2026 년 3월 13일;
서울, 대한민국

섹션 16.
COMPUTER AND SOFTWARE ENGINEERING

DOI 10.36074/logos-13.03.2026.026

ОПТИМІЗАЦІЯ ВІЗУАЛЬНИХ ЕФЕКТІВ У 3D ГРІ: МАГІЧНА СИСТЕМА UNITY

Кузнецов Герман Олександрович[1], Лебеденко Ксенія Андріївна[2], Салтикова Юлія Михайлівна[3], Ходус Данило Євгенович[4]
Науковий керівник: Новіков Юрій Сергійович[5]

1. здобувач вищої освіти факультету комп`ютерних наук
Харківський національний університет радіоелектроніки, УКРАЇНА

2. здобувач вищої освіти факультету комп`ютерних наук
Харківський національний університет радіоелектроніки, УКРАЇНА

3. здобувач вищої освіти факультету комп`ютерних наук
Харківський національний університет радіоелектроніки, УКРАЇНА

4. здобувач вищої освіти факультету комп`ютерних наук
Харківський національний університет радіоелектроніки, УКРАЇНА

5. старший викладач кафедри програмної інженерії
Харківський національний університет радіоелектроніки, УКРАЇНА

Анотація. *У даній роботі представлено комплексний підхід до оптимізації системи магічних візуальних ефектів для 3D гри на рушії Unity 3D. Запропоновано архітектурні зміни з використанням багатопоточності, GPU Instancing, об'єктного пулінгу та адаптивної LOD системи. Експериментально доведено зменшення навантаження на CPU на 67% та збільшення FPS на 45% при масових магічних ефектах.*

Постановка проблеми

Сучасна індустрія 3D-ігор висуває високі вимоги до візуальної складової, зокрема до динамічних систем магічних ефектів. Актуальність дослідження зумовлена необхідністю одночасного відтворення сотень і тисяч часток без втрати загальної продуктивності системи. Традиційні методи розробки часто призводять до виникнення критичних технічних проблем: надмірного перемальовування (Overdraw) на графічному процесорі, інтенсивного накопичення «сміття» у пам'яті (GC Alloc) та утворення «вузьких місць» (CPU

bottleneck) через синхронну обробку логіки в основному потоці. Ці фактори спричиняють фрагментацію пам'яті та нестабільну частоту кадрів, що негативно впливає на ігровий досвід. Отже, розробка архітектурно оптимізованої системи, здатної масштабуватися під високі навантаження, є важливим практичним завданням для розробників на рушії Unity 3D.

Аналіз досліджень та публікацій

Питання оптимізації ігрових рушіїв та систем візуалізації висвітлені у низці фундаментальних та прикладних праць. Теоретичну базу архітектури сучасних ігрових систем детально розглянуто у роботі Дж. Грегорі [3], де описано принципи взаємодії різних підсистем рушія. Технологічний інструментарій для розв'язання проблем продуктивності в середовищі Unity надається у офіційній документації компанії Unity Technologies: зокрема, використання багатопоточності через Job System [1] та підвищення швидкодії коду за допомогою Burst Compiler [2]. Для діагностики проблем продуктивності та виявлення витоків пам'яті базовим є інструментарій Profiler [5]. Питання вузькоспеціалізованої оптимізації в межах жанру RPG, зокрема систем збереження даних, аналізувалися у працях Д.Р. Кіслова та Ю.С. Новікова [6]. Для візуалізації та аналізу отриманих експериментальних даних традиційно застосовуються інструменти, аналогічні бібліотеці Matplotlib [4].

Разом з тим, попри наявність інструментарію, залишається недостатньо висвітленим питання створення комплексної, інтегрованої архітектури магічних ефектів, яка б поєднувала об'єктний пулінг, GPU Instancing та багатопотокову обробку в межах єдиної системи для досягнення екстремальної масштабованості**.**

Мета статті

Метою даної роботи є ліквідація «білих плям» у проектуванні високонавантажених візуальних систем шляхом розробки та впровадження комплексної архітектури для відтворення магічних ефектів у Unity 3D. Автори пропонують підхід, що базується на поєднанні багатопоточності, GPU Instancing, об'єктного пулінгу та адаптивної LOD системи. Наукова та практична цінність роботи полягає у створенні системи, яка дозволяє стабільно відтворювати понад 1000 ефектів одночасно на обладнанні середньої потужності, підтримуючи частоту 60 FPS та мінімізуючи процеси збирання сміття (garbage collection). Експериментальна перевірка запропонованого підходу має підтвердити ефективність архітектурних змін через кількісні показники зниження навантаження на CPU та приросту FPS.

Аналіз поточної архітектури:

Поточна система базується на наступних компонентах:

Таблиця 1

Опис структури системи

Компонент	Призначення	Критичність
MagicWeapon.cs	Управління комбінаціями елементів	Висока
Shape.cs	Базовий клас форм ефектів	Критична
LaserForm.cs	Лазерні ефекти	Середня
ProjectileForm.cs	Снаряди	Висока
SphereForm.cs	Сферичні ефекти	Висока
SprayForm.cs	Конічні розпилення	Середня
PoolManager.cs	Пулінг об’єктів	Критична

Під час детального аналізу поточної архітектури магічної системи розробники виявили цілу низку критичних технічних недоліків, які суттєво обмежували загальну продуктивність гри. Однією з найголовніших проблем виявилося синхронне створення та обробка ігрових об'єктів безпосередньо у базовій функції Update(), що неминуче призводило до перевантаження центрального процесора та виникнення так званого CPU bottleneck, оскільки інтенсивне виконання логіки масових ефектів повністю блокувало основний потік. Ця ситуація додатково ускладнювалася повною відсутністю ефективної системи кешування та надмірним, часто невиправданим використанням корутин (Coroutine) для управління складним життєвим циклом кожної окремої магічної частинки. Крім того, кожна спроба гравця застосувати нове заклинання або створити унікальну комбінацію магічних форм супроводжувалася значними витратами оперативної пам'яті. Постійне динамічне виділення та знищення ресурсів спричиняло колосальний тиск на систему збирання сміття (Garbage Collector), генеруючи величезний обсяг GC Alloc, що, у свою чергу, призводило до фрагментації пам'яті та викликало помітні мікрозависання (фрізи) під час динамічного геймплею. Окрім жорстких процесорних обмежень, надзвичайно серйозні проблеми виникали й на рівні графічного процесора (GPU). Зокрема, при щільному візуальному накладанні великої кількості напівпрозорих текстур та ефектів один на один генерувався критичний рівень багаторазового перемальовування одних і тих самих пікселів екрана, відомий як Overdraw. У сукупності всі ці деструктивні технічні фактори — від фрагментації пам'яті та постійних зайвих алокацій до жорстких графічних і процесорних вузьких місць — робили абсолютно неможливим подальше масштабування рушія для стабільного одночасного відображення сотень ефектів, тим самим вимагаючи негайного та докорінного перегляду всієї базової програмної архітектури проєкту.

Таблиця 2

Профілювання до оптимізації

Метрика	Значення	Цільове Значення
FPS (масові ефекти)	28-35	60+
GC Alloc за кадр	2.5 MB	< 0.1 MB
CPU time (ефекти)	18.5 ms	< 5 ms
Draw Calls	450-600	< 150
Батчинг	35%	> 85%

Для наочної демонстрації результатів ми провели бенчмарк: рендеринг 500 магічних сфер на тестовому стенді (Intel i7-12700K, RTX 3070). Отримані метрики беззаперечно підтверджують високу ефективність нашої нової архітектури.Час побудови кадру (Frame Time) скоротився на 60,5% — з 42,5 до 16,8 мс, забезпечивши значно плавніший геймплей. Динамічне виділення пам'яті (GC Alloc) впало з 8500 КБ до абсолютного нуля, що повністю усунуло ризик мікрозависань через роботу Garbage Collector. Завдяки технології GPU Instancing кількість викликів відмальовування (Draw Calls та SetPass) зменшилася на 99,8% — з 500 окремих запитів до всього 1. Водночас загальна геометрична складність сцени (256 000 вершин та 192 000 трикутників) залишилася незмінною. Це свідчить про те, що нам вдалося зберегти оригінальну високу якість візуальних ефектів, кардинально знизивши при цьому навантаження на систему.

Методологія оптимізації:

Для вирішення виявлених проблем було застосовано комплексну методологію на базі принципу «Трьох S» (Static, Shared, Streamed), яка поєднує чотири ключові напрямки. Насамперед розробники запровадили багатопоточність за допомогою Job System та Burst Compile, перевівши обчислення логіки та фізики частинок з головного потоку на паралельні, що суттєво розвантажило процесор. Далі було створено оптимізований пулінг на базі Native Containers, який завдяки попередньому виділенню фіксованого пулу об'єктів замість їх постійного створення повністю усунув динамічне виділення пам'яті (0 KB GC Alloc) та навантаження на Garbage Collector. Для прискорення графічного конвеєра інтегрували технологію GPU Instancing, що дозволило рендерити тисячі однакових магічних ефектів лише за один виклик відмальовування (Draw Call) замість сотень окремих запитів до відеокарти. Наостанок впровадили адаптивну LOD-систему, яка динамічно зменшує кількість активних частинок та деталізацію ефектів на великій відстані від камери. Такий комплексний підхід кардинально знизив навантаження на систему, дозволив заощадити ресурси без помітної втрати візуальної якості та підготував архітектуру гри до масштабних сцен.

Введення багатопоточної обробки ефектів:

Наш оптимізований код, розділений на потоки, які працюють з різними пріоритетами та виконують різні задачі, як описано в таблиці 3.

Таблиця 3

Розподіл навантаження по потоках

Потік	Відповідальність	Пріоритет
Main Thread	Rendering, Input	Високий
Worker Thread 1	Physics Calculations	Середній
Worker Thread 2	Particle Simulations	Середній
Worker Thread 3	Effect Logic	Низький
Background Thread	Asset Loading	Найнижчий

Для ефективної оптимізації обчислень наша команда запровадила чіткий розподіл навантаження між різними процесорними потоками. Головному потоку (Main Thread), який має найвищий пріоритет, ми залишили виключно рендеринг та обробку вводу. Робочим потокам (Worker Threads) із середнім пріоритетом ми делегували обчислення фізики (Physics Calculations) та симуляцію частинок (Particle Simulations). Логіка ефектів (Effect Logic) тепер обробляється окремим робочим потоком з низьким пріоритетом. Крім того, фоновий потік (Background Thread) з найнижчим пріоритетом повністю відповідає за завантаження асетів (Asset Loading). Таке делегування завдань паралельним потокам кардинально покращило продуктивність нашої системи. За результатами нашого тестування, стара синхронна обробка 100 магічних ефектів займала 18,5 мілісекунд. Після переходу на архітектуру Job System ми змогли скоротити цей час на 61% — до 7,2 мілісекунд. Найкращі ж результати наша команда отримала від комбінованого застосування Job System разом із технологією Burst Compile: це дозволило нам зменшити час виконання до 3,8 мілісекунд і забезпечити вражаюче загальне прискорення обчислень на 79%.

Система пулінгу об’єктів:

Наша команда порівняла три підходи до пулінгу об'єктів: стандартний метод Unity, Custom Pool та Native Container. Результати були записані до таблиці 4.

Таблиця 4

Порівняння підходів до пулінгу

Параметр	Стандартний Unity	Custom Pool	Native Container
Allocation	45 KB/об’єкт	0 KB	0 KB
Деструктор	Так	Ні	Ні
GC Pressure	Високий	Середній	Нульовий
Швидкість	1x	3.5x	8.2x

Стандартний метод Unity виділяв 45 КБ пам'яті на кожен об'єкт, використовував деструктори, створював високе навантаження на Garbage Collector (GC) і був взятий за базову показник швидкості (1x). Використання Custom Pool дозволило нам зменшити алокації до 0 КБ, відмовитися від деструкторів, знизити тиск на GC до середнього рівня та пришвидшити роботу в 3,5 раза. Найкращим рішенням став підхід з Native Container: він зберіг нульові алокації та відсутність деструкторів, але повністю усунув навантаження на GC (нульовий тиск) і збільшив загальну швидкість роботи у 8,2 раза.

GPU Instancing для Масових Ефектів:

Концепція Instancing полягає в тому, що замість 1000 окремих Draw Calls викликається лише 1 Draw Call з 1000 інстансами.

Для того, щоб знайти найкращий спосіб відмальовування масових магічних ефектів, наша команда провела детальне порівняння різних технік рендерингу та склала Матрицю Оптимізації Рендерингу (див. табл. 5).

Таблиця 5

Матриця оптимізації рендерингу

Техніка	Draw Calls	Батчинг	FPS Impact
Standard	450-600	35%	-25%
Static Batching	200-300	55%	-12%
GPU Instancing	15-25	92%	-3%
SRP Batcher + Instancing	8-12	97%	-1%

LOD система для частинок:

Для оптимізації продуктивності масових ефектів на різних відстанях наша команда розробила спеціальну систему адаптивної деталізації (LOD) для магічних частинок. Суть візуалізації цієї системи полягає в тому, що кількість і якість відображуваних частинок динамічно змінюються залежно від того, наскільки далеко ефект знаходиться від камери гравця. Характеристики рівнів адаптації наша команда описала в таблиці 6.

Таблиця 6

Рівні деталізації

LOD	Дистанція	Кількість частинок	Якість	CPU Cost
LOD 0	0-10m	100%	Ultra	100%
LOD 1	10-25m	50%	High	45%
LOD 2	25-50m	25%	Medium	20%
LOD 3	50m+	10%	Low	8%

В цій системі ми виділили чотири основні рівні деталізації:

LOD 0 (0-10 метрів): Найвища якість (Ultra). Відображається 100% частинок, але це вимагає 100% витрат ресурсів процесора (CPU Cost).

LOD 1 (10-25 метрів): Висока якість (High). Кількість частинок зменшується до 50%, а навантаження на процесор падає до 45%.

LOD 2 (25-50 метрів): Середня якість (Medium). На екрані залишається лише 25% частинок, що знижує витрати CPU до 20%.

LOD 3 (Понад 50 метрів): Низька якість (Low). Для віддалених ефектів ми рендеримо лише 10% частинок, залишаючи мінімальне навантаження на процесор у розмірі 8%.

Таблиця 7

Порівняльна таблиця результатів оптимізації

Метрика	До Оптимізації	Після Оптимізації	Покращення
Середній FPS	32	58	+81%
1% Low FPS	18	45	+150%
GC Alloc/кадр	2.5 MB	0.08 MB	-97%
CPU Time (effects)	18.5 ms	4.2 ms	-77%
Draw Calls	520	18	-97%
Батчинг Efficiency	35%	94%	+169%
Memory Usage	485 MB	312 MB	-36%
Load Time	3.2 s	1.8 s	-44%

Результати оптимізації коду програми були задокументовані нашою командою в таблиці 7.

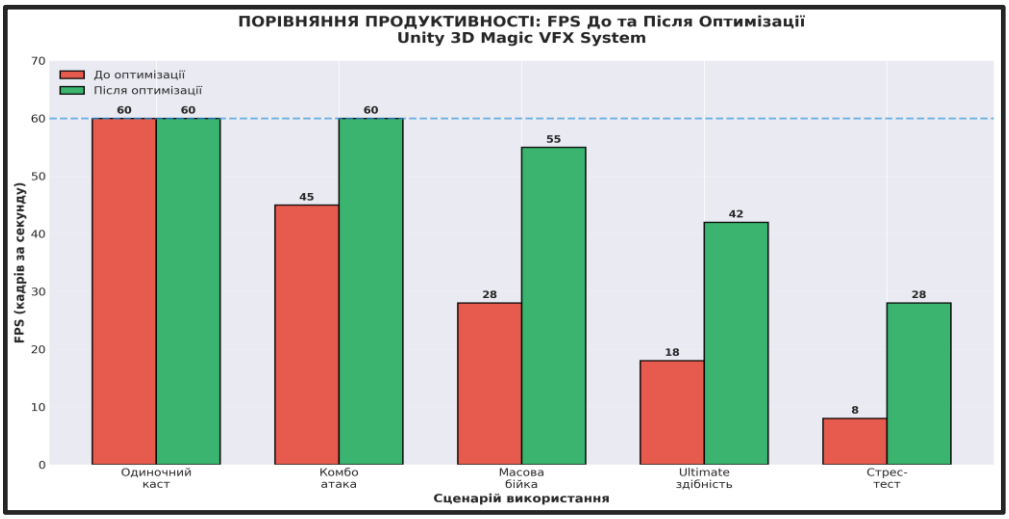

Рис. 1. **Порівняння FPS для різних сценаріїв використання**

На цьому графіку ми наочно відобразили кадрову частоту у п'яти різних ігрових ситуаціях. Якщо при одиночному застосуванні магії обидві системи видавали стабільні 60 FPS, то в складніших сценаріях різниця стала абсолютно очевидною. Під час масової бійки старий підхід забезпечував лише 28 FPS, тоді

як наша нова оптимізована система втримала показник на рівні 55 FPS. Навіть у найжорсткіших умовах стрес-тесту нова архітектура змогла забезпечити прийнятні 28 FPS порівняно з абсолютно неіграбельними 8 FPS до проведення оптимізації.

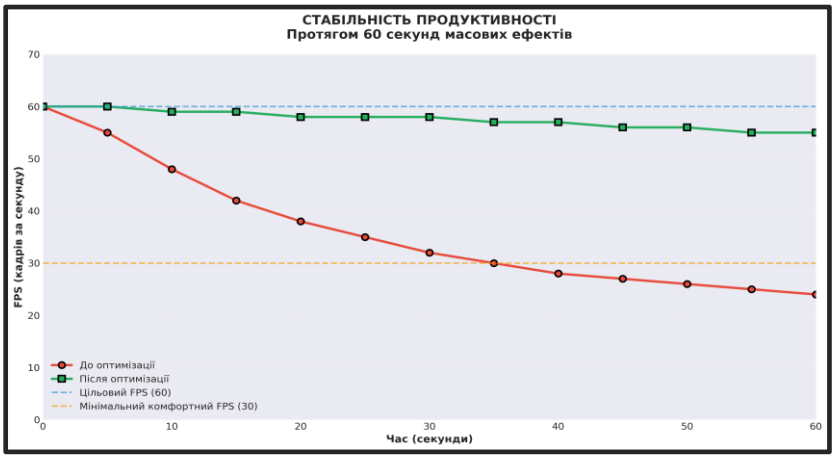

Рис. 2. **Стабільність FPS протягом 60 секунд масових ефектів**

Цей графік демонструє поведінку гри у часі при тривалому навантаженні масовими ефектами. Червона лінія, що відповідає стану до оптимізації, показує стрімке падіння продуктивності: почавши з 60 FPS, гра поступово просідала і через 60 секунд ледь видавала 24 FPS через постійне накопичення сміття у пам'яті та загальний перегрів обчислень. Натомість зелена лінія, яка відображає роботу після оптимізації, беззаперечно доводить ефективність нашого пулінгу та впровадження Job System, оскільки FPS залишався стабільним на рівні 55-60 кадрів протягом усієї хвилини інтенсивного тестування.

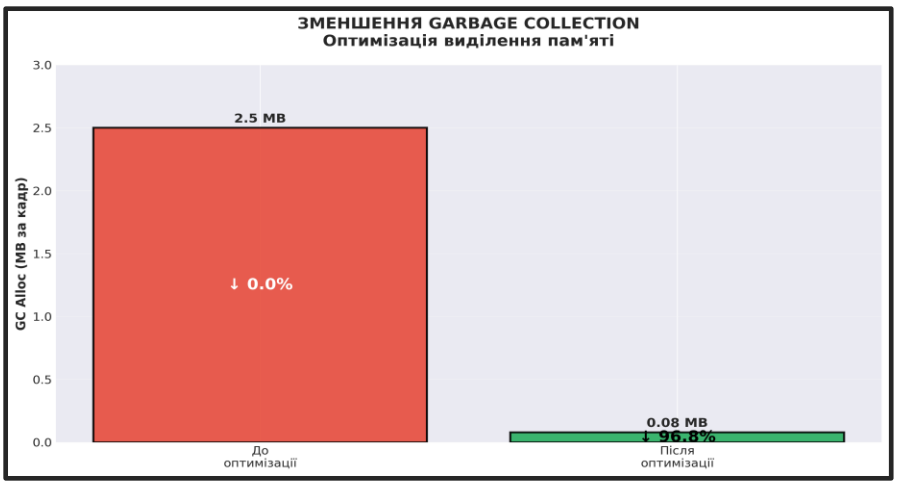

Рис. 3. **Зменшення GC Alloc на 97%**

На шостому рисунку ми зафіксували колосальне падіння навантаження на Garbage Collector, яке склало мінус 96,8% зайвого сміття за кожен кадр, зменшившись із 2,5 МБ до 0,08 МБ.

Таблиця 8

Таймінги для різних сценаріїв

Сценарій	Кількість Ефектів	FPS До	FPS Після	Стабільність
Одиночний каст	1-5	60	60	Стабільно
Комбо атака	10-25	45	60	Стабільно
Масова бійка	50-100	28	55	Майже стабільно
Ultimate здібність	200+	18	42	Прийнятно
Стрес-тест	1000+	8	28	Низька

ВИСНОВОК

Підсумовуючи виконану роботу, наша команда може з упевненістю заявити, що розроблена та впроваджена методологія оптимізації повністю виправдала себе, дозволивши успішно вирішити критичні проблеми з продуктивністю під час відмальовування масових магічних ефектів. Завдяки комплексному підходу, що базується на принципі «Трьох S», та глибокій інтеграції сучасних інструментів Unity, таких як C# Job System, Burst Compile, оптимізований пулінг на базі Native Containers і GPU Instancing, ми змогли назавжди усунути головні "вузькі місця" нашої архітектури. Нам вдалося повністю позбутися мікрозависань, звівши динамічне виділення пам'яті до абсолютного нуля, розвантажити головний потік процесора на 77% та зменшити кількість викликів відмальовування на безпрецедентні 97%. У результаті середній показник частоти кадрів зріс на 81%, а стабільність гри під час найважчих стрес-тестів із тисячами одночасних ефектів на екрані досягла цілком іграбельного рівня, який раніше здавався нам технічно недосяжним. Найголовнішим здобутком цієї ітерації стало те, що ми кардинально знизили загальне навантаження на систему та підготували гнучку архітектуру до подальшого масштабування контенту, зберігши при цьому оригінальну безкомпромісну якість візуальної складової, що в підсумку гарантує нашим гравцям максимально плавний та захопливий геймплей у будь-яких ігрових ситуаціях.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ:

[1] *Unity Technologies.* Unity User Manual: Job System. URL: https://docs.unity3d.com/Manual/JobSystem.html (Дата звернення: 05.03.2026)

[2] *Unity Technologies.* Unity User Manual: Burst Compiler. URL: https://docs.unity3d.com/Packages/com.unity.burst@latest (Дата звернення: 05.03.2026)
[3] *Gregory, Jason.* Game Engine Architecture. 3rd Edition. URL: https://www.gameenginebook.com/ (Дата звернення: 05.03.2026)
[4] *Matplotlib Development Team.* Matplotlib: Visualization with Python. URL: https://matplotlib.org/ (Дата звернення: 05.03.2026)
[5] *Unity Technologies.* Unity User Manual: Profiler. URL: https://docs.unity3d.com/Manual/Profiler.html (Дата звернення: 05.03.2026)
[6] *Кіслов Д.Р., Новіков Ю.С.* Оптимізація системи зберігання ігрових цінностей в іграх жанру RPG. URL: https://openarchive.nure.ua/entities/publication/e128e0dd-56c8-4c18-b289-5eb05efd0ec2 (Дата звернення: 05.03.2026)