Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження методів оптимізації та проектувальних рішень для створення складних ігрових програмних систем
(тема)

Виконав:
студент (ка) 2 курсу, групи ІПЗм-22-4

Косенко Б.А.
(прізвище, ініціали)

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

Керівник доц. Мар'їн С.О.
(посада, прізвище, ініціали)

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

2024 р.

Харківський національний університет радіоелектроніки

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

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

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Косенко Борису Андрійовичу
(прізвище, ім’я, по батькові)

1. Тема роботи «Дослідження методів оптимізації та проектувальних рішень для створення складних ігрових програмних систем.»
затверджена наказом університету від 29.03. 2024р. № 250 Ст
2. Термін подання студентом роботи до екзаменаційної комісії 18.06.2024
3. Вихідні дані до роботи
Аналіз методів оптимізації та їх впливу на продуктивність та якість ігрових систем , формування вимог, розробка та тестування оптимізованих модулів для різних компонентів ігрових систем, застосування новітніх технологій, таких як ECS (Entity Component System), DOTS (Data-Oriented Technology Stack) у Unity, порівняння продуктивності та якості різних підходів і методів оптимізації, загальний аналіз результатів дослідження, мова програмування C#, середовище розробки Rider, платформа Unity.
4. Перелік питань, що потрібно опрацювати в роботі
Огляд існуючих технічних рішень, практична реалізація проекту, тестування та аналіз результатів, фрагменти коду, графічні матеріали.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз предметної галузі та постановка задачі	30.03 – 14.04.24	*виконано*
2	Аналіз та вибір API для дослідження	15.04 – 24.04.24	*виконано*
3	Аналіз та моделювання предметної області	25.04 – 28.04.24	*виконано*
4	Планування експериментів	29.04 – 08.05.24	*виконано*
5	Програмна реалізація кожного з обраних для дослідження API	09.05 – 19.05.24	*виконано*
6	Експериментальні дослідження	20.05 – 25.05.24	*виконано*
7	Аналіз результатів експериментальних досліджень та розробка рекомендацій	25.05 – 28.05.24	*виконано*
8	Написання та оформлення статті та тез доповіді	28.05 – 31.05.24	*виконано*
9	Підготовка пояснювальної записки	01.06 – 09.05.24	*виконано*
10	Підготовка презентації та доповіді	10.06 – 12.06.24	*виконано*
11	Нормоконтроль	13.06 – 14.06.24	*виконано*
12	Рецензування	14.06 – 15.06.24	*виконано*
13	Занесення диплома в електронний архів	16.06.2024	*виконано*
14	Попередній захист	17.06.2024	*виконано*
15	Допуск до захисту у зав. кафедри	18.06.2024	*виконано*

Дата видачі завдання 30 березня 2024р.

Студент (ка) _____ Косенко Б.А.
(підпис)

Керівник роботи _____ доц. Мар'їн С.О.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка містить: 71 с., 21 рис., 3 табл., 26 джерел.

ОПТИМІЗАЦІЯ, ПРОЕКТУВАННЯ, ТЕСТУВАННЯ, BURST, DOTS, ECS, JOB SYSTEM, UNITY.

Об’єктом дослідження є складні ігрові програмні системи, що включають в себе велику кількість об’єктів та складних фізичних обчислень

Метою роботи є проведення дослідження та аналіз методів оптимізації і проектувальних рішень, спрямовані на підвищення продуктивності та ефективності складних ігрових програмних систем у середовищі Unity.

Методами розробки та проектування є аналіз існуючих методів оптимізації та проектувальних підходів в ігровій розробці, експериментальне порівняння ефективності різних підходів на основі відповідних критеріїв, таких як продуктивність, швидкодія та масштабованість, розробка інноваційних методів оптимізації та проектування, спрямованих на покращення продуктивності та ефективності складних ігрових програмних систем, тестування розроблених методів на практиці та аналіз отриманих результатів.

У результаті кваліфікаційної роботи було розроблено тестову сцену з великою велику кількістю об’єктів та обчисленням їх руху в кожному кадрі. За допомогою ECS, Job System та BURST цю сцену було оптимізовано для одночасно знаходження 300 000 об’єктів. Практична перевірка розробленого рішення на платформі Unity та оцінка ефективності на різних умовах та завданнях. Тестування проекта у двух реалізацій середовища виконання (Scripting Backend) для роботи з кодом мовою C#: традиційна Mono та сучасне рішення від Unity розробників – IL2CPP.

OPTIMIZATION, PROJECTING, TESTING, BURST, DOTS, ECS, JOB SYSTEM, UNITY.

The object of research is complex game software systems that include a large number of objects and complex physical calculations

The purpose of the work is to conduct research and analysis of optimization methods and design solutions aimed at increasing the productivity and efficiency of complex game software systems in the Unity environment.

Development and design methods include the analysis of existing optimization methods and design approaches in game development, experimental comparison of the effectiveness of different approaches based on relevant criteria, such as performance, speed and scalability, development of innovative optimization and design methods aimed at improving the performance and efficiency of complex game software systems, testing the developed methods in practice and analysis of the obtained results.

As a result of the qualification work, a test scene was developed with a large number of objects and the calculation of their movement in each frame. Using ECS, Job System, and BURST, this scene was optimized to find 300,000 objects simultaneously. Practical verification of the developed solution on the Unity platform and evaluation of its effectiveness under various conditions and tasks. Testing of the project in two implementations of the execution environment (Scripting Backend) for working with C# code: traditional Mono and a modern solution from Unity developers - IL2CPP.

Я, Косенко Борис Андрійович, студент(ка) гр. ІПЗм-22-4, здобувач вищої освіти на другому (магістерському) рівні кафедри «Програмна інженерія», заявляю: моя кваліфікаційна робота на тему «Дослідження методів оптимізації та проектувальних рішень для створення складних ігрових програмних систем», що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElArKhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений(на) з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в

допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

ЗМІСТ

ВСТУП

Ігрова індустрія, нинішній гігант із оборотом у 200 мільярдів доларів, можливо, є сьогодні одним із найважливіших та інноваційних секторів технологій. Його важливість для культури, соціальних мереж та розваг неможливо недооцінити. Термін «індустрія розваг» більше не використовується тільки для Голлівуду та кіноіндустрії, оскільки ігри тепер надають одну з найбільш захоплюючих форм розваг більш ніж трьом мільярдам людей по всьому світу. Сучасна ігрова індустрія, створена постійно зростаючими ігровими компаніями, розширює межі технологій, створюючи передові та захоплюючі розваги [1].

Зі збільшенням кількості гравців зростає і конкуренція у сфері розробки ігор, що у свою чергу сприяє появі нових ігрових програм. В наш час практично всі можливі ігрові механіки вже були створені кимось і реалізовані в одному з тисяч проектів. Однак, просто надати якісну гру вже не достатньо для завоювання користувальницької уваги.

Сьогодні гравців необхідно дивувати, пропонуючи їм щось унікальне та захоплююче. Це призводить до того, що ігри стають все більш масштабними та амбітними у своїх ідеях та реалізації. Збільшується складність як процесу розробки, а й подальшої підтримки таких гігантських ігрових проектів.

Розробка та підтримка AA та AAA проектів є викликом для багатьох розробників, особливо в умовах зростаючої конкуренції та вимогливих смаків гравців [2]. Відсутність якісної оптимізації може призвести до невдалого випуску гри на ринок.

З розвитком технологій апаратного забезпечення персональних комп’ютерів та мобільних пристроїв збільшуються попит на програмне забезпечення, яке могло б ефективно використовувати потужності сучасних графічних адаптерів [3].

Отже оптимізація продуктивності ігрового проекту — один із вирішальних факторів для масштабування, оскільки різні пристрої мають різні апаратні можливості та обмеження. Продуктивність означає, наскільки плавно та ефективно працює гра, без затримок, збоїв та розрядки акумулятора. Щоб

оптимізувати продуктивність гри необхідно використовувати різні методи, такі як скорочення кількості викликів малювання, оптимізація розміру та якості ресурсів, використання ефективних алгоритмів та структур даних, а також реалізація кешування та об'єднання в пули [4]. Також необхідно використовувати інструменти профілювання для вимірювання та аналізу продуктивності проекту на різних платформах та виявлення потенційних вузьких місць та проблем.

Підтримка таких проектів також є значущим аспектом. Після випуску розробники повинні надавати постійні оновлення, виправляти помилки та взагалі удосконалювати гру, щоб утримати та залучати аудиторію [5].

Швидкість впровадження нових механік та вирішення наявних помилок залежить від обраного комплексу проектних рішень. Це охоплює всі частини проекта: створення та налаштування ігрових об'єктів, їх управління, впровадження залежностей, підхід до розподілу бізнес-логіки, управління ігровим циклом та станами, складність ієрархії класів, абстракції ігрових об'єктів, отримання доступу до інших компонентів у об'єкті чи сцені, складність компонентів ігрових об'єктів, характеристики об'єктів у грі, модифікації, доступ до тимчасових та постійних даних, серіалізація даних та їх безпека, передача даних між об'єктами у сцені та між сценами, покриття тестами та інше.

При цьому вибір одних рішень може перекривати можливість використання інших. Наприклад, якщо бізнес-логіка буде в ігрових компонентах, прикріплених до об'єктів на сцені, то це викличе труднощі для покриття такої логіки автоматичними текстами. У такому разі доведеться розділити логіку та візуальну модель, що у свою чергу збільшить час, необхідний на впровадження нових механік, але зменшить ризик виникнення помилок.

Також при проектуванні потрібно розробляти рішення (правила, підходи, гайдлайни) не тільки для ігрової логікі, але і для всього, з чим проект пов'язаний: назва файлів і папок, структура зберігання ассетів, структура збірок, утиліти для розробки, тестування, робота із системою контролю версії, білд проекту та його деплой, робота зі сторами та виливка хотфіксів, моніторинг метрик, помилок та реагування на них [6].

Розробка додаткових сервісів та шарів абстракції, кастомних утиліт для редактора, покриття тестами може зайняти значний час розробки та окупити його лише у довгостроковій перспективі. Іноді цілі команди розробників займаюся проектуванням заради проектування там, де це зовсім не потрібно забуваючи, що в першу чергу вони розробляють гру. Саме тому важливо знаходити баланс між дійсно необхідними проектними рішеннями та тими, що на даний момент не несуть практичної користі.

Отже, грамотний підхід до проектування продукту може сприяти швидкому впровадженню нових механік, уникненню основної проблеми швидкої розробки - появі великої кількості багів. У такому випадку розробники матимуть можливість витратити більше часу на вдосконалення ігрових механік проекту. Оптимізація продуктивності має безпосередній вплив на фінансовий успіх проекту, оскільки вона визначає кількість пристроїв, на яких гра зможе оптимально функціонувати. Профайлінг проекту, хоча і є тривалим і складним процесом, є необхідною складовою розробки, яку необхідно враховувати впродовж всього проекту, а не залишати на останній момент.

Якісно розроблений оптимізаційний план із систематичним усуненням вузьких місць у проекті буде значно ефективнішим, ніж хаотичні виправлення. Це сприятиме вдосконаленню продукту та забезпечить його стабільну роботу, що відіграє ключову роль у забезпеченні задоволення користувачів та, в кінцевому рахунку, фінансовому успіху проекту.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз предметної галузі

Обсяг світового ринку відеоігор оцінювався в 248,52 мільярда доларів США в 2023 році і, за прогнозами, досягне 560,11 мільярда доларів США до 2030 року, при цьому середньорічний темп зростання становитиме 12,81% з 2022 по 2030 рік.

Одним із найпопулярніших видів розваг в даний час є відеоігри, які дають гравцям можливість поринути в ігровий процес. За оцінками Управління міжнародної торгівлі Міністерства торгівлі США, світовий ринок засобів масової інформації та розваг досяг 1,9 трильйона доларів у 2016 році, а наступного року ця цифра збільшилася до 2 трильйонів доларів. Поряд із розширенням світового медіа- та розважального бізнесу очікується очікуване зростання світової індустрії відеоігор. Очікується, що розвиток технологій та постійні інновації як в апаратному, так і в програмному забезпеченні продовжуватимуть сприяти зростанню ринку протягом прогнозованого періоду. Ці досягнення спрямовані на покращення рендерингу графіки у реальному часі [7]. Очікується, що такі фактори як повсюдна доступність смартфонів, зростання рівня проникнення Інтернету та зручність завантаження ігор з Інтернету сприятимуть розширенню галузі.

Масові розраховані на багато користувачів онлайн-ігри (MMO), безкоштовні ігри (F2P) і розраховані на багато користувачів ігри стають все більш популярними в результаті збільшення числа людей, які воліють грати в відеоігри онлайн. Очікується, що ця модель збережеться протягом кількох наступних років [8]. В результаті творці відеоігор приділяють більше уваги ефективності та сумісності обладнання. У той же час зміна споживчих переваг та зростання рівня дискреційних доходів у всьому світі сприяють широкомасштабному впровадженню передових ігрових консолей, оснащених складними функціями, такими як запис та обмін, а також кросплатформовий ігровий процес [9]. Ці фактори сприяють поширенню сучасних ігрових консолей, що оснащені складними функціями [10].

Ігрова індустрія продовжує демонструвати вражаюче зростання завдяки кільком ключовим чинникам. Одним із головних стимулів є постійне вдосконалення графіки, звукових ефектів та апаратних можливостей, обіцяючи гравцям більш реалістичний та захоплюючий ігровий досвід. Еволюція ігрової механіки, поява нових жанрів та інноваційних методів оповідання також відіграють важливу роль, захоплюючи та тримаючи увагу гравців протягом багатьох годин.

Не можна недооцінювати вплив розрахованих на багато користувачів режимів і позитивного досвіду, який гравці діляться з друзями і сім'єю. Відгуки та рекомендації як особисто, так і в онлайн-спільнотах стають потужним інструментом залучення нових учасників [11]. Разом з тим, доступність ігор на мобільних пристроях, консолях та ПК зробила процес гри ще зручнішим, дозволяючи гравцям насолоджуватися улюбленими іграми у будь-який час та в будь-якому місці.

Також еволюція цифрового розповсюдження та можливості розрахованих на багато користувачів мереж створили плідний ґрунт для міжнародних змагань та кіберспорту. Професійні ліги, турніри та інші події не тільки виявляють майстерність гравців, але й приваблюють величезну кількість глядачів, захоплених духом суперництва та прагнуть побачити визначні виступи віртуальних атлетів. Кіберспорт, який став своєрідною аналогією традиційному спорту, продовжує завойовувати популярність і визнання, стверджуючи свою роль у сучасній культурі розваг та змагань [12].

В цілому, ігрова індустрія на даний момент залишається однією з найбільш динамічно розвиваються і впливових галузей у світі розваг.

1.2 Застосування Unity платформи

Розквіт Unity приніс істотні зміни в ігрову індустрію. Спочатку орієнтований на інді-розробників, цей інтуїтивно зрозумілий двигун став каталізатором великого зростання в галузі. Його доступний робочий процес та можливість багатоплатформної публікації, вартість яких на сьогоднішній день

оцінюється більш ніж у 6 мільярдів доларів, лідирують у секторі, захоплюючи 60% ринку серед розробників ігор, які прагнуть використати його революційні можливості в реальному часі для графічного рендерингу.

Подальше розширення можливостей Unity, що забезпечує високоякісну візуалізацію та покращену оптимізацію продуктивності, привернула увагу великих студій розробки ігор. Вони, які раніше дотримувалися пропрієтарних двигунів або Unreal Engine, тепер активно використовують Unity, визнаючи його переваги, такі як швидкий час ітерації у виробничих циклах, що дозволяє заощадити ресурси для додаткової розробки [13]. Проекти з великими бюджетами та амбітними масштабами, такі як Electronic Arts, Microsoft та навіть студії Activision, впроваджують Unity у свої проекти.

Очевидно, що Unity стає все більш універсальним двигуном для розробки ігор на ПК, консолях та мобільних пристроях. Популярність Unity серед інді-розробників, які набувають популярності завдяки своїм іграм і конкурують із визнаними гігантами індустрії, підтверджує його домінуючу позицію [14]. Це відкриває перспективу, в якій більшість ігрового контенту буде розроблено з використанням Unity для просування інновацій. Рух став настільки просунутим, що він здатний підтримувати як проривні інді-ігри, так і великомасштабні ААА-проекти, привертаючи увагу студій розробки ігор до його потенціалу.

Успіх Unity у сфері відеоігор привертає увагу й інших галузей, що працюють із 3D-технологіями, таких як кіноіндустрія, автомобільний дизайн, архітектура та багато інших [15]. У цих сферах 3D-інструменти Unity в реальному часі виявляються дуже корисними для попереднього перегляду роботи.

Кінематографісти все частіше використовують Unity для компонування сцен із цифровими персонажами, поєднуючи їх із реальними кадрами. Це допомагає візуалізувати сцени та розробляти кадри перед зйомкою, а також створювати прототипи візуальних ефектів, що прискорює процес просування ідей, не вимагаючи очікування результатів тестового рендеру.

Unity також став незамінним інструментом у виробництві великих фільмів, включаючи фільми із франшизи «Зоряні війни». Було також створено

короткометражні фільми та експериментальні проекти, присвячені розширенню можливостей Unity у кіновиробництві. Багато аніматорів та студій використовують інструменти ігрового движка Unity для створення кінематографічних історій через 3D-анімацію з елементами інтерактивності[16].

В області автомобільного дизайну Unity використовується для швидкого тестування різних варіантів зовнішнього вигляду та матеріалів, а архітектурні фірми використовують його для створення тривимірних посібників з будівель для турів клієнтів. Розробники продуктів також тестують свої проекти у доповненій реальності за допомогою мобільних додатків Unity.

Від автосалонів віртуальної реальності до тест-драйвів віртуальної реальності та інтерактивних екранних візуалізаторів Unity дає можливість покращити якість обслуговування клієнтів та оптимізувати внутрішні робочі процеси в різних галузях [17]. 3D-движок у реальному часі допомагає швидко розробляти прототипи, тестувати різні конфігурації та представляти реалістичні попередні зображення, що значно заощаджує час та ресурси.

1.3 Проблематика розробки у Unity середовищі

Велика перевага ігрової студії Unity – швидкість. Створення 3D-відеоігор з великою кількістю графіки, персонажів та деталей складно. Великим студіям може знадобитися багато років, щоб створити одну гру. Але в движку Unity є інструменти, які дозволяють дизайнерам швидко тестувати ідеї та вносити зміни. Замість того, щоб чекати нових збірок, розробники можуть оновитися прямо всередині движка Unity, щоб побачити, що працює найкраще. Ця творча гнучкість прискорює створення великих ігор та надає розробникам, гейм-дизайнерам та незалежним ігровим студіям інструменти та функції, що дозволяють їм втілити свою творчу уяву у реальність [18].

На жаль, як потужний інструмент, він також має свої проблеми. Часто можно зіткнутися з різними неприємностями: від збоїв Unity та виправлень помилок до ширших проблем при розробці ігор.

Оптимізація проекту Unity – це складне завдання, що включає використання ресурсів, питання ефективності сценаріїв та контроль ігрових ресурсів. Досягнення продуктивності гри в Unity вимагає поєднання методів оптимізації та певної тактики. Основні області - це оптимізація дозволу ресурсів та розмірів файлів, скорочення кількості викликів малювання за рахунок пакетної обробки та використання вбудованих інструментів Unity, таких як Profiler, для виявлення вузьких місць у продуктивності.

Бібліотеки ресурсів з великою кількістю ресурсів є серйозною проблемою при розробці ігор, пов'язану з часом завантаження та іншими проблемами продуктивності. Управління цими ресурсами має вирішальне значення для підтримки рівня продуктивності та забезпечення швидких та плавних ігор [19]. Такі методи, як використання пакетів ресурсів для завантаження на вимогу, використання оптимізованих форматів файлів для текстур і моделей, точне налаштування параметрів імпорту ресурсів значно зменшують обсяг пам'яті і збільшують час завантаження.

Також треба звернути уваги на ШІ, який дуже активно використовується в великих проектах. Одним з найчастіше використовуваних алгоритмів для NPC є пошук шляху. Пошук шляху або pathfinder — визначення комп’ютером найоптимальнішого маршруту між двома точками. З ним часто можна зіткнутися при розробці ігор, наприклад, якщо в них є вороги, що вільно пересуваються. Потрібно не просто прагнемо знайти найкоротшу відстань, а також необхідно враховувати і тривалість руху. Для пошуку цього шляху можна використовувати алгоритм пошуку за графом, який застосовується, якщо карта є графом. A* часто використовується як алгоритм пошуку за графом [20]. Даний алгоритм досить вимогливий до ресурсів пристрою користувача та його оптимізація є першорядним завданням у будь-якому великому проекті.

Ще одна проблема – це виділення пам'яті. Без управління пам'яттю можуть виникнути проблемам з продуктивністю та забезпеченню безперебійної роботи проектів Unity на будь-якій цільовій платформі. Для вирішення цієї проблеми використовують пули які дозволять уникнути непотрібних виділень у методах, що

часто викликаються, які зменшують вплив складання сміття, об'єднує об'єкти в пули, щоб мінімізувати накладні витрати на створення і знищення, а також керує переходами між сценами для вивантаження непізнаних ресурсів[21].

Хоча принципи Unity прості для розуміння, дуже легко припуститися фундаментальних помилок. Такі помилки можуть сповільнити процес розробки, особливо при переході від етапу початкового прототипу до фінальної версії.

1.4 Постановка задачі

Нашою задачею є пошук та аналіз рішення, яке дозволить зручно масштабувати великий проект на платформі Unity без значної втрати продуктивності. Це рішення повинно відповідати вимогам до складних ігрових програмних систем, а саме вміти обробляти велику кількість одночасних об'єктів та складних фізичних обчислень. Це важливо, оскільки висока продуктивність і ефективність системи у процесі масштабування забезпечить плавну роботу великого проекту та задовільнення потреб користувачів.

Нам необхідно зібрати та аналізувати дані щодо різних можливих рішень та їх впливу на ефективність проекту. Проаналізувати ці дані та провести оцінку ефективності, швидкодії та масштабованості рішення за різних умов та інтерпретувати результати.

Це дослідження спрямовано на використання інструментів з високим рівнем контролю та детермінізму з метою забезпечення максимальної швидкодії та ефективності проекту. Використання таких інструментів дозволить нам забезпечити стабільну роботу системи навіть при великій кількості одночасних обчислень та об'єктів.

Нами будуть розглянуті основні «вузькі» місця проектів та способи їх вирішення. Фактори, що впливають на впливають на продуктивність та масштабованість системи, будуть ретельно проаналізовані з метою знаходження оптимальних рішень для покращення роботи системи та підвищення її ефективності.

Глибокий аналіз цих аспектів сприяє сприяє розумінню потенційних проблем та визначенню найкращих шляхів їх вирішення, що є ключовим для успішного розроблення та масштабування складних ігрових програмних систем на платформі Unity.

2 АНАЛІЗ ІНСТРУМЕНТІВ У ДОСЛІДЖЕННІ

2.1 DOTS

Unity DOTS (Data-Oriented Technology Stack) – це стек технологій, представлена Unity Technologies, яка покликана змінити спосіб створення ігор та інших інтерактивних програм. DOTS пропонує новий підхід до розробки, заснований на організації даних та паралельному виконанні, щоб забезпечити високу продуктивність та масштабованість.

По суті DOTS реалізує архітектурний шаблон Entity Component System (ECS). Це нова архітектура, яка відрізняється від класичної об'єктно-орієнтованої моделі програмування. ECS поділяє дані та логіку, що спрощує паралельне виконання та оптимізацію роботи додатків.

Об'єктно-орієнтоване програмування (ООП) несе у собі значну спадщину таких шаблонів, як успадкування та інкапсуляція, і навіть досвідчені програмісти можуть робити архітектурні помилки у розпал розробки, що призводить до рефакторингу або заплутаної логіки у довгострокових проектах [22].

Навпаки, ECS забезпечує просту та інтуїтивно зрозумілу архітектуру. Все природним чином розпадається на ізольовані компоненти та системи, що спрощує розуміння та розробку з використанням такого підходу; навіть розробники-початківці швидко схоплюють цей підхід з мінімальними помилками.

У ECS замість складних ієрархій успадкування створюються ізольовані компоненти та системи поведінки [23]. Ці компоненти та системи можна легко додавати або видаляти, що дозволяє гнучко змінювати характеристики та поведінку об'єктів – такий підхід значно підвищує можливість повторного використання коду.

Ще однією ключовою перевагою ECS є оптимізація продуктивності. ECS дані зберігаються в пам'яті безперервним і оптимізованим чином, при цьому ідентичні типи даних розташовуються близько один до одного. Це оптимізує доступ до даних, зменшує промахи в кеші та покращує шаблони доступу до пам'яті. Більш того, системи, що складаються з окремих блоків даних, легше

розпаралелювати між різними процесами, що призводить до виняткового збільшення продуктивності порівняно з традиційними підходами[24].

Застосування Unity DOTS дозволить нам створити високопродуктивний проект, здатний обробляти великі обсяги даних та взаємодіяти з ними в реальному часі. Він особливо корисний для створення ігор з великою кількістю об'єктів на екрані або вимогливих до продуктивності програм.

2.2 Unity DOTS Technology Stack: Компоненти та можливості

Ядром DOTS є пакет Entities, який полегшує перехід від звичних MonoBehaviours та GameObjects до підходу Entity Component System. Цей пакет формує основу розробки з урахуванням DOTS.

Пакет Unity Physics представляє новий підхід до роботи з фізикою в іграх, забезпечуючи неймовірну швидкість за рахунок розпаралелених обчислень.

Крім того, пакет Havok Physics for Unity забезпечує інтеграцію із сучасним двигуном Havok Physics. Цей двигун забезпечує високопродуктивне виявлення зіткнень і фізичну симуляцію, забезпечуючи підтримку таких популярних ігор, як The Legend of Zelda: Breath of the Wild, Doom Eternal, Death Stranding, Mortal Kombat 11 та інших.

Пакет Entities Graphics орієнтований на рендеринг у DOTS. Він забезпечує ефективний збір даних рендерингу та безперешкодно працює з існуючими конвеєрами рендерингу, такими як універсальний конвеєр рендерингу (URP) або конвеєр рендерингу високої чіткості (HDRP) [25].

Деякі технології, тісно пов'язані з DOTS, можуть використовуватися в рамках DOTS та за її межами. Пакет Job System надає зручний спосіб написання коду з паралельними обчисленнями. Він заснований на поділі роботи на невеликі фрагменти, які називають завданнями, які виконують обчислення на власних даних. Система завдань рівномірно розподіляє ці завдання потоками для ефективного виконання. Для забезпечення безпеки коду система завдань підтримує обробку типів даних [26].

Ще одним важливим пакетом є Burst Compiler, який можна використовувати із системою завдань для генерації високооптимізованого коду. Хоча компілятор Burst має певні обмеження використання коду, він забезпечує значний приріст продуктивності.

2.3 Profiler

Unity Profiler надає нам потужний інструмент для аналізу продуктивності та моніторингу ключових аспектів проекту. Він включає кілька важливих панелей, таких як CPU Usage, яка відображає відсоткове використання процесора за різними категоріями, включаючи рендеринг, сценарії (скрипти), фізику та інші [27]. GPU Usage дозволяє відстежувати завантаження графічного процесора (GPU), включаючи час рендерингу, пропускну здатність та використання різних графічних API (див. рис. 1).

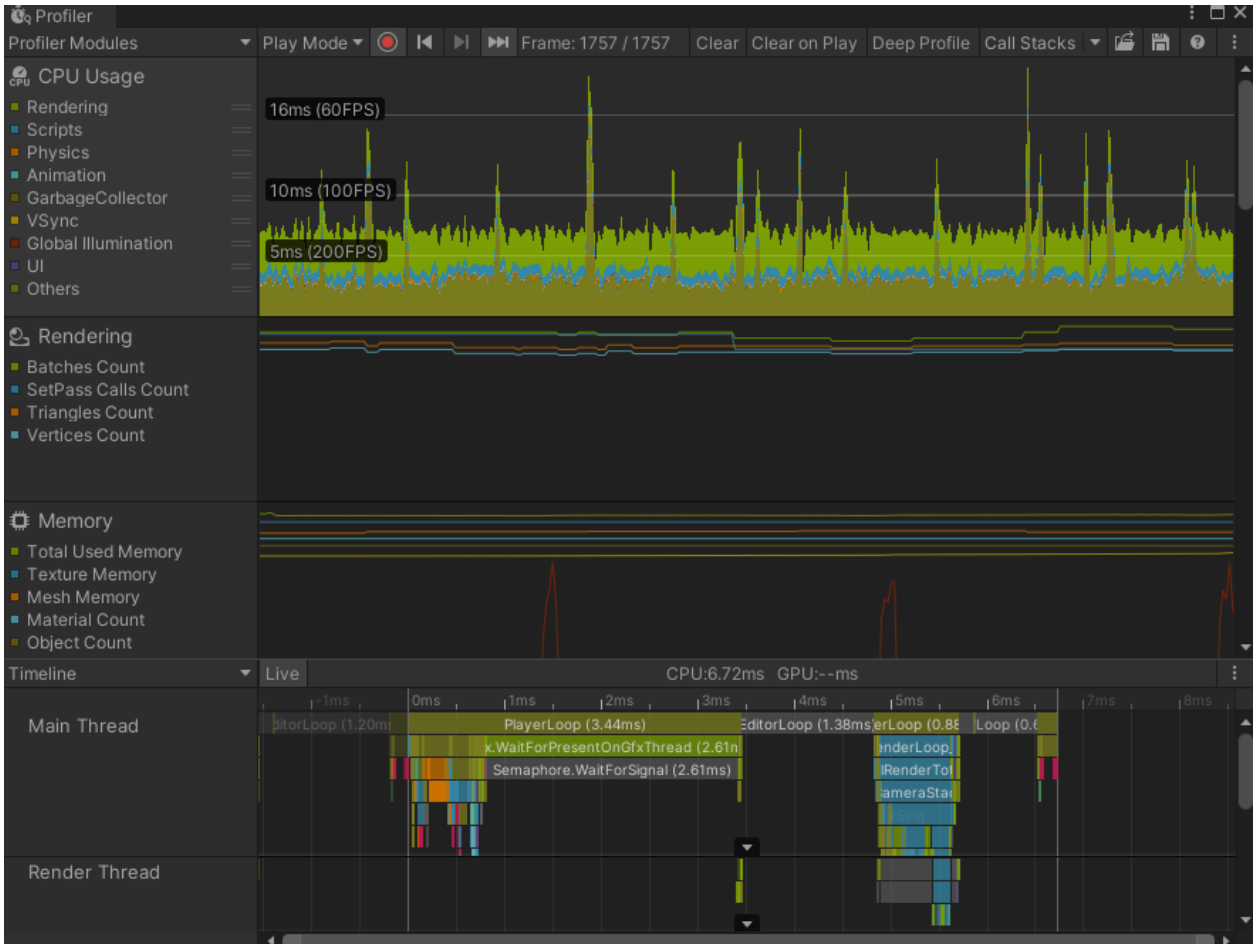

Рисунок 1 – Візуальний вигляд профайлеру (виконано самостійно)

Memory Allocation представляє інформацію про виділену пам'ять, частоту та обсяг алокацій в реальному часі. Потім Rendering Statistics моніторить кількість трикутників, відмальованих кадром, та інші статистичні дані про процес рендерингу. Крім того, панель Audio дає інформацію про завантаження аудіосистеми Unity, включаючи кількість аудіоканалів та час обробки звукових ефектів [28]. Нарешті Profiler Markers дозволяє розробникам додавати власні маркери в коді для більш детального аналізу та відстеження продуктивності конкретних ділянок програми. Це дозволяє переглядати характеристики продуктивності, унікальні для вашої програми, в контексті безпосередньо у вікні профільника.

Цей комплексний інструментарій допоможе нам виявляти вузькі місця у проекту та оптимізувати їх для досягнення кращої продуктивності та якості роботи.

3 РОЗРОБКА ТЕХНІЧНОГО РІШЕННЯ. ТЕОРЕТИЧНЕ ПОДАННЯ

3.1 Методи оптимізації

У нашому дослідженні ми розглянемо та порівняємо кілька методів підвищення продуктивності: використання загального менеджера частинок, кешування, багатопоточність, GPU instancing та Entity Component System (ECS). Важливо, щоб симуляція великої кількості частинок виконувалася плавно та ефективно, не створюючи навантаження на процесор та не уповільнюючи роботу основного потоку.

Використання загального менеджера частинок дозволить централізовано керувати всіма частинками, оптимізуючи їх оновлення та рендеринг, що знизить накладні витрати та покращує контроль за станом частинок. Кешування забезпечить швидкий доступ до даних, що часто використовуються, що значно прискорить виконання операцій і оптимізує використання пам'яті. Багатопотоковість розподілить обчислення між декількома потоками, ефективно використовуючи ресурси процесора та знижуючи навантаження на основний потік, звільняючи його для виконання інших важливих завдань, таких як обробка введення користувача та рендеринг.

Застосування GPU instancing дозволить нам ефективно малювати багато об'єктів за один виклик, знижуючи навантаження на процесор і шину пам'яті, що особливо корисно при роботі з тисячами частинок. Це оптимізує процес рендерингу та підтримує високу продуктивність. Використання ECS забезпечить високу масштабованість, дозволяючи легко керувати та оновлювати велику кількість об'єктів. Компактне зберігання даних та їх організація для ефективного доступу знизить накладні витрати. Інтеграція із Burst Compiler дозволить генерувати високоефективний машинний код, досягаючи максимальної продуктивності.

Наше дослідження спрямовано на демонстрацію переваг кожного з цих підходів та оцінку їхнього впливу на продуктивність симуляції частинок.

3.2 Модель для тестування

Для наочної демонстрації переваг різних підходів до оптимізації проведемо симуляцію великої кількості частинок зі складними обчисленнями руху у кожному кадрі.

Реалізуємо ефект розльоту частинок при наближенні курсору та їх подальшому поверненні у вихідне положення. Для опису поведінки частинок нам достатньо виділити три принципи: прагнення скоротити відстань до початкової точки, прагнення відлетіти від курсору миші та загасання руху. Нам не потрібні точні фізичні взаємодії та формули, потрібні лише загальні принципи, за якими частка поводитиметься заданим чином. Для спрощення ми не враховуватимемо масу частки.

Для того, щоб частка прагнула повернутися у вихідне становище, ми можемо використовувати закон Гука: сила, спрямована до початкової позиції, буде лінійно пропорційна відстані до неї. Якщо частка відлітає вдвічі далі від початкової позиції, сила, що притягує її назад, буде вдвічі сильнішою, все просто.

Частинки повинні якось взаємодіяти з курсором та дозволяти собі відходити від вихідного становища. Ми використовуємо гравітацію зі зворотним знаком: частинки відштовхуються силою, обернено пропорційною квадрату відстані між положенням курсору миші та поточною позицією частинки, спрямованої від курсору до частки. Формула має вигляд: $F = -C/r^2$, де C - деяка константа, що регулює взаємодію.

Якщо ми обмежимося цими двома формулами, то частка коливатиметься нескінченно після встановлення початкової амплітуди, оскільки в рамках цієї моделі енергія не втрачається. Для імітації згасання ми можемо застосувати силу в'язкого опору згідно із законом Стокса, яка буде лінійно пропорційна швидкості руху частинки та спрямована у протилежному напрямку. При цьому ми візьмемо до уваги обмеження, що накладаються на застосування цієї формули, оскільки нас цікавить лише принципова поведінка, а не абсолютна точність фізичної взаємодії.

У результаті ми отримуємо формулу сили, що діє на частку в довільний час (формула 3.1):

$$\overrightarrow{F_s} = C_a * (\vec{x} - \vec{x}_0) - \frac{C_r}{\|\vec{x} - \vec{x}_r\|^2} * \frac{\vec{x} - \vec{x}_r}{\|\vec{x} - \vec{x}_r\|} - C_d * \vec{v} \quad (3.1)$$

де x, v — поточна позиція та швидкість частки,

x_0 — початкове положення,

x_r — позиція курсору,

C_a, C_r, C_d — коефіцієнти тяжіння, відштовхування та згасання відповідно.

Тепер теоретично описавши поведінку частинок ми можемо розпочати практичну реалізацію проекту.

4 РОЗРОБКА ТЕХНІЧНОГО РІШЕННЯ. ПРОГРАМУВАННЯ ПОВЕДІНКИ

4.1 MonoBehaviour

Реалізуємо симуляцію частинок з використанням лише компонентів MonoBehaviour, без менеджерів та паралельних обчислень. У цьому підході кожна частка буде представлена окремим об'єктом із власним скриптом MonoBehaviour, який керуватиме її поведінкою.

Створимо префаб для частинки, що складається з об'єкта з компонентом SpriteRenderer для візуалізації. Додаємо скрипт ParticleBehavior до префабу частки. У цьому скрипті реалізуємо всю логіку поведінки частинок (див. рис. 2): повернення до вихідної позиції, відштовхування від курсору та загасання руху. На сцені створюємо безліч екземплярів префабу частинок та розставляємо їх у випадкових позиціях (див.рис.2).

```
Vector3 mousePosition = mainCamera.ScreenToWorldPoint(Input.mousePosition);
mousePosition.z = 0;

Vector3 toInitial = initialPosition - transform.position;
Vector3 force = toInitial * returnForce;

Vector3 fromMouse = transform.position - mousePosition;
float distanceToMouse = fromMouse.magnitude;
if (distanceToMouse > 0)
{
    force += fromMouse.normalized * (repulsionForce / (distanceToMouse * distanceToMouse));
}

velocity += force * Time.deltaTime;
velocity *= damping;
transform.position += velocity * Time.deltaTime;
```

Рисунок 2 – Логіка поведінки частинки (виконано самостійно)

Як результат, кожен об'єкт MonoBehaviour додає накладні витрати на пам'ять та продуктивність. При великої кількісті частинок це призводить до значного падіння продуктивності. Виклик методу Update для кожного об'єкта в кожному кадрі є неефективним, особливо при роботі з сотнями або тисячами частинок. Так само в даному випадку дані частинок розподілені по багатьох об'єктах, що ускладнює ефективне використання кеш-пам'яті процесора.

Без використання Unity Jobs System або інших методів паралельного виконання, всі обчислення виконуються послідовно на основному потоці, що обмежує продуктивність.

Використання тільки MonoBehaviour для реалізації симуляції частинок є неефективним підходом до завдань, що потребують обробки великої кількості об'єктів. Цей метод призводить до значних накладних витрат на управління об'єктами та обчислення, що знижує загальну продуктивність та масштабованість системи. Використання більш сучасних підходів, таких як ECS, Jobs System та Burst Compiler, дозволить значно покращити продуктивність та ефективніше використовувати ресурси.

4.2 Сценарій менеджера

Використання загального менеджера частинок має кілька значних переваг, особливо у контексті симуляції великої кількості частинок із загальною логікою поведінки. Насамперед це спрощення логіки. Замість того, щоб кожна частка мала власні скрипти з однаковою або схожою логікою, ми маємо один скрипт, який керує всіма частинками. Це полегшує код і робить його більш зрозумілим.

Так само все налаштування параметрів (наприклад, сила повернення, сила відштовхування та коефіцієнт загасання) знаходиться в одному місці. Це полегшує тестування та налаштування параметрів без необхідності змінювати кожен об'єкт окремо.

Щодо оптимізації це зниження навантаження на систему та зменшення витрат пам'яті. Керування великою кількістю об'єктів через один скрипт знижує навантаження на процесор, оскільки зменшує кількість викликів методів Update() та інших функцій одночасно. Використання одного менеджера дозволяє уникнути дублювання даних та методів, що заощаджує пам'ять [29].

Даний підхід спрощує налагодження та тестування, а також дає нам гнучкість і масштабованість. Всі можливі помилки та баги концентруються в одному скрипті, що спрощує їх пошук та виправлення, а додавання нових

частинок до сцени стає простим, тому що потрібно лише створити новий об'єкт та додати його до списку частинок у менеджері (див. рис. 3).

```
Vector3 mousePosition = Camera.main.ScreenToWorldPoint(Input.mousePosition);
mousePosition.z = 0;

foreach (var particle in particles)
{
    Vector3 force = Vector3.zero;

    Vector3 toInitial = particle.InitialPosition - particle.Object.transform.position;
    force += toInitial * returnForce;

    Vector3 fromMouse = particle.Object.transform.position - mousePosition;
    float distanceToMouse = fromMouse.magnitude;
    if (distanceToMouse > 0)
    {
        force += fromMouse.normalized * (repulsionForce / Mathf.Pow( f: distanceToMouse,  p: 2));
    }

    particle.Velocity += force * Time.deltaTime;

    particle.Velocity *= damping;

    particle.Object.transform.position += particle.Velocity * Time.deltaTime;
}
```

Рисунок 3 – Винесення логіки поведінки частки під управління менеджера (виконано самостійно)

Таким чином, використання спільного менеджера для частинок в Unity дозволяє створити ефективну систему, що легко налаштовується і розширюється, яка спрощує управління частинками та їх поведінкою.

4.3 Кешування

Використання кешування при вирішенні задачі симуляції частинок у Unity надає нам ще кілька значних переваг.

При симуляції частинок багато обчислень виконуються багаторазово, наприклад, обчислення сил, що діють на кожну частинку, та оновлення їх позицій. Кешування проміжних результатів або даних, що часто використовуються, може істотно знизити навантаження на процесор.

Приклад кешованих даних для нашої системи:

```
public GameObject particlePrefab;
public int particleCount = 100;
public float returnForce = 0.1f;
public float repulsionForce = 100.0f;
public float damping = 0.98f;

private List<Particle> particles = new List<Particle>();
private Vector3 mousePosition;
private Vector3[] forces;
```

Також ми мінімізуємо повторні обчислення. Наприклад, відстань між часткою та курсором може кешуватися на кожному кадрі, щоб уникнути повторних обчислень, особливо якщо вона використовується в кількох місцях.

З точки зору пам'яті ми знижуємо аллокацію і покращуємо ефективність використання кеш-пам'яті процесора. Постійне створення та видалення об'єктів (наприклад, тимчасових векторів) може викликати фрагментацію пам'яті та уповільнити роботу програми. Кешування об'єктів, таких як вектори або масиви, допомагає уникнути зайвих алокацій та звільнень пам'яті (див. рис. 4). Доступ до даних, які зберігаються поруч у пам'яті (наприклад, у масиві), швидше, ніж доступ до розкиданих даних. Кешування допомагає тримати дані в пам'яті процесора, що прискорює їхню обробку. Як результат ми отримуємо більш стабільну частоту кадрів і збігаємо стрибків продуктивності, склежуючи піки навантаження на процесор.

```
forces = new Vector3[particleCount];
for (int i = 0; i < particleCount; i++)
{
    GameObject particleObj = Instantiate(particlePrefab,
        position: Random.insideUnitCircle * 5, Quaternion.identity);

    particles.Add( item: new Particle(particleObj, particleObj.transform.position));
}
```

Рисунок 4 – Кешування об’єктів (виконано самостійно)

Кешування є важливим методом оптимізації задач, що вимагають інтенсивних обчислень, таких як симуляція частинок. Воно допомагає знизити навантаження на процесор, покращити використання пам'яті та забезпечити

стабільну та передбачувану продуктивність, що особливо важливо для підтримки високого FPS та загальної якості роботи програми.

4.4 GPU instancing

GPU instancing дозволяє малювати безліч об'єктів з однаковим мішом і матеріалом в одному виклику змальовування, що знижує навантаження на CPU і дозволяє GPU більш ефективно обробляти графіку. У контексті симуляції частинок це може бути особливо корисним. Додамо в наші дані поля, які відповідають за візуальний вигляд частинок:

```
private List<Particle> particles = new List<Particle>();
private Matrix4x4[] matrices;
private Vector4[] colors;
private MaterialPropertyBlock propertyBlock;
private Mesh particleMesh;
private Material particleMaterial;
private Vector3 mousePosition;
private Vector3[] forces;
```

В першу чергу ми знизимо кількість викликів малювання. GPU instancing дозволяє малювати безліч об'єктів в одному виклику, що знижує overhead на CPU та зменшує кількість draw calls. GPU може ефективно обробляти велику кількість об'єктів паралельно, що прискорює рендеринг сцени. Також CPU не потрібно виконувати обчислення трансформацій для кожного об'єкта окремо. Це дозволяє звільнити ресурси інших обчислювальних завдань.

Використання GPU instancing дозволить легко масштабувати кількість частинок, що ідеально підходить для складних візуальних ефектів та симуляцій (див. рис. 5).

Використання GPU instancing у Unity для симуляції частинок дозволяє значно підвищити продуктивність, особливо під час роботи з великою кількістю об'єктів. Цей підхід зменшує навантаження на CPU, ефективно використовує можливості GPU та забезпечує масштабованість та стабільність продуктивності, що робить його ідеальним для створення складних та візуально насичених ефектів.

```
for (int i = 0; i < particles.Count; i++)
{
    Particle particle = particles[i];
    Vector3 force = Vector3.zero;

    Vector3 toInitial = particle.InitialPosition - particle.Object.transform.position;
    force += toInitial * returnForce;

    Vector3 fromMouse = particle.Object.transform.position - mousePosition;
    float distanceToMouse = fromMouse.magnitude;
    if (distanceToMouse > 0)
    {
        force += fromMouse.normalized * (repulsionForce / Mathf.Pow( f: distanceToMouse,  p: 2));
    }

    forces[i] = force;
}

for (int i = 0; i < particles.Count; i++)
{
    Particle particle = particles[i];
    particle.Velocity += forces[i] * Time.deltaTime;
    particle.Velocity *= damping;
    particle.Object.transform.position += particle.Velocity * Time.deltaTime;

    matrices[i] = particle.Object.transform.localToWorldMatrix;
}

propertyBlock.SetVectorArray( name: "_Color", colors);
Graphics.DrawMeshInstanced(particleMesh,  submeshIndex: 0, particleMaterial, matrices, particleCount, propertyBlock);
```

Рисунок 5 – Застосуємо метод оптимізації викликів малювання (виконано самостійно)

4.5 Jobs System

Використання Unity Jobs System та Burst Compiler разом із GPU instancing може значно підвищити продуктивність нашої симуляції частинок. Unity Jobs System дозволяє виконувати обчислювальні завдання паралельно ефективно використовуючи багатоядерні процесори, а Burst Compiler забезпечує високу продуктивність, генеруючи високоефективний машинний код.

Оновлення позицій, швидкостей та сил, що діють на частинки тепер виконується паралельно для кожної частки, що значно прискорює ці обчислення. Основний потік (main thread) звільняється від цих важких обчислень, що дозволяє йому швидше обробляти інші завдання, такі як введення користувача і рендеринг (див. рис. 6).

```
positions = new NativeArray<Vector3>(particleCount, Allocator.Persistent);
velocities = new NativeArray<Vector3>(particleCount, Allocator.Persistent);
initialPositions = new NativeArray<Vector3>(particleCount, Allocator.Persistent);
matrices = new NativeArray<Matrix4x4>(particleCount, Allocator.Persistent);
propertyBlock = new MaterialPropertyBlock();
```

Рисунок 6 – Використання NativeArray дозволить використовувати дані у багатопотоковому коді (виконано самостійно)

Підготовка даних для GPU instancing (матриць трансформації) також виконується паралельно, що прискорює процес підготовки та дозволяє швидше передати дані на GPU. За рахунок прискорення обчислень та більш ефективного використання ресурсів процесора загальний час кадру теж скорочується, що призводить до збільшення частоти кадрів (див. рис. 7).

```
mousePosition = Camera.main.ScreenToWorldPoint(Input.mousePosition);
mousePosition.z = 0;

var job = new UpdateParticlesJob
{
    deltaTime = Time.deltaTime,
    returnForce = returnForce,
    repulsionForce = repulsionForce,
    damping = damping,
    mousePosition = mousePosition,
    positions = positions,
    velocities = velocities,
    initialPositions = initialPositions,
    matrices = matrices
};

JobHandle handle = job.Schedule( arrayLength: particleCount,  innerloopBatchCount: 64);
handle.Complete();

Graphics.DrawMeshInstanced(particleMesh,  submeshIndex: 0, particleMaterial,
    matrices.ToArray(), particleCount, propertyBlock);
```

Рисунок 7 – Паралельне оновлення стану частинок (виконано самостійно)

Використання багатопоточності через Unity Jobs System та Burst Compiler надає значні переваги у продуктивності, особливо для завдань, що потребують інтенсивних обчислень та обробки великої кількості об'єктів. У нашому випадку це дозволяє ефективно оновлювати позиції і швидкості частинок паралельно,

покращуючи загальний час виконання і забезпечуючи плавний і чуйний інтерфейс користувача.

4.6 ECS

Використання Entity Component System (ECS) у Unity для вирішення задачі симуляції частинок може значно підвищити продуктивність завдяки високоефективному керуванню даними та паралельним обчисленням.

ECS зберігає дані компонент у масивоподібних структурах, що дозволяє ефективніше використовувати кеш процесора. Ці дані розташовуються послідовно у пам'яті, що зменшує кеш-промахи та збільшує швидкість доступу до них.

ECS підтримує автоматичне паралельне виконання систем, що дозволяє ефективно використовувати багатоядерні процесори та легко інтегрується з Burst Compiler, що забезпечує високу продуктивність.

4.6.1 Компоненти

Враховуючи, що в нашій моделі частинки незалежні одна від одної, а їхня поведінка визначається лише їх характеристиками та загальними константами, обчислення оновленого стану ідеально підходить для паралелізації.

Суть написання програми в рамках ECS полягає в поділі коду на компоненти (Components), які описують стан, системи (Systems), що описують поведінку та взаємодію компонентів, та сутності (Entities) – об'єкти, що містять набір компонентів.

Нам будуть потрібні наступні компоненти: початкове положення частинки (її атрактор), швидкість частинки, прискорення частки, дані для рендеру.

Компоненти ECS містять виключно чисті дані об'єктів, без жодної краплі логіки. Наведемо базовий шаблон, який загалом відбиває суть компонентів. Кожен такий компонент містить прості дані у вигляді векторів, матриць чи звичайних чисел. Шаблон класа компонента:

```
public struct Data : IComponentData
{
    public DataType data1;
    public DataType data2;
    public DataType data3;
}
```

Тепер коли у нас є структури, що зберігаються в собі всі можливі дані, потрібні для ігрової логіки, можна перейти до розробки систем.

4.6.2 Системи

Системи виконують лише виключно логіку обробки даних. Нам буде потрібна система, яка оновлює швидкість об'єктів у грі на основі їх прискорення та минулого часу, використовуючи паралельні обчислення для оптимізації продуктивності.

Для початку Entities.ForEach() метод виконує ітерацію по всіх сутностях, які містять певні компоненти (в даному випадку VelocityData та AccelerationData) та виконує зазначені операції для кожної сутності. Тут оновлюється швидкість кожної сутності на основі прискорення та часу, що минув.

Потім ScheduleParallel() метод запускає виконання операцій паралельно всім сутностям. ECS можуть виконуватися безліч операцій одночасно, що підвищує продуктивність (див. рис. 8).

```
public class VelocityUpdateSystem : SystemBase
{
    protected override void OnUpdate()
    {
        var time = Time.DeltaTime;
        Entities.ForEach((ref VelocityData velocity, in AccelerationData acceleration) ⇒
            {
                velocity.Value += time * acceleration.Value;
            }) // ForEachLambdaJobDescription
            .ScheduleParallel();
    }
}
```

Рисунок 8 – Система оновлення швидкості (виконано самостійно)

Після оновлення швидкості об'єктів ми можемо оновити їхню фактичну позицію. Принцип полягає у тому, що у кожному кадрі оновлює становище всіх

об'єктів, які мають компонент швидкості (VelocityData). Для цього використовується цикл Entities.ForEach, який перебирає кожен об'єкт із компонентом Translation (описує положення об'єкта у просторі) та VelocityData (описує швидкість об'єкта).

Усередині циклу відбувається зміна положення об'єкта на основі його поточної швидкості і часу з попереднього кадру. Для цього використовується формула: нове становище = старе становище + (час * швидкість).

Після оновлення всіх об'єктів, виконання даної системи паралельно розподіляється по кількох потоках за допомогою методу .ScheduleParallel(), що дозволяє оптимізувати продуктивність під час роботи з великою кількістю об'єктів (див. рис. 9).

```
public class MoveSystem : SystemBase
{
    protected override void OnUpdate()
    {
        var time = Time.DeltaTime;
        Entities.ForEach((ref Translation t, in VelocityData velocity)
                ⇒ { t.Value.xy += time * velocity.Value; }) // ForEachLambdaJobDescription
            .ScheduleParallel();
    }
}
```

Рисунок 9 – Система зміни фактичної позиції (виконано самостійно)

Наступною розробим систему оновлення прискорення, що діє відповідно до виведеної формули (див. рис. 10).

Принцип роботи системи полягає у обчисленні прискорення для кожного об'єкта на основі його положення, швидкості та впливу зовнішніх факторів, таких як тяжіння та відштовхування. Клас AccelerationSystem успадковується від SystemBase, базового класу для систем ECS в Unity.

В Update методі викликається кожен кадр оновлення стану системи. Усередині цього методу відбувається розрахунок прискорення об'єктів.

```
public static float2 FindNearestPointOnLine(in float2 origin, in float2 end, in float2 point)
{
    var heading = end - origin;
    var magnitudeSqMax = math.distancesq( x: origin,  y: end);
    heading = math.normalizesafe(heading);

    //Do projection from the point but clamp it
    var lhs = point - origin;
    var dotP = math.dot( x: lhs,  y: heading);
    dotP = math.clamp( x: dotP,  a: 0f,  b: magnitudeSqMax);
    return origin + heading * dotP;
}
```

Рисунок 10 – Пошук найближчої точки на прямій (виконано самостійно)

Далі виконується ітерація по всіх сутностях, які містять певні компоненти (в даному випадку AccelerationData, AttractorPosData, Translation і VelocityData) та виконується зазначені операції для кожної сутності. Тут відбувається розрахунок прискорення кожної сутності на основі різних факторів, таких як тяжіння, відштовхування та згасання (див. рис. 11).

```
protected override void OnUpdate()
{
    var repulsionMode = Globals.SettingsHolder.SettingsModel.RepulsionMode;
    float2 repulsorPos = Globals.Repulsor.Position;
    float2 repulsorPrevPos = float.IsNaN(_repulsorPrevPos.x) ? repulsorPos : _repulsorPrevPos;
    _repulsorPrevPos = repulsorPos;

    var attractionPower = Globals.SettingsHolder.SettingsModel.Attraction;
    var repulsionPower = Globals.SettingsHolder.SettingsModel.Repulsion;
    var dampingPower = Globals.SettingsHolder.SettingsModel.Damping;
    Entities.ForEach((ref AccelerationData acceleration,
            in AttractorPosData attractorPosition, in Translation t, in VelocityData velocity) =>
        {
            var attraction = (attractorPosition.Value - t.Value.xy) * attractionPower;

            var repulsorPosition = repulsionMode == SettingsModel.RepulsionModes.Point
                ? repulsorPos
                : FindNearestPointOnLine( origin: repulsorPos,  end: repulsorPrevPos,  point: t.Value.xy);

            var distSqr = math.clamp( x: math.distancesq( x: repulsorPosition,  y: t.Value.xy),
                a: Globals.MinRepulsionDist * Globals.MinRepulsionDist,  b: float.MaxValue);

            var repulsion = -math.normalizesafe( x: repulsorPosition - t.Value.xy) / distSqr * repulsionPower;

            var damping  = -velocity.Value * dampingPower;

            acceleration.Value = attraction + repulsion + damping;
        }) // ForEachLambdaJobDescription
        .ScheduleParallel();
}
```

Рисунок 11 – Обчислення прискорення (виконано самостійно)

Цей метод ScheduleParallel запускає виконання операцій паралельно всім сутностей. ECS можуть виконуватися безліч операцій одночасно, що підвищує продуктивність

Як результат після обчислення всіх впливів прискорення підсумовується та присвоюється об'єкту. Виконання даної системи також паралельно розподіляється на кількох потоках для оптимізації продуктивності.

4.6.3 Рендер

Хоча потрібно відобразити потенційно величезну кількість частинок, можна використовувати один і той же квадратний mesh з двох полігонів і той самий матеріал для всіх частинок. Це дозволить відрендерити їх усі за один draw call. Однак є одна проблема: у всіх частинок зазвичай різні кольори, і без цього картинка, що вийшла, буде нудною.

Стандартний Unity-шейдер «Sprites/Default» підтримує GPU Instancing для оптимізації рендеру об'єктів з різними спрайтами та кольорами. Він об'єднує їх в один draw call, але для цього потрібно задавати посилання на текстуру та колір для кожного об'єкта зі скрипту, до чого ми не маємо доступу з ECS.

Як альтернатива, можна використовувати метод Graphics. DrawMeshInstanced, який дозволяє малювати один mesh кілька разів за один draw call з різними параметрами матеріалу, використовуючи той же GPU Instancing.

Розробимо систему відмальовування спрайтових частинок Unity з використанням методу інстансингу підвищення продуктивності. Спочатку створюється екземпляр MaterialPropertyBlock, який дозволяє передавати кольори спрайтів на шейдер. Потім витягується міш квадрата (quadMesh) та матеріал (material), що використовується для рендерингу спрайтів.

Далі виходить список entityQuery, що містить компоненти ParticleData, після чого відбувається ітерація за даними спрайтів у блоках (кожен блок розміром BatchCount). Для кожного блоку даних формуються списки матриць трансформації та кольорів, що передаються у функцію Graphics.DrawMeshInstanced. Для кожного блоку даних викликається метод

Graphics.DrawMeshInstanced, який здійснює рендеринг екземплярів міша квадрата (quadMesh) із застосуванням матеріалу (material) та переданих кольорів. Цей метод рендерингу дозволяє суттєво збільшити продуктивність за рахунок використання інстансингу, що дозволяє рендерити безліч спрайтів за один прохід (див. рис. 12).

```
protected override void OnUpdate()
{
  var materialPropertyBlock = new MaterialPropertyBlock();
  var quadMesh = Globals.Quad;
  var material = Globals.ParticleMaterial;
  var shaderPropertyId = Shader.PropertyToID( name: "_Color");

  var entityQuery = GetEntityQuery( params componentTypes: typeof(ParticleData));
  var animationData = entityQuery.ToComponentDataArray<ParticleData>(Allocator.TempJob);
  var layer = LayerMask.NameToLayer("Particles");

  for (int meshCount = 0; meshCount < animationData.Length; meshCount += BatchCount)
  {
    var batchSize = math.min( x: BatchCount,  y: animationData.Length - meshCount);
    _matrixList.Clear();
    _colorList.Clear();

    for (var i = meshCount; i < meshCount + batchSize; i++)
    {
      var particleData = animationData[i];
      _matrixList.Add(particleData.matrix);
      _colorList.Add(particleData.color);
    }

    materialPropertyBlock.SetVectorArray(shaderPropertyId, _colorList);
    Graphics.DrawMeshInstanced(
      quadMesh,
       submeshIndex: 0,
      material,
      _matrixList,
      materialPropertyBlock,
       castShadows: ShadowCastingMode.Off,  receiveShadows: false, layer);
  }
  animationData.Dispose();
}
```

Рисунок 12 – Відображення частинок на екрані (виконано самостійно)

Для того, щоб використовувати цей метод для рендерингу групи об'єктів, необхідно зібрати масив матриць трансформації та параметрів матеріалу, які будуть змінюватися. Система рендеру виконується після TrsMatrix

CalculationSystem, яка розраховує матрицю трансформації для кожної частки (див. рис. 13).

```
public class TrsMatrixCalculationSystem : SystemBase
{
    protected override void OnUpdate()
    {
        var scale = Globals.SettingsHolder.SettingsModel.ParticlesScale;
        Entities.ForEach((ref ParticleData data, in Translation translation) ⇒
            {
                data.matrix = Matrix4x4.TRS( pos: translation.Value, Quaternion.identity,
                    s: new Vector3( x: scale,  y: scale,  z: scale));
            }) // ForEachLambdaJobDescription
            .ScheduleParallel();
    }
}
```

Рисунок 13 – Розрахунок матриць трансформації (виконано самостійно)

Закінчивши розробку всіх основних систем проекту, ми можемо приступити до його тестування на продуктивність.

5 ТЕСТУВАННЯ

5.1 Вплив вибраних оптимізаційних рішень на продуктивність

5.1.1 Умови тестування

В рамках нашого дослідження ми будемо тестувати та порівнювати кілька оптимізаційних методів для симуляції та відтворення великої кількості частинок у Unity. Ці методи включають використання спільного менеджера частинок, кешування, багатопоточність, GPU instancing, а також Entity Component System (ECS) з інтеграцією Jobs та Burst компілятора. Нашою метою є виявлення переваг та недоліків кожного підходу в умовах, наближених до реальних завдань розробки.

Для об'єктивного та всебічного аналізу продуктивності, ми будемо тестувати кожен метод в однакових умовах.

Тестування буде проводитися на ПК із середніми характеристиками, що включають процесор із чотирма ядрами, інтегровану графічну карту та 8 ГБ оперативної пам'яті. Ми будемо тестувати кожен метод з різними обсягами частинок: 10,000, 50,000 та 100,000. Це дозволить оцінити масштабованість та ефективність методів зі збільшенням складності сцени.

Для оцінки ефективності використовуватимемо метрики продуктивності. Основний показник - це FPS, що використовується для оцінки загальної продуктивності і плавності роботи програми. Також час кадру, що витрачається на обробку і рендеринг одного кадру, що вимірюється в мілісекундах.

Вимірюємо обсяг пам'яті , що використовується за різних методів, щоб оцінити їх ефективність в управлінні ресурсами (див. рис. 14).

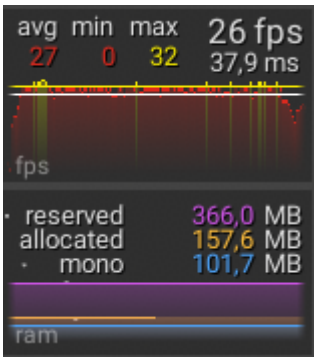

Рисунок 14 – Монітор статистиці (виконано самостійно)

Специфікації тестового пристрою:

- Full HD (1920x1080);
- Intel® Pentium® Gold7505 2GHz;
- Intel® UHD Graphics;
- 8GB GDDR4 RAM.

5.1.2 Проведення тестування

Проведемо кілька сценаріїв тестування. Спочатку виміряємо статичні частинки, які спокоїться, щоб оцінити базові накладні витрати на управління частинками та їх відмальовування. Потім динамічні частинки, які будуть піддаватися впливу різних сил (повернення до вихідної позиції, відштовхування від курсору, згасання), щоб оцінити продуктивність методів в динамічному середовищі і разом з ними інтерактивні частинки, які будуть реагувати на переміщення курсору миші, імітуючи взаємне користування, щоб перевірити ефективність методів за умов реального використання.

Передбачається, що використання багатопоточності та GPU instancing призведе до значного покращення продуктивності порівняно з базовою реалізацією на одному потоці. Інтеграція ECS з Jobs та Burst компілятором має надати найкращі результати завдяки оптимізованому управлінню даними та високого ступеня паралелізму. Однак, реальна продуктивність залежатиме від багатьох факторів, включаючи специфіку апаратної платформи та складності сцени.

Перехід від індивідуальних MonoBehaviour-скриптів до більш оптимізованих методів керування частинками значно покращує продуктивність. Використання загального менеджера частинок забезпечує хорошу масштабованість та стабільність. Використання кешування дозволяє досягти найкращих результатів, забезпечуючи високий FPS навіть при обробці 100,000 частинок. Нам вдалося підвищити кількість кадрів з 75 до 95 при 20 000 об'єктах. І з 6 до 17 за 100 000 об'єктів. Якщо при порівняно невеликій кількості об'єктів ми

отримали порівняно невеликий приріст у 26%, то при 100 000 об'єктах спостерігається практично триразовий приріст продуктивності (див. рис. 15).

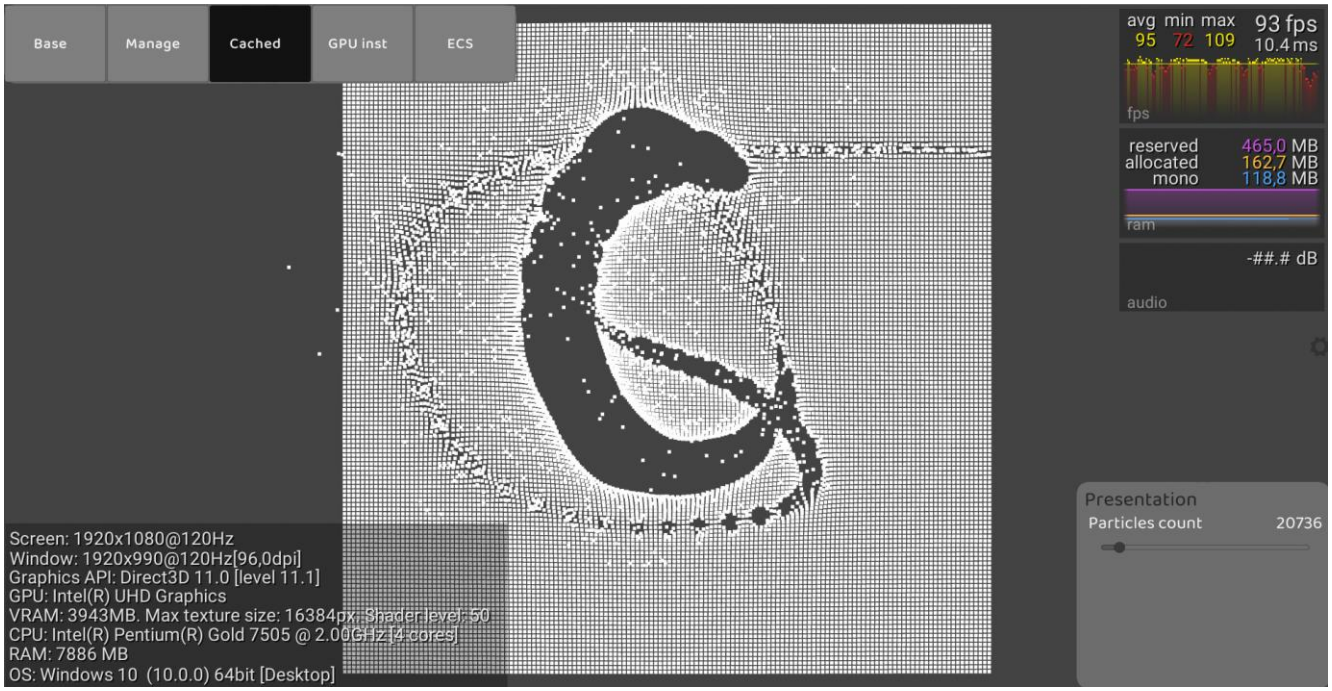

Рисунок 15 – Тестування продуктивності, сценарій з єдиним циклом оновлення та кешуванням даних 20000 об'єктів (виконано самостійно)

Продовжуючи дослідження методів оптимізації, ми протестували використання GPU instancing для симуляції частинок. Цей метод дозволяє малювати багато однотипних об'єктів з мінімальними накладними витратами на виклики від CPU до GPU.

GPU instancing дозволяє малювати безліч об'єктів в одному виклику, що знижує overhead на CPU та зменшує кількість draw calls. GPU може ефективно обробляти велику кількість об'єктів паралельно, що прискорює рендеринг сцени.

Тестування проводилося з тими самими наборами даних: 20,000, 40,000 та 100,000 частинок. GPU instancing показав значно кращі результати порівняно з попередніми методами. Цей метод дозволяє ефективно використовувати потужності графічного процесора, значно знижуючи навантаження на центральний процесор, що збільшило частоту кадрів у 3 рази при 20000 об'єктах та у 2 рази при 100000 об'єктах у порівнянні з попередніми методами (див. рис. 16).

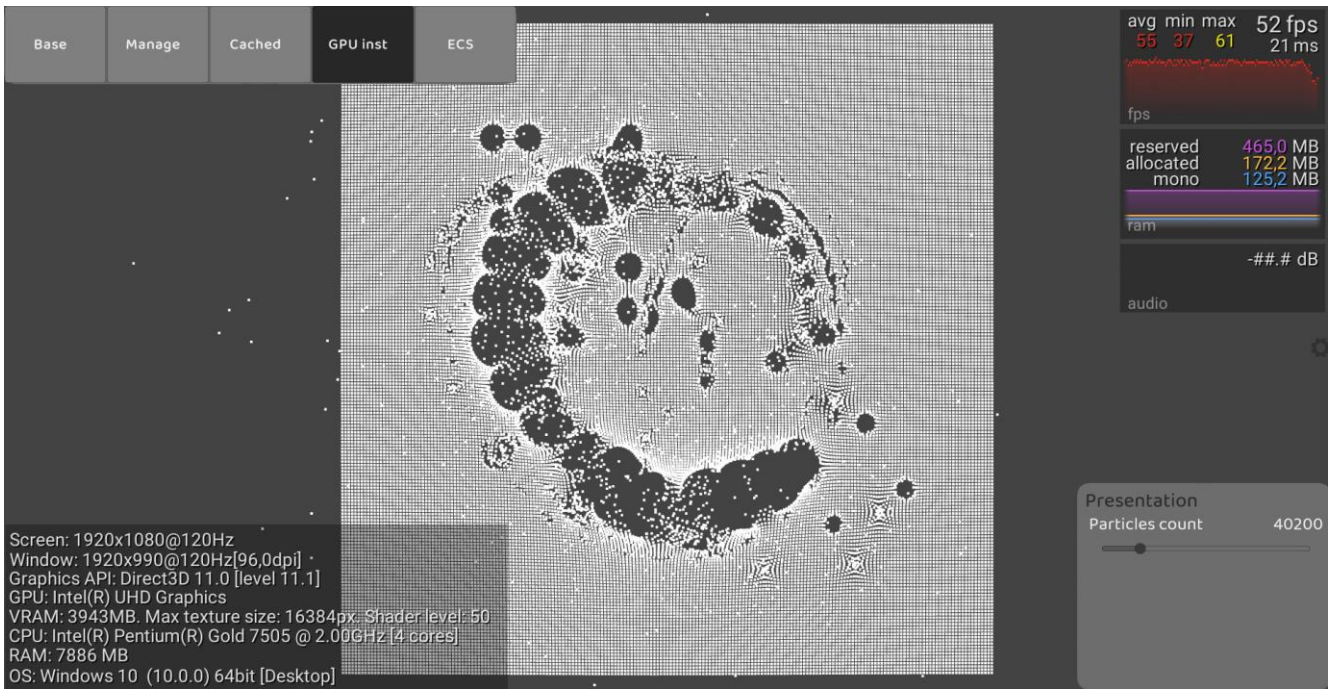

Рисунок 16 – Тестування продуктивності, сценарій з GPU instancing 40000 об'єктів (виконано самостійно)

Останній і найбільш розвинений метод оптимізації, який ми використовували, включає використання Entity Component System (ECS) у поєднанні з Unity Job System та Burst компілятором. Цей підхід дозволяє максимально ефективно використовувати можливості паралельних обчислень та оптимізацію коду на рівні машинних інструкцій. Тестування проводилося з тими самими наборами даних: 20,000, 40,000 та 100,000 частинок. Впровадження ECS разом з Jobs та Burst компілятором дозволило нам досягти найвищої продуктивності серед усіх протестованих методів.

Особливо хочется звернути увагу на високі показники частоти кадрів при 100 000 об'єктах. 160 кадрів в секунду - це вражаючий результат, який на голову перевершує всі попередні сценарії.

Цей підхід забезпечує максимальну паралелізацію та ефективність управління пам'яттю, що дозволяє обробляти та малювати велику кількість частинок з мінімальними накладними витратами (див. рис. 17).

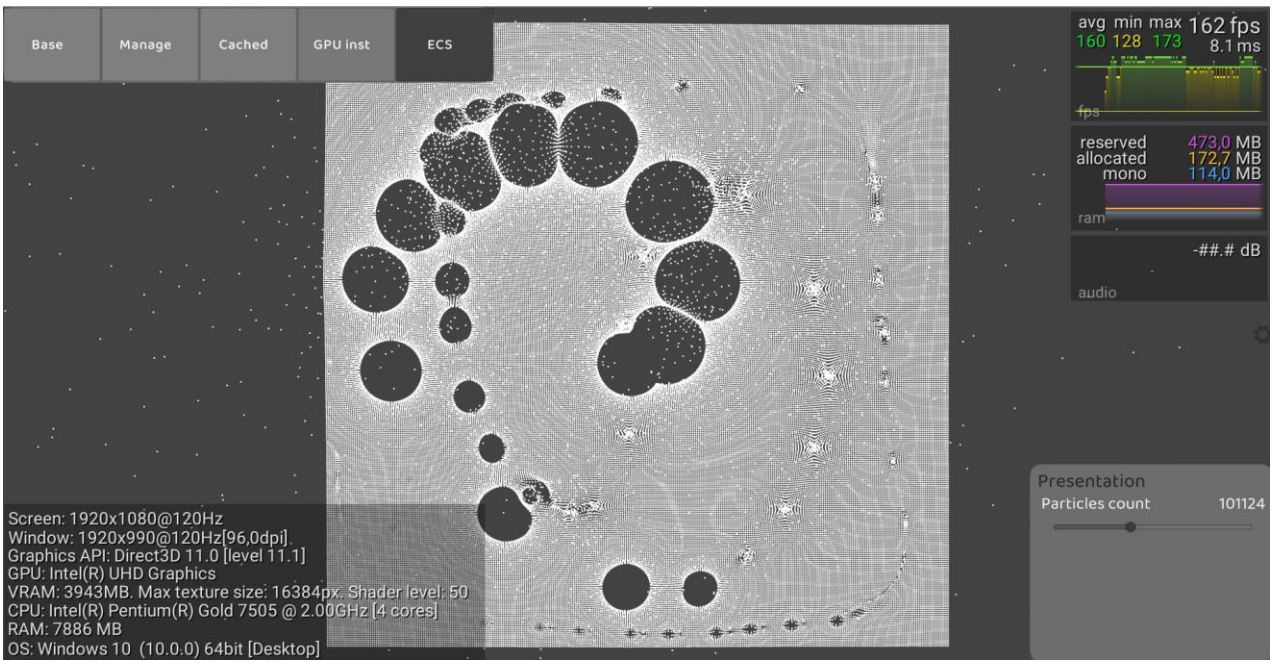

Рисунок 17 – Тестування продуктивності, сценарій з ECS+ Jobs + Burst 100000 об'єктів (виконано самостійно)

Завершивши етап проведення тестування різних методів оптимізації симуляції частинок Unity, ми отримали чітке уявлення про те, як кожен підхід впливає на продуктивність. Наші експерименти показали, що перехід від індивідуальних MonoBehaviour-скриптів до використання спільного менеджера та кешування суттєво покращує FPS. GPU instancing забезпечив ще більшу продуктивність за рахунок ефективного використання ресурсів графічного процесора. Найвищі результати продемонстрував метод ECS у поєднанні з Jobs та Burst компілятором, що забезпечив максимальну паралелізацію та оптимізацію, що дозволило досягти найкращих показників FPS та масштабованості. Завершивши тестування, занесемо всі дані в таблицю і приступимо до аналізу результатів.

5.1.3 Результати тестування

У нас вийшла таблиця, яка включає наступні сценарії тестування: Mono, спільний менеджер частинок, кешування, багатопоточність, GPU instancing, а також Entity Component System (ECS) з інтеграцією Jobs та Burst компілятора. Для оцінки ефективності обраних оптимізаційних підходів було обрано два

основних критерія: середня частоту кадрів, якої вдалося досягти при різній кількості об'єктів на сцені та час, що витрачається на обробку та рендеринг одного кадру, що вимірюється в мілісекундах (див. табл. 5.1).

Таблиця 5.1 – Результати тестування за різних сценаріїв. Метрикою продуктивності виступає середня частота кадрів (таблиця виконана самостійно)

Сценарій тестування	Кількість частинок - 20000	Кількість частинок - 40000	Кількість частинок - 100000
Індивідуальні MonoBehaviours	75	15	6
Спільний менеджер	90	26	17
Кешування	95	30	20
GPU Instansing	300	55	37
GPU Instansing + Jobs + Burst	600	110	70
ECS + Jobs + Burst	650	270	160

Індивідуальні MonoBehaviours показали найнижчі результати, особливо зі збільшенням кількості частинок. При 100,000 частинок FPS впав до 6, що робить симуляцію практично неграбної. Введення загального менеджера частинок дозволило значно покращити продуктивність. усуває багато накладних витрат, пов'язаних з керуванням великою кількістю об'єктів, але все ще не є оптимальним для великих сцен. Кешування даних частинок додає невеликий приріст продуктивності. Це поліпшення особливо помітне зі збільшенням кількості частинок, оскільки зменшує накладні витрати доступу до даних. Застосування GPU instancing дало значний приріст продуктивності. За рахунок використання потужності графічного процесора вдалося досягти 300 FPS за 20,000 частинок. Проте, зі збільшенням кількості частинок, приріст був настільки значним, що вказує на наявність вузьких місць при високих навантаженнях. Додавання

багатопоточності та оптимізації через Jobs та Burst компілятор до GPU instancing подвоїло ефективність у 20,000 частках та помітно покращило результати у великих обсягах даних. Це підкреслює ефективність паралельних обчислень та оптимізації на рівні машинного коду. ECS + Jobs + Burst продемонстрував найкращі результати (див. табл. 5.2), досягаючи 650 FPS при 20,000 частках і зберігаючи високу продуктивність навіть за 100,000 частинок (160 FPS).

Таблиця 5.2 – Результати тестування за різних сценаріїв. Метрикою продуктивності виступає час кадру у мілісекундах (таблиця виконана самостійно)

Сценарій тестування	Кількість частинок - 20000	Кількість частинок - 40000	Кількість частинок - 100000
Індивідуальні MonoBehaviours	14	76	120
Спільний менеджер	11	48	63
Кешування	10	46	58
GPU Instansing	4,8	21	39
GPU Instansing + Jobs + Burst	2,3	6,9	15
ECS + Jobs + Burst	2	4,4	8,1

Наше тестування показало, що для досягнення максимальної продуктивності та масштабованості у симуляціях з великою кількістю частинок у Unity, найбільш ефективним методом є використання ECS у поєднанні з Jobs та Burst компілятором. Цей підхід забезпечує високу продуктивність за рахунок оптимального використання ресурсів як CPU, так і GPU і демонструє відмінну масштабованість навіть при збільшенні кількості частинок до 100,000.

Застосування методів оптимізації, таких як GPU instancing та багатопоточність, також значно покращує продуктивність, але не настільки, як ECS з Jobs і Burst. Індивідуальні MonoBehaviour-скрипти та загальний менеджер частинок можуть бути використані для менш вимогливих сцен, але вони не забезпечують необхідної продуктивності для великих симуляцій.

5.2 Залежність продуктивності від середовища виконання

5.2.1 Умови тестування

В даний час у Unity доступні два варіанти реалізації середовища виконання (Scripting Backend) для роботи з кодом мовою C#: традиційна Mono та сучасне рішення від Unity розробників – IL2CPP. Mono використовує JIT-компілятор, а IL2CPP використовує AOT-компілятор.

JIT-компілятори не компілюються під час створення програми. Компіляція відбувається під час запуску програми на пристрої цільового користувача. Windows підтримує компілятор, але не останні мобільні телефони.

Компілятори AOT компілюються під час збирання і, отже, повільніше, ніж JIT-компілятори. Підтримується як на ПК, так і мобільних платформах.

Давайте порівняємо продуктивність складання для цих двох реалізацій середовища виконання: Mono та IL2CPP.

Специфікації тестового пристрою:

- Full HD (1920x1080);
- AMD Ryzen 5 4600H;
- GTX1650 4GB;
- 16GB GDDR4 RAM 3200MHz.

5.2.2 Проведення тестування

Переконаймося, що всі необхідні компоненти та залежності DOTS встановлені та налаштовані коректно.

Запустимо тестування продуктивності програми на двох різних Scripting Backend: Mono та IL2CPP. Для цього створимо сценарій, в якому буде багато

частинок, що реалізують описаний ефект. Запустимо додаток і виміряємо середній FPS (кількість кадрів на секунду) за різної кількості частинок для обох варіантів Scripting Backend.

Порівняємо середній FPS для кожної кількості частинок під час використання Mono та IL2CPP. Звернімо увагу зміну продуктивності зі збільшенням кількості частинок. Проаналізуємо отримані результати та зробимо висновки про те, який варіант Scripting Backend забезпечує кращу продуктивність для цього проекту.

Завантажимо зображення, що складається зі 100 000 об'єктів у сбірку Mono. Підключимо спеціальний інструмент для вимірювання продуктивності, щоб у реальному часі відстежувати частоту кадрів та споживання пам'яті. На моніторі продуктивність бачимо в середньому 72 кадри на секунду (див. рис. 18).

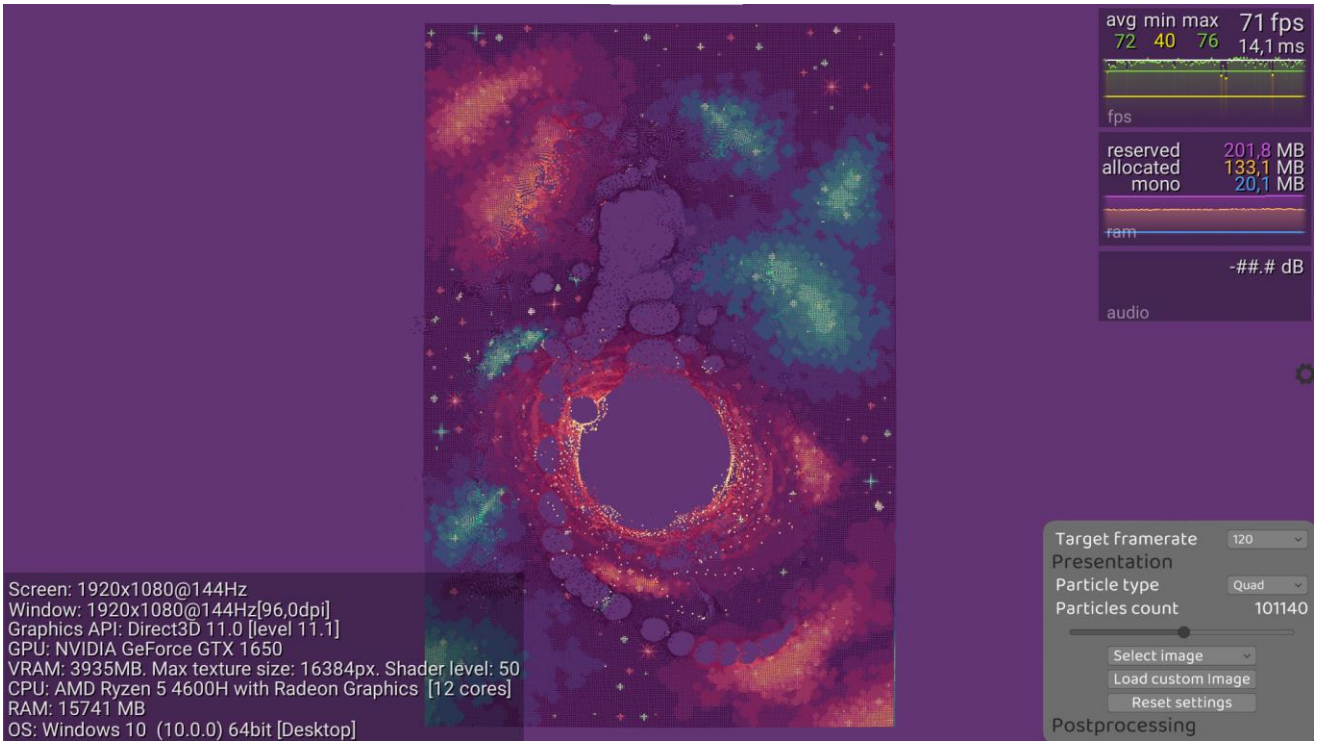

Рисунок 18 – Тестування продуктивності, сбірка Mono 100 000 об'єктів (виконано самостійно)

Mono JIT обмежений у тому, який аналіз та оптимізацію він може виконувати, оскільки він робить це "на льоту". Він також часто не має «загальної

картини» того, що відбувається у нашій програмі, і тому зазвичай може виконувати лише «локальну» оптимізацію.

Спробуємо збільшити зображення ще на 100 000 об'єктів у сбіркці Mono. Підключимо спеціальний інструмент для вимірювання продуктивності, щоб у реальному часі відстежувати частоту кадрів та споживання пам'яті. На моніторі продуктивність бачимо в середньому 32 кадри на секунду (див. рис. 19).

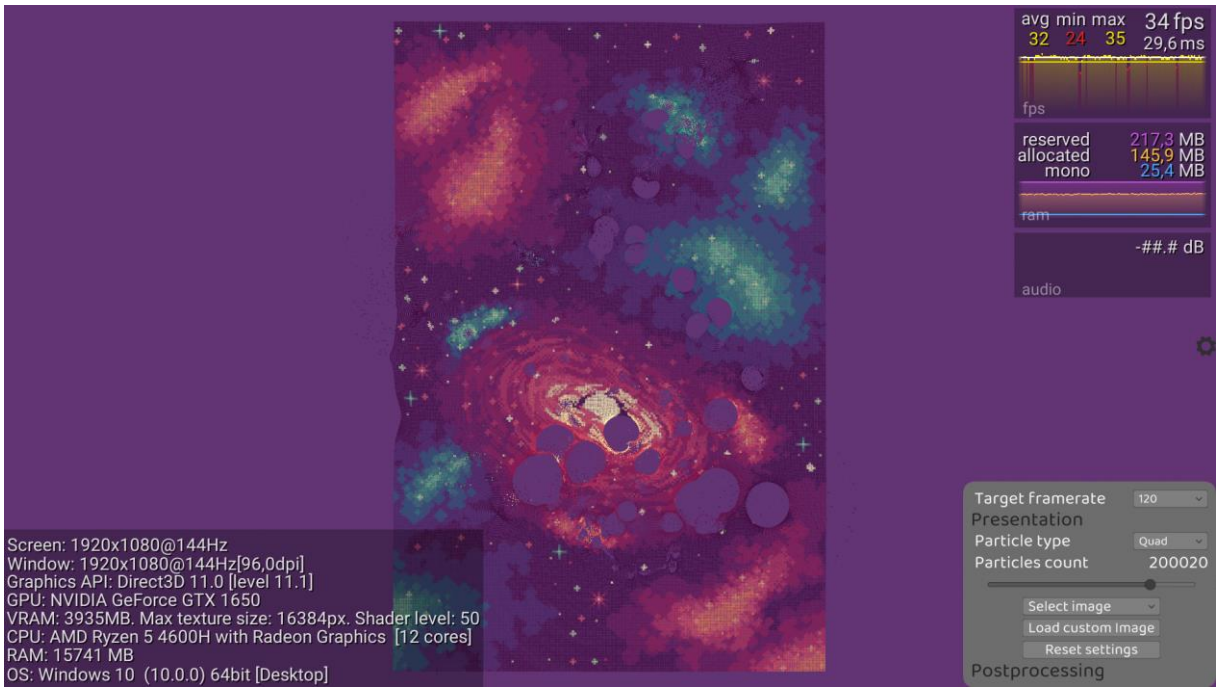

Рисунок 19 – Тестування продуктивності, сбірка Mono 200 000 об'єктів (виконано самостійно)

Завантажимо зображення, що складається зі 300 000 об'єктів у сбірку Mono. Підключимо спеціальний інструмент для вимірювання продуктивності, щоб у реальному часі відстежувати частоту кадрів та споживання пам'яті. На моніторі продуктивність бачимо в середньому 23 кадри на секунду (див. рис. 20). Спостерігається значне навантаження на центральний процесор, що спричиняє зниження частоти кадрів. Споживання оперативної пам'яті також досягає високих значень, що вказує на необхідність оптимізації системи для обробки такої кількості об'єктів.

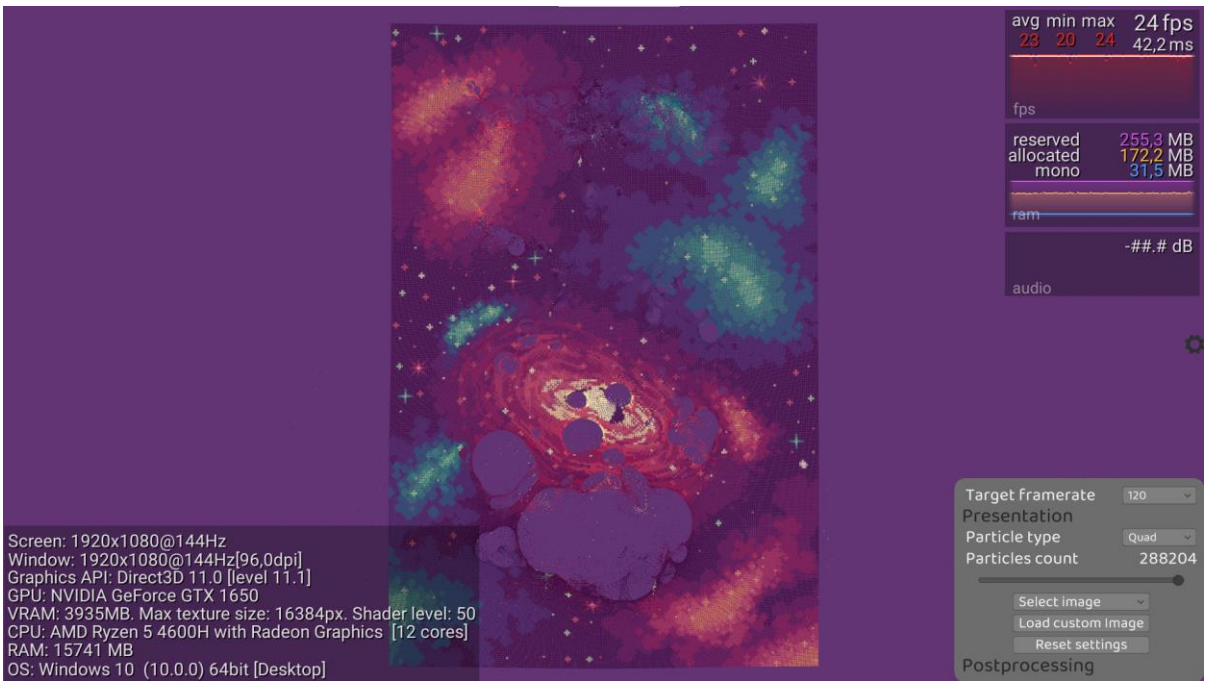

Рисунок 20 – Тестування продуктивності, сбірка Mono 300 000 об'єктів (виконано самостійно)

Завантажимо зображення, що складається зі 200 000 об'єктів у сбірку IL2CPP. Перевага IL2CPP полягає у попередній компіляції нашого коду за допомогою оптимізуючого, сучасного та зрілого компілятора C++. Але оскільки IL2CPP прагне поводитися так само, якби ми виконали свою програму в моно, він може зрештою згенерувати код, який компілятору не обов'язково легко оптимізувати. Підключимо спеціальний інструмент для вимірювання продуктивності, щоб у реальному часі відстежувати частоту кадрів та споживання пам'яті. На моніторі продуктивність бачимо в середньому 48 кадрів на секунду

Завершивши тестування, занесемо всі дані в таблицю і приступимо до аналізу результатів.

5.2.3 Результати тестування

У нас вийшла невелика таблиця, яка включає два різних сценарії тестування і середню частоту кадрів, якої вдалося досягти при різній кількості об'єктів на сцені (див. табл. 5.3).

Таблиця 5.3 – Результати тестування двох реалізацій Mono та IL2CPP (таблиця виконана самостійно)

Кількість об'єктів	Середній FPS, Mono	Середній FPS, IL2CPP
30000	115	115
60000	113	114
100 000	72	107
200 000	32	48
290 000	23	34

На підставі цих даних можна зробити висновок що, в рамках цього с експерименту, IL2CPP виграє у Mono приблизно на 30%.

Згідно з профайлером, буквально весь час, що витрачається на кадр, йде на систему рендеру, розрахунки інших систем обходяться практично «безкоштовно» (див. рис. 21).

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
▼ PlayerLoop	89.9%	0.2%	2	400 B	39.36	0.12
▼ SimulationSystemGroup	83.4%	0.0%	1	96 B	36.50	0.00
▼ UpdateFunction.Invoke()	83.4%	0.0%	1	96 B	36.50	0.00
▼ Default World Unity.Entities.SimulationSystemGroup	83.4%	0.0%	1	96 B	36.49	0.02
▶ Default World ECS.Systems.SpriteSheetRenderer	82.8%	81.7%	1	96 B	36.24	35.75
Default World ECS.Systems.AccelerationSystem	0.2%	0.2%	1	0 B	0.09	0.09
Default World ECS.Systems.VelocityUpdateSystem	0.0%	0.0%	1	0 B	0.03	0.03
Default World ECS.Systems.TrsMatrixCalculationSystem	0.0%	0.0%	1	0 B	0.02	0.02
Default World ECS.Systems.MoveSystem	0.0%	0.0%	1	0 B	0.02	0.02
▶ Default World Unity.Transforms.TransformSystemGroup	0.0%	0.0%	1	0 B	0.02	0.01
▶ Default World Unity.Entities.BeginSimulationEntityCommandBufferSystem	0.0%	0.0%	1	0 B	0.01	0.01
▶ Default World Unity.Entities.LateSimulationSystemGroup	0.0%	0.0%	1	0 B	0.01	0.01
▶ Default World Unity.Entities.EndSimulationEntityCommandBufferSystem	0.0%	0.0%	1	0 B	0.00	0.00
Default World CompanionGameObjectUpdateTransformSystem	0.0%	0.0%	1	0 B	0.00	0.00
Default World CompanionGameObjectUpdateSystem	0.0%	0.0%	1	0 B	0.00	0.00
Default World EditorCompanionGameObjectUpdateSystem	0.0%	0.0%	1	0 B	0.00	0.00
EntityCommandBuffer Playback	0.0%	0.0%	1	0 B	0.00	0.00

Рисунок 21 –Загальний час, який знадобився проекту для відтворення 1 кадру. (виконано самостійно)

ECS загалом і Unity DOTS зокрема відмінні інструменти для певних сценаріїв та класів завдань. Свою роль ефективній обробці величезної кількості даних вони виконують чудово і дозволяють створювати симуляції, на які в їх відсутність пішло б значно більше зусиль.

ВИСНОВКИ

У ході виконання роботи був проведений аналіз методів оптимізації, спрямовані на підвищення продуктивності та ефективності складних ігрових програмних систем, що включають в себе велику кількість об’єктів та складних фізичних обчислень.

На підставі цього аналізу був обраний стек технологій DOTS і розроблений проект для проведення тестування та оцінки ефективності обраного рішення.

Хоча Unity DOTS можливо не є універсальним рішенням для кожного проекту, воно безперечно має великий потенціал для тих, хто зацікавлений у розробці, орієнтованій на продуктивність та масштабованість. Я вважаю, що Unity DOTS заслуговує на увагу, оскільки ECS може довести свою цінність у багатьох ігрових проектах.

Платформа Unity DOTS, з її фокусом на дані та багатопотоковим підходом, пропонує величезний потенціал для оптимізації продуктивності ігор у середовищі Unity. Використовуючи архітектуру Entity Component System та технології, які є складовою частиною, такі як Job System та Burst Compiler, розробники можуть досягти нових рівнів продуктивності та масштабованості.

Unity DOTS є потужною платформою, яка може значно поліпшити досвід розробки ігор, забезпечуючи підвищення продуктивності, паралельні обчислення та готовність до майбутньої обробки на багатоядерних системах. Вивчення та розгляд можливостей впровадження Unity DOTS дозволить розробникам повністю використовувати сучасне обладнання та оптимізувати продуктивність ігор у середовищі Unity.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Гриневич К. В. Методи оптимізації середи розробки ігор unity у жанрі «First-person Shooter» / К. В. Гриневич, Л. А. Власенко // Сучасні напрями розвитку інформаційно-комунікаційних технологій та засобів управління : тез. доп. дванадцатої міжнародної науково-технічної конференції, 27–28 квітня 2022 р. – Т. 2. – Баку–Харків–Жиліна, 2022. – С. 167.

2. Jason Gregory. Game Engine Architecture. 3rd edition. CRC Press, 2018. 1240 p.

3. Афанасьєва І. В. Фреймворк для рендерінгу 3d сцен на платформах, що підтримують Metal API / І. В. Афанасьєва, О. С. Перов // Сучасні напрями розвитку інформаційно-комунікаційних технологій та засобів управління : тез. доп. дванадцатої міжнародної науково-технічної конференції, 27–28 квітня 2022 р. – Т. 2. – Баку–Харків–Жиліна, 2022. – С. 117.

4. Andrew Rollings, Dave Morris. Game Architecture and Design. 2rd edition. New Riders, 2004. 926 p.

5. Julian Gold. Object-oriented Game Development. Pearson Education, 2004. 426p.

6. Daniel Sánchez, Crespo Dalmau. Core Techniques and Algorithms in Game Programming. New Riders, 2004. 854 p.

7. Hocking, Joseph. Unity in Action: Multiplatform Game Development in C#. Manning Publications, 2018. 560 p.

8. Thorn, Alan. Pro Unity Game Development with C#. Apress, 2019. 520 p.

9. Nystrom, Robert. Game Programming Patterns. Genever Benning, 2014. 354 p.

10. Smith, Matt. Unity 2020 Cookbook: Over 160 recipes to take your 2D and 3D game development to the next level, 4th Edition. Packt Publishing, 2020. 470 p.

11. Thorn, Alan. Pro Unity High-Performance C#. Apress, 2018. 510 p.

12. Gregory, Jason. Game Engine Architecture. CRC Press, 2018. 864 p.

13. Godbold, Dr. Ashley. Mastering Unity 2020 Game Development with C#: Build advanced games and applications with Unity 2020 and C#. Packt Publishing, 2020. 560 p.

14. Adams, E. Fundamentals of Game Design / E. Adams. – New Riders, 2014. – 640 p.

15. Gregory, J. Game Engine Architecture / J. Gregory. – CRC Press, 2018. – 1040 p.

16. Millington, I., Funge, J. Artificial Intelligence for Games / I. Millington, J. Funge. – CRC Press, 2016. – 864 p.

17. Takahashi, D., Bentley, P. Real-Time Rendering / D. Takahashi, P. Bentley. – CRC Press, 2010. – 576 p.

18. Nystrom, R. Game Programming Patterns / R. Nystrom. – Genever Benning, 2014. – 354 p.

19. Mitchell, T. M. Machine Learning / T. M. Mitchell. – McGraw Hill, 1997. – 414 p.

20. Кириченко І. В. Порівняння ефективності алгоритмів пошуку шляху при розробці ігрового штучного інтелекту / І. В. Кириченко, В. Д. Рошка // Бионика интеллекта. – 2021. – № 2 (97). – С. 39–45. – DOi: https://doi.org/30837/ bi.

21. Shirley, P., Marschner, S. Fundamentals of Computer Graphics / P. Shirley, S. Marschner. – A K Peters/CRC Press, 2009. – 784 p.

22. Hillis, D. W. The Art of Game Design: A Book of Lenses / D. W. Hillis. – CRC Press, 2001. – 512 p.

23. Salen, K., Zimmerman, E. Rules of Play: Game Design Fundamentals / K. Salen, E. Zimmerman. – MIT Press, 2003. – 688 p.

24. Unity Technologies. Unity Manual and Scripting API / Unity Technologies. – Unity Technologies, 2021.

25. Jönsson, R. Learn ECS and Job System with Unity 2020: A Practical Guide to Writing Performant and Maintainable Code / R. Jönsson. – Packt Publishing, 2020. – 350 p.

26. Eberly, D. H. 3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics / D. H. Eberly. – CRC Press, 2007. – 1040 p.

27. Blow, J. Game Development: Harder Than You Think / J. Blow. – Queue, 1(10), 28-37, 2004.

28. Van Verth, J. M., Bishop, L. M. Essential Mathematics for Games and Interactive Applications: A Programmer's Guide / J. M. Van Verth, L. M. Bishop. – CRC Press, 2015. – 864 p.

29. Rollings, A., Adams, E. Andrew Rollings and Ernest Adams on Game Design / A. Rollings, E. Adams. – New Riders, 2003. – 648 p.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ ЗА НАУКОВИМИ НАПРЯМАМИ НАУКОВЦІВ КАФЕДРИ ПРОГРАМНОЇ ІНЖЕНЕРІЇ

1. Гриневич К. В. Методи оптимізації середи розробки ігор unity у жанрі «First-person Shooter» / К. В. Гриневич, Л. А. Власенко // Сучасні напрями розвитку інформаційно-комунікаційних технологій та засобів управління : тез. доп. дванадцатої міжнародної науково-технічної конференції, 27–28 квітня 2022 р. – Т. 2. – Баку–Харків–Жиліна, 2022. – С. 167.

3. Афанасьєва І. В. Фреймворк для рендерінгу 3d сцен на платформах, що підтримують Metal API / І. В. Афанасьєва, О. С. Перов // Сучасні напрями розвитку інформаційно-комунікаційних технологій та засобів управління : тез. доп. дванадцатої міжнародної науково-технічної конференції, 27–28 квітня 2022 р. – Т. 2. – Баку–Харків–Жиліна, 2022. – С. 117.

20. Кириченко І. В. Порівняння ефективності алгоритмів пошуку шляху при розробці ігрового штучного інтелекту / І. В. Кириченко, В. Д. Рошка // Бионика интеллекта. – 2021. – № 2 (97). – С. 39–45. – DOi: https://doi.org/30837/ bi.