналізу плавності кадрів, а не лише на середній FPS [8; 9].
Для профілювання GPU та потоків виконання придатні Nsight System і подібні системні профайлери. У документації NVIDIA зазначено, що Nsight System не потребує змін у застосунку для запуску профілювання, хоча невеликі доопрацювання можуть суттєво покращити якість даних [10]. Це робить профайлер зручним інструментом для вимірювання того, як саме працюють потоки завантаження, обробки та рендерингу.
Для побудови тестового сценарію користю використовувати заздалегідь підготовлені сцени з великою кількістю текстур і моделей. Наприклад, для

моделювання навантаження підходять міські сцени з великою кількістю будівель або природні ландшафти з великою кількістю об’єктів середовища. Такі сцени зручно використовувати як базовий набір для експериментів, оскільки вони дозволяють порівнювати поведінку системи в умовах різної щільності ресурсів.

Отже, у межах цієї роботи логічно обрати технологічний стек, який дозволяє поєднати швидке прототипування, контроль над ресурсами та можливість отримати об’єктивні показники продуктивності. Саме інструменти стрімінгу, профілювання і вимірювання frame time повинні стати основною практичної реалізації та подальших експериментів.

1.5 Висновки до розділу 1

У першому розділу було розглянуто проблему рендерингу великих сцен, основні причини перевантаження відеопам’яті та наслідки неефктивного управління графічними ресурсами. Показано, що просте статичне завантаження ресурсів не забезпечує стабільної роботи великих сцен, а отжк, потребує застосування динамічних і адаптивних підходів. Було також розглянуто основні методи оптимізації, серед яких streaming, mipmapping, LOD, virtual texturing і sparse residency, Unity SVT і Vulkan sparse resources є прикладами сучаниз механізмів, які підтверджують практичну актуальність розбивання текстур не частиини та керування їхньою резидентністю.

Проведений аналіз показав, що найбільш перспективний напрямок є саме адаптивне керування пам’яттю GPU, оскільки воно дозволяє узгодити обмеження апаратного середовища з вимогами до якості зображення. На відміну від простих методів оптимізації, адаптивний менеджер пам’яті враховує актуальність ресурсів, стан сцени, напрямок руху камери, пріоритети текстур і можливості апаратної платформи. Це створює основу для побудови гнучкої системи, яка буде ефективно працювати як на більш потужних, так і на менш продуктивних ПК.

З точки зору подальших розділів дипломної роботи цей аналіз є базою для проєктування власного механізму підвантаження та вивантаженнч ресурмва, реаліхції прототипу і проведення експериментів. Саме результати такого практичного моделювання дадуть змогу оцінити, наскільки адаптивна система покращує плавність рендерингу, зменшує кількість просідань frame time та раціоналне використання VRAM.

Розділ 2 Проєктування системи оптимізації рендерингу великих сцен через адаптивне керування пам’яттю GPU

2.1 Постановка задачі до системи

У межах даної кваліфікаційної роботи необхідно розробити прототип системи, яка забезпечує ефективне керування графічними ресурсами під час рендерингу великої сцени. Основна мета полягає в тому, щоб зменшити перевантаження відеопам’яті, знизити кількість фризів і підвищити плавність відобравження без суттєвої втрати візуальної якості. Для цього система повинна в режимі реального часу визначати, які ресурси необхідно залишити в VRAM, які слід завантажити наперед, а які можна вивантажити без негативного впливу на сприйняття сцени.

На практиці така задача виникає тоді, коли сцена містить значну кількість текстур, моделей, деталей, матеріалів та інших графічних елементів. Якщо всі ці ресурси завантажувати одночасно, навіть досить потужна система може зіткнутись з нестачею відеопам’яті. У результаті виникає додатковий обмін даними між оперативною пам’яттю та VRAM, що суттєво знижує продуктивність. Саме тому в даній роброті розглядається не статичний, а адаптивний підхід до керування пам’яттю GPU.

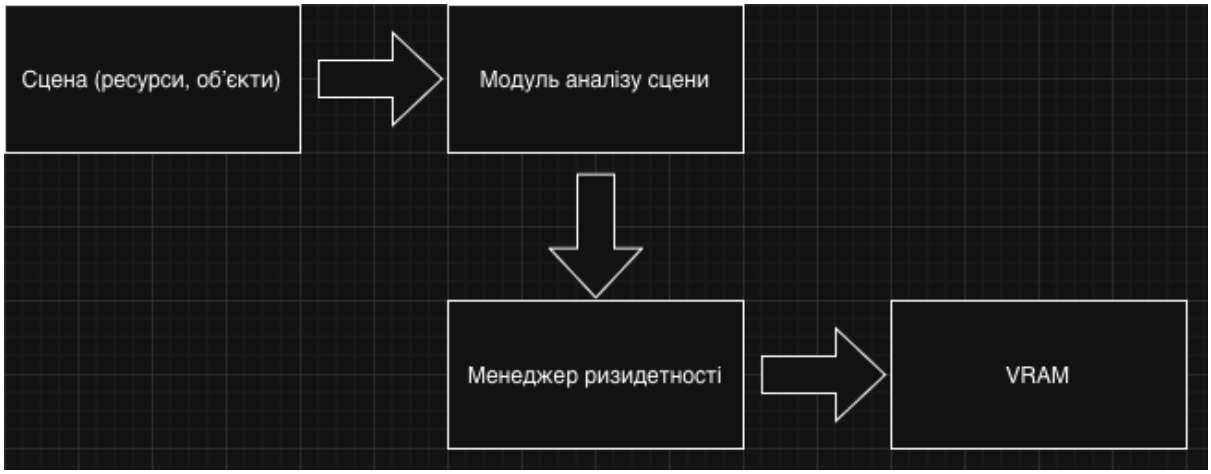

Рисунок 2.1 Загальна архітектура системи

Для побудови такої системи необхідно визначити набір вимог. По-перше, система повинна працювати асинхронно, тобто підвантаження ресурсів не повинно блокувати основний рендеринговий потік. По-друге, потрібен механізм пріоритетизації ресурсів, який ураховуватиме їхню видимість, відстань до камери, площу на екрані та ймовірність подальшого використання. По-третє, система повинна бути придатною для тестування на різних апаратних конфігураціях, оскільки поведінка менеджера пам’яті може суттєво

відрізнитися залежно від обсягу VRAM, швидкодії накопичувача й характеристик GPU.

Ще однією вимогою є масштабованість. Система має працювати не лише на одній тестовій сцені, а й на інших подібних середовищах, де структура ресурсів може відрізнятися. Саме тому в межах проєкту необхідно розробити модульну архітектуру, у якій кожен елемент виконує обмежене коло задач. Такий підхід дозволяє спростити реалізацію, полегшує тестування й дає можливість у майбутньому розширювати систему без повного переписування коду.

Таким чином, задача проєктування зводиться до створення адаптивного механізму, який на основі набору вхідних параметрів курує життєвим циклом графічних ресурсів і забезпечує стабільний рендеринг великих сцен. Виходячи з цього, далі доцільно розглянути архітектуру майбутньої системи та її основні компоненти.

2.2 Аналіз вхідних даних та критеріїв пріоритету ресурсів

Для того щоб система могла приймати обґрунтовані рішення щодо завантаження або вивантаження ресурсів, вона повинна отримувати та аналізувати набір параметрів, які характеризують як сам ресурс, так і поточний стан сцени. Ці параметри можна умовно поділити на кілька груп: геометричні, візуальні, просторові та технічні.

Група параметрів	Параметр	Опис	Вплив на пріоритет
Геометричні	Розмір ресурсу (байт)	Обсяг пам’яті займає текстура або модель	Чим більший розмір, тим нижчий пріоритет при нестачі VRAM
Геометричні	Тип об’єкта	Текстура, меш, матеріал, мі-рівень	Визначає стратегію завантаження
Геометричні	Кількість полігонів	Кількість трикутників у моделі	Чим більше полігонів тим вища вартість завантаження
Візуальні	Видимість у фрустумі	Чи потрапляє об’єкт у поле зору камери	Якщо об’єкт невидимий, то пріоритет 0
Візуальні	Площа на екрані (ScreenCovarage)	Частина екрану, яку займає об’єкт	Чим більша площа, тим вищий пріоритет

Візуальні	Відстань до камери	Відстань у світових одиницях	Чим більша відстань, тим нижчий пріоритет
Просторові	Напрямок руху камери	Вектор переміщення камери	Використовується для prefetching
Просторові	Швидкість камери	Метрів на секунду	Чим вища швидкість, тим ширша зона prefetch
Технічні	Вільна VRAM	Доступний обсяг відеопам’яті (МБ)	Чим менше пам’яті, тим активніше вивантаження
Технічні	Навантаження GPU (%)	Поточне завантаження графічного процесора	Визначає агресивність eviction

Таблиця 2.1 вхідні дані для оцінювання пріоритету ресурсу

До геометричних параметрів належать розмір ресурсу, тип об’єкта, кількість полігонів, кількість підресурсів і тип текстури. Наприклад, велика текстура високої роздільності потребує більше місця в пам’яті, ніж дрібний елемент декору, тому її завантаження повинно бути обґрунтованим. Водночас сам по собі великий розмір не завжди означає високий пріоритет: якщо об’єкт не видно в кадрі, система може тимчасово відкласти його завантаження.

До візуальних параметрів належать видимість об’єкта, розмір його проєкції на екрані, відстань до камери, а також ступінь перекриття іншими об’єктами. Ці значення дозволяють визначити, наскільки ресурс важливий для поточного кадру. Наприклад, об’єкт, що займає значну частину екрана, має вищий пріоритет, ніж той, який знаходиться на віддаленій периферії сцени. Саме тому в реальній системі пріоритет часто обчислюється як комбінація кількох ознак, а не одним критерієм.

До просторових параметрів належать положення камери, напрямок її руху, швидкість переміщення та ймовірної видимості в наступних кадрах. Саме ці дані використовуються для механізму prefetching, коли система намагається завантажити ресурси ще до того, як вони стануть потрібними. Такий підхід допомагає забезпечити плавність та уникати затримок, але

потребує акуратного налаштування, щоб не перевантажити пам’ять зайвими даними.

До технічних параметрів належать поточний обсяг вільної VRAM, швидкість обміну з накопичувачем та поточний рівень навантаження на GPU. Ці показники визначають, наскільки агресивно система може діяти в конкретний момент. Наприклад, якщо VRAM майже заповнена, алгоритм повинен активніше вивантажувати низькопріоритетні ресурси. Якщо ж вільної пам’яті достатньо, можна дозволити ширший буфер попереднього завантаження.

Для більш зручного аналізу всі ресурси доцільно зберігати у вигляді структурного запису, який містить ідентифікатор, тип, розмір, час останнього використання, пріоритет та стан резидентності. Це дозволяє швидко виконувати пошук і приймати рішення в межах одного циклу рендерингу.

Поле	Призначення
ResourceID	Унікальний ідентифікатор ресурсу
ResourceType	Тип ресурсу(текстура, модель, міп-рівень, тайл)
Size	Обсяг пам’яті, який займає ресурс
LastAccessTime	Час(секунди від старту) останього звернення
Priority	Поточний рівень важливості
ResidentState	Ознака, чи знаходиться ресурс у VRAM
LoadState	Стан заванатаження: очікує, завантажується, готовий
ScreenCovarage	Площа, яку ресурс займає на екрані

Таблиця 2.2 Пиклад структури запису про ресурс

Для практичної реалізації доцільно використовувати числову інтегральну оцінку пріоритету. Вона може обчислуватися як зважена сума кількох нормалізованих показників. Наприклад, більшу вагу можна надати видимості та площі на екрані, а меншу – відстані та прогнозу руху. Важливо щоб формула була простою для обчислення в реальному часі, оскільки менеджер пам’яті повинен працювати дуже швидко.

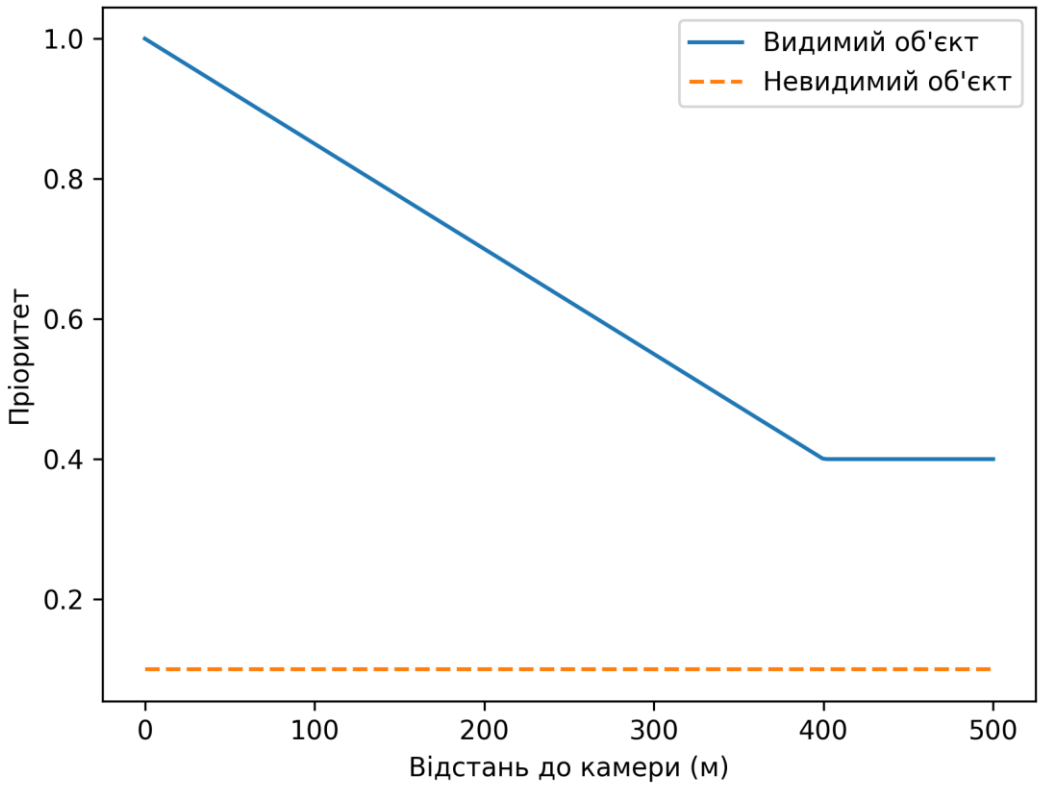

Рисунок 2.2 Приклад залежності пріоритету від видимості та відстані

Отже, під час побудови системи необхідно враховувати не один параметр, а сукупність ознак, які дозволяють визначити реальну цінність ресурсу для поточного кадру. Саме це і становить основу адаптивного керування пам’яттю GPU.

2.3 Загальна архітектура адаптивного менеджера пам’яті GPU

Запропонована система будується за модульним принципом і складається з кількох взаємопов’язаних компонентів. Така структура дозволяє розділити логіку контролю пам’яті, аналізу сцени, завантаження ресурсів і профілювання результатів. Основною частиною системи є менеджер резидентності, який ухвалює рішкння щодо стану кожного ресурсу в режимі реального часу.

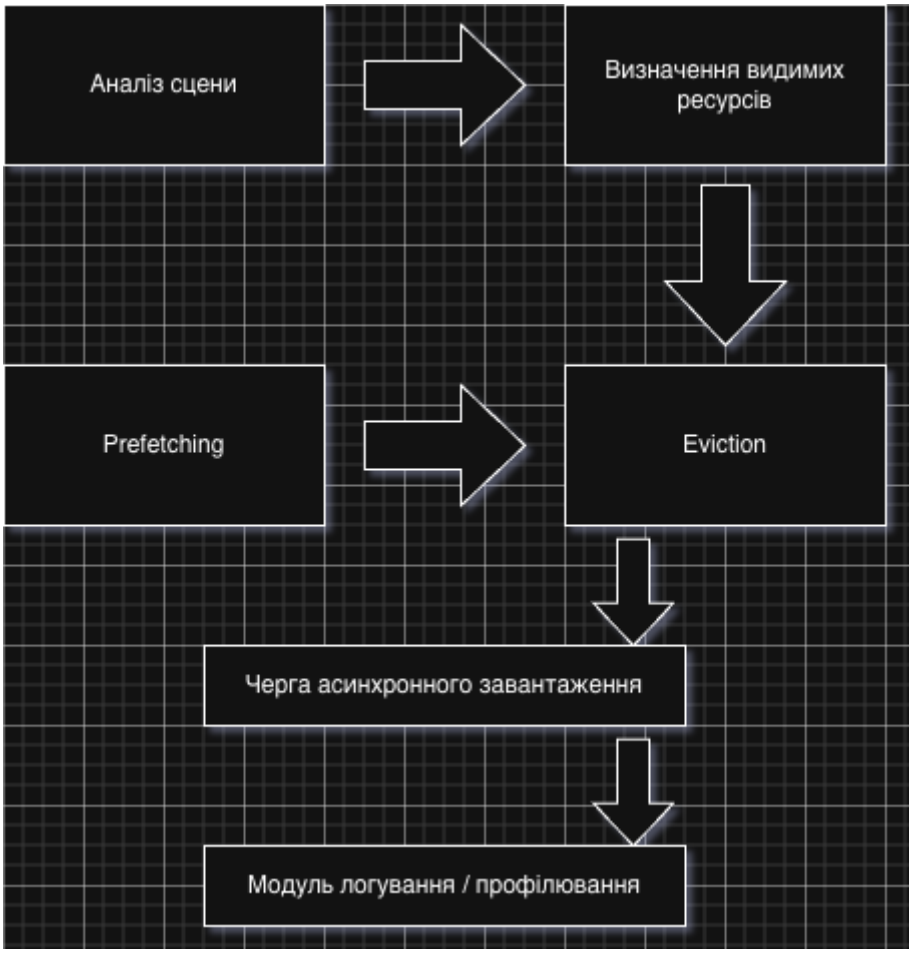

Рисунок 2.3 Детальна архітекртура модулів системи

Модуль аналізу сцени відповідає за збір інформації про об’єкти, що знаходяться в полі зору камери. Він визначає, які ресурси активно використовуються, які з них потраплять до фрустуму, а які можуть стати потібними в найближчі кадри. На основі цих даних формується список кандидатів на завантаження або вивантаження.

Менеджер резидентності є центральним вузлом усієї системи. Саме він приймає рішення про те, які ресурси повинні бути розміщенні у VRAM, а які можуть бути тимчасово виключені з активного набору. Для цього він використовує дані від інших модулів і порівнює їх із поточним обсягом доступної пам’яті. Якщо запас VRAM достатній, система може залишити більше ресурсів у резидентному стані. Якщо пам’яті не вистачає, виконується вивантаження ресурсів із найнижчим пріоритетом.

Черга асинхронного завантаження забезпечує незалежну обробку запитів на під вантаження. Це особливо важливо для великих сцен, оскільки синхронне завантаження може зупиняти рендиренговий цикл і призводити до візуальних артефактів.

Модуль prefeching аналізує напрямок руху камери і прогнозує, які ресурси можуть знадобитися найближчим часом. У найпростішому випадку він завантажє сусідні тайли текстур або ресурси з області, куди рухається камера. У більш складному варіанті він враховує історію переміщення користувача та інтенсивність взаємодії зі сценою.
Модуль логування і профілювання фіксує основні показники роботи системи: кількість під вантаження тайлів, число вивантажених ресурсів, поточне використання VRAM, середній час обробки запиту та кількість фризів. Ці дані будуть використані в екстримальній частині роботи.

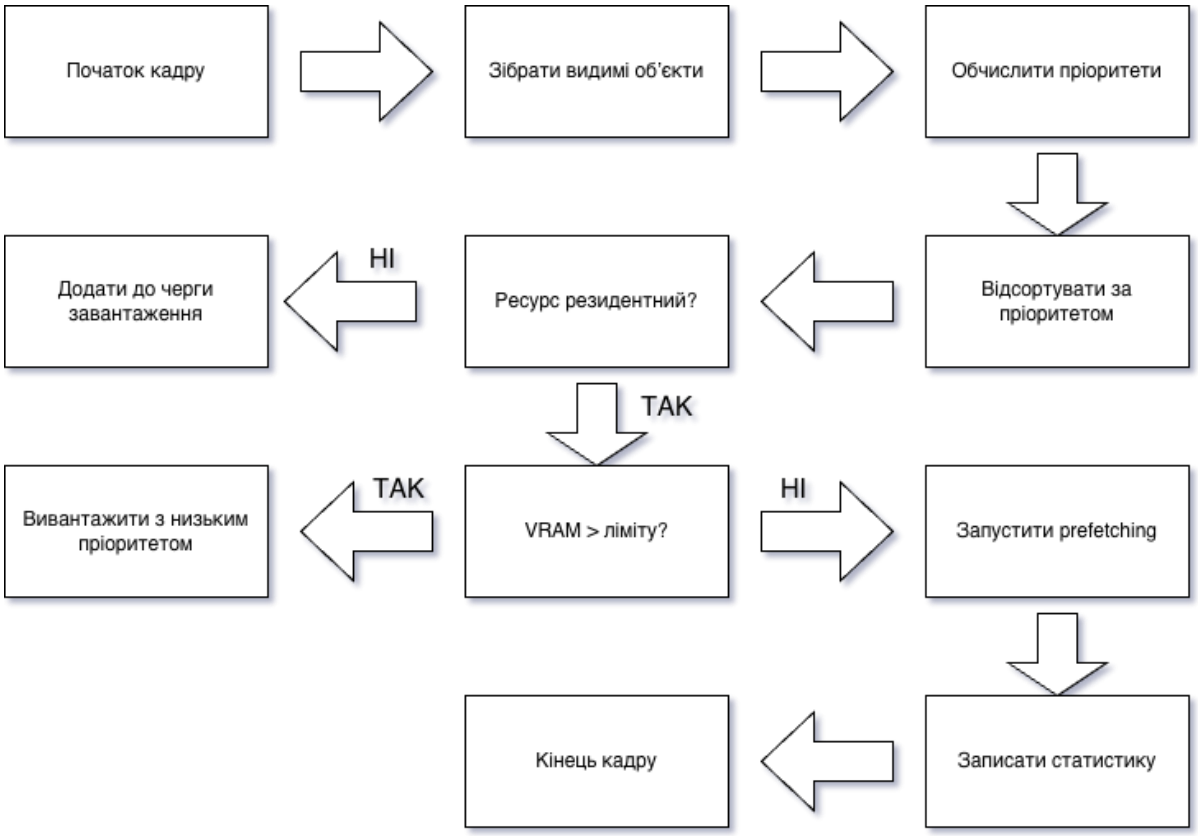

Рисунок 2.4 Логіка роботи адаптивного менеджера

Таким чином, архітектура системи повинна забезпечити одночасно гнучкість, швидкодію та зручність подальшого тестування. Саме модульна структура є найбільш придатною оскільки вона дозволяє реалізувати працездатний прототип без надмірної складності.

2.4 Логіка рототи менеджера резидентності

Менеджер резидентності є ключовим програмним компонентом, який контролює життєвий цикл графічних ресурсів. Його основне завдання полягає в тому, щоб у кожний момент часу підтримувати актуальний набір даних у

VRAM і не допустити перевищення допусттимого обсягу пам’яті. При цьому він повинен ухвалювати рішення швидко, оскільки працює в межах одного або кількох кадрів.
Логіка робрти менеджера може бути подана у вигляді наступної послідовності дій. Спочатку система збирає інформацію про видимі об’єкти сцени. Потім кожному ресурсу призначається оцінка пріоритету. Далі виконується порівняння поточного використання VRAM із доступним лімітом. Якщо пам’яті достатньо, ресурс переводиться в резидентний стан або залишається в ньому. Якщо ні, система формує список кандидатів на вивантаження та звільняє пам’ять для нових даних.

У вигляді псевдокоду основна функція має приблизно такий вигляд:

```
HandleFrame():
   visibleResources ← DetectVisibleResources()
   for each resource in visibleResources:
      resource.priority ← CalculatePriority(resource)

   SortResourcesByPriority()

   for each resource in visibleResources:
      if resource is not resident:
         RequestLoad(resource)

   if VRAM usage > limit:
      EvictLowPriorityResources()

   PrefetchNextResources()
   LogStatistics()
```

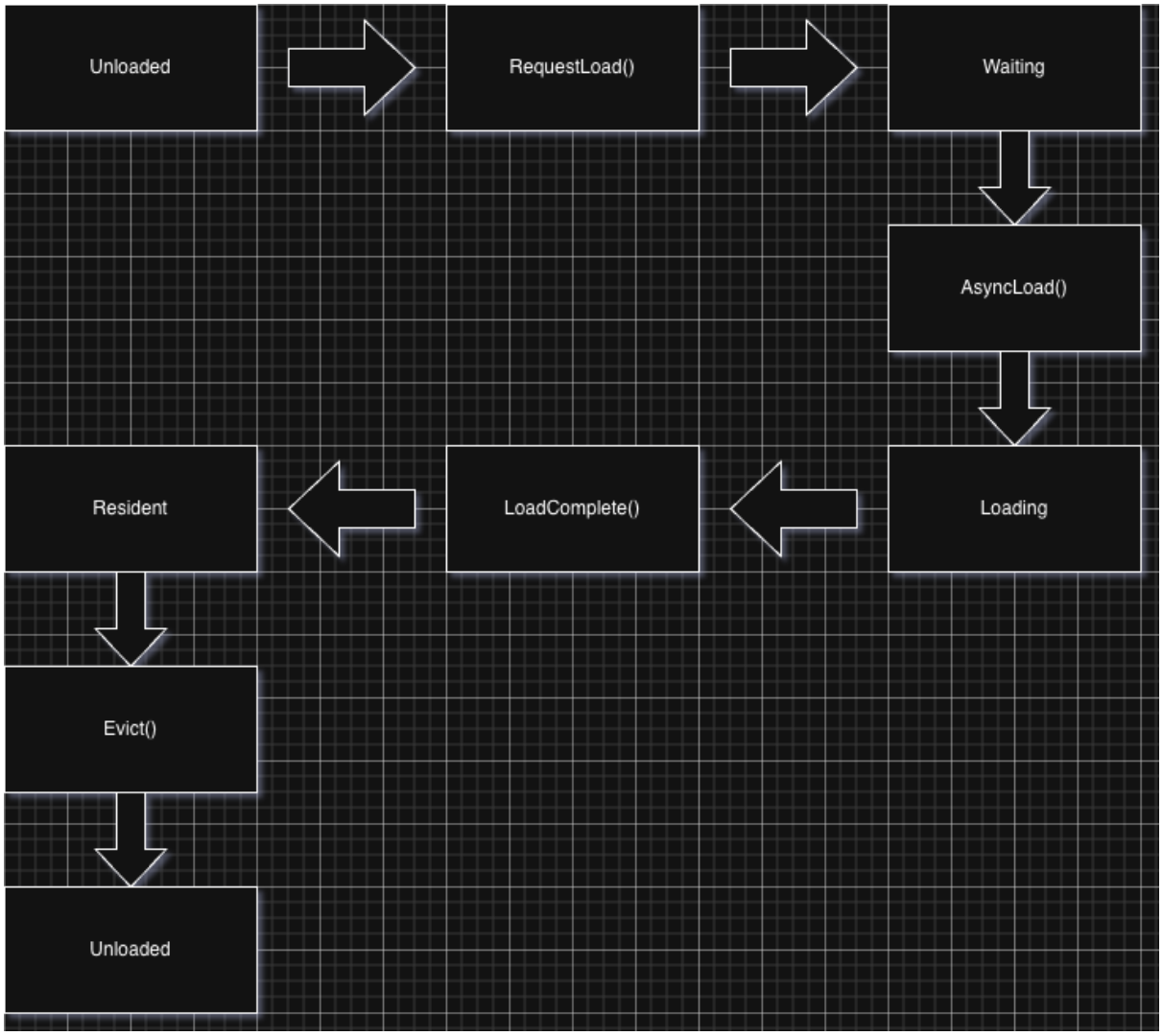

Рисунок 2.5 Схема роботи менеджера резидентності (стани ресурсу)

У цьому алгоритмі важливим є, те що приймаються один раз на весь рівень. Вони оновлюються для кожного кадру або через певний проміжок часу. Саме така динаміка і робить систему адапттивною. Крім того, окремо слід передбачити буфер ресурсів, які вже завантажуються, але ще не готові до використання. Це дозволить уникнути дублювання запитів і зменшити навантаження на систему введення-виведення.

Для вивантаження ресурсів доцільно використовувати політику, яка поєднує LRU-підхід із просторовим аналізом. Інакше кажучи, найменш недавно використані ресурси можуть вивантажуватися першими, але лише за умови, що вони не мають високого пріоритету або не потрапляють до прогнозованої області видимості. Такий компроміс дозволяє уникати ситуації, коли ресурс знову потрібен одразу після вивантаження.

Політика	Переваги	Недоліка

FIFO	Простота реалізації	Не враховує реальну важливість ресурсу
LRU	Добре працює для багатьох сценаріїв	Не враховує майбутню видимість
Priority-based	Враховує цінність ресурсу	Потребує точного налаштування ваг
Hybrid LRU + priority	Баланс між простотою та якістю	Складніша реалізація

Таблиця 2.3 Порівняння можливих політик eviction

Отже, менеджер резидентності є не просто механізмом кешування, а центральним елементом усієї системи оптимізації. Саме від нього залежить, наскільки стабільно працюватиме рендеринг великої сцени.

2.6 Висновки до розділу 2

У другому розіділі було сформульовано задачу проєктування системи оптимізації рендерингу великих сцен через адаптивне керування відеопам’яттю GPU. Визначено основні вимоги до системи, наведено перелік вхідних даних, які впливають на рішення про завантаження або вивантаження ресурсів, та описано загальну архітектуру майбутнього прототипу.

Показано, що основою системи є менеджер резидентності, який працює в режимі реального часу та оперує такими параметрами, як видимість ресурсу, відстань до камери, площа на екрані, стан VRAM і поточний рівень навантаження. Окремо було розглянуто логіку асинхронного завантаження, механізм prefetching та політики вивантаження ресурсів. Усе це створює підґрунття для практичної реалізації, яку буде описано в наступному розділі.

Таким чином, у межах проєктування вдалося визначити структуру системи, розподіл функцій між модулями та загальні принципи прийняття рішень. Це дозволяє перейти до наступного етапу – програмної реалізації прототипу і проведення експериментів.

Розділ 3. Експериментальне дослідження та аналіз результатів

3.1 Опис тестового середовища

Результати експериментального дослідження безпосередньо залежать від конфігурації обладнання та програмного забезпечення, на яких проводяться вимірювання. Через це у першу чергу необхідно зафіксувати характеристики тестового стенда, що дозволить як інтерпретувати отримані числа в межах цієї роботи, так і відтворити дослідження в інших умовах. Тестовий стенд був типовим робочим комп'ютером середнього класу, що відповідає реальним умовам, у яких можуть працювати ігрові або візуалізаційні застосунки. Повний перелік апаратних параметрів наведено у таблиці 3.1.

Компонент	**Значення**
Операційна система	Windows 11 Pro 25H2, build 26200.8457
Оперативна пам'ять	32 ГБ DDR4
Відеокарта	NVIDIA GeForce RTX 4060 Ti
Обсяг відеопам'яті	16 ГБ GDDR6
Версія драйвера GPU	32.0.15.9144 (від 02.12.2025)
Накопичувач проєкту	Seagate ST2000DM008, HDD 2 ТБ, 7200 об/хв

Таблиця 3.1 Апаратна конфігурація тестового стенда

З точки зору цієї роботи важливо виділити дві характеристики цього стенда. Перша – обсяг відеопам'яті 16 ГБ, який відповідає сучасному середньому класу і дозволяє розмістити доволі великий обсяг ресурсів, але водночас залишає простір для перевантаження у режимі без оптимізації. Це робить порівняння адаптивної та статичної стратегій змістовним: різниця між ними проявляється у фактичних показниках, а не лише теоретично. Друга характеристика – накопичувач у вигляді HDD, а не SSD. Це створює суттєво більші затримки при асинхронному завантаженні ресурсів, ніж було б на твердотільному диску, і дозволяє більш яскраво побачити роботу streaming-механізмів у несприятливих для них умовах. Якщо адаптивна система демонструє стабільну роботу на HDD, то на SSD її поведінка буде ще плавнішою.

Програмне середовище проєкту обрано таким чином, щоб максимально відповідати сучасним стандартам розробки ігрових застосунків з фотореалістичним рендерингом. Усі ключові параметри наведено у таблиці 3.2.

Параметр	**Значення**
Версія Unity	6000.3.13f1 (Unity 6.3 LTS)

Render Pipeline	High Definition RP (HDRP)
Версія HDRP	17.3.0
Scripting Backend	Mono
API Compatibility Level	.NET Standard 2.1
Quality Level	Ultra

Таблиця 3.2 Параметри Unity та HDRP

Версія Unity 6.3 LTS обрана як остання довгопідтримувана редакція, у якій уже стабілізовані всі нові механізми HDRP, включно з GPU Resident Drawer та оновленим Virtual Texturing. Сам рендер-пайплайн HDRP взято тому, що він орієнтований саме на сцени з високою деталізацією та значним обсягом ресурсів, що цілком відповідає темі роботи [11]. Quality Level встановлено у режим Ultra, оскільки тільки на високих налаштуваннях якості ефект від адаптивного керування пам'яттю стає кількісно відчутним: на низьких налаштуваннях усі текстури автоматично стискаються та зменшуються у роздільності, і різниця між режимами фактично нівелюється.

Scripting Backend Mono та API Compatibility Level .NET Standard 2.1 обрано за замовчуванням для прискорення розробки. У продуктивній збірці використовується IL2CPP, але оскільки експеримент виконується безпосередньо в Editor Play Mode, рівень оптимізації коду не є визначальним фактором, а Mono забезпечує швидші ітерації під час розробки.

Таким чином, тестовий стенд відображає реалістичну робочу станцію розробника або користувача, який не використовує найдорожче обладнання, але має достатні ресурси для роботи з HDRP-сценами. Подальші підрозділи містять описи самої сцени, методики експерименту та отриманих числових результатів.

3.2 Архітектура експериментальної сцени

Для проведення вимірювань необхідна сцена, яка одночасно є достатньо великою, щоб виявити обмеження статичного підходу, і досить контрольованою, щоб гарантувати відтворюваність результатів. Виходячи з цього, для роботи було обрано не природний ландшафт чи готовий ассет-пак, а синтетичну сцену, що повністю генерується у коді при запуску. Такий підхід дозволяє у будь-який момент змінити кількість об'єктів, їхнє розташування або набір текстур, не порушуючи інших частин системи.

Створення сцени виконує компонент SceneGenerator, який під час події Awake генерує 500 простих об'єктів типу Cube і розкладає їх рівномірно у прямокутній зоні розмірами 350 на 500 метрів. Для рівномірності використано не випадкове, а сіткове розміщення з невеликим випадковим зміщенням

всередині кожної комірки. Завдяки цьому уникнуто характерної проблеми чисто випадкової генерації – кластеризації, коли частина об’єктів накопичується в одному кутку, а інша частина зони залишається порожньою. Загальні параметри сцени наведено у таблиці 3.3.

Параметр	**Значення**
Кількість об’єктів	500
Ширина зони (вісь X)	350 м
Глибина зони (вісь Z)	500 м
Мінімальний масштаб	1.5
Максимальний масштаб	4.0
Тип об’єктів	Cube (стандартний primitive Unity)
Кількість унікальних матеріалів	5
Облікова пам’ять на об’єкт (SizeMB)	16 МБ

Таблиця 3.3 Параметри тестової сцени

Кожен згенерований об’єкт автоматично отримує два додаткові компоненти: MaterialLoader, який відповідає за асинхронне завантаження текстур, та ResourceRegistrar, який реєструє об’єкт у менеджері резидентності. Тип матеріалу обирається випадково з п’яти наперед підготовлених наборів. Кожен набір складається з трьох текстур у форматі PNG роздільності 2048×2048: дифузна (Albedo), нормальна (NormalGL у OpenGL-форматі) та маска шорсткості (Roughness), що відповідає сучасній PBR-моделі. Конкретно у роботі використано матеріали Bricks045, Concrete034, Metal032, Wood049 та Ground037 – цеглу, бетон, метал, дерево та землю. Таке різноманіття дозволяє побачити поведінку системи на ресурсах різного типу, а сама кількість унікальних текстур (всього 15) робить навантаження на VRAM прогнозованим.

Для зйомки тестового сценарію використано віртуальну камеру, яка рухається сценою за наперед заданою траєкторією. Цей рух реалізовано через компонент CameraPath і набір з восьми контрольних точок – так званих waypoints. Камера послідовно переміщується від одного waypoint до наступного зі швидкістю 35 одиниць за секунду, плавно повертаючись у бік цільової точки. Точні координати усіх waypoints наведено у таблиці 3.4.

Точка	**X**	**Y**	**Z**
WP_00	0	60	-50
WP_01	140	25	80
WP_02	140	10	250
WP_03	-140	10	400
WP_04	-140	40	200

WP_05	0	70	100
WP_06	140	20	480
WP_07	0	80	250

Таблиця 3.4 Координати waypoints камери

Траєкторію було спроєктовано таким чином, щоб камера послідовно охопила різні зони сцени: спочатку огляд згори (WP_00 і WP_05, WP_07 на висоті 60–80 м), потім близькі прольоти крізь скупчення об’єктів (WP_02 і WP_03 на висоті лише 10 м), а далі повернення вгору. Такий патерн руху максимально активізує адаптивну логіку системи: при низьких прольотах у фрустум камери одночасно потрапляє багато об’єктів, а при огляді згори частина з них залишає кадр і має бути вивантажена. Загальна тривалість маршруту обмежена параметром TestDuration у 90 секунд, але на практиці камера проходить усі точки і зупиняється раніше – орієнтовно за 50 секунд. Загальний вигляд сцени з виду зверху наведено на рисунку 3.1.

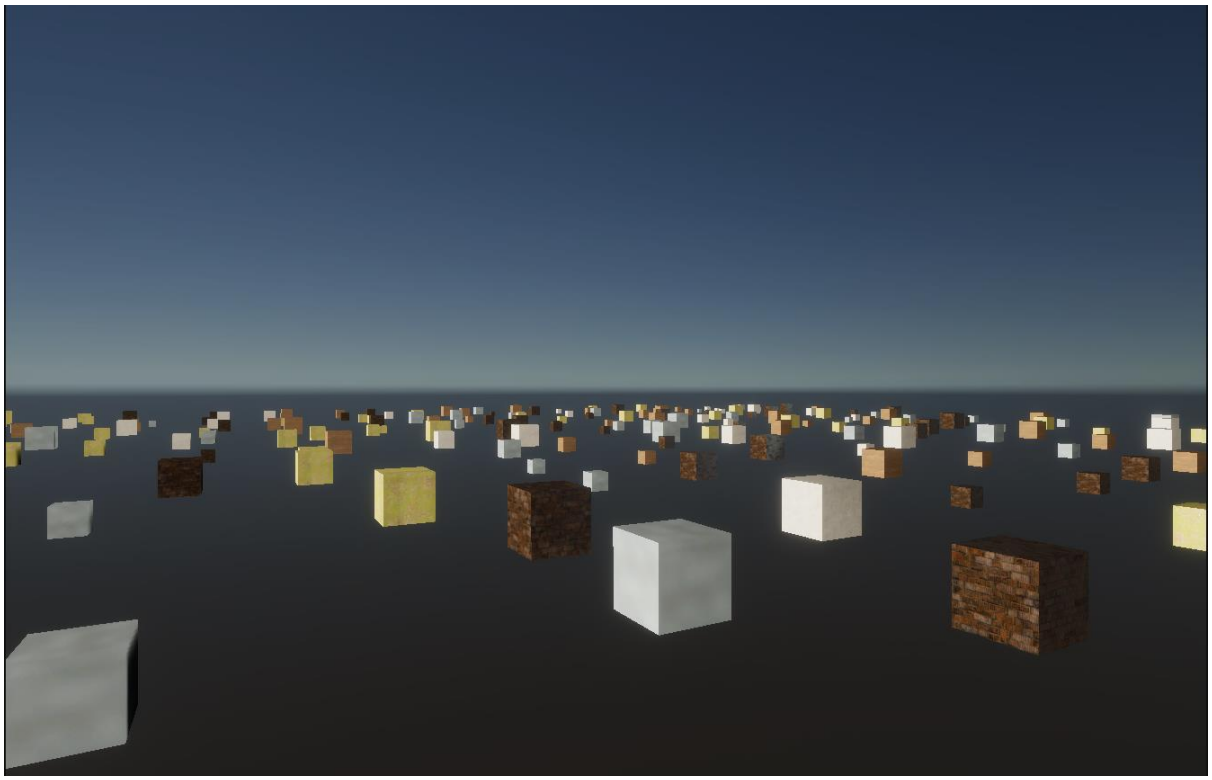

Рисунок 3.1 Загальний вигляд згенерованої тестової сцени

Таким чином, тестова сцена побудована за принципом мінімальної достатності: 500 простих об’єктів з п’ятьма матеріалами, розкиданих у прямокутній зоні з відомими розмірами, і камера з фіксованою траєкторією. Завдяки цьому при кожному запуску система отримує однакові умови, що є необхідною передумовою для об’єктивного порівняння режимів роботи у наступних підрозділах.

3.3 Методика проведення експериментів

Для об’єктивної оцінки роботи розробленої системи необхідно сформувати методику, яка дозволить отримувати порівнянні та повторювальні результати. У межах даної роботи експеримент організовано як прямий порівняльний тест двох режимів роботи системи на одній і тій же сцені, з однаковою траєкторією камери та однаковими початковими умовами. Такий підхід дає можливість відокремити вплив самого механізму адаптивного керування пам’яттю від інших чинників, пов’язаних із роботою рушія, рендер-пайплайну чи особливостями апаратного забезпечення.

Першим режимом є ADAPTIVE - повноцінна робота системи, де менеджер резидентності керує життєвим циклом ресурсів у реальному часі. У цьому режимі активним є визначення видимих ресурсів через фруструм камери, асинхронне завантаження матеріалів, оновлення часу останнього доступу обчислення пріоритету ресурсів та LRU - вивантаження тих які не використовувалися протягом заданого часу. Другим режимом є BASELINE - контрольний варіант, у якому усі ресурси сцени завантажуються одразу на старті за допомогою компонента StaticPreloader, а менеджер резидентності залишається вимкненим. Перемикання між цими двома режимами реалізовано у скрипті BaselineController. Натискання клавіші Tab у методі Update перемикає прапорець прапорець AdaptiveModeEnable і викликає один з двох допоміжних методів - SetAdaptiveModeEnabled або SetBaselineMode. Кожен із них активує або деактивує менеджер резидентності, налаштовує StaticPreloader на відповідну поведінку (вивантаження всього набору ресурсів при переході в ADAPTIVE або повне статичне завантаження при переході в BASELINE) та оновлює інформаційний напис екрані. Відповідну реалізацію наведено на рисунку 3.2.

```
0 references
void Update()
{
    if (Input.GetKeyDown(KeyCode.Tab))
    {
        if (AdaptiveModeEnabled)
            SetBaselineMode();
        else
            SetAdaptiveMode();
    }
}

2 references
private void SetAdaptiveMode()
{
    AdaptiveModeEnabled = true;
    Manager.enabled = true;

    if (Preloader != null)
    {
        Preloader.IsBaseline = false;
        Preloader.UnloadAll();
    }

    if (Logger != null)
        Logger.LoggingEnabled = true;

    UpdateModeText();
    UnityEngine.Debug.Log("[BaselineController] ADAPTIVE mode");
}

1 reference
private void SetBaselineMode()
{
    AdaptiveModeEnabled = false;
    Manager.enabled = false;

    if (Preloader != null)
    {
        Preloader.IsBaseline = true;
        StartCoroutine(Preloader.PreloadAll());
    }

    if (Logger != null)
        Logger.LoggingEnabled = true;

    UpdateModeText();
    UnityEngine.Debug.Log("[BaselineController] BASELINE mode");
}
```

Рисунок 3.2 Лістинг коду BaselineController: перемикання режимів

Конкретні значення параметрів менеджера резидентності, з якими проводилися експерименти, наведено у таблиці 3.5. Ці значення підбиралися експериментально таким чином, щоб ліміт VRAM був достатньо тісним для активації механізму вивантаження, але не настільки малим, щоб система постійно перебувала у стані перевантаження.

Параметр	**Значення**	**Призначення**
VramLimitBytes	1.5 ГБ	Максимальний обсяг VRAM під курування
UpdateEveryFrames	5	Періодичність виклику HandleFrame
EvictAfterSeconds	4	Час бездіяльності перед вивантаженням
MaxConcurrentLoads	10	Максимум одночасних завантажень

Таблиця 3.5 Параметри конфігурації ResidencyManager

Для збереження умов відтворюваності експерименту в коді SceneGenerator застосовано фіксований seed випадкового генератора. Це означає, що під час кожного запуску генерується одна й та сама конфігурація об’єктів: їхні позиції, масштаби, набори текстур та орієнтація залишається незмінними. Камера у обох режимах рухається по однаковій траєкторії з тими ж самими waypoints, що дозволяє говорити про порівняння саме поведінки системи, а не випадкових змін у сцені. Ключову частину коду генератора показано на рисунку 3.3.

```
1 reference
private void GenerateScene()
{
    // Use grid to spread objects evenly (no clustering)
    int cols = Mathf.CeilToInt(Mathf.Sqrt(
        ObjectCount * (AreaWidth / AreaDepth)));
    int rows = Mathf.CeilToInt((float)ObjectCount / cols);

    float cellW = AreaWidth / cols;
    float cellD = AreaDepth / rows;

    int idx = 0;
    for (int r = 0; r < rows && idx < ObjectCount; r++)
    {
        for (int c = 0; c < cols && idx < ObjectCount; c++)
        {
            // Position with jitter inside cell
            float baseX = -AreaWidth / 2f + c * cellW + cellW * 0.5f;
            float baseZ = StartZ + r * cellD + cellD * 0.5f;

            float jitterX = UnityEngine.Random.Range(
                -cellW * 0.3f, cellW * 0.3f);
            float jitterZ = UnityEngine.Random.Range(
                -cellD * 0.3f, cellD * 0.3f);

            float scale = UnityEngine.Random.Range(MinScale, MaxScale);

            var go = GameObject.CreatePrimitive(PrimitiveType.Cube);
            go.name = $"SceneObject_{idx:D3}";
            go.transform.position = new Vector3(
                baseX + jitterX,
                scale * 0.5f,
                baseZ + jitterZ);
            go.transform.localScale = Vector3.one * scale;

            int t = UnityEngine.Random.Range(0, AlbedoPaths.Length);

            var loader = go.AddComponent<MaterialLoader>();
            loader.AlbedoPath = AlbedoPaths[t];
            loader.NormalPath = NormalPaths[t];
            loader.RoughnessPath = RoughnessPaths[t];

            var reg = go.AddComponent<ResourceRegistrar>();
            reg.Manager = Manager;
            reg.Loader = loader;
            reg.SizeMB = 16;

            idx++;
        }
    }

    UnityEngine.Debug.Log(
        $"[SceneGenerator] Generated {idx} objects in grid " +
        $"{cols}x{rows} over {AreaWidth}x{AreaDepth}m");
}
```

Рисунок 3.3 Лістинг коду SceneGenerator: генерація об’єктів з фіксованим seed

Тривалість одного прогону становить 60 секунд від моменту запуску камери; цього часу достатньо, щоб камера встигла пройти всі вісім контрольних точок і повернулися у вихідну зону. Перед кожним новим запуском стан системи скидається, щоб уникнути впливу попереднього запуску на результат.

Збір даних відбувається покадрово через через компонент FrameLogger записує у CSV-файл шість основних метрик. Реалізацію основного циклу логування показано на рисунку 3.4.

```
void Update()
{
    if (!LoggingEnabled) return;

    float ms = Time.deltaTime * 1000f;
    float trackedMb = Manager.GetUsedVramMB();
    float actualMb = VramMonitor != null
        ? VramMonitor.AllocatedVramMB
        : 0f;
    int resident = Manager.GetResidentCount();
    int pending = Manager.GetPendingCount();
    string mode = Manager.enabled ? "ADAPTIVE" : "BASELINE";

    _buffer.AppendLine(string.Format(CultureInfo.InvariantCulture,
        "{0},{1:F2},{2:F2},{3:F2},{4},{5},{6}",
        _frameCount++, ms, trackedMb, actualMb,
        resident, pending, mode));

    if (_frameCount % FlushEveryFrames == 0)
        Flush();
}
```

Рисунок 3.4 Лістинг коду FrameLogger: збір та запис метрик

Перелік метрик, які логуються під час експерименту, наведено у таблиці 3.6.

Метрика	**Одиниці**	**Опис**
FrameTime	мс	Тривалість поточного кадру, обчислення через Time.deltaTime
TrackedVRAMMB	МБ	Внутрішня оцінка системи: сума розмірів резидентних ресурсів
ActualVRAMMB	МБ	Реальне виділення GPU-пам’яті через Profiler.GetAllocatedMemoryForGraphicsDriver
ResidentCount	шт.	Поточна кількість ресурсів у черзі на завантаження
PendingCount	шт.	Кількість ресурсів у черзі на завантаження
Mode	-	Поточний режим роботи(ADAPTIVE або BASELINE)

Таблиця 3.6 Метрики, що логуються під час експерименту

Особливу увагу слід приділяти різниці між TrackedVRAMMB та ActualVRAMMB. Перше значення відображає лише ту пам’ять, яку контролює

сам менеджер резидентності, тобто суму розмірів зареєстрованих та активних на даний момент ресурсів. Друге значення показує фактичний обсяг пам’ятті, який виділив графічний драйвер. У цей обсяг входять також внутрішні буфери HDRP, тіні, post-procrssing, мапи глибини, render targets та інші службові ресурси, які створюються рушієм незалежно від нашої системи. Таке розділення дозволяє побачити, наскільки точно наш менеджер відображає реальний стан VRAM, а також оцінити загальний вплив системи на сумарне споживання пам’яті.

Для забезпечення коректного запису даних у файл, FrameLogger використовується кожні 60 кадрів. Це знижує вплив операцій вводу/виводу на час кадру і робить вимірювання чистішими. Окремо було передбачено, що десяткові розділювачі у файлі мають відповідати міжнародному стандарту (крапка, а не кому), оскільки робота виконувалася у середовищі з українською локаллю, де за замовчування кома. Без цього виправлення CSV-файл був би неможливий для коректного аналізу у Python чи Excel.

Окрім кількісних метрик, у процесі тестування використовується візуальний контроль через DebugOverlay. Цей компонент відображає на екрані поточні показники FPS, кількість резидентних та pending-ресурсів, обсяг використаної VRAM та назву активного режиму, як показано на рисунку 3.5. Це дозволяє у реальному часі переконатися, що система працює очікувано і що перемикання режимів дійсно змінює поведінку менеджера. Зокрема, у режимі ADATIVE значення ResidentCount має змінюватися під час руху камери, тоді як у режимі BASELINE воно повинно залишатися постійним після завершення початкового завантаження.

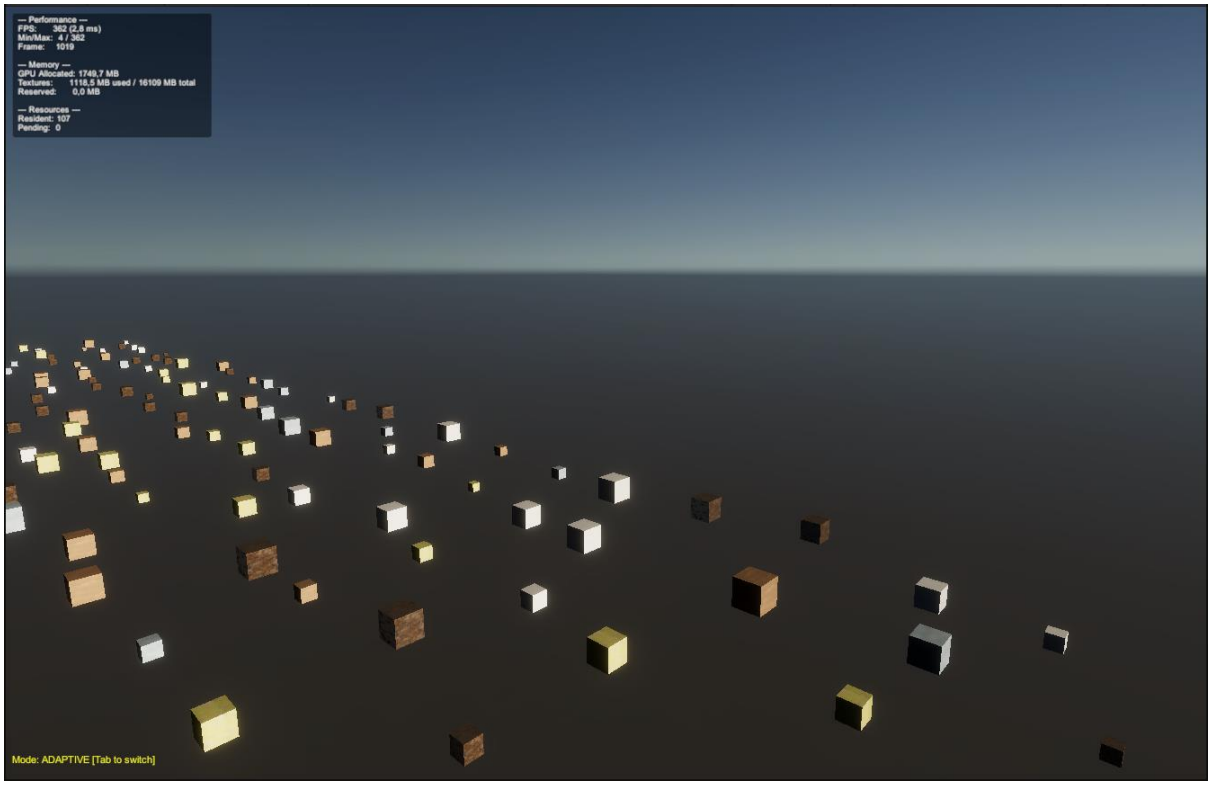

Рисунок 3.5 Інтерфейс системи з Debug-панеллю під час виконання тесту

Після завершення кожного 90-секундного прогону CSV-файл копіюється у окрему директорію з відповідною назвою (results_adptive.csv та results_baseline.csv) для подальшого аналізу. Обробка даних виконується мовою Python з використанням бібліотек pandas, numpy та matplotlib [12; 13; 14]. На етапі обробки відкидається перші 120 кадрів кожного запису, які відповідають фазі warm-up, тобто моменту, коли система ще не встигла повністю сформувати свій активний набір ресурсів. Для решти кадрів обчислюються середнє значення, медіана, 95-й та 99-й перцентилі, мінімум і максимум для кожної з метрик. На основі цих значень будуються графіки часових залежностей, гістограми розподілів та порівняльні діаграми, які наведено у наступних підрозділах.

Таким чином, розроблена методика забезпечує контрольоване середовище для порівняння двох режимів роботи системи. Завдяки фіксованим параметрам сцени, однаковій траєкторії камери, ідентичній конфігурації ресурсів та одночасному логуванню кількох метрик можна об’єктивно оцінити, наскільки адаптивне керування пам’яттю впливає на споживання VRAM і продуктивність рендерингу. Подальші підрозділи містять конкретні числові результати, отримані в ході описаних експериментів.

3.4 Результати вимірювання продуктивності відеопам’яті

У цьому підрозділі наведено основні результати тестування системи у частині, яка стосується споживання відеопам’яті. Саме цей блок результатів є ключовим для оцінки ефективності адаптивного керування, оскільки головна мета системи - зменшити обсяг VRAM, який одночасно займається ресурсами сцени, без помітної втрати якості рендерингу. Дані, наведені нижче, отримано на остнові CSV-файлів, які логувалися компонентом FrameLogger під час двох незалежних 60-секундних прогонів - у режимі ADAPTIVE та у режимі BASELINE. Кількість записаних кадрів у кожному прогоні складає близько 13-14 тисяч, що дає достатню статистичну базу для аналізу.

Безпосередньо за зміну рівня VRAM відповідає метод HandleFrame у класі ResidencyManager. Він виконується через кожні UpdateEveryFrames кадрів і реалізує три послідовні дії: визначення видимих ресурсів, завантаження тих, які стали видимими, і вивантаження тих, які давно не використовувались. Відповідний фрагмент коду наведено на рисунку 3.6.

```
1 reference
private void HandleFrame()
{
    var visible = VisibilityDetector
                      .GetVisibleResources(_allResources);

    // Build visible set for fast lookup
    var visibleSet = new HashSet<string>();
    foreach (var res in visible)
        visibleSet.Add(res.ResourceID);

    // Load visible unloaded resources
    foreach (var res in visible)
    {
        res.Priority = CalculatePriority(res);

        if (!res.IsResident &&
            _loaders.TryGetValue(res.ResourceID, out var loader) &&
            loader != null)
        {
            res.State = LoadState.Loading;
            loader.LoadAndApply();
            res.IsResident = true;
            res.State = LoadState.Ready;
        }
    }

    // Unload non-visible resident resources (LRU)
    foreach (var res in _allResources)
    {
        if (!res.IsResident) continue;
        if (visibleSet.Contains(res.ResourceID)) continue;

        float stale = Time.time - res.LastAccessTime;
        if (stale > EvictAfterSeconds)
        {
            if (_loaders.TryGetValue(
                    res.ResourceID, out var loader))
            {
                loader.Unload();
                res.IsResident = false;
                res.State = LoadState.Unloaded;

                UnityEngine.Debug.Log(
                    $"[ResidencyManager] Evicted: " +
                    $"{res.ResourceID} " +
                    $"({stale:F1}s inactive)");
            }
        }
    }

    // VRAM pressure eviction
    if (GetUsedVramBytes() > VramLimitBytes)
        EvictLowPriority(visibleSet);
}
```

Рисунок 3.6 Лістинг коду методу HandleFrame менеджера резидентності

Реальне використання відеопам’яті у часі відображає метрика ActualVRAMMB, яка отримується через виклик Profiler.GetAllocatedMemoryForGraphicsDriver і показує фактичне виділення пам’яті графічним драйвером. Її динаміку для обох режимів наведено на рисунку 3.7. У режимі BASELINE значення майже одразу досягає рівня близько 4300 МБ і залишається стабільним протягом усього тесту, оскільки всі 500 об’єктів завантажуються одразу. У режимі ADAPTIVE крива стартує з рівня близько 300 МБ і коливається у діапазоні 1500–2500 МБ, повторюючи характер руху камери.

Рисунок 3.7 Реальне використання відеопам’яті у часі

Метрика TrackedVRAMMB є внутрішньою оцінкою системи – сумою номінальних розмірів усіх резидентних ресурсів. У нашій конфігурації кожен об’єкт зареєстровано як 16 МБ, тож при 500 завантажених ресурсах TrackedVRAMMB у BASELINE становить рівно 8000 МБ. На рисунку 3.8 видно, що у ADAPTIVE TrackedVRAMMB і ActualVRAMMB рухаються синхронно, але на різних рівнях. Внутрішня оцінка завищена через те, що HDRP застосовує стиснення текстур у форматах BC7 і BC5, тоді як облік ведеться за номінальним розміром. Для роботи цінні обидва показники: один характеризує логічний стан менеджера, інший – фактичний вплив на систему.

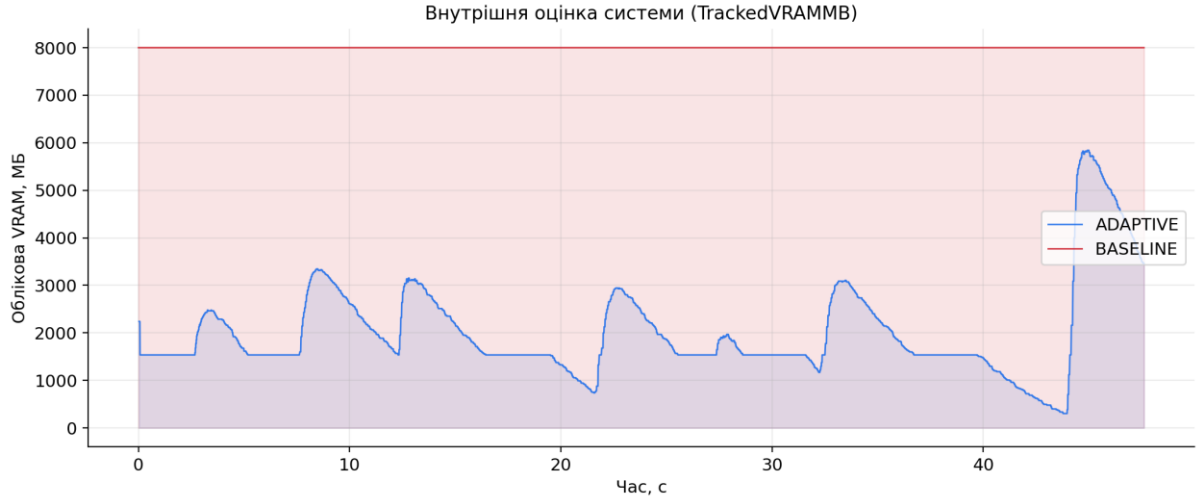

Рисунок 3.8 Внутрішня оцінка системи у часі

Розподіл значень ActualVRAMMB за весь час тесту наведено на рисунку 3.9. У BASELINE розподіл має вузький пік навколо 4327 МБ зі стандартним відхиленням лише 13 МБ, що підтверджує статичний характер режиму. У ADAPTIVE розподіл значно ширший, охоплює діапазон 300–3000 МБ і має

концентрацію значень у зоні 1500–1800 МБ. Така форма свідчить про постійне балансування системи між завантаженням і вивантаженням ресурсів.

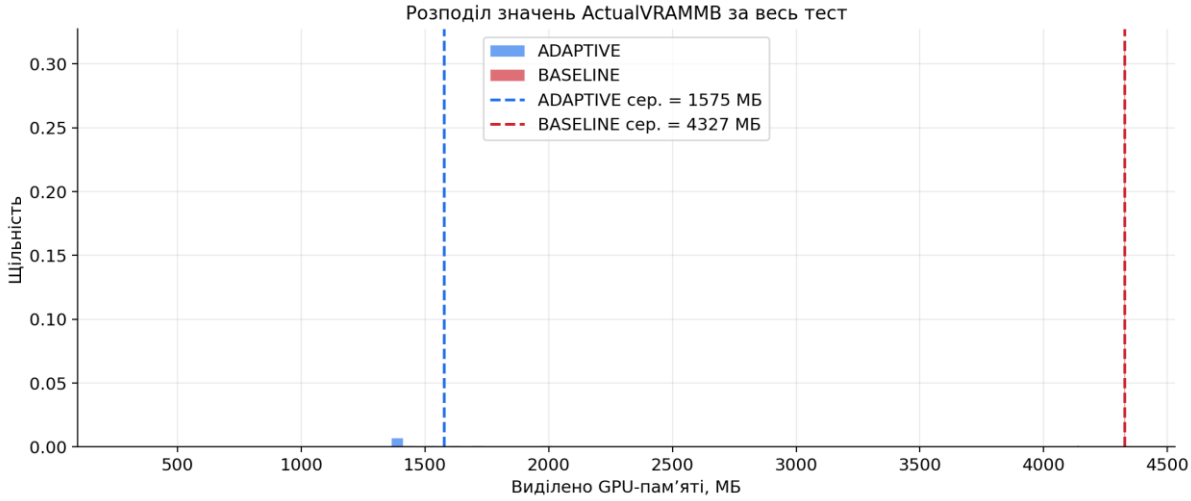

Рисунок 3.9 Розподіл значень ActualVRAM за весь тест

Поведінку самого менеджера резидентності найкраще ілюструє динаміка кількості завантажених об'єктів (рисунок 3.10). У BASELINE значення ResidentCount стале і дорівнює 500. У ADAPTIVE крива має пилкоподібну форму з діапазоном 19–365 і середнім 120. Це означає, що адаптивна система у середньому тримає у пам'яті близько 24% об'єктів сцени, причому максимум жодного разу не сягає 500, що підтверджує коректну роботу LRU-вивантаження.

Рисунок 3.10 Динаміка кількості резидентних ресурсів

Підсумкові числові показники наведено у таблиці 3.7. Значення обчислено за період після фази warm-up.

Показник	**ADAPTIVE**	**BASELINE**
ActualVRAMMB, середнє	1613 МБ	4327 МБ
ActualVRAMMB, мінімум	299 МБ	4137 МБ
ActualVRAMMB, максимум	3079 МБ	4328 МБ

TrackedVRAMMB, середнє	1923 МБ	8000 МБ
ResidentCount, середнє	120	500
ResidentCount, діапазон	19-365	500-500

Таблиця 3.7 Статистика використання VRAM

Економія реальної відеопам’яті, обчислена як відношення середніх значень ActualVRAMMB, складає 2752 МБ, або 63,6%. Тобто адаптивний менеджер виконує рендеринг тієї самої сцени з майже втричі меншим фактичним обсягом GPU-пам’яті. У термінах внутрішньої оцінки TrackedVRAMMB різниця сягає 76%. Така економія підтверджує головну гіпотезу роботи, а вплив адаптивного підходу на продуктивність розглянуто у наступному підрозділі.

3.5 Результати вимірювання продуктивності рендерингу

Економія відопам’яті, отримана у попередньому підрозділі, є важливим, але не єдиним показником ефективності системи. Адаптивний механізм у режимі реального часу постійно завантажує та вивантажує ресурси, і ці операції теоретично можуть впливати на час рендерингу окремих кадрів. Тому необхідно окремо оцінити, як саме адаптивне керування пам’яттю позначається на продуктивності, тобто на часі рендерингу одного кадру (FrameTimeMs) та похідний від нього метриці - частоті кадрів (FPS).

Перед розглядом графіків доцільно показати фрагмент коду, що відповідає за найбільш помітне з точки зору продуктивності рішення - вивантаження низько пріоритетних ресурсів при перевантаженні VRAM. Цей метод викликається лише тоді, коли поточне використання пам’яті перевищує заданий ліміт VramLimitBytes. Він сортує всі ресурси за пріоритетом, обчисленим у методі CalculatePriority, та послідовно вивантажує ті, що мають найнижчий пріоритет, поки обсяг звільненої пам’яті не перевищить необхідний цільовий показник. Реалізацію наведено на рисунку 3.11.

```
private void EvictLowPriority(HashSet<string> visibleSet)
{
    _allResources.Sort((a, b) =>
        a.Priority.CompareTo(b.Priority));

    long freed = 0;
    long target = GetUsedVramBytes() - VramLimitBytes;

    foreach (var res in _allResources)
    {
        if (freed >= target) break;
        if (!res.IsResident) continue;
        if (visibleSet.Contains(res.ResourceID)) continue;

        if (_loaders.TryGetValue(res.ResourceID, out var loader) && loader != null)
        {
            loader.Unload();
            res.IsResident = false;
            res.State = LoadState.Unloaded;
            freed += res.SizeBytes;
        }
    }
}
```

Рисунок 3.11 Лістинг коду методу EvictLowPriority

Загальний розподіл часу кадру для обох режимів зручніше за все аналізувати у вигляді boxplot, який наведено на рисунку 3.12. Цей тип графіка показує медіану, межі першого та третього квартилей, а також межі типового діапазону значень, виключаючи поодинокі викиди.

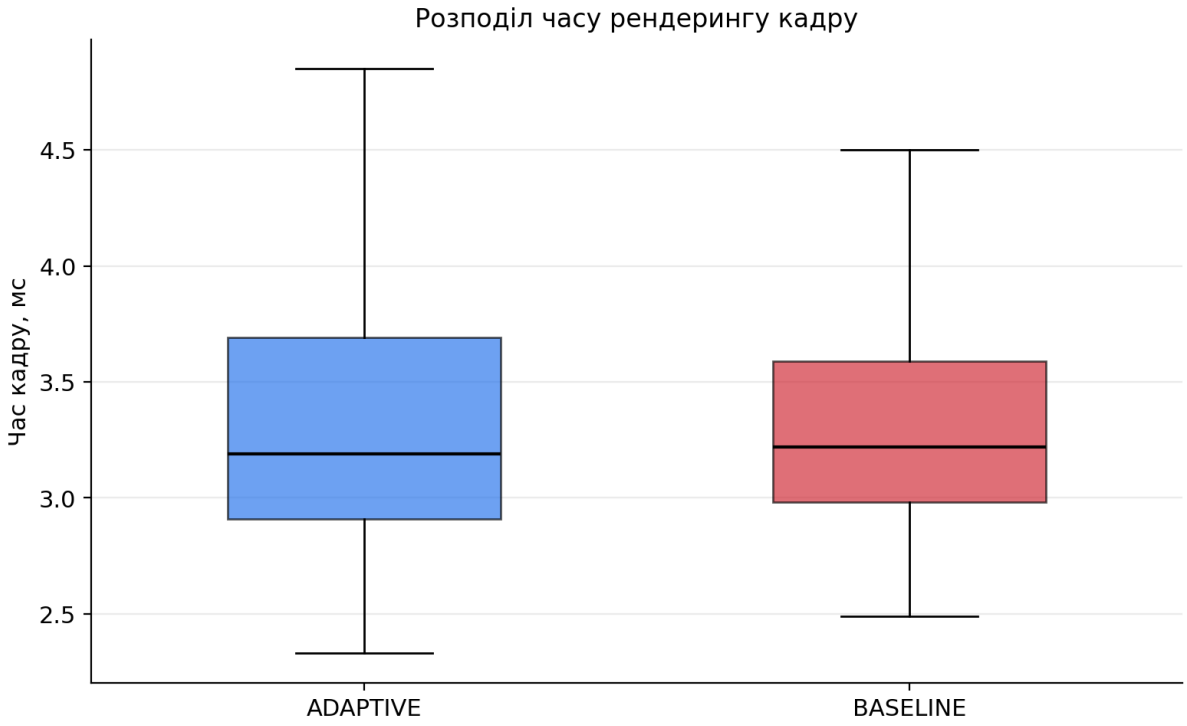

Рисунок 3.12 Розподіл часу рендерингу кадру

На графіку видно, що медіани двох режимів практично співпадають і складають близько 3.2 мс. Це означає, що у типовому кадрі обидва режими виконують рендеринг приблизно з однаковою швидкістю. Однак ширина боксу для ADAPTIVE помітно більша, ніж для BASELINE усі ресурси вже завантажено на старті, тоді як у режимі ADAPTIVE під час руху камери постійно виконуються операції завантаження або вивантаження, які додають невеликі затримки до окремих кадрів.

Більш детально розподіл часу кадру показано на гістограмі з рисунку 3.13. Для зручності перегляду значення обрізано на рівні 20 мс, оскільки кадрів, що перевищують цю межу, у ADAPTIVE менше одного відсотка, а у BASELINE окремі великі викиди пов'язані з моментом початкового завантаження, а не з нормальною роботою.

Рисунок 3.13 Гістограма часу кадру

Розподіл BASELINE виглядає компактним та сконцентрованим у вузькому діапазоні навколо 3-4 мс. Розподіл ADAPTIVE також має основний пік у тому самому місці, але має помітний хвіст у бік більших значень. Цей хвіст і є вартістю адаптивної логіки: окремі кадри, які потрапляє момент завантаження або вивантаження ресурсу, потребують більше часу на обробку. Динаміку частоти кадрів у часі наведено на рисунку 3.14. Для зменшення шуму застосовано згладжування ковзним середнім по 50 кадрах – без нього графік був би занадто щільним для візуального сприйняття.

Рисунок 3.14 - Динаміка частоти кадрів у часі

Обидві криві стабільно тримаються на рівні приблизно 280-300 FPS, що значно перевищує цільовий рівень у 60 FPS, позначений горизонтальною пунктирною лінією. Час від часу крива ADAPTIVE опускається нижче, що відповідає моментам активного перерозподілу ресурсів, але навіть у найнижчих точках частота кадрів залишається на рівні близько 100 FPS. Це підтверджує, що адаптивна логіка не призводить до візуально помітних просідань.

Числові показники продуктивності зведено у таблиці 3.8.

Показник	**ADAPTIVE**	**BASELINE**
FrameTimeMs, середнє	3.77	3.37
FrameTimeMs, медіана	3.19	3.22
FrameTimeMs, p95	8.19	4.29
FrameTimeMs, p99	10.84	5.13
FPS, середнє	296	305
FPS, p1 (нижній перцентиль)	92	195

Таблиця 3.8 Статистика часу кадру та FPS

Аналіз цих чисел дозволяє побачити чіткий компроміс. Середній час кадру у ADAPTIVE лише на 0.4 мс вищий, ніж у BASELINE, а медіани взагалі практично співпадають. Найбільш помітна різниця проявляється у хвостах розподілу: 95-й перцентиль ADAPTIVE удвічі вищий за BASELINE (8,19 проти 4,29 мс), а 99-й – більш ніж у два рази (10,84 проти 5,13 мс). Це означає, що приблизно один кадр зі ста у режимі ADAPTIVE буде виконуватися удвічі довше, ніж типовий кадр.

З практичної точки зору ці затримки не є критичними. Навіть найгірші 1% кадрів у ADAPTIVE укладаються у часовий бюджет 16,67 мс, що відповідає 60 FPS – загальноприйнятому порогу плавності рендерингу для більшості застосунків. Тобто адаптивна система зберігає достатній рівень

продуктивності навіть у моменти найактивнішої роботи менеджера резидентності.

Таким чином, отримані результати показують, що економія VRAM на рівні 63,6% описана у. попередньому підрозділі, досягається ціною незначного збільшення розкиду часу кадру. Середня продуктивність залишається практично такою ж, як і у статичному режимі, а більші затримки виникають лише у короткі моменти активного переходу через скупчення об’єктів. Цей компроміс є прийнятним для більшості сценаріїв використання є саме обсяг доступної відеопам’яті, а не часовий бюджет рендерингу.

3.6 Аналіз поведінки адаптивного алгоритму та зведене порівняння

У попередніх підрозділах було розглянуто результати по окремих метриках - використано VRAM та часу частину рендерингу. У цьому підрозділі ці результати зводяться у єдиному порівнянні, а також розглядаються механізми, які забезпечують отриману поведінку системи. Ключовими серед них є визначення видимих ресурсів, оскільки саме видимість є основним критерієм рішенням про резидентність кожного об’єкта.

Цю функцію виконує метод GetVisibleResources класу VisibilityDetector. Він не тільки виявляє об’єкти у фрустумі камери, а й оновлює для кожного видимого ресурсу його геометричні характеристики: площу на екрані, відстань до камери та час останнього доступу. Саме на основі цих значень потім обчислюється пріоритет, який використовується у методі EvictLowPriority. Реалізацію наведено на рисунку 3.15.

```
public List<ResourceRecord> GetVisibleResources(
                                List<ResourceRecord> all)
{
    GeometryUtility.CalculateFrustumPlanes(_cam, _frustumPlanes);

    var visible = new List<ResourceRecord>();

    foreach (var rec in all)
    {
        if (!_rendererCache.TryGetValue(rec, out var rend))
            continue;
        if (rend == null) continue;

        bool inFrustum = GeometryUtility.TestPlanesAABB(
            _frustumPlanes, rend.bounds);

        if (!inFrustum) continue;

        rec.ScreenCoverage = GetScreenCoverage(rend);
        rec.DistanceToCamera = Vector3.Distance(
            _cam.transform.position,
            rend.bounds.center);
        rec.WorldPosition = rend.bounds.center;
        rec.LastAccessTime = Time.time;

        visible.Add(rec);
    }

    return visible;
}
```

Рисунок 3.15 Лістинг коду методу GetVisibleResources

Важливою деталлю реалізації є кешування посилань Renderer для кожного зареєстрованого ресурсу. Без такого кешу довелося б щокадру викликати GameObject.Find або обходити сцену в пошуках потрібних компонентів, що при 500 об’єктах призводило б до помітних просідань FPS. Завдяки тому, що пошук рендерерів виконується один раз під час реєстрації, а потім використовується кеш, GetVisibleResources залишається швидким навіть для великої кількості об’єктів. Це і є однією з причин, чому ADAPTIVE-режим зберігає прийнятну продуктивність, описану у підрозділі 3.5.

Якщо подивитися на графік кількості резидентних ресурсів з рисунку 3.10, можна побачити характерну кореляцію між положенням камери та кількістю об’єктів у пам’яті. Коли камера на високих waypoints (WP_00, WP_05, WP_07 на висоті 60–80 м) має широкий огляд, кількість видимих

об’єктів і відповідно ResidentCount зростає. Коли камера проходить низькі точки (WP_02 і WP_03 на висоті 10 м), у фрустум потрапляє локальна група об’єктів, і кількість резидентних знижується. Таким чином, динаміка ResidentCount є прямим наслідком роботи методу GetVisibleResources, який щокадру переоцінює список видимих ресурсів.

Зведене порівняння обох режимів за усіма ключовими метриками наведено у таблиці 3.9.

Показник	**ADAPTIVE**	**BASELINE**	**Перевага ADAPTIVE**
ActualVRAMMB, середнє	1575 МБ	4327 МБ	–2752 МБ (–63,6%)
ActualVRAMMB, максимум	3079 МБ	4328 МБ	–1249 МБ
TrackedVRAMMB, середнє	1915 МБ	8000 МБ	–6085 МБ (–76,1%)
ResidentCount, середнє	120	500	–380 (–76,0%)
FrameTimeMs, середнє	3,77 мс	3,37 мс	+0,40 мс (+11,9%)
FrameTimeMs, p95	296	305	–9 (–3,0%)

Таблиця 3.9 Зведення порівняння ефективності двох режимів

Для наочного сприйняття ключові показники подано також у вигляді стовпчикової діаграми (рисунок 3.16).

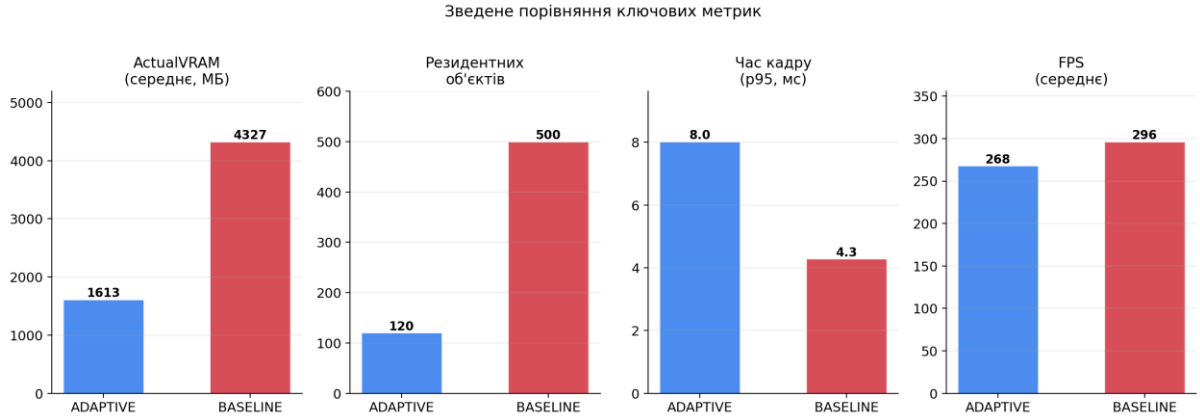

Рисунок 3.16 Діаграма зведено порівняння ключових метрик

З таблиці та діаграми видно, що адаптивна система виграє у двох групах показників і програє в одній. Перша перевага – фактичне використання VRAM, де економія сягає 63,6%, або 2752 МБ у середньому. Друга перевага – кількість одночасно завантажених ресурсів, яка зменшується у понад чотири рази. Програш проявляється у показнику p95 часу кадру, де ADAPTIVE приблизно вдвічі повільніший за BASELINE – 8,19 мс проти 4,29 мс.

Натомість середня частота кадрів практично не відрізняється (296 проти 305 FPS), а різниця у середньому часі кадру складає лише 0,4 мс.

З точки зору практичного використання такий компроміс є вигідним. Зниження кількості резидентних ресурсів дозволяє запускати ту саму сцену на конфігураціях з меншим обсягом VRAM, або ж масштабувати її розмір без необхідності модернізації обладнання. Ціною є невелике збільшення мінливості часу кадру, яке не виходить за межі норми навіть у найгірших випадках: жоден з обчислених перцентилів (p95, p99) не перевищує бюджет 16,67 мс, що відповідає 60 FPS. У застосунках, де якість плавності рендерингу обмежується саме цим порогом, адаптивний підхід забезпечує одночасно і прийнятну продуктивність, і суттєву економію пам’яті.

Слід зазначити, що отримані результати залежать від конкретних параметрів конфігурації менеджера – ліміту VRAM, частоти оновлення HandleFrame та часу бездіяльності перед вивантаженням. При інших значеннях цих параметрів баланс між економією та продуктивністю може зміщуватися в той чи інший бік. Наприклад, зменшення EvictAfterSeconds робить систему агресивнішою у вивантаженні, що збільшує економію VRAM, але одночасно підвищує кількість повторних завантажень тих самих ресурсів і відповідно – p95 часу кадру. Підбір параметрів для конкретного застосунку є окремою інженерною задачею, яка виходить за межі даної роботи.

Таким чином, проведені експерименти підтвердили, що розроблена система адаптивного керування пам’яттю GPU здатна суттєво знизити споживання відеопам’яті без істотного впливу на середню продуктивність рендерингу. Виявлені обмеження – підвищена варіативність часу окремих кадрів і залежність балансу від конкретних параметрів – є типовими для систем streaming і не зменшують практичну корисність запропонованого рішення.

3.7 Висновки до розділу 3

У третьому розділі виконано експериментальне дослідження розробленої системи адаптивного керування пам’яттю GPU. Тестування проводилося у середовищі Unity 6.3 LTS з рендер-пайплайном HDRP 17.3.0 на синтетичній сцені з 500 об’єктами та фіксованою траєкторією руху камери, що дозволило отримати порівнянні та відтворювані результати.

Для оцінки ефективності системи проведено два прямі порівняльні прогони. У режимі ADAPTIVE працював повний цикл менеджера резидентності з визначенням видимих ресурсів, асинхронним завантаженням, LRU-

вивантаженням та евікцією за пріоритетом. У режимі BASELINE усі ресурси сцени завантажувалися статично на старті, а адаптивна логіка була повністю вимкнена. Покадровий збір метрик дозволив побудувати статистично значущі розподіли часу кадру, фактичного використання GPU-пам'яті та кількості резидентних об'єктів.

Основним результатом експерименту є зниження середнього фактичного використання відеопам'яті з 4327 МБ у BASELINE до 1575 МБ у ADAPTIVE, що відповідає економії 63,6%, або 2752 МБ. Середня кількість одночасно резидентних ресурсів зменшилася з 500 до 120, тобто система у середньому тримала у пам'яті лише 24% об'єктів сцени замість усіх 100%. Отриманий результат підтверджує, що адаптивне керування дозволяє виконувати рендеринг тієї самої сцени з істотно меншим обсягом фізично виділеної відеопам'яті.

Аналіз продуктивності показав, що економія VRAM досягається ціною помірного збільшення варіативності часу кадру. Середній час рендерингу у ADAPTIVE зріс лише на 0,4 мс (з 3,37 до 3,77 мс), а 95-й перцентиль – з 4,29 до 8,19 мс, що приблизно вдвічі вище. Водночас навіть у найгірших випадках час кадру не перевищував 16,67 мс, тобто система завжди залишалася у межах бюджету 60 FPS. Середня частота кадрів знизилася лише на 3% (з 305 до 296 FPS), що з практичної точки зору є несуттєвим.

У ході роботи виявлено й окремі обмеження запропонованого підходу, які слід враховувати при подальшому використанні системи. По-перше, отримані результати є справедливими для конкретної конфігурації параметрів менеджера – ліміту VRAM 1,5 ГБ, періодичності HandleFrame у п'ять кадрів та часу бездіяльності 4 секунди. При інших значеннях баланс між економією та продуктивністю буде іншим. По-друге, експеримент проводився у середовищі Editor Play Mode, тобто без оптимізації IL2CPP, яка застосовується у продуктивній збірці. По-третє, як накопичувач використовувався HDD, що збільшує затримки асинхронного завантаження – на SSD поведінка системи була б ще плавнішою.

Підсумовуючи, результати цього розділу демонструють, що розроблена система адаптивного керування пам'яттю GPU виконує поставлене у роботі завдання – забезпечує суттєву економію відеопам'яті без помітного зниження продуктивності рендерингу великої сцени. Це підтверджує практичну цінність запропонованого підходу та доцільність його подальшого розвитку.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було досліджено проблему ефективного керування відеопам’яттю GPU при рендерингу великих сцен та розроблено прототип системи, що реалізує адаптивний підхід до цього завдання. Робота охопила як теоретичний аналіз, так і практичну реалізацію з подальшим експериментальним підтвердженням ефективності запропонованого рішення.

У першому розділі проаналізовано основні причини перевантаження VRAM у великих сценах і розглянуто існуючі підходи до оптимізації – streaming, mipmapping, LOD, virtual texturing та sparse residency. Показано, що окремі методи вирішують лише частину задачі, а найкращий результат досягається при їх поєднанні з механізмом адаптивного керування. Сформульовано ключові поняття – резидентність ресурсу, пріоритетизація, eviction policy, prefetching та асинхронне I/O – які стали теоретичною основою для подальшого проєктування.

У другому розділі сформульовано вимоги до системи, визначено критерії пріоритету ресурсів і запропоновано модульну архітектуру. Виділено основні компоненти системи: менеджер резидентності, модуль визначення видимості, асинхронну чергу завантаження, модуль prefetching та підсистему профілювання. Описано логіку роботи менеджера у вигляді послідовних кроків (визначення видимих об’єктів, обчислення пріоритету, прийняття рішень про завантаження або вивантаження) та запропоновано гібридну політику вивантаження, що поєднує LRU за часом бездіяльності з евікцією за пріоритетом при перевищенні ліміту VRAM.

У третьому розділі описано програмну реалізацію прототипу у середовищі Unity 6.3 LTS з рендер-пайплайном HDRP 17.3.0, а також проведено експериментальне дослідження. Тестування виконано на синтетичній сцені з 500 об’єктами та фіксованою траєкторією камери, що забезпечило відтворюваність результатів. Для порівняння реалізовано два режими роботи: ADAPTIVE з повним циклом адаптивного керування та BASELINE зі статичним завантаженням усіх ресурсів на старті.

Основним результатом експерименту є зниження середнього фактичного використання відеопам’яті з 4327 МБ у BASELINE до 1575 МБ у ADAPTIVE, що відповідає економії 63,6%, або 2752 МБ. Середня кількість одночасно резидентних ресурсів зменшилася зі 500 до 120 – тобто система у середньому тримала у пам’яті лише 24% об’єктів сцени. Ці результати

підтверджують, що адаптивне керування дозволяє виконувати рендеринг тієї самої сцени з істотно меншим обсягом фактично виділеної відеопам’яті.

Аналіз продуктивності показав, що економія VRAM досягається ціною помірного збільшення варіативності часу кадру. Середній час рендерингу зріс лише на 0,4 мс (з 3,37 до 3,77 мс), а 95-й перцентиль – з 4,29 до 8,19 мс. Середня частота кадрів знизилась лише на 3% (з 305 до 296 FPS). При цьому навіть у найгірших випадках час кадру не перевищував 16,67 мс, тобто система завжди залишалася у межах бюджету 60 FPS. Виявлений компроміс є типовим для систем streaming і не зменшує практичної цінності запропонованого рішення.

Поставлена у роботі мета досягнута: розроблено прототип системи адаптивного керування пам’яттю GPU та експериментально підтверджено його ефективність. Усі завдання роботи виконано в повному обсязі.

Серед напрямків подальшого розвитку слід відзначити інтеграцію з механізмом MipMapStreaming для роботи з рівнями деталізації текстур, додавання просторового індексу (octree або BVH) для прискорення визначення видимості у сценах з тисячами об’єктів, а також впровадження прогнозування траєкторії камери для більш точної роботи модуля prefetching. Розроблену систему можна використовувати як основу для оптимізації реальних проєктів на Unity, що працюють зі сценами великого обсягу.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Akenine-Möller T., Haines E., Hoffman N., Pesce A., Iwanicki M., Hillaire S. Real-Time Rendering. 4th ed. Boca Raton : CRC Press, 2018. 1198 p.
2. Williams L. Pyramidal Parametrics. Computer Graphics (SIGGRAPH ’83 Proceedings). 1983. Vol. 17, No. 3. P. 1–11. DOI: 10.1145/964967.801126.
3. Luebke D., Reddy M., Cohen J. D., Varshney A., Watson B., Huebner R. Level of Detail for 3D Graphics. San Francisco : Morgan Kaufmann, 2003. 432 p.
4. Streaming Virtual Texturing : Unity Manual. URL: https://docs.unity3d.com/6000.3/Documentation/Manual/svt-streaming-virtual-texturing.html (дата звернення: 01.06.2026).
5. How Streaming Virtual Texturing works : Unity Manual. URL: https://docs.unity3d.com/6000.3/Documentation/Manual/svt-how-it-works.html (дата звернення: 01.06.2026).
6. Sparse Resources : Vulkan Documentation / Khronos Group. URL: https://docs.vulkan.org/spec/latest/chapters/sparsemem.html (дата звернення: 01.06.2026).
7. How Virtual Textures Really Work : вебсайт. URL: https://www.shlom.dev/articles/how-virtual-textures-really-work/ (дата звернення: 01.06.2026).
8. Unreal Engine Optimization Guide: Profiling Fundamentals : Intel Developer Zone. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/unreal-engine-optimization-profiling-fundamentals.html (дата звернення: 01.06.2026).
9. PresentMon : performance analysis tool / Intel GameTechDev. GitHub. URL: https://github.com/GameTechDev/PresentMon (дата звернення: 01.06.2026).
10. NVIDIA Nsight Systems User Guide : NVIDIA Documentation. URL: https://docs.nvidia.com/nsight-systems/UserGuide/ (дата звернення: 01.06.2026).

11. Using the High Definition Render Pipeline : Unity Manual. URL: https://docs.unity3d.com/6000.3/Documentation/Manual/high-definition-render-pipeline.html (дата звернення: 01.06.2026).
12. Harris C. R., Millman K. J., van der Walt S. J. et al. Array programming with NumPy. Nature. 2020. Vol. 585. P. 357–362. DOI: 10.1038/s41586-020-2649-2.
13. Hunter J. D. Matplotlib: A 2D Graphics Environment. Computing in Science & Engineering. 2007. Vol. 9, No. 3. P. 90–95. DOI: 10.1109/MCSE.2007.55.
14. McKinney W. Data Structures for Statistical Computing in Python. Proceedings of the 9th Python in Science Conference (SciPy 2010). 2010. P. 56–61.