Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Моделі, методи та засоби оптимізації ресурсів в ігрових застосунках

(тема)

Виконав:
здобувач 2 року навчання,
групи СПм-23-3
Данило СЕРГЄЄВ
(власне ім'я, прізвище)

Спеціальність
123 «Комп'ютерна інженерія»
(код і повна назва спеціальності)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма
Системне програмування
(повна назва освітньої програми)

Керівник: проф. Тетяна ФЕСЕНКО
(посада, власне ім'я, прізвище)

Допускається до захисту

Завідувач кафедри ЕОМ _____ Андрій КОВАЛЕНКО
(підпис) (власне ім'я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти другий (магістерський)

Спеціальність 123 «Комп'ютерна інженерія»
(код і повна назва)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Системне програмування
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
"____" _____ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Сергєєву Данилу Володимировичу
(прізвище, ім'я, по батькові)

1. Тема роботи Моделі, методи та засоби оптимізації ресурсів в ігрових застосунках

затверджена наказом по університету від "21" квітня 2025 р. № 296 Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 16 червня 2025 р.

3. Вхідні дані до роботи 1) основні технології Unity: ScriptableObject, Custom Editor, Unity Profiler, Frame Debugger, FrameTimingManager;
2) інструменти оптимізації продуктивності: Burst Compiler, Job System, Compute Shaders;
3) середовище розробки та інструменти: Unity, C#, Visual Studio, GitHub.

4. Перелік питань, що потрібно опрацювати у роботі
1) проведення аналізу ресурсів, що будуть підлягати оптимізації;
2) виконання аналізу існуючих рішень в предметній області;
3) вибір методів та технологій, що будуть використовуватись при створенні власних моделей;
4) створення моделей для оптимізації ресурсів;
5) тестування моделей та аналіз результатів.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій

Слайд-презентація – 16 слайдів

6. Консультанти розділів роботи (заповнюється за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Постановка задачі, визначення об'єкта та	21.04.2025-25.04.2025	Виконано
	предмета		
2	Теоретичне обґрунтування методів	25.04.2025-01.05.2025	Виконано
	оптимізації		
3	Розробка моделей оптимізації	01.05.2025-20.05.2025	Виконано
4	Оформлення матеріалів кваліфікаційної роботи	20.05.2025-30.05.2025	Виконано
5	Подання кваліфікаційної роботи керівникові	30.05.2025-10.06.2025	Виконано
	та її попередній захист		
6	Подання кваліфікаційної роботи на	10.06.2025-12.06.2025	Виконано
	рецензування		

Дата видачі завдання " 21 " квітня 2025 р.

Здобувач _____
(підпис)

Керівник роботи _____ проф. Тетяна ФЕСЕНКО
(підпис) (посада, власне ім'я, прізвище)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 104 с., 57 рис., 13 табл., 1 дод., 25 джерел.

ОПТИМІЗАЦІЯ, ІГРОВА СИСТЕМА, ЦЕНТРАЛЬНИЙ ПРОЦЕСОР, ГРАФІЧНИЙ ПРОЦЕСОР, АВТОМАТИЗАЦІЯ, UNITY.

Метою кваліфікаційної роботи є розробка моделей та методів, що націлені на вирішення оптимізаційних проблем при розробці ігрової системи. Основними завданнями дослідження стали аналіз можливих критичних секцій, які створюють проблеми з продуктивністю, створення моделей та методів для обробки таких ресурсів, тестування створених систем.

В ході роботи було проведено аналіз найбільш значущих ресурсів в ігрових додатках, на основі якого було встановлено вектор напрямку для розробки моделей та методів. Проведено огляд сучасних рішень, що мають відношення до обраної теми. Побудовано три напрямки розвитку моделей.

Перша модель вирішує проблему надмірного використання ресурсів процесора. Друга досліджує взаємозв'язок між завантаженням пам'яті та архітектурою структур даних. Третя спрямована на проблему надмірних витрат часу розробника. На основі цих напрямків було створено три системи.

Перша – це модель оптимізації CPU шляхом вивантаження найбільш критичних обчислювальних операцій на GPU. Друга – набір методів для проектування оптимальної архітектури даних. Третя – набір моделей для створення більш структурованих та систематизованих інтерфейсів для генерації та конфігурації ігрових об'єктів. Кожна система була протестована з використанням різних формату тестів, а ефективність запропонованих рішень була проаналізована на основі результатів тестування.

ABSTRACT

Master’s thesis: 104 pages, 57 figures, 13 tables, 1 appendices, 25 sources.

OPTIMIZATION, GAME SYSTEM, CENTRAL PROCESSING UNIT, GRAPHICS PROCESSING UNIT, AUTOMATION, UNITY.

The aim of this qualification work is the development of models and methods aimed at solving optimization problems in the design of a game system. The main objectives of the research include the analysis of potential critical sections that cause performance issues, the creation of models and methods for handling such resources, and testing the developed systems.

During the course of the work, an analysis of the most significant resources in game applications was conducted, based on which a direction vector for developing models and methods was established. Contemporary solutions relevant to the chosen topic were reviewed. Three development directions for models were constructed.

The first model addresses the issue of excessive CPU resource usage. The second investigates the relationship between memory load and the architecture of data structures. The third targets the problem of excessive developer time usage. Based on these directions, three systems were created.

The first is a model for CPU optimization by offloading the most critical computational operations to the GPU. The second is a set of methods for designing an optimal data architecture. The third is a set of models for creating more structured and systematic interfaces for game object generation and configuration. Each system was tested using various formats, and the effectiveness of the proposed solutions was analyzed based on the test results.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

ГП – графічний процесор

ПК – персональний комп’ютер

ЦП – центральний процесор

ШІ – штучний інтелект

CPU – центральний процесор (Central Processing Unit)

FPS – кадри за секунду (Frames Per Second)

GPU – графічний процесор (Graphics Processing Unit)

HLSL – мова високорівневого шейдингу (High-Level Shading Language)

IL – проміжна мова (Intermediate Language)

LLVM – інфраструктура для компіляції (Low-Level Virtual Machine)

NPC – неігровий персонаж (Non-Player Character)

PC – персональний комп’ютер (Personal Computer)

RAM – оперативна пам’ять (Random Access Memory)

SIMD – одноінструкційна багатоданна обробка (Single Instruction, Multiple Data)

URP – універсальний рендер-пайплайн (Universal Render Pipeline)

VRAM – відеопам’ять (Video Random Access Memory)

ВСТУП

Сфера розробки ігрових застосунків стрімко росте з року в рік. Такий стрімкий розвиток вносить також необхідність у постійному дослідженні впливу різних компонентів на продуктивність системи, що розроблюється. Створення нових моделей оптимізації та їх впровадження до реальних проєктів може сприяти підвищенню продуктивності та їх ефективності.

Метою цієї кваліфікаційної роботи є створення моделей, методів та засобів, що націлені на вирішення основних проблем, що пов’язані із недостатньою оптимізацією при розробці ігрових застосунків. Для досягнення поставленої мети необхідно вирішити наступні задачі:

- дослідження роботи вбудованих інструментів для створення освітлення;
- розробка власної моделі, що вирішує проблему надмірного використання ресурсів при роботі з вбудованими рішеннями;
- аналіз роботи створеної моделі та співвідношення графічних результатів;
- дослідження впливу різних факторів на завантаження пам’яті;
- створення алгоритму для оптимізації оперативної пам’яті;
- проведення аналізу результатів застосування алгоритму;
- створення систем, націлених на автоматизацію процесів;
- порівняння результатів створених моеделей.

Об’єктом роботи є методи та моделі оптимізації ресурсів при розробці ігрових застосунків. Предметом є розробка систем та засобів спрямованих на автоматизацію процесів та зниження навантаження на апаратні ресурси комп’ютера: центральний процесор та оперативну пам’ять.

Для досягнення поставлених задач необхідно використовувати компаративні аналізи експериментальних результатів, моделювання систем та їх тестування.

1 ТЕОРЕТИЧНІ ОСНОВИ ОПТИМІЗАЦІЇ РЕСУРСІВ В ІГРОВИХ ЗАСТОСУНКАХ

1.1 Аналіз і значення ресурсів в ігрових застосунках

1.1.1 Визначення ресурсів у ігрових системах

Розробка гри доволі важкий процес не тільки зі сторони написання скриптів для основної логіки, але й з точки зору оптимізації. Загалом оптимізація є невід’ємною частиною глобального процесу створення системи. Пов'язано це з тим, що надмірне використання обчислювальних ресурсів може призвести до негативних результатів. Саме тому починаючи процес оптимізації необіхдно визначитись які взагалі ресурси споживає будь-яка гра.

Першим необхідно розглянути ЦП (центральний процесор). ЦП є основою будь-якого комп’ютера, й елементом, що виконує команди та відповідає за обробку даних [7, 14]. Альтернативно, його можна назвати мостом між різними частинами девайсу. Основним компонентом процесора є ядро, що також можна назвати одиницею продуктивності. Кожне таке ядро має властивість виконувати одну послідовність певних команд самостійно. Іншими словами, якщо процесор має багато ядер, то це означає що кількість завдань, що можуть виконуватися паралельно, буде більшою [7]. Кожна програма, у тому ж числі й гра, використовує центральний процесор. Тому зменшення навантаження на нього є майже найважливішим етапом у порівнянні з іншими.

Далі йде ГП (графічний процесор). Це інший вид процесора, що націлений на прискорення різного формату операцій, починаючи від рендеренгу графіки закінчуючи розрахунками великих об’ємів даних або ж розробкою ШІ (штучного інтелекту) [5, 14]. Будь-яка гра використовує

графічний процесор в певних мірах. Якщо це добре оптимізована система, то вона буде використовувати не тільки для обробки зображення, але й для певних великих розрахунків [15]. Нерідко розробники ігор забувають про таку можливість, використовуючи для цього ЦП. Тому гарно налаштована комунікація цих двох компонентів породжує покращення продуктивності.

Наступними ресурсами йде пам’ять: RAM (оперативна пам’ять) та VRAM (відеопам’ять). Перший тип використовує будь-яка гра як тимчасове та доволі швидке сховище для різного роду інформації [16]. Тут зберігаються майже усі дані гри, що необхідні для подальшої обробки. Кількість оперативної пам’яті завжди обмежена, тому ще одним кроком оптимізації є зменшення використання об’ємів цього компоненту. Що ж стосується відеопам’яті – вона має аналогічний принцип роботи з попередньою, проте використовується для операцій, що пов’язанні з роботою графічного процесора. Це окремий вид пам’яті, що є в більшості дискретних відеокартах. Проте бувають і такі комп’ютери, що використовують тільки інтегровану відеокарту. В такому випадку використовується певна частина оперативної пам’яті [1].

Кожна система має також свій розмір. Сучасні ігри мають тенденцію займати гігабайти дискового простору. Частіш за все це є вагомою проблемою для користувача, адже комп’ютер містить не тільки якусь одну програму. Це система різних додатків, що також займають певні об’єми пам’яті. Тому проблема зменшення розміру ігор не є менш важливою у порівнянні з іншими.

Потрібно також не забувати про людські ресурси. Здебільшого це час, за який розробник повинен створити нову логіку, виправити помилки або ж доповнити контент новими елементами. Так це не є технічним ресурсом, проте він залишається доволі важливим.

1.1.2 Споживання технічних ресурсів підсистемами гри

Для оптимізації необхідно чітко розуміти які процеси використовують той чи інший ресурс.

Першим можна визначити центральний процесор. Зазвичай ігрові додатки використовують його для того, щоб розраховувати якісь фізичні властивості об'єктів. Наприклад для визначення зіткнення об'єктів, для застосування сил гравітації, для створення взаємодії між елементами. Також ЦП використовується для обробки певної логіки неігрових персонажей. Це може бути обробка поведінки або ж знаходження шляху для руху. Також важливим процесом є обробка усіх вхідних даних від гравця. Це можуть бути різні обробники подій, починаючи від кліків миші та натискання клавіш закінчуючи обробкою великого менеджеру різноманітних подій. Ще можна додати процес перенесення певних фізичних даних до оперативної або відео пам’яті.

Також важливим процесом є рендеринг – це обчислення, генерація та вивід візуальної інформації на екран користувача [6]. Для цього ігри використовують ресурси вже графічного процесора. Потрібно пам'ятати, що в деяких випадках можна використати ГП і для розрахунків певних даних або ж сортування різного виду масивів.

Наступним йде використання RAM та VRAM. Здебільшого в іграх використовується перший тип пам’яті для збереження інформації про кожен об’єкт на ігровій сцені. Це можуть бути позиції, стани, анімції. Також в оперативній пам’яті зберігається виконуваний код та значення його змінних. Що ж стосуєтсья відеопам’яті, то тут в більшій мірі зберігаються текстури. Інколи це сховище можна використовувати і для іншого роду інформації: проміжні результати розрахунків, великі масиви даних, буфери і так далі.

1.1.3 Навантаження на ресурси

Неефективне використання тих чи інших ресурсів може призвести до неочікуваних результатів. Найчастіше для ЦП це є використання великої кількості фізичних об'єктів зі складною системою обробки зіткнень. Або ж постійне оновлення стану всіх об'єктів на ігровій сцені. Сюди також можна додати неефективну реалізацію різних алгоритмів, що не використовує систему кешування.

Для графічного процесора це може бути високополігональні моделі для 3D ігор, або ж велика кількість спрайтів для 2D. Якщо ігрова система використовує освітлення або шейдери, то це також може стати проблемою.

Для оперативної пам’яті найбільшим навантаженням є масиви великих даних. Для відеопам’яті це частіш за все не оптимізовані за розміром текстури.

Якщо казати про людські ресурси, то сюди можна віднести неефективне використання часу розробника для створення якогось нового контенту, що здебільшого є дублікатом якогось існуючого. Або використання ручного налаштування властивостей певних об’єктів.

1.1.4 Значення ефективного управління ресурсами

FPS (frames per second) є майже головним показником продуктивності гри. Чим більший цей показник, тим кращою є система. Неоптимальне використання будь-якого з ресурсів впливає на стабільність та кількість цього критерію. Під словом «стабільність» мається на увазі ситуації, коли FPS, за різних умов коливається в певному діапазоні, без різких коливань, або критичних змін від середнього показника.

Наприклад, якщо гра перевантажує центральний процесор, це може призвести до того, що певним командам буде потрібно більше часу на виконання, що в свою чергу зменшить кількість FPS гри. Якщо ж казати про

графічний процесор, то він також впливає на результат. Якщо система надмірно використовує полігони або текстури, то це може призвести до того, що грі доведеться витрачати більше часу на обробку та формування кадрів.

При виникненні перевантаження різного роду пам'яті, можуть також виникати лаги або фризи. Це відбувається через велику затримку завантаження інформації до сховища.

Треба також пам'ятати, що ігри створюються під різні платформи: ПК, мобільні пристрої, консолі. Тому питання управління ресурсами стає загостреним. Наприклад якщо розглянути можливості мобільних пристроїв, можна побачити, що вони мають менше пам'яті, гірший ЦП та ГП ніж ПК. Все це необхідно враховувати при розробці ігрової системи. Адже перевантаження може викликати перегрівання системи, що в свою чергу може призвести до поломки компонентів. Якщо ж говорити про консолі, то тут у розробників ще більше обмежень, адже здебільшого у них фіксовані значення характеристик.

Неоптимальне використання часу розробників, може призвести до того, що сам процес розробки ігрової системи може уповільнитися, що в свою чергу призводить до порушення певних дедлайнів. Недотримання терміну може негативно вплинути на бюджет компанії і в свою чергу вплинути на подальшу розробку. Цей процес може стати циклічним. Тому оптимізація цього виду ресурсів є важливим етапом.

1.2 Класифікація підходів до оптимізації ресурсів

1.2.1 Класифікація за типом оптимізованих ресурсів

До цієї класифікації можна віднести оптимізацію обчислювальних ресурсів. Серед таких є ЦП, ГП та пам’ять RAM та VRAM. Наприклад для оптимізації центрального процесора можна використати різні підходи паралелізації розрахунків [2, 12]. Виконати зменшення кількості викликів

оновлення стану об’єктів [9], спростити певну логіку алгоритмів. Використати можливості ГП для виконання великих розрахунків.

Для оптимізації графічного процесора достатньо використовувати різні підходи зменшення часу для рендерингу. Зазвичай такий процес відбувається по різному та залежить від конкретного ігрового рушія. Проте загальними правилами є зменшення або полегшення логіки шейдерів [21], оптимізація освітлення ігрової сцени, стискання або кластеризація даних.

Що ж стосується зменшення навантаження на пам’ять, то тут існує доволі багато підходів, починаючи від оптимізації структур даних [9] до використання пулінгу об’єктів. Проте здебільшого це залежить від мови програмування та завдяки якому рушію створюється гра. Загальними правилами є зменшення кількості або стискання даних, перевикористання, вивільнення тієї інформації, що вже ніколи не буде використовуватись [9].

Також до цієї класифікації можна віднести підходи, що націлені на зменшення енергоспоживання. Особливо такий підхід важливий при розробці мобільних застосунків. Базується він на мінімізації фонового навантаження центрального процесора, зменшенню ефектів у грі та керуванням кількості FPS.

1.2.2 Класифікація за методами оптимізації

Цю класифікацію можна розбити на кілька умовних частин. Першою є алгоритмічна оптимізація. Головними підходами є зменшення складності деяких алгоритмів, використання спеціалізованих об’єктів для збільшення швидкості пошуку. Для першого частіш за все досліджують різні варіанти сортування та виконують компаративний аналіз, метою якого є визначення найшвидшого варіанту для конкретної задачі. Для другого ж підходу частіш за все досліджують які структури даних дозволяють пришвидшити пошук елементів або виконати швидку фільтрацію.

Наступною частиною можна виділити оптимізацію графіки та

рендеренгу. Це система різних підходів, що дозволить зменшити час на рендеринг одного кадру. Одним з підходів може бути зменшення кількості об'єктів, які необхідно згенерувати за один кадр. Також сюди можна включити різні підходи оптимізації створення світлових ефектів, або ж взагалі зменшити їх кількість. Ще одним підходом є техніка зменшення розміру текстур, або зменшення їх деталізацій, що залежить від дистанції об'єкта до гравця.

Іншою частиною є оптимізація фізики. Можливо тут будуть найважчі підходи. Їх ціль не тільки зменшити навантаження на різні компоненти, але й прискорити обробку фізичних властивостей об’єктів без зміни їх результату. Іншими словами, необхідно дійти до золотої середини між швидкістю та результатом. Для цього можна використовувати більш прості колайдери, для обробки зіткнень. Також сюди входять техніки для зменшення кількостей фізичних об’єктів, які необхідно опрацювати за одну умовну одиницю часу. Інколи розробники використовують підхід написання власної фізики, зі своєю логікою розрахунків руху та дії різних фізичних сил на об’єкти.

Останньою частиною можна виділити автоматизацію. Це по суті також процес оптимізації, проте здебільшого він націлений на зменшення навантаження на людський ресурс. Тут є багато підходів: створення різних допоміжних інструментів, використання можливостей рушія та його вдосконалення, створення алгоритмів для автоматичного конструювання ігрових сцен.

1.3 Сучасні інструменти та технології для аналізу та моніторингу ресурсів в ігрових системах

1.3.1 Інструменти аналізу продуктивності в Unity

Майже більшість сучасних ігрових рушіїв пропонують різні варіації інструментів для моніторингу витрачених ресурсів. Частіш за все це

деталізовані графіки, одиницею виміру яких є FPS. Вивчення такого графіку може не тільки допомогти розібратися у життєвому циклу проєкту, але й дозволяє визначити слабкі місця у програмі.

Ось наприклад Unity, має доволі багату систему моніторингу ресурсів. Основний компонент Profiler [13] має можливість одночасно відсліджувати навантаженість на центральний процесор, графічний процесор, пам’ять, дозволяють також відсліджувати рендеринг, фізику та анімацію (рисунок 1.1). Такий інструмент надає можливість розробникам виявити певні навантажені зони та за допомогою деталізованого списку викликів, дозволяє знайти які методи або класи більше всього витратили ресурсів. Також має доволі легку конфігурацію. Розробникам надано можливість обирати ті ресурси, які вони хочуть відстежувати в цей момент. Додатково можна в будь-який момент зупинити аналіз, перейти до будь-якої точки та продивитися усю інформацію про цей кадр.

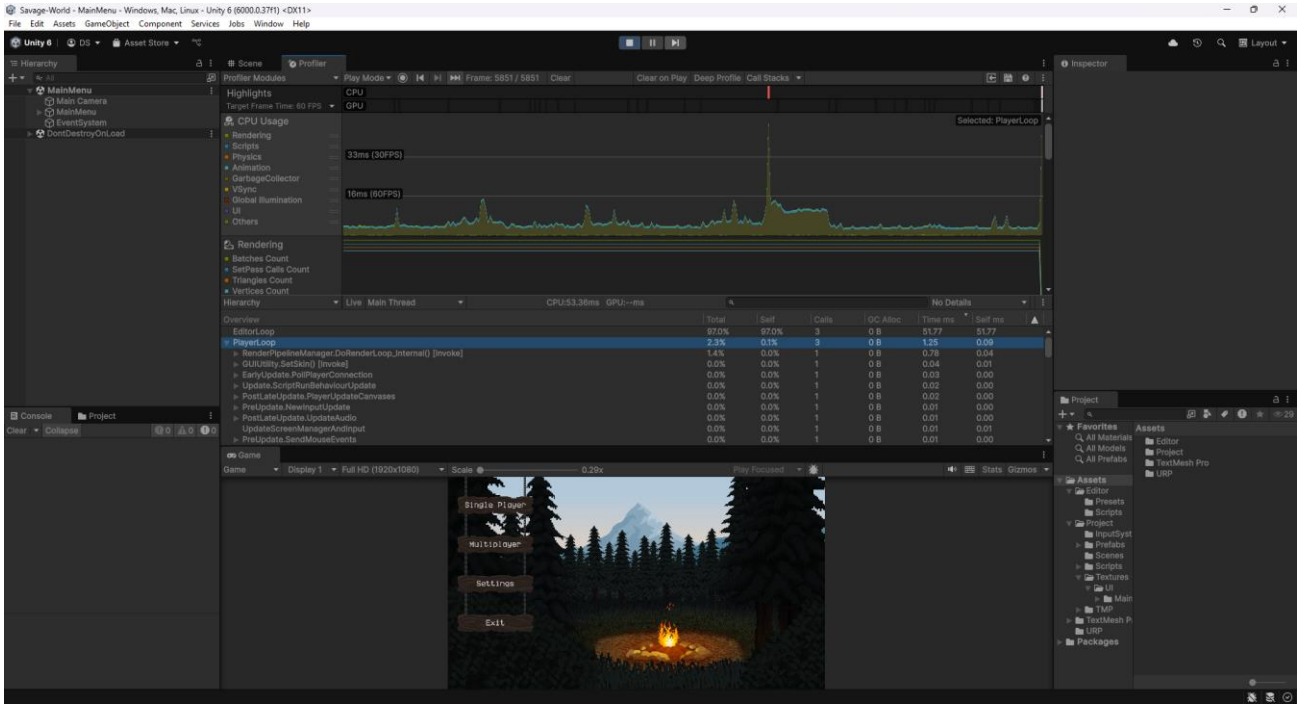

Рисунок 1.1 – Вигляд Unity Profiler

Інший доволі цікавий приклад це Frame Debugger [19]. Це інструмент, що дозволяє отримати інформацію про кожен сформований кадр та крок за

крокок проаналізувати кожну подію. Це є корисним коли розробникам необхідно визначити на якому етапі рендерингу виникають певні артефакти. Цей аналізатор складається з кількох частин: процес, що додсліджується, скроллер подій, кнопки для попередніх та наступних подій, частина вікна з ієрархією подій, та половина вікна з інформацією про подію (рисунок 1.2).

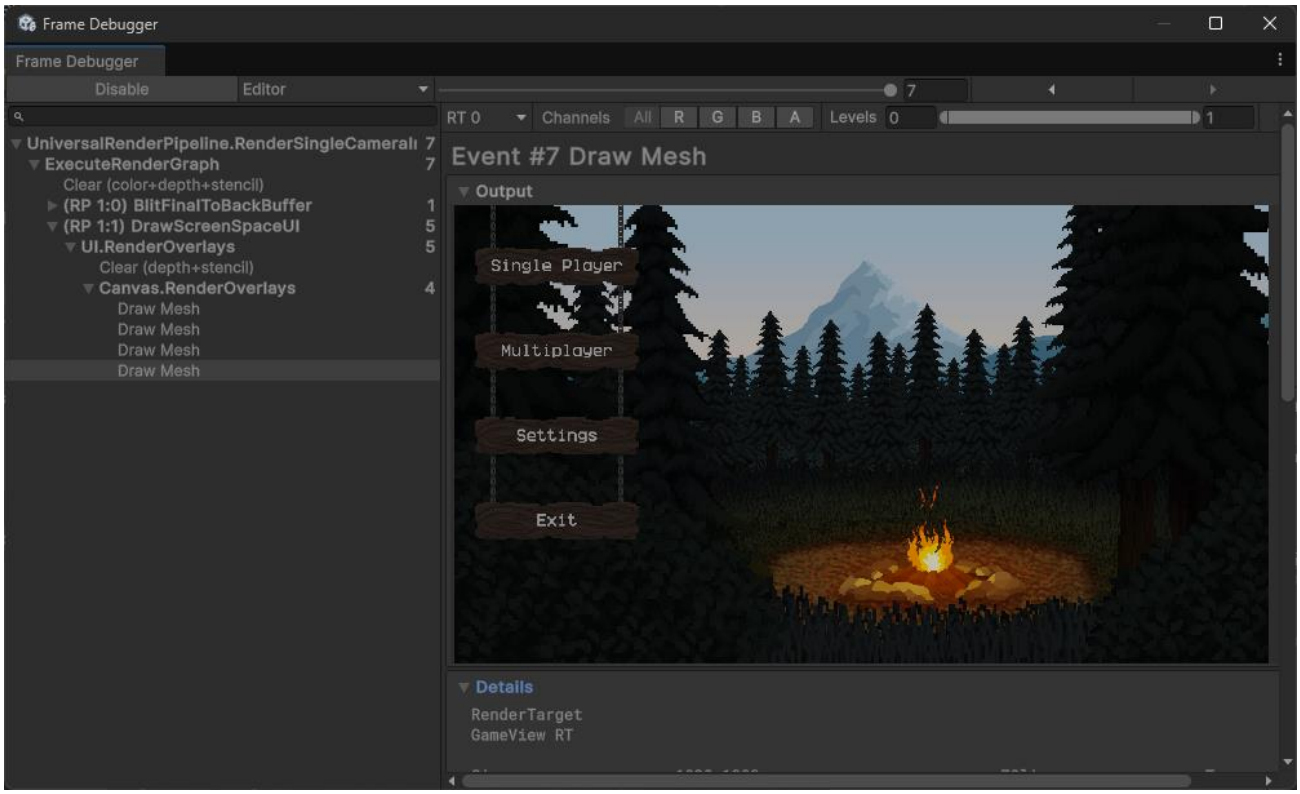

Рисунок 1.2 – Вигляд Frame Debugger

1.3.2 Інструменти аналізу продуктивності в Unreal Engine

Іншим не менш популярним представником ігрового рушія є Unreal Engine. Він містить кілька варіантів профілювання, одним з яких є Unreal Insights. Це складна система інструментів, що дозволяє збирати дані для аналізу продуктивності гри. Принцип роботи доволі простий. Він відловлює ігрові події, зберігає усю можливу інформацію про них а потім, надає певне графічне представлення з метою виявлення не оптимізованих частин [22]. Завдяки детальній інформації від офіційних джерел, такий інструмент доволі

легко налаштувати під свої потреби. Так само як в Unity, Unreal Insights містить вікно Timing Insights, що дозволяє побачити інформацію про завантаженість центрального та графічного процесорів (рисунок 1.3). Що ж стосується аналізу пам’яті, то тут також є можливість використати компонент Memory Insights (рисунок 1.4). Попри основні частини, цей інструмент також можна використовувати для аналізу навантаження мережі, завантаження даних до проєкту та іншу різну інформацію.

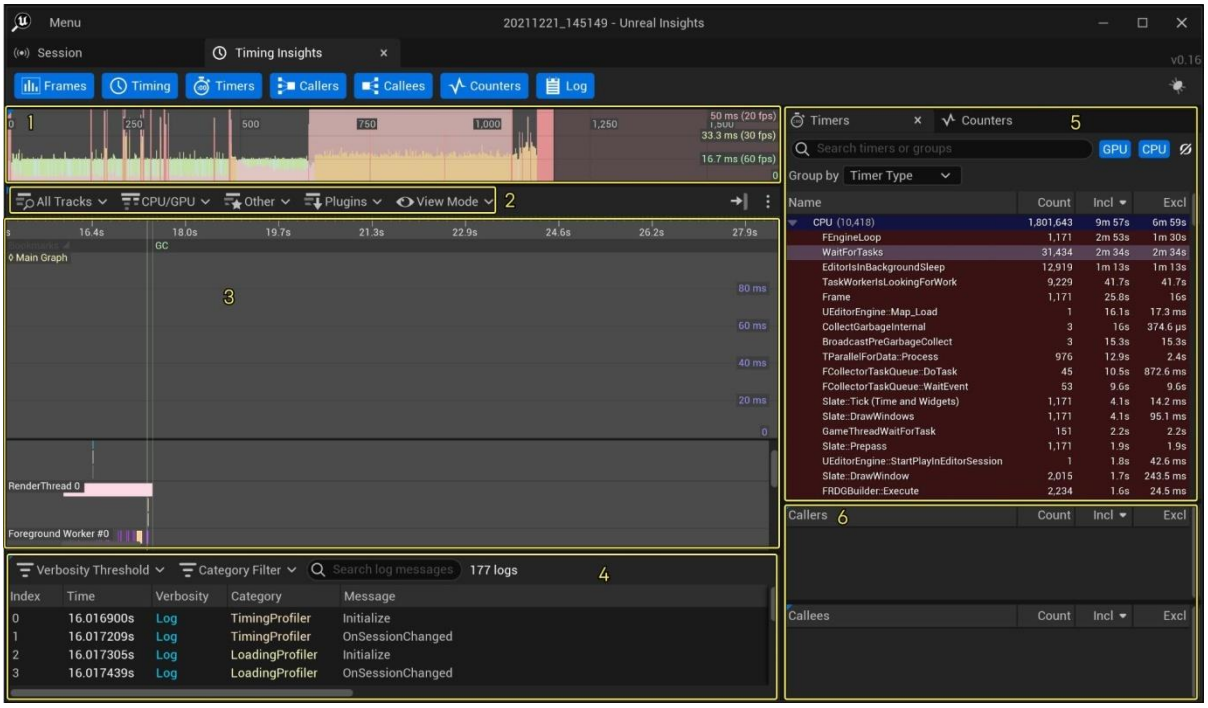

Рисунок 1.3 – Компонент Timing Insights для аналізу CPU та GPU

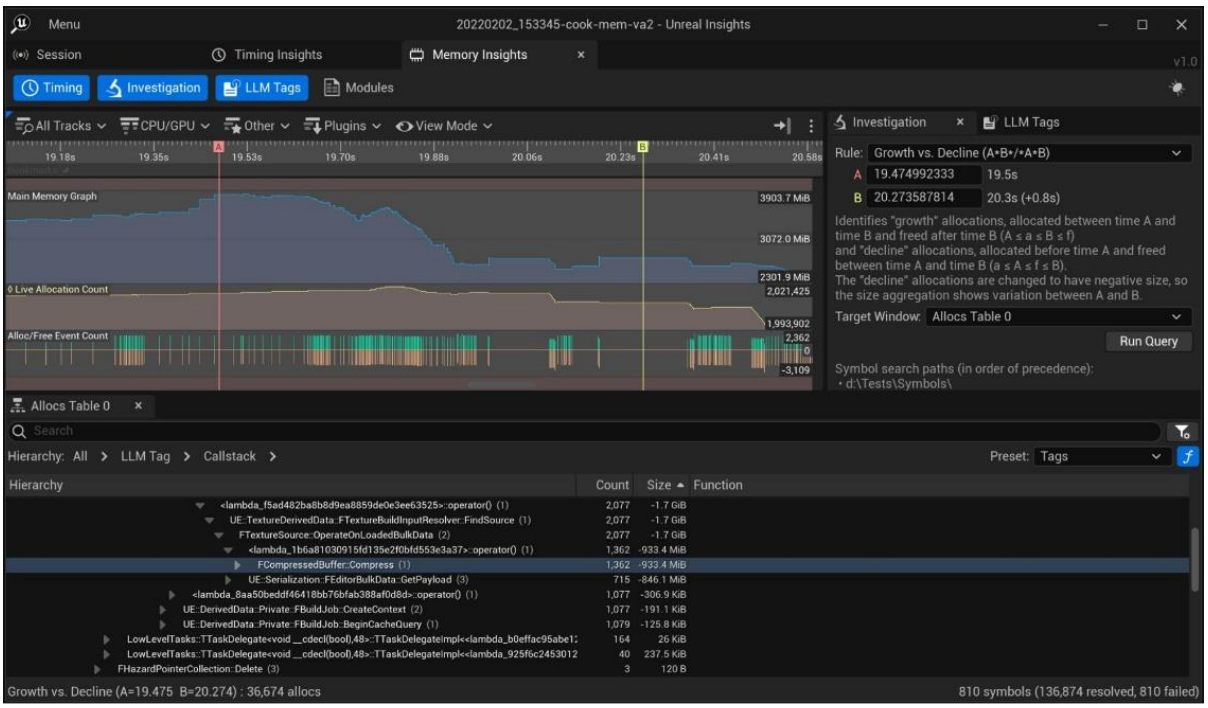

Рисунок 1.4 – Компонент Memory Insights для аналізу пам’яті

1.3.3 Сторонні інструменти для аналізу продуктивності

Якщо використання вбудованих в рушій інструментів не є достатнім, для розробників надано можливість використовувати різні сторонні програми для збору, відстеження та створення аналітичної моделі. Першим представником є NVIDIA Nsight Systems. Цей інструмент дозволяє виконувати аналіз продуктивності на апаратному рівні [11]. Тут можна побачити різного роду інформацію про події, деталі про роботу центрального та графічного процесора, роботу операційної системи та інше (рисунок 1.5). Такий інструмент здебільшого використовується для більш глибокого аналізу, адже потребує певних знань.

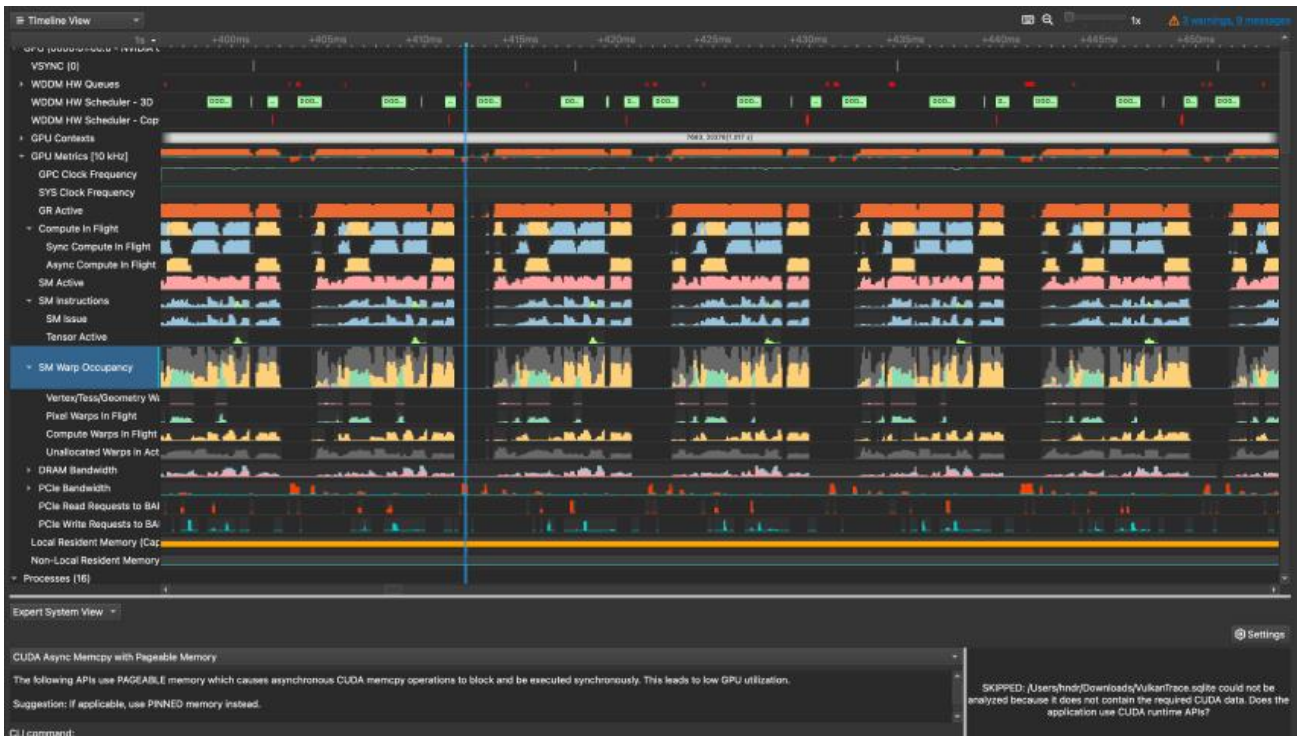

Рисунок 1.5 – Вигляд інструменту NVIDIA Nsight Systems

Іншим представником може стати RenderDoc. Це інструмент з відкритим кодом, що дозволяє відловлювати кадр та з легкістю виконати його аналіз (рисунок 1.6). Доступний під будь-яку платформу та має певну інтеграцію для роботи з додатками Unity або ж Unreal Engine. Цей інструмент надає інформацію про роботу центрального процесора, а саме скільки та який метод викликався. Дозволяє отримати певну інформацію про роботу графічного процесора, про кількість шейдерів, що були спрацьовані в поточному кадрі [17]. Також можна побачити, які об'єкти або примітиви були створені в цей момент. Також в інтернет просторі є і інші представники різних аналізаторів. Деякі націлені на дослідження тільки одного ресурсу, а бувають і такі, що використовуються для аналізу всієї системи. Вибір залежить лише від конкретних потреб

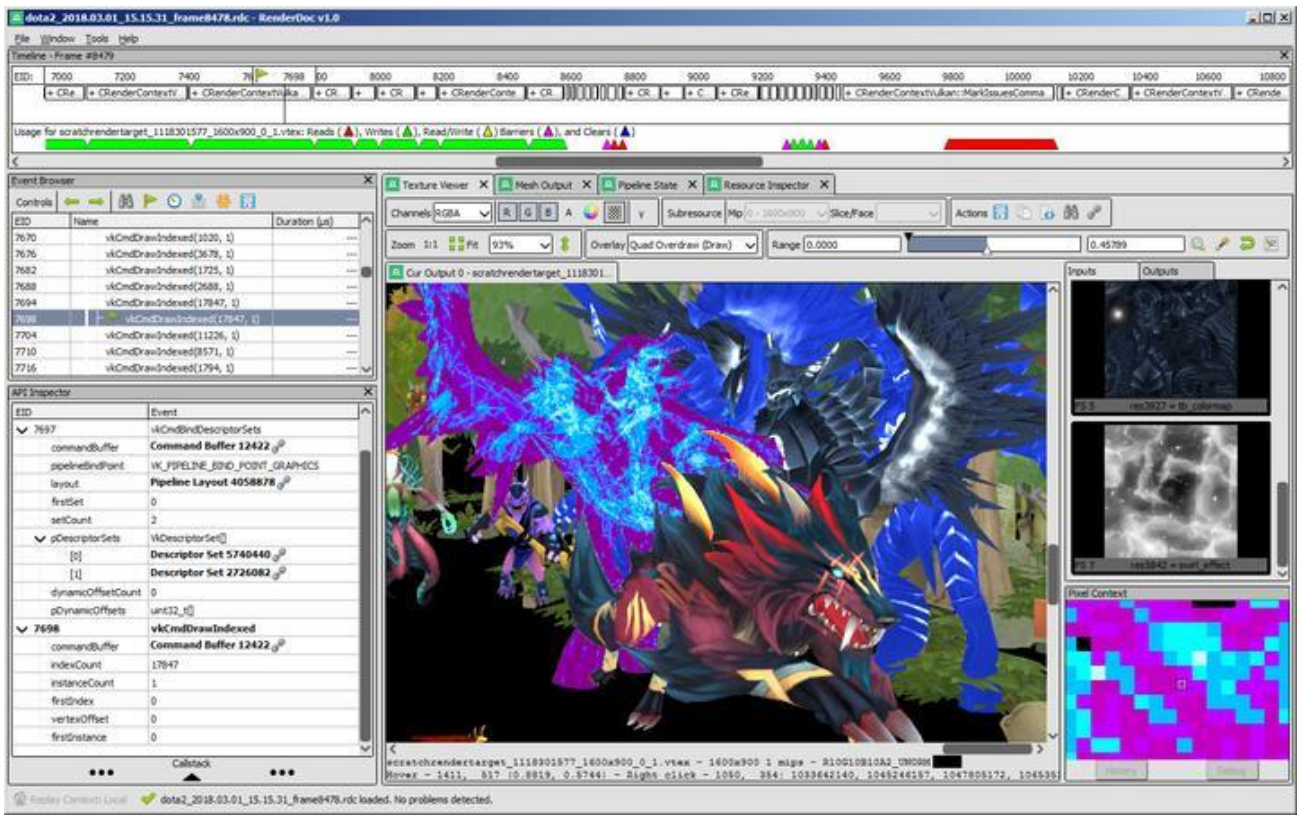

Рисунок 1.6 – Робота інструменту RenderDoc

1.4 Постановка задачі дослідження

При створенні ігрового застосунку, розробники рідко використовують методи оптимізації на початкових етапах. Проєкт стає великим через що з’являється багато залежностей, що зменшують гнучкість та продуктивність. Також часто зустрічається ситуації коли розробники спочатку роблять функціонал, а оптимізацію відкладають на кінець. Розробка за таким алгоритмом часто створює основну проблему ігор – надмірне використання ресурсів.

Використання тих чи інших методів оптимізації також залежить від специфіки гри. Якщо основною ціллю застосунку є часте оновлення освітлення або важка робота з графікою, що нерідко зустрічається в сучасних іграх, то оптимізація здебільшого виконується для конкретних ресурсів. Це пов’язано з тим, що для цього використовуються великі та важкі шейдери, або ж деталізовані текстури. Проте існують і такі додатки, що для

збереження інформації про ігрову сцену використовують великі масиви даних. В основному це також створює потребу виконувати обробку цієї системи. Не треба забувати й про візуальну якість, що також використовує певні ресурси. В такому випадку необхідно ретельно поставитися до питання оптимізації, адже залежностей між компонентами тут буде в рази більше.

Основною метою цього дослідження є створення різного роду моделей для оптимізації майже всіх необхідних ресурсів: центральний процесор, графічний процесор, пам’ять та недостатня автоматизація. Основними проблемами, що підштовхнули проаналізувати та розробити технології стало:

- надмірне використання оперативної пам’яті через збереження масиву інформації розміром більше 20 мільйонів елементів;
- велике навантаження на центральний процесор для обробки ігрового світу;
- недостатня швидкість та велике навантаження на процесор при процедурній генерації ігрової сцени;
- додаткове використання потужності центрального процесора для розрахунку мапи освітлення;
- недостатня автоматизація створення нового контенту.

2 МЕТОДИ ТА ЗАСОБИ ОПТИМІЗАЦІЇ РЕСУРСІВ В ІГРОВИХ ЗАСТОСУНКАХ

2.1 Методи оптимізації продуктивності процесора

2.1.1 Використання Job System

Нерідко при розробці комп’ютерної системи виникає задача впровадження або розробки системи для роботи з паралелізацією центрального процесора. Для цього можна використовувати вже готові бібліотеки класів, або ж вбудовані можливості ігрових рушіїв. Такоє трапляється ситуація, коли для досягнення паралелізму розробники пишуть власні рішення, використовуючи для цього бібліотеку Threads. Це є гарною практикою для того, щоб зрозуміти роботу паралелізації, проте під час роботи з цим інструментарієм часто виникають Race Conditions, незакриті потоки та Deadlocks.

Компанія Unity ретельно розглянула цю проблематику та запропонувала рішення – Job System [2]. Така система дозволяє писати не тільки багатопоточний код, але й забезпечує повну безпеку від гонок за ресурсами або ж дедлоками. Не менш корисною функцією є можливість налаштування кількості ядер, що будуть використовувати при паралелізмі.

Робота системи доволі проста – масив інформації ділиться на невеликі частини, що будуть оброблюватися власним робочим потоком (рисунок 2.1) (ще його називають Worker Thread) [2]. Цей процес відбувається автоматично на початку роботи системи. Тому розробникам немає потреби у ручному розділенні даних з метою їх паралельної обробки. Робота цього механізму доволі схожа з роботою класу Parallel з бібліотеки TPL (Task Parallel Library), що також робить певне розбиття даних на блоки та оброблює їх в різних потоках. Проте на відміну від Parallel, Job System забезпечує розробників від

гонок за ресурсами або ж ситуацій, коли один потік блокує інший.

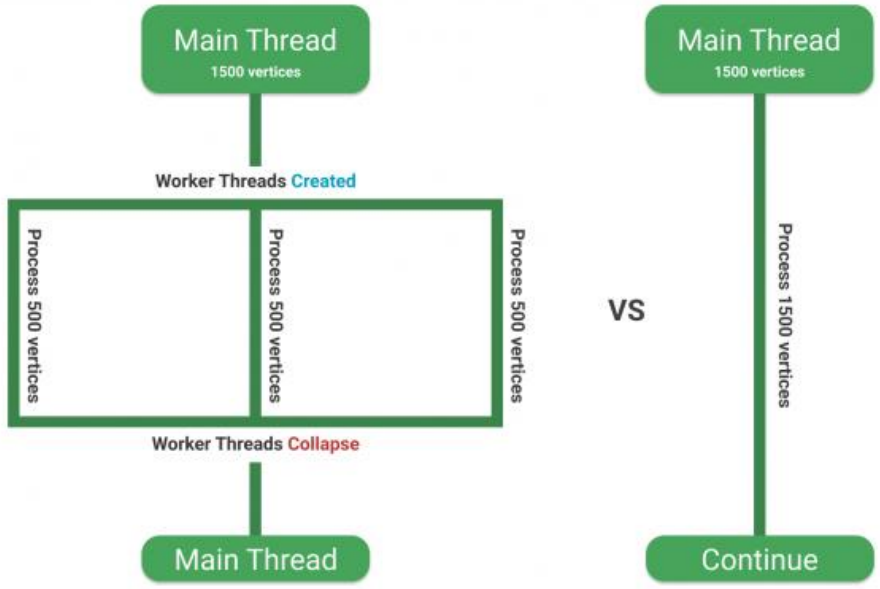

Рисунок 2.1 – Порівняння поділу задач в однопоточному режимі та за допомогою JobSystem

Для того, щоб почати працювати з цією системою, необхідно створити структуру, що вспадковується від інтерфейсів IJob, IJobParallelFor та IJobParallelForTransform [2]. Перший відповідає за стандартну роботу паралелізації. Він дозволяє створити одну паралельну задачу для одного об’єкта. Обробка буде працювати окремо від основного потоку. IJobParallelFor, дозволяє розбити масив даних на сегменти та виконувати над ними певний набір операцій. Останній тип використовується для роботи з положенням об’єктів, що робить його зручним у використанні, адже загалом зміна властивостей об’єктів Unity заборонена поза основним потоком.

Для роботи з системою Job, прийнято використовувати спеціальний та безпечний набір даних Native Array, що якраз і забезпечує в більшості випадків безпеку роботи системи. Такий об’єкт має кілька властивостей. Найбільш важливим є життя цього об’єкту: постійний, короткочасний або ж взагалі розрахований виключно для задач в Job.

2.1.2 Застосування атрибуту Burst Compiler

Для додаткового покращення та оптимізації навантаження, Unity пропонує використовувати спеціальний механізм оптимізації проміжного коду – Burst Compiler [8]. Основна його робота полягає у тому, що базуючись на певний набір даних, він оптимізує IL код та перетворює його на LLVM.

Що ж мається на увазі під певним набором даних. Для коректної роботи та взагалі для можливості роботи, розробникам необхідно дотримуватися певних правил у використанні типів даних та операцій. Найголовнішим обмеженням є використання класів. Burst забороняє використовувати будь-якого формату посилальні дані. Він працює лише з примітивами: bool, char, byte, short, int, long, float, double. Також заборонено використовувати string та decimal з причини того, що перший варіант це по суті є посилальний тип, а другий через те, що немає прямого еквівалента у машинних інструкціях. Для оптимізації, Burst Compiler також виконує перетворення векторів у формат SIMD.

Не менш важливим є можливість використання типів перелічень. Проте для такого варіанту поки що не можливо використовувати перелічення як флаги.

Говорячи про структури, що створюються розробниками, то Burst також може їх оптимізувати, за умови того, що вони використовують лише ті типи даних, які не є забороненими. Не заборонено також використовувати деякі атрибути організації даних у структурі. Burst дозволяє використовувати Generic типи структур але за умови роботи лише зі структурами.

Тепер щодо колекцій. Для роботи з ними необхідно використовувати лише Native Array, адже використання звичайних масивів не дозволяється. Тому необхідно доречно поставитись до цього питання, адже в деяких випадках копіювання даних з загального масиву до нативного може бути ресурсозатратним.

Говорячи про синтаксис, Burst підтримує майже всі основні операції.

Він може працювати з різного роду логічних операцій таких як if else statement, switch case, цикли for та while, а також операції для роботи з циклами: break та continue. Підтримує також методи, що є розширенням функціоналу. В деяких випадках можна використовувати коди, що не є безпечними за замовчуванням. Дозволено також створювати нові екземпляри структур, використовувати ключові слова ref, для передачі структури як посилання, та out для видачі структури з методу. Можна також працювати зі статичними полями, що встановлені в режим тільки для зчитування.

Burst Compiler здебільшого дозволяє працювати з виключеннями, а саме використання операції throw. Проте працювати з нею можна не на повному рівні. Попри це, працювати з блоком try catch finally – заборонено. Через це випливає заборона на цикл foreach, так як він потребує цей блок для обробки виключень.

Говорячи про інші заборони, то тут можна віднести обмеження у статичних полів. Їх не можна використовувати як такі, що не є тільки для зчитування. З цього правила також виникає інше – статичним полям не можна присвоювати нове значення. Також через те, що масиви не повинні взагалі використовуватись у звичайному для них вигляді, то обробка або отримання доступу до них з різних частин коду є забороненою.

Виходячи з цього великого набору правил, перед розробниками постає питання, а коли ж доречно використовувати такий атрибут. Відповідь проста – здебільшого він створений на роботу з Job System [23]. Адже половина умов буде автоматично виконана при роботі з цією системою.

Для початку роботи, необхідно створити структуру екземпляр, що вспадковується від одного з інтерфейсів Job System. Після цього, необхідно чітко пройтись по всім обмеженням та додати лише необхідні дані. Останнім кроком є використання атрибуту BurstCompile над структурою. Такий процес дозволить оптимізувати за можливості код, що зробить його в рази швидшим. Якщо ж розробник допустив на якомусь етапі помилку, то атрибут буде попереджати, що в структурі використовуються елементи, що є

недозволеними. В такому випадку функціонал Burst буде припинено, і замість нього буде використовуватись неоптимізований IL код.

2.1.3 Робота з життєвим циклом в Unity

Розуміння життєвого циклу в Unity не тільки дозволяє уникнути певних проблем з ініціалізацією та обробкою даних, але й дозволяє чітко виявити місця до яких необхідно поставитись доречно. Найголовнішими функціями, з якими необхідно вміти працювати – Update та FixedUpdate (рисунок 2.2). Перший метод викликається кожен ігровий кадр, що робить його в більшості випадків нестабільним. Наприклад, якщо гра працює в режимі 60 кадрів на секунду, то це означатиме, що система буде викликати функцію Update рівно 60 разів на секунду. Другий варіант оновлення є закріплений за часом. В більшості випадків він позначається як fixedDeltaTime та за замовчуванням дорівнює 0.02 секунди. Це означає що кожну секунду, Unity буде стабільно викликати функцію рівно 50 разів незалежно від кількості ігрових кадрів.

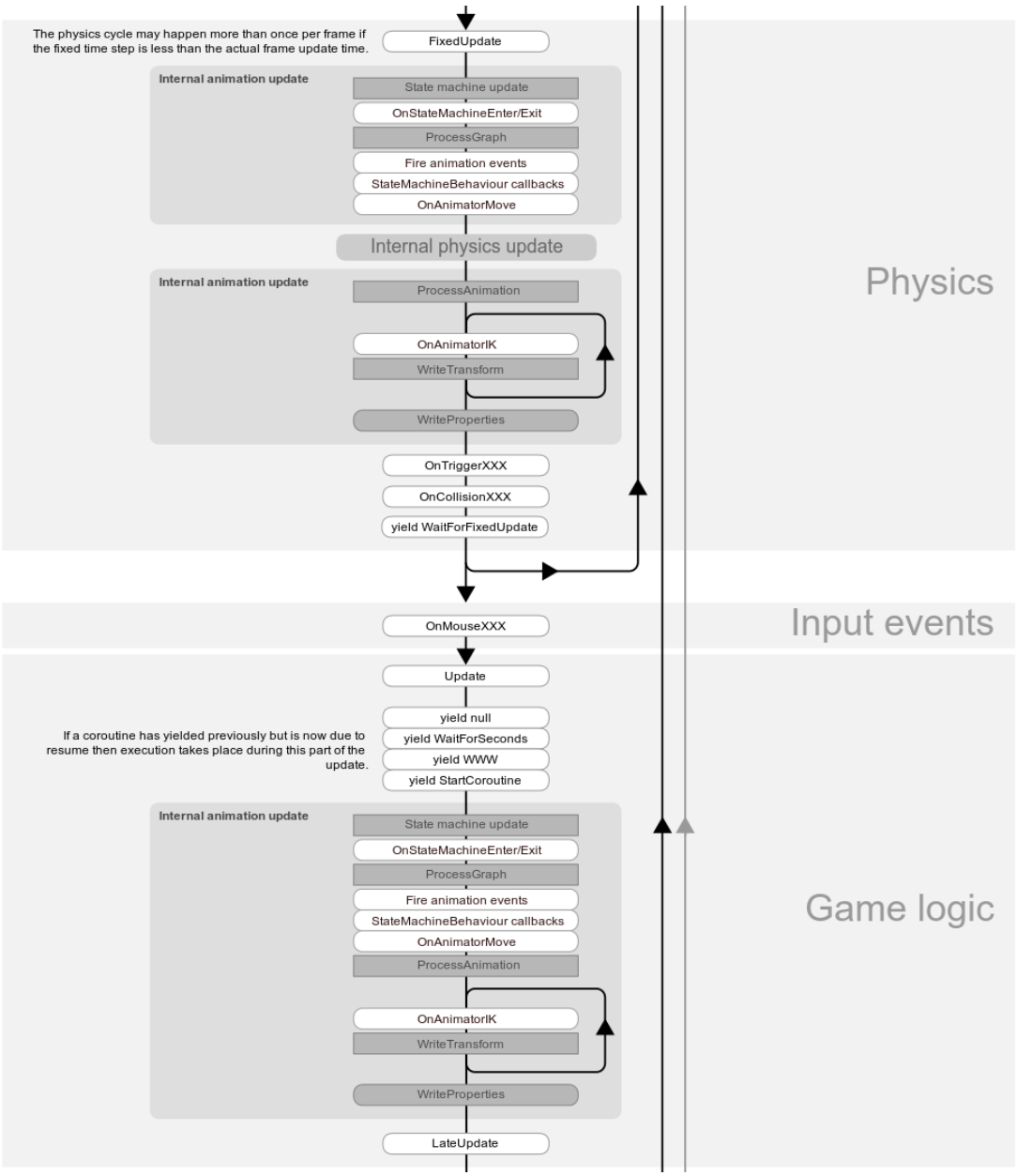

Рисунок 2.2 – Цикл виклику функцій FixedUpdate та Update

Головними рекомендаціями у роботі з цими двома методами є те, що основні фізичні властивості об’єктів, а саме переміщення, робота з Rigidbody і так далі, відбувається на момент FixedUpdate. Це виникає через потребу у стабільній обробці фізичних параметрів та застосуванню їх незалежно від кількості ігрових кадрів. Адже якби фізика використовувала залежний від кадрів варіант, то рух об’єктів був би не гладким та не стабільним, а в деяких випадках був би різким або періодичним. Говорячи про метод Update, то тут доречніше буде використовувати лише ті методи, які працюють з графікою,

текстом та відображенням об’єктів.

Чим доречніше обрана функція, тим швидше та безпечніше буде працювати код. В деяких випадках чітке розуміння та дотримання певних умовних обмежень може призвести до певного роду оптимізації не тільки центрального процесора, але й інших компонентів системи.

2.1.4 Використання GPU для великих розрахунків

Нерідко виникає проблема, коли використання паралелізації центрального процесора є недостатнім для вирішення однієї задачі. На допомогу цьому приходить рішення роботи та обробки даних з використанням графічного процесора.

Unity дозволяє легко та просто створити розрахунковий шейдер (програма для обробки даних на графічному процесорі) та передати туди необхідні дані. Називається він Compute Shader [18]. Така програма використовує шейдерну мову HLSL, що робить її в деяких випадках схожу на програми написані на мові програмування C або C++.

Для того, щоб почати працювати з цим інструментом, в Unity необхідно створити екземпляр цього шейдеру. Зайшовши до редактору коду, необхідно визначити які функції буде видно для програм на центральному процесорі. Робиться це за допомогою директиви pragma kernel та додавання назви методу.

Синтаксис програми не є надто складним. Загальні типи даних майже такі самі як і в мові C#, проте також є і окремі структури інформації, такі як float3 або float4, що можна представити як тип даних Color, що містить RGB або RGBA значення кольору пікселя.

Для того, щоб використовувати створений шейдер, необхідно вже в ігровому об’єкті створити екземпляр шейдеру, вказати всі покажчики на необхідні методи та створювати набори даних, що будуть передаватися з центрального процесора до пам’яті графічного.

Також необхідно пам’ятати про дві властивості такої паралелізації – вибір розміру сітки для функцій на стороні GPU та розмір сегментів даних на стороні CPU. Досягти максимального успіху допоможуть лише експериментальна практика. Пов’язано це з тим, що не має якогось конкретного обмеження з приводу розмірів. Це вирішується на етапі конструювання системи.

2.2 Методи оптимізації взаємодії об'єктів.

2.2.1 Pooling об’єктів

Найкращий та найпростішим методом оптимізації об’єктів є використання пулінгу об’єктів. Основна задача такого процесу – збереження неактивних об’єктів з метою подальшого перевикористання у проєкті.

Уявімо ситуацію. На сцені присутні 10 ворожих неігрових персонажей (надалі буде використано термін NPC). Кожен такий NPC повинен створювати об’єкти, що націлені на гравця. При зіткненні такого тіла з персонажем, він повинен зникати та виконувати якусь дію з гравцем. Якщо такий процес відбувається надто часто, то постійне створення нового об’єкту, збір пам’яті та виділення нової пам’яті може витрачати додаткові ресурси, які можна витратити на інші процеси. В такому випадку на допомогу можна використати пулінг об’єктів. Структура такої системи доволі проста (рисунок 2.3). В якості ядра використовується швидка колекція, що зберігає усі неактивні об’єкти, які можна потім перевикористати. Розмір такого масиву може бути як статичний, так і динамічний. Робота з тілами в такому випадку буде трохи відрізнятись. Якщо масив статичний, то для отримання першого вільного об’єкта достатньо лише пошукати його в колекції. У протилежному випадку, необхідно виконувати створення нового екземпляру, у разі якщо вільних об’єктів більше не існує.

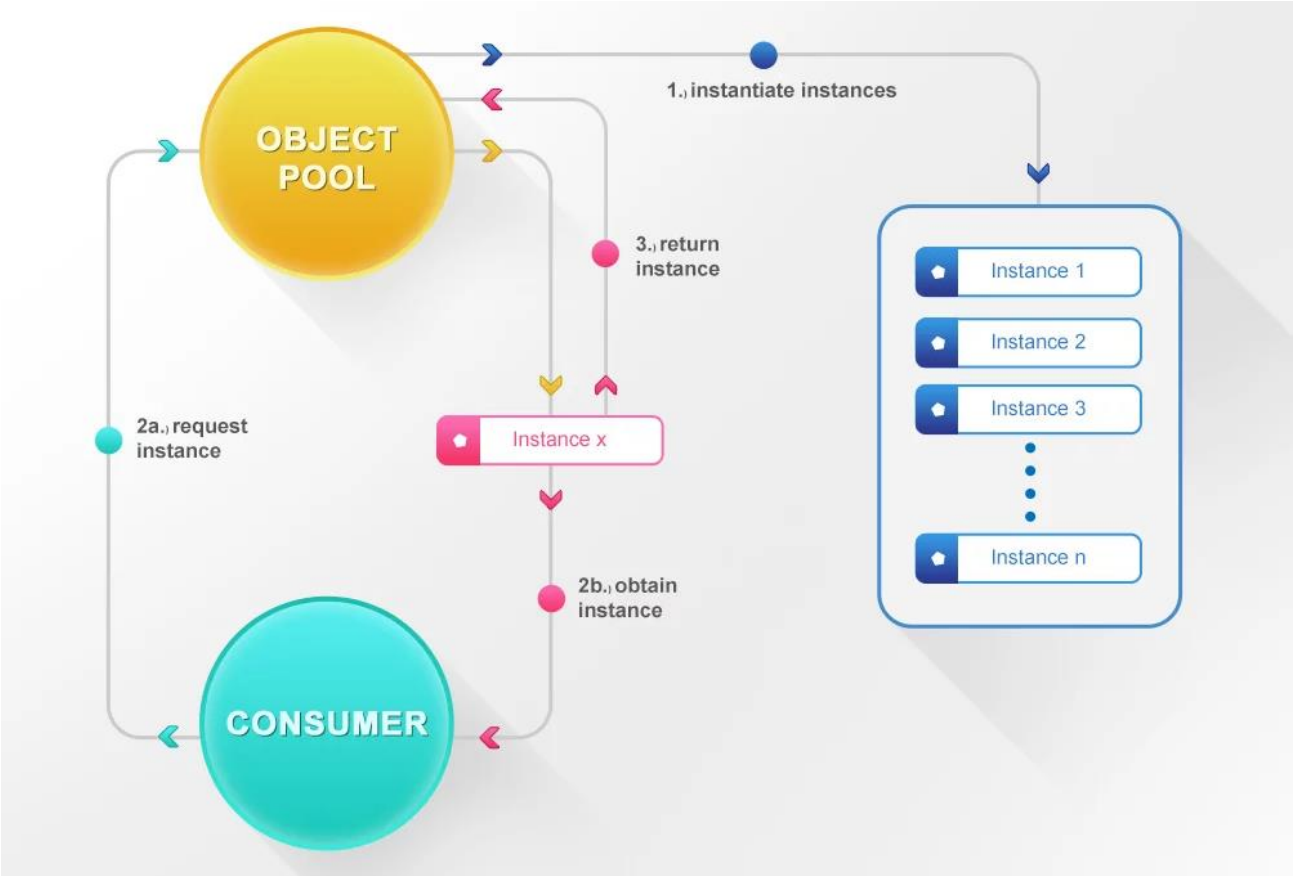

Рисунок 2.3 – Графічне представлення алгоритму роботи пулу об’єктів та його структури

Кожен варіант масивів має як свої плюси так і мінуси. Наприклад при використання динамічних колекцій, розробникам немає потреби піклуватися про те, що в якийсь момент буде недостатньо вільних об’єктів. Але з іншого боку, при короткочасному масштабному використанні, це може призвести до того, що в пам’яті будуть міститися об’єкти, що майже ніколи не використовуються. При роботі зі статичними масивами, така проблема зникає, проте для коректної роботи, необхідно придумати певний механізм очікування у випадку, якщо в пулі немає вільних об’єктів.

2.2.2 Оптимізація обробки інформації

Доволі часто при розробці ігрової системи, неоптимізованим рішенням може стати обробка елементів гри. Часті оновлення фізичних властивостей, постійні розрахунки зіткнень, обробка даних, що не знаходяться в полі зору гравця – шлях до використання надлишкових ресурсів. Для уникнення таких проблем є ряд підходів.

Одним з таких є Occlusion culling (рисунок 2.4) [20, 24]. Використовується він для того, щоб зменшити навантаження на процесор виконуючи вилучення певної групи елементів з рендеренгу. Іншими словами це алгоритм, який дозволяє зробити об’єкти, які не знаходяться в полі зору гравця неактивними. Робиться це з метою зменшення кількості даних, що оброблюються як центральним процесором під час обробки інформації, так і графічним, під час створення зображення. Такий підхід можна вбудувати в будь-якому ігровому рушії, адже це є загальним правилом для усіх.

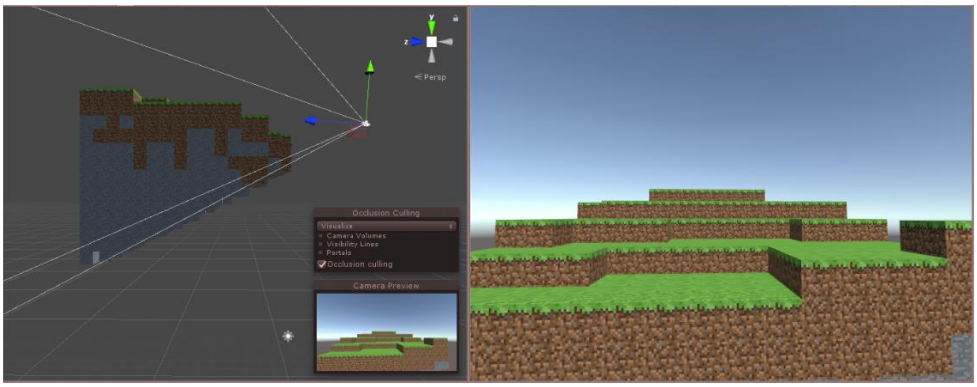

Рисунок 2.4 – Приклад роботи алгоритму Occlusion Culling

Принцип роботи доволі простий. Необхідно розрахувати, які елементи знаходяться за межами головної камери, після чого вимкнути рендеринг цих об’єктів. Такий підхід дозволяє оптимізувати кількість фізичних розрахунків для системи об’єктів.

2.3 Автоматизація процесів та оцінка оптимізації ресурсів

2.3.1 Custom Editors, як інструмент покращення рушія

Під час налаштування системи, перед розробниками доволі часто постає проблема недостатньої автоматизації або ж неструктурованості даних для конфігурації. Для вирішення цієї задачі, компанія Unity надає можливість змінювати ігровий рушій під власні потреби.

Одним з таких підходів є написання редакторських скриптів, що дозволяють додавати нові та корисні елементи до інтерфейсу. Така технологія надає розробникам можливість структурувати інформацію за потреби, додати певного роду кнопки для роботи або ж автоматичного конфігурування [10].

Для того, щоб почати користуватися технологією, необхідно для початку чітко визначитись під яку групу об’єктів буде використовуватись користувацький редактор. Після цього, необхідно створити клас, що буде вспадковуватись від класу UnityEditor.Editor. Це дозволить перезаписати функції, що використовуються під час створення редакторів в Unity. Головною умовою такого підходу є місце де знаходяться скрипти. Для коректної роботи та створення видимості нових редакторів, файли з кодом повинні зберігатися в папці Editor. Ця папка не є в наявності за замовчуванням, тому її частіш за все створюють власноруч.

На сьогодні, Unity підтримує два варіанти створення редакторів: legacy та з використанням новітнього підходу – UI Builder (рисунок 2.5). Перший варіант частіш за все використовується в старих системах. Другий же надає доступ до зручного інтерфейсу, де розробники у режимі реального часу можуть створювати UI та налаштовувати стилі його елементів. Такий процес можна порівняти зі створенням веб сторінки, використовуючи HTML та CSS.

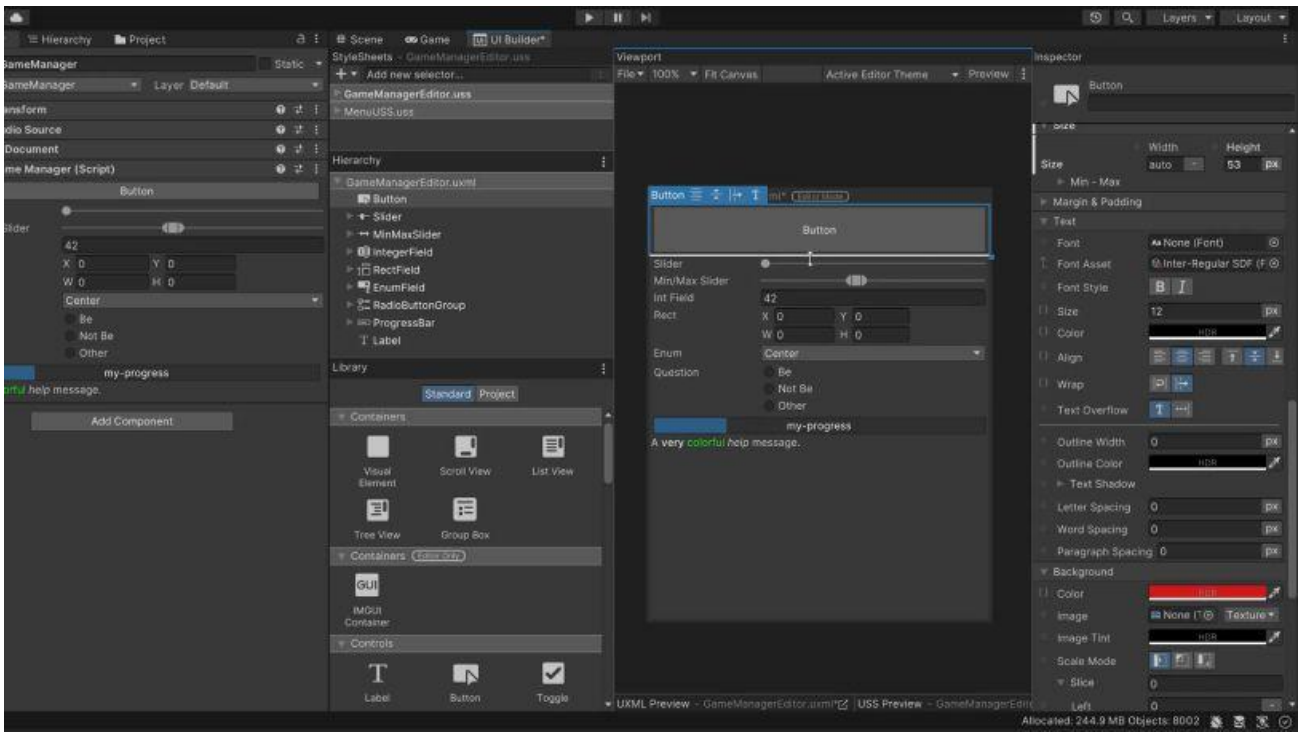

Рисунок 2.5 – UIBuilder від компанії Unity

Доволі часто при роботі з таким підходом, об’єкти повинні бути можливими для серіалізації. Для цього над необхідними для обробки полями класу необхідно поставити атрибут SerializeField, що зробить його видимим при обробці інформації у редакторі.

2.3.2 Розробка Editor Window, для налаштування та маніпуляцій даними

Розробка звичайних редакторів для конкретних об’єктів вже є непоганим результатом. Таке рішення в деяких випадках допомагає зробити інформацію про об’єкт більш структурованою та інтуїтивно зрозумілою. Такий підхід дозволяє значно зменшити час, що витрачається розробниками на ручну монотонну роботу, та прискорити сам процес розробки ігор.

Проте, для організації цілої системи різного роду об’єктів, на допомогу приходить створення вікон редакторів (рисунок 2.6). Це інший підтип інструментів в Unity. Основна його задача – створення нового вікна для

маніпуляцій даними.

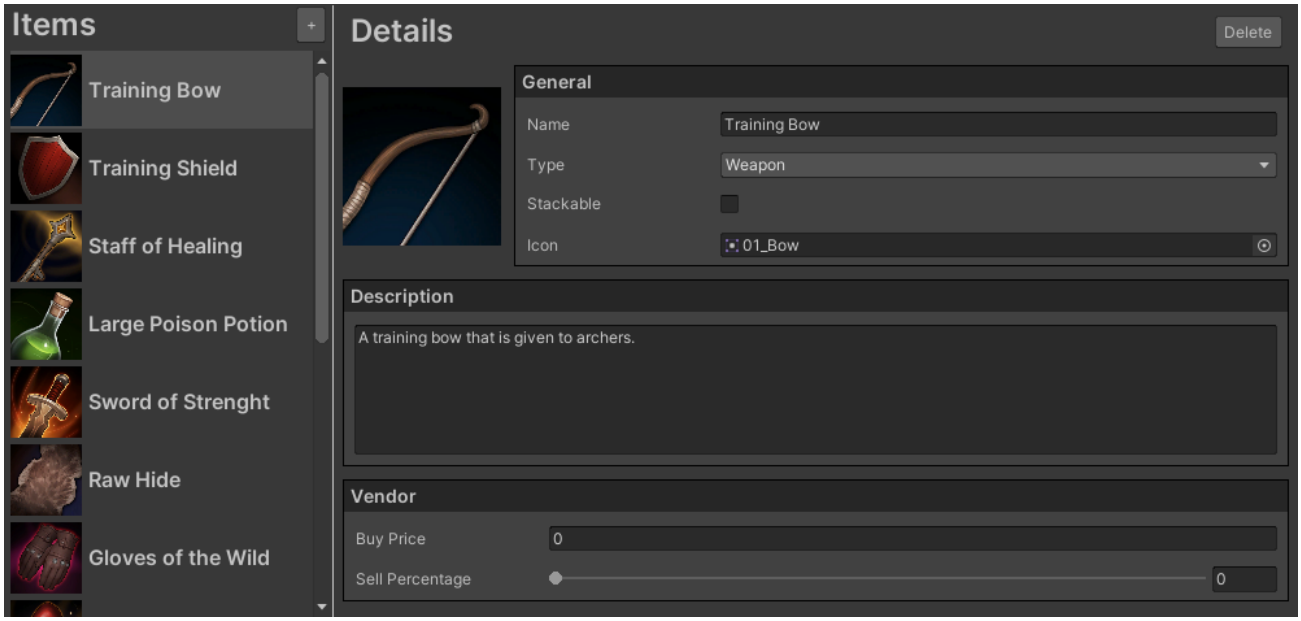

Рисунок 2.6 – Приклад використання Editor Window для покращення роботи із системою об’єктів

Таке рішення можна застосовувати в будь-якому напрямку розробки, починаючи від графічного представлення алгоритмів, закінчуючи менеджером об’єктів і так далі. За складністю написання та реалізації такого типу інструментів, розробникам необхідно мати певне уявлення про кодову складову системи. Документація Unity збагачена інформацією про можливості та функціонал, проте найкращим методом є експерименти.

Для того, щоб почати працювати з вікнами, необхідно створити в папці Editor клас, що буде вспадкований від класу UnityEditor.EditorWindow. Після цього як у варіанті з редакторами, є можливість використовувати два формати створення – застарілий (більше підходить для старих версій Unity) та новітній із використанням UI Builder.

2.3.3 Атрибути та їх графічне представлення для швидкої роботи з об’єктами

Інколи, замість ручного налаштування редакторів або вікон, буває необхідно якось швидко та просто додати користувацький елемент інтерфейсу. На допомогу у вирішенні цієї проблеми можна використовувати комбінацію методів створення атрибутів для елементів класів або структур, та спеціальний інструмент PropertyDrawer.

Таке рішення, дозволяє швидко та просто створювати власний інтерфейс для полів, методів та властивостей. Для того, щоб почати роботу, необхідно створити власне атрибут. Робиться це із використанням бібліотеки System. Загалом атрибути можна робити під будь-які об’єкти: поля, властивості, методи. Головною задачею атрибутів є надання певного додаткового контексту тим чи іншим елементам та обробка цього контексту в різного роду обробниках.

Таким обробником у цьому випадку є PropertyDrawer. Клас, що був створений для розробки користувацького інтерфейсу для конкретних об’єктів (рисунок 2.7). В нашому випадку такими об’єктами будуть атрибути.

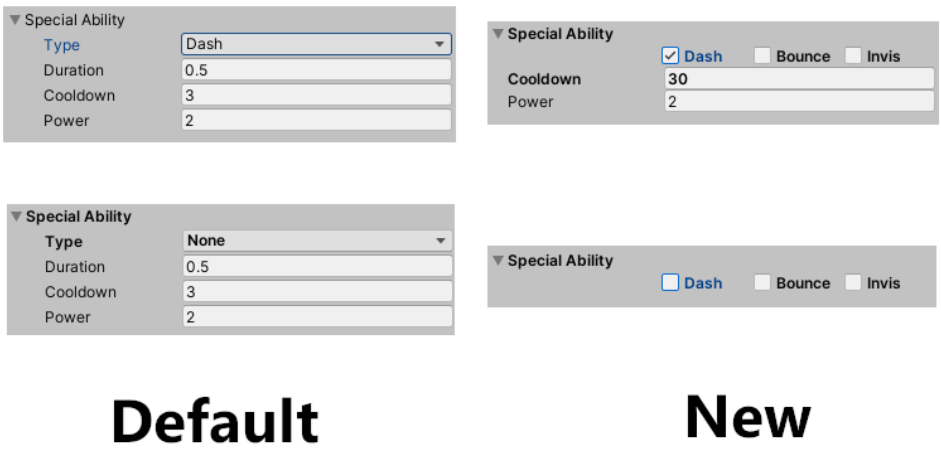

Рисунок 2.7 – Приклад використання PropertyDrawer

Використовуючи таку комбінацію, можна швидко та легко впорядковувати навіть цілу систему об’єктів, для цього необхідно лише створити атрибути для контейнеризації елементів та у PropertyDrawer виконати логіку їх об’єднання в один візуальний об’єкт (рисунок 2.8). Іншим можливим рішенням може бути використання методів як кнопок, які можна натиснути та перевірити роботу тих чи інших алгоритмів.

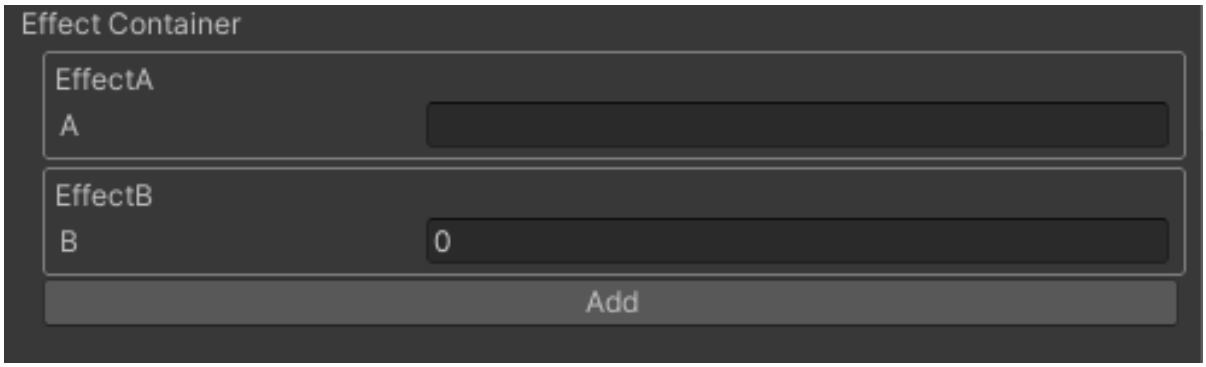

Рисунок 2.8 – Контейнеризація об’єктів за допомогою PropertyDrawer

3 МОДЕЛІ ОПТИМІЗАЦІЇ РЕСУРСІВ В ІГРОВИХ ЗАСТОСУНКАХ

3.1 Опис наукової проблеми

Створення ігрового застосунку часто супроводжується не оптимізованими процесами, що надмірно можуть використовувати не тільки можливі ресурси девайсу, але й людські ресурси – час. Головними помилками, що розробники допускають є конфлікти інтеграції вбудованих технологій, використання повільних алгоритмів, неорганізовані структури даних.

Першою на розгляд проблемою є використання освітлення в ігрових системах. Більшість ігрових рушіїв мають власні технології, що використовують різні методи створення мапи світла для ігрової сцени. Проте частіш за все вони зосередженні на використання для 3D проєктів. Якщо ж виконується інтеграція до 2D перспективи, то тут виникають певні конфлікти, що супроводжуються проблемами з оптимізацією, через відмінності логіки роботи та обробки даних. Особливо це помітно при конкретних жанрах, де ігровий світ може динамічно змінюватись.

Наступною проблемою є оптимізація ресурсів для ігор, що використовують велику кількість елементів. Для цього дослідження такою кількістю є масив розміром від 20 до 100 мільйонів елементів. Частіш за все, через відсутність певного досвіду у розробці такого роду ігор, розробники не розуміють як правильно необхідно налаштувати свою структуру даних, щоб вона використовувала якомога менше пам’яті, проте не втрачала своєї інформативності. Також, необхідно пам’ятати, що робота з легкими та простими даними дозволяє зменшити навантаження не тільки на оперативну пам’ять.

Іншою проблемою є недостатньо оптимізовані для розробників інтерфейси взаємодії з великими системами об’єктів. Це доволі поширена проблема, адже здебільшого ігрові рушії пропонують базові та прості вікна

налаштувань. В деяких випадках це є корисним. Проте, якщо один об’єкт містить багато різного роду властивостей, його вигляд буде хаотичним, а налаштування займатиме надмірний час у розробників. Також необхідно пам’ятати про те, що в ігровій системі таких об’єктів може налічуватись від тисячі до десятків тисяч. Це спричиняє певні проблеми пошуку за критеріями, за властивостями і за характером поведінки.

3.2 Модель комбінації CPU та GPU для паралельної обробки освітлення

Більшість сучасних ігрових рушіїв мають власну систему освітлення. Частіш за все вони вже є оптимізованими. Але трапляються випадки, коли їх використання не є доречним у певному контексті. Так от наприклад система URP (Universal Render Pipeline) від компанії Unity за середніми показниками є кращим рішенням. В Unreal Engine також є своя система – Global Illumination. Проте більшість з них використовуються для 3D ігор. У випадку коли гра має перспективу 2D, вони стають більш обмеженими у використанні. Розглядаючи систему URP, вона має певний рівень інтеграції з таким варіантом ігор, проте здебільшого не надає достатньої гнучкості та оптимізації. Для подальшої розробки та модифікації системи було проведено компаративний аналіз технології URP та власно створеної (таблиця 3.1) з метою не тільки створити оптимізований продукт, але й такий, що не буде гіршим за якістю результату ніж вбудований. Для цього необхідно чітко визначитись з головними критеріями порівняння:

- сумісність з 2D іграми – дозволить визначити рівень можливої інтеграції до конкретних проєктів;
- гнучкість – дозволяє визначити цільову платформу;
- масштабованість – рівень можливості для виконання певних змін від розробників;
- легкість впровадження – ступінь простоти впровадження системи;
- рівень оптимізації – показник, що дозволяє визначити рівень

оптимізації системи в 2D іграх;

- динамічність – можливість системи динамічно оновлювати освітлення через постійні зміни ігрової сцени.

Таблиця 3.1 – Порівняльна таблиця загальних характеристик URP та піксельного світла

Критерії порівняння	Технологія піксельного освітлення	Universal Render Pipeline
1	2	3
Сумісність з 2D іграми	Повний рівень сумісності. Технологія створена конкретно для 2D-ігор	Сумісна частково. Надає не всі можливості у порівнянні з 3D.
Гнучкість	Середня. Створена для роботи під платформу PC та для виконання конкретних задач. Має можливість бути інтегрованої до інших платформ з мінімальними змінами	Висока. Здебільшого створена для роботи на різних платформах. Не має чіткої можливості використання під конкретну задачу
Масштабованість	Висока. Дозволяє змінити/ удосконалити технологію під власні потреби	Середня. Дозволяє змінювати систему, проте потребує більших знань.
Легкість впровадження	Середній рівень. Технологію доволі просто впровадити, проте необхідно чітко визначитись з даними.	Легкий рівень. Доволі легко впровадити у проєкт. Надає можливості використання готових об’єктів/ компонентів

Продовження таблиці 3.1

1	2	3
Рівень оптимізації	Високий. Пов’язане це з тим, що система націлена на роботу з фрагментом масиву інформації, та використовує різні методи паралелізації розрахунків.	Середній. Оптимізоване здебільшого під 3D проєкти. Також має певний рівень оптимізації для 2D, за умови статичності ігрової сцени.
Динамічність	Повністю динамічна система. Реагує на будь-які зміни від гравця та змінює світлову мапу відносно цих змін.	Здебільшого передбачена на використання статичної сцени. Важко налаштувати для реагування на дії гравця

В результаті порівняння можна зробити висновок, що технологія URP здебільшого використовується в 3D перспективі. Є певна інтеграція з 2D, проте вона знаходиться на базовому рівні та здебільшого потребує статичної сцени. Якщо ж гра повинна динамічно змінювати ігровий світ – налаштування та використання можуть стати проблематичними. Порівнюючи роботу систем, можна сказати, що URP (рисунок 3.1) та власна модель (рисунок 3.2) надають схожі графічні результати.

Рисунок 3.1 – Приклад освітлення з використанням URP

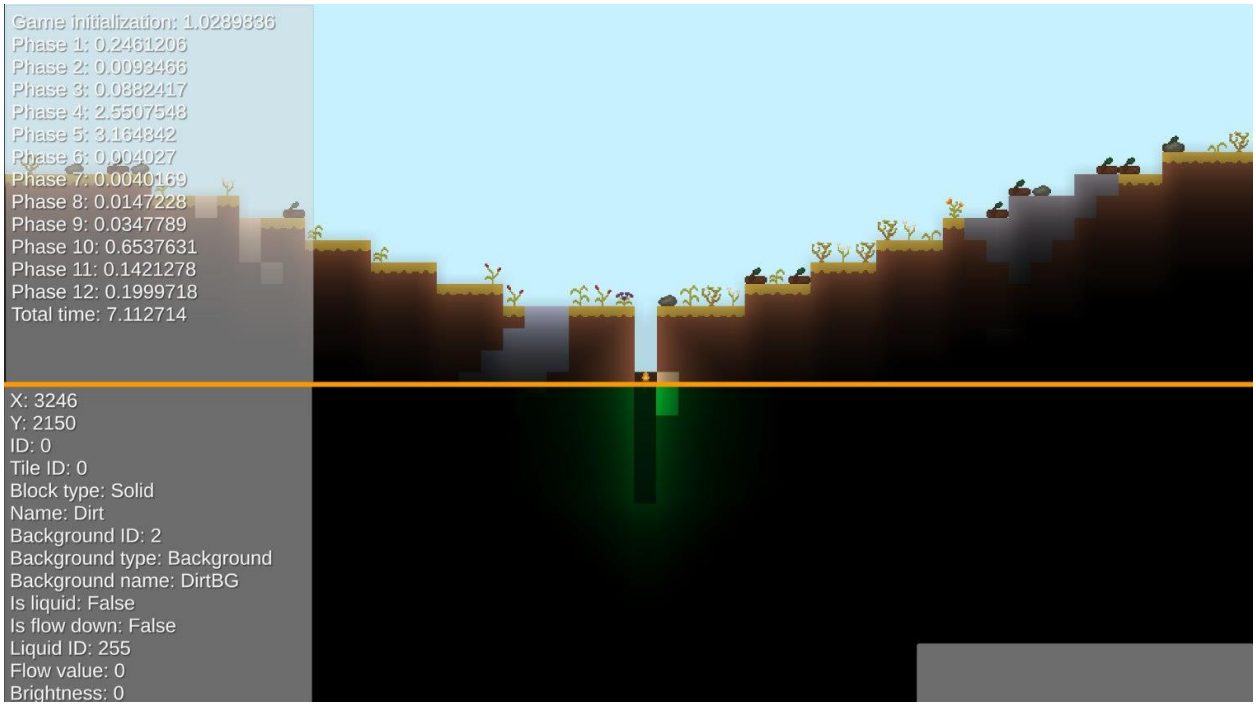

Рисунок 3.2 – Приклад освітлення з використанням створеної технології

Так як графічні результати можна вважати гарними, необхідно більш детально розглянути оптимізацію. Виходячи з порівняльного аналізу, можна зробити висновок, що система URP загалом є оптимізованою для 3D ігор або 2D за умови статичності ігрової сцени. Якщо ж сцена є динамічно змінюваною, то це може призвести до надмірного використання певних ресурсів. Виникає це через потребу постійного оновлення великої кількості

світлових об’єктів. Для визначення того, яка система є більш оптимізованою, необхідно також провести аналіз (таблиця 3.2) використовуючи різного роду критерії.

Таблиця 3.2 – Порівняльна таблиця оптимізаційних якостей URP та піксельного світла

Критерії порівняння	Технологія піксельного освітлення	Universal Render Pipeline
1	2	3
Навантаження на CPU	Низьке. Технологія використовує Job System для паралелізації та атрибут BurstCompile для оптимізації IL коду	Помірне. Потребує більше ресурсів при великій кількості об’єктів.
Навантаження на GPU	Середнє. Система використовує ComputeShader для виконання більшості розрахунків, та виконує розділення даних між потоками.	Високе. Особливо навантаження відбувається при великій кількості джерел світла.
Навантаження на RAM	Низьке. Система майже не використовує оперативну пам’ять для обробки даних.	Низький. Технологія здебільшого не використовує RAM.
Навантаження на VRAM	Помірне. Система використовує єдиний буфер фіксованого розміру та один RenderTexture, що дозволяє напряму змінювати пікселі текстури.	Високе. Особливо при великій кількості різного роду світлових об’єктів.

Продовження таблиці 3.2

1	2	3
Продуктивність при великій кількості джерел світла	Висока. Система працює з усіма джерелами під час єдиної обробки	Середня. При додаванні нових джерел, система буде витрачати більше ресурсів на їх обробку.
Продуктивність при частому оновленні	Висока. Система створювалась з метою оновлення більше 100 разів на секунду. Проте цей параметр можна змінювати залежно від потреб.	Середня. Через складність розрахунків, система потребує більше часу на обробку світлових об’єктів.

Виходячи вже з цього порівняння, можна зробити висновок, що URP попри свої великі можливості є менш оптимізованим у випадку, коли гра потребує динамічно змінювати ігровий світ та інформацію про нього. Сама по собі система працює на високому рівні. Проте з 2D іграми виникають певні конфліктні ситуації, особливо в іграх жанру «Sandbox».

Особливість створеної моделі полягає у тому, що вона пропонує більш оптимізоване рішення у порівнянні з існуючим. Сильними сторонами є графічно приємний результат, швидкість, зменшене навантаження на ресурси, повна сумісність з 2D перспективою та високий рівень динамічності. Обмеженнями є те, що технологія націлена на роботу з конкретними жанрами ігор, що передбачають постійну зміну ігрового світу. Іншими словами модель є більш оптимізованою за URP для ігор жанру «Sandbox» і усіх можливих похідних у перспективі 2D. Окрім цього модель може слугувати як допоміжний матеріал тим розробникам, які планують створювати власні оптимізовані системи освітлення. Завдяки цьому вони можуть навчитися виконувати паралелізацію, працювати з шейдерами та

побачать деякі хитрощі у роботі з результатами розрахунків. Проте такий результат є лише побічним.

Модель базується на розподіленні обчислювальних процесів між центральним процесором та графічним. Основною метою є швидка обробка великого об’єму інформації для створення єдиної мапи освітлення кожен ігровий кадр.

Для моделювання було обрано ігровий рушій Unity та IDE Visual Studio 2022. В якості жанру гри обрано «Sandbox», що є основним обмеженням моделі. Тому, для імплементування в інші проєкти необхідно звернути увагу на цю умову.

Система використовує паралелізацію центрального процесора із використанням Job System. Таке рішення дозволить гнучко та безпечно налаштувати багатопоточний код. Якщо ж система використовується для інших ігрових рушіїв відмінних від Unity, то там можна використати або вбудовану в мову програмування паралелізацію, або ж використати рішення рушія. Основна мета цього кроку – збір необхідної інформації для її подальшої обробки (лістинг 3.3).

Лістинг 3.3 – Приклад паралельного збору інформації

```
public void Execute(int index)
{
    TilesManager worldDataManager = TilesManager.Instance;
    WorldCellDataGPU data = new();
    int x = index % _width;
    int y = index / _width;
    int dx = _startX + x;
    int dy = _startY + y;

    if (_isColoredMode)
    {
        data.BlockLightColor =
worldDataManager.GetBlockLightColor(dx, dy);
        data.WallLightColor =
worldDataManager.GetWallLightColor(dx, dy);
    }
    else
    {
        data.BlockLightIntensity =
```

```
worldDataManager.GetBlockLightIntensity(dx, dy);
        data.WallLightIntensity =
worldDataManager.GetWallLightIntensity(dx, dy);
    }

    data.Flags = 0;
    if (worldDataManager.IsPhysicallySolidBlock(dx, dy))
data.Flags += 1;
    if (worldDataManager.IsLiquidFull(dx, dy)) data.Flags += 2;
    if (worldDataManager.IsDayLightBlock(dx, dy)) data.Flags +=
4;

    _worldData[index] = data;
}
```

Додатково для прискорення можна використати спеціальний атрибут BurstCompile, що дозволяє провести оптимізацію IL коду за допомогою LLVM та отримати ще більше прискорення. Проте також необхідно пам’ятати про всі його обмеження.

Після цього необхідно налаштувати GPU для подальшого використання. По-перше, треба визначитись завдяки якій технології створити скрипт для обробки. При створенні моделі було використано ComputeShader та мову HLSL, так як Unity може використовувати цей тип шейдерів як розрахунковий. Для початку необхідно визначитись які дані необхідно передавати. Це буде збірка, що сформована на попередньому кроці. Після цього, необхідно знайти в шейдері необхідні функції та передати дані з центрального процесора до графічного (лістинг 3.4).

Лістинг 3.4 – Фрагмент коду ініціалізації шейдеру та передачі даних

```
_fillLightBrightnessHandler =
_computeShader.FindKernel("FillLightBrightness");
_spreadLightBrightnessHandler =
_computeShader.FindKernel("SpreadLightBrightness");
_applyLightBrightnessHandler =
_computeShader.FindKernel("ApplyLightBrightness");
_fillLightColoredHandler =
_computeShader.FindKernel("FillLightColored");
_spreadLightColoredHandler =
_computeShader.FindKernel("SpreadLightColored");
_applyLightColoredHandler =
_computeShader.FindKernel("ApplyLightColored");
```

```
_computeShader.SetInt("_width", _width);
_computeShader.SetInt("_height", _height);

_computeShader.SetBuffer(_fillLightColoredHandler, "_worldData",
_worldDataComputeArray);
_computeShader.SetBuffer(_fillLightColoredHandler, "_colors",
_colorsComputeArray);
_computeShader.SetBuffer(_fillLightBrightnessHandler,
"_worldData", _worldDataComputeArray);
_computeShader.SetBuffer(_fillLightBrightnessHandler,
"_brightness", _brightnessComputeArray);
_computeShader.SetBuffer(_spreadLightColoredHandler,
"_worldData", _worldDataComputeArray);
_computeShader.SetBuffer(_spreadLightColoredHandler, "_colors",
_colorsComputeArray);
_computeShader.SetBuffer(_spreadLightBrightnessHandler,
"_worldData", _worldDataComputeArray);
_computeShader.SetBuffer(_spreadLightBrightnessHandler,
"_brightness", _brightnessComputeArray);
_computeShader.SetBuffer(_applyLightColoredHandler, "_colors",
_colorsComputeArray);
_computeShader.SetBuffer(_applyLightBrightnessHandler,
"_brightness", _brightnessComputeArray);
_computeShader.SetTexture(_applyLightColoredHandler, "_texture",
_lightMapRenderTexture);
_computeShader.SetTexture(_applyLightBrightnessHandler,
"_texture", _lightMapRenderTexture);
```

Тепер необхідно перейти до самого шейдеру. Структура доволі проста. Є кілька змінних, що будуть поширюватися між потоками графічного процесора, та є кілька методів, що направлені на обробку сегменту даних. Для того, щоб використовувати функцію, необхідно для початку налаштувати кількість потоків що будуть виділятися під конкретну задачу. Після цього йде основна логіка обробки (лістинг 3.5).

Лістинг 3.5 – Фрагмент шейдеру обробки світла

```
[numthreads(250, 1, 1)]
void FillLightBrightness(uint3 id : SV_DispatchThreadID)
{
    float wallLightIntensity = _worldData[id.x +
id.y].wallLightIntensity;
    float blockLightIntensity = _worldData[id.x +
id.y].blockLightIntensity;
    int isSolid = _worldData[id.x + id.y].flags & 1;
    int isFullLiquidBlock = _worldData[id.x + id.y].flags & 2;
```

```
    int isDayLightBlock = _worldData[id.x + id.y].flags & 4;

    if (isSolid != 0 || isFullLiquidBlock != 0)
        _brightness[id.x + id.y] = blockLightIntensity;
    else if (isDayLightBlock != 0)
    {
        wallLightIntensity = lerp(0, 1, _dayLightValue);
        if (wallLightIntensity >= blockLightIntensity)
            _brightness[id.x + id.y] = wallLightIntensity;
        else
            _brightness[id.x + id.y] = blockLightIntensity;
    }
    else
    {
        if (wallLightIntensity > blockLightIntensity)
            _brightness[id.x + id.y] = wallLightIntensity;
        else
            _brightness[id.x + id.y] = blockLightIntensity;
    }
}
```

Використання будь-якої функції з шейдеру також потребує певного простого налаштування. Це є вирішення розміру сегменту. Бувають ситуації, коли задача повинна розглядати кожен елемент як окремий, або ж як сукупність, що створюють стовпець або рядок. Для цього необхідно при використанні функції Dispatch, задати кількість робочих груп в різних осях (лістинг 3.6).

Лістинг 3.6 – Приклад встановлення розміру для різних задач

```
_computeShader.Dispatch(_spreadLightColoredHandler, (int)(_width
/ x), 1, 1);
_computeShader.Dispatch(_applyLightColoredHandler, (int)(_width
/ x), (int)(_height / y), 1);
```

Дотримання усіх вищеперчислених вимог дозволяє створити доволі гнучку та швидку систему. Також додатково, для уникнення переносу даних з графічного процесора назад до центрального, можна використовувати об’єкт RenderTexture. Він дозволяє напряму з’єднати результат розрахунків з графічним супроводженням об’єкта, дозволяючи уникнути додаткових операцій переносу.

Тепер щодо загального алгоритму роботи цієї моделі. Він складається лише з кількох кроків (рисунок 3.3): фаза збору даних на ЦП, фаза обробки в ГП та фаза постобробки завдяки допоміжним шейдерам (цей пункт є оптимальним та залежить від потреб розробників). Фаза обробки це доволі обширне поняття в цьому контексті, адже це залежить від конкретних задач. Проте за замовчуванням відбувається спочатку фільтрація за типом блоку. Потім розповсюдження та наостанок встановлення результату до текстури. Для оптимізації кожен крок використовує графічний процесор та необхідну для цього етапу кількість потоків. Таке рішення дозволяє швидко та безпечно опрацювати необхідні дані.

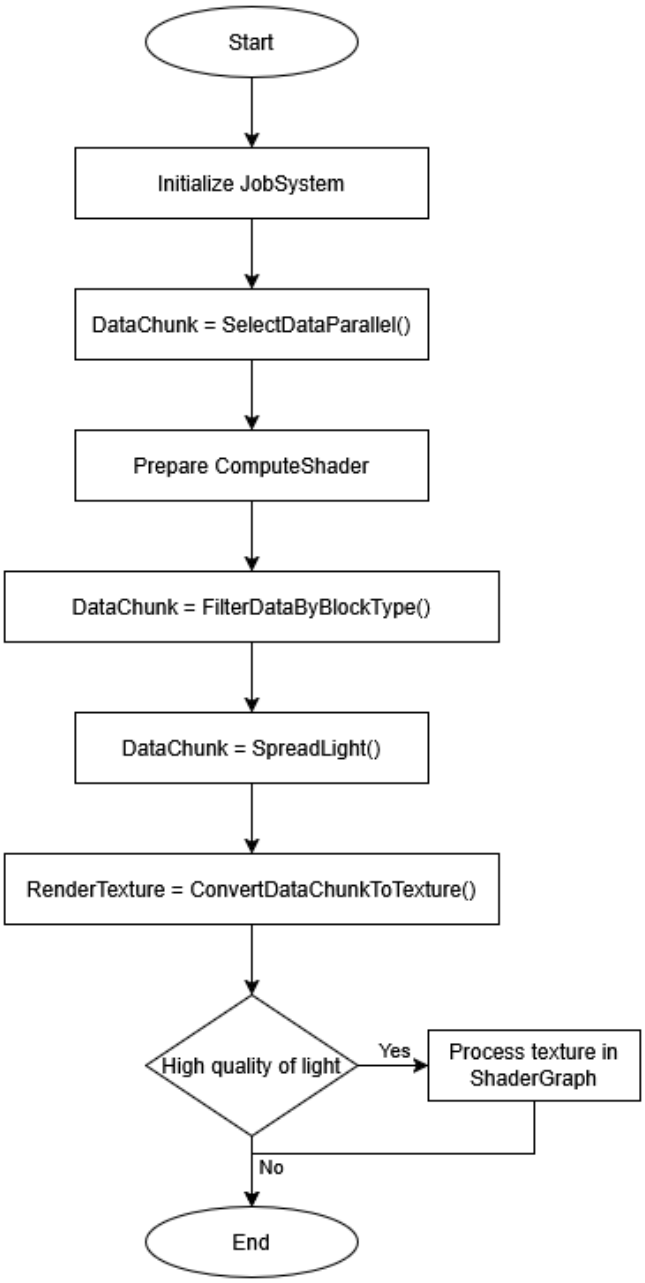

Рисунок 3.3 – Блок схема алгоритму формування мапи світла

Для більш покращеного розуміння було створено IDEF0 нотацію двох рівнів (рисунки 3.4-3.5), що показує які існують вхідні та вихідні дані, залежності та технології, що використовуються при роботі.

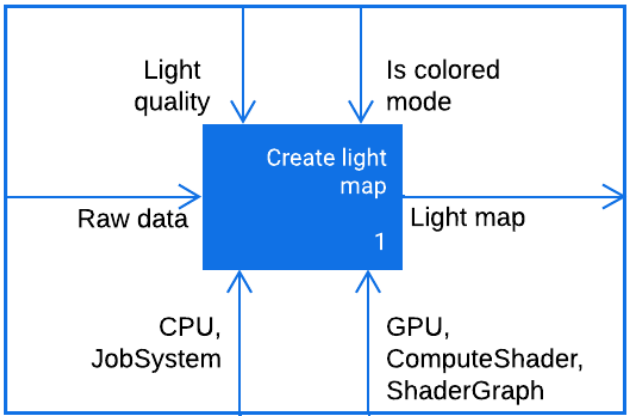

Рисунок 3.4 – IDEF0 нотація 1 рівня

Для можливості проведення більш детального аналізу, цю нотацію було переведено на рівень вище (рисунок 3.5). Для цього додано всі основні етапи генерації світлових ефектів. Тут вже можна більше детально побачити, яка фаза використовує конкретний ресурс.

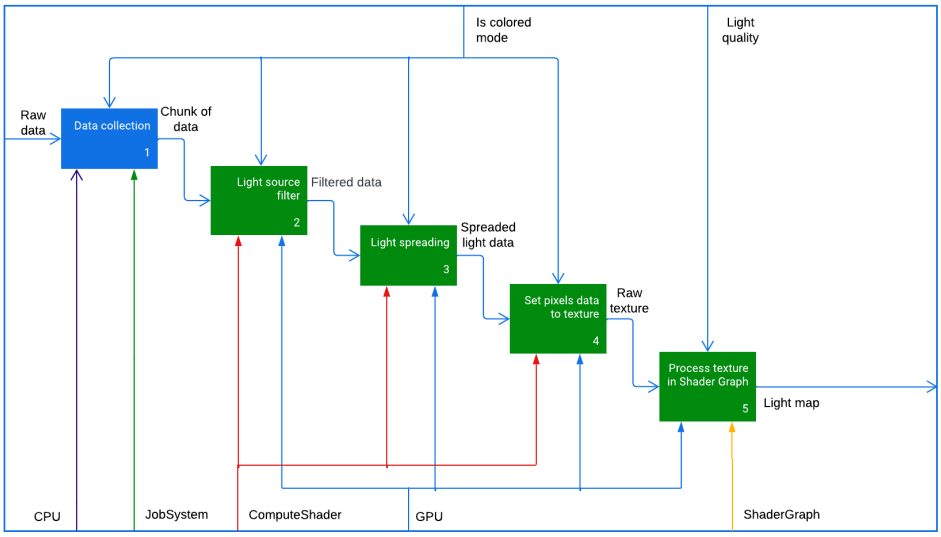

Рисунок 3.5– IDEF0 нотація 2 рівня

Для аналізу результату було створено невелику утиліту для виміру часу. Вона допомогла побудувати таблицю результатів (таблиця 3.3) для порівняння роботи за різних умов. Головними вхідними параметрами та критеріями порівняння є якість освітлення (низька, середня, висока) та режим кольору (монохромний, кольоровий). У якості вихідного результату – час в мілісекундах.

Таблиця 3.3 – Час роботи моделі за різних умов

Якість	Колір	Час обробки даних, мс
Низька	Ні	0.5615
Низька	Так	0.5645
Середня	Ні	0.5653
Середня	Так	0.5671
Висока	Ні	0.5712
Висока	Так	0.5723

Для того, щоб побачити наскільки технологія є швидшою за URP, було також виконано кілька десятків тестів (таблиця 3.4). Але URP не надавав гарного візуального світла, тому таке порівняння буде базуватись на кількості джерел світла та на часі їх обробки. Тепер вхідним параметром є кількість джерел, а вихідним – час в мілісекундах. Модель використовувала найкращу якість та кольоровий режим. Також ці результати можна порівняти за допомогою гістограми (рисунок 3.6).

Таблиця 3.4 – Порівняльна таблиця часу виконання від кількості джерел світла

Кількість джерел світла	Час URP, мс	Час моделі, мс	SpeedUp (NVIDIA GeForce GTX 1650)
10	0.9655	0.5687	1.6977
15	1.3471	0.5702	2.3625
25	1.5369	0.5809	2.6457
50	1.8476	0.5918	3.1220
100	2.5873	0.6238	4.1476
200	3.4878	0.6576	5.3038

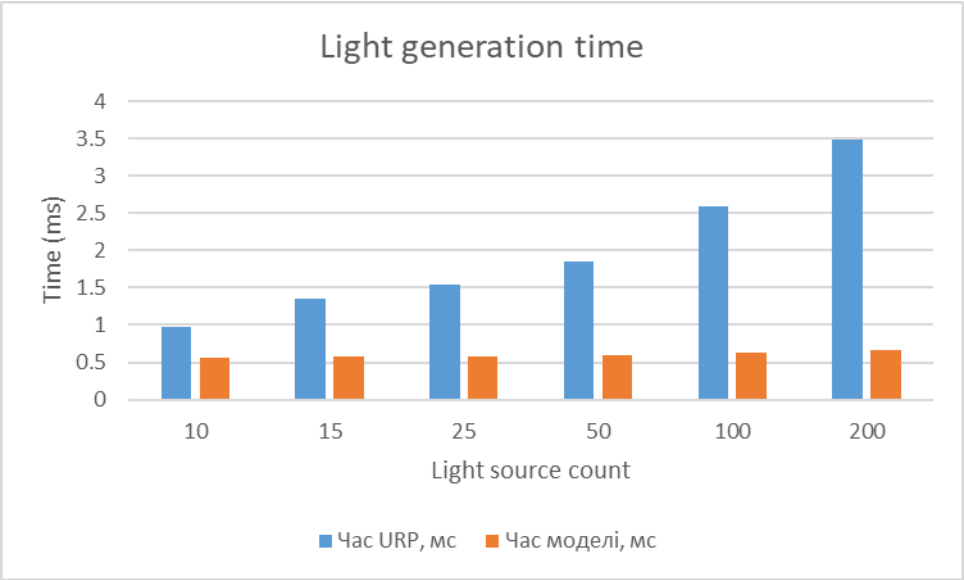

Рисунок 3.6 – Гістограма часу виконання різних технологій

3.3 Модель оптимізації пам’яті

Часто в результуючому проєкті можна зустріти ситуацію, коли ігрова система займає багато оперативної пам’яті. Це одна з найпоширеніших проблем при розробці. Відбувається це через те, що деякі розробники-початківці допускають помилку при побудуванні структури даних. Частіш за все вони думають, якщо працює то вже добре. Проте при збільшенні проєкту збільшується і потреба в ресурсах. Це може призвести до того, що продукт буде займати увесь доступний йому простір в оперативній пам’яті. В інтернет просторі існує безліч рекомендацій з приводу того як взагалі необхідно використовувати та організовувати пам’ять, проте більшість розробників може не зрозуміти, що й коли використовувати при розробці ігрової системи. Саме тому було проведено дослідження впливу різних факторів: розміщення елементів у пам’яті, їх розмір та доречність використання тих чи інших типів даних. На основі цього дослідження побудовано певний алгоритм дій, що стане у нагоді тим розробникам, які тільки починають свій шлях у напрямку геймдева. Дотримання усіх вимог цього набору правил, допоможуть значно зменшити навантаження на оперативну пам’ять. Особливо це корисно в тому випадку коли гра націлена

на збереження інформації про ігровий світ у масиві великих розмірів (від 20 до 100 мільйонів елементів). Також додатково можна розглянути різного роду алгоритми компресії для роботи з даними [3].

3.3.1 Оптимізація структури об’єктів

Самим найпершим кроком є оптимізація структури необхідних об’єктів. Для цього було виконано дослідження впливу використання класу та структури при організації масивів даних.

В більшості випадків, використання структур вважається швидшим у порівнянні із класами. Пов’язано це з алгоритмом розташування у пам’яті та з життєвим циклом. Також структури не використовують додаткового виділення пам’яті як класи, що також значно збільшує шанси на їх використання.

Для того, щоб додатково зменшити кількість необхідної пам’яті в структурі потрібно використовувати тільки типи значень. Якщо використовувати посилання то це може значно порушити необхідний порядок змінних, що в свою чергу призведе до значного збільшення пам’яті. Загалом структура повинна використовуватись для тих об’єктів, що представляють данні, а не поведінку ігрових об’єктів.

Якщо говорити про клас, то його використання при роботі з масивами даних повинно зменшитись до мінімуму. Класи частіш за все використовуються для об’єктів, що представляють собою певну поведінку в грі. Це можуть бути різного роду менеджери, системи, контролери, поведінки різних об’єктів і так далі. Через те, що класи доволі погано організовують змінні, виникають певні непотрібні виділення пам’яті, що в свою чергу можуть призвести до надлишкового використання RAM.

В якості прикладу в лістингу 3.7 наведено клас та структуру, що мають однаковий набір даних. Також слід пам’ятати, що класи це тип посилання, тому на нього також необхідно виділити пам’ять під заголовок (рисунок 3.7).

Зі структурою в такому випадку все простіше, адже вона не виділяє додаткових байтів інформації під заголовок (рисунок 3.8). Це робить її компактною та більш швидшим типом даних.

Лістинг 3.7 – Приклад класу та структури

```
public class DataClass
{
    public byte byte1;
    public byte byte2;
    public int int1;
    public int int2;
}

public struct DataStruct
{
    public byte byte1;
    public byte byte2;
    public int int1;
    public int int2;
}
```

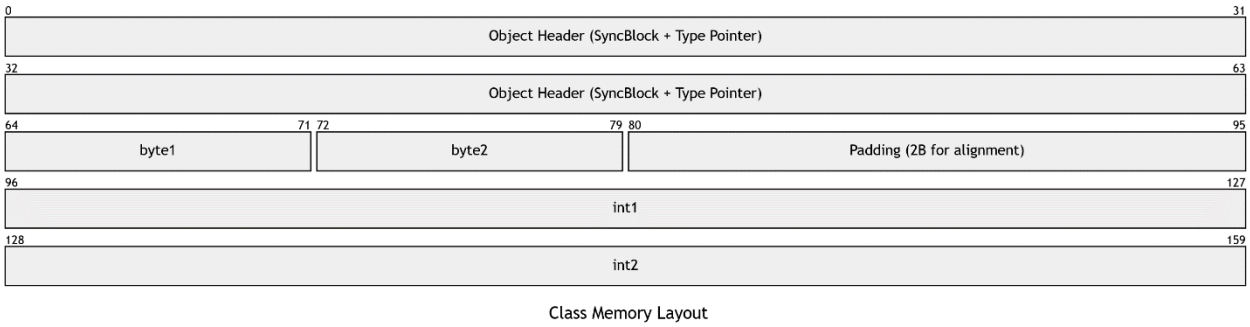

Рисунок 3.7 – Приблизний вигляд одного екземпляру класу в пам’яті

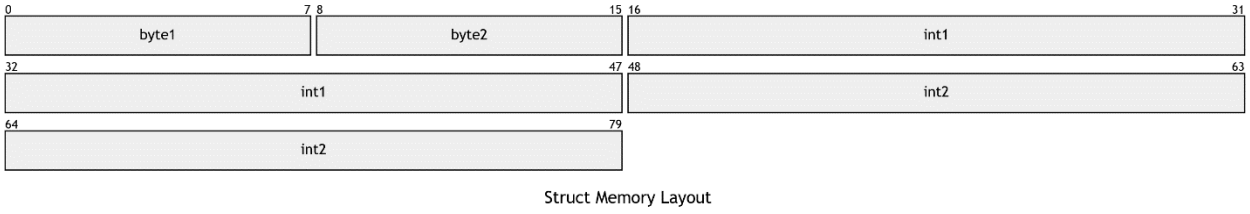

Рисунок 3.8 – Розміщення екземпляру структури в пам’яті

Якщо виконати неважкі математичні розрахунки, то можна легко порахувати кількість необхідної інформації для масивів. Взяти до прикладу

масив в 20 мільйонів елементів. Для структури, використовуючи формулу 3.1, екземпляр займатиме приблизно 12 байт інформації (10 байтів для даних та 2 байти на вирівнювання). Це буде приблизно 240 мегабайт для масиву (формула 3.3). Якщо ж використовувати клас, то екземпляр займатиме 28 байт (16 байт на заголовок, 10 на дані та 2 на вирівнювання) (формула 3.2), а розмір всіх елементів в купі 560 мегабайт. Масив буде розміром від 160 до 320 мегабайт (залежить від розміру посилання). Тоді загальний розмір буде приблизно 880 мегабайт, що майже в 4 рази більший.

$$\text{Size of Structure} = \text{Size of Fields} + \text{padding} \quad (3.1)$$

$$\text{Size of Class} = \text{Size of Fields} + \text{Header} + \text{Padding} \quad (3.2)$$

$$\text{Size of Array} = (\text{Count} * \text{Reference Size}) + (\text{Count} * \text{Size of Object}) \quad (3.3)$$

Для підтвердження правила, було виконано низку експериментів. Для цього використовувались оптимізовані дані та різна кількість елементів у масиві. Це означає, що значення вирівнювання в обох випадках дорівнює 0.

Таблиця 3.5 – Порівняльна таблиця розміру масиву для різного типу даних

Кількість елементів, млн	Клас, Мб	Структура, Мб	Різниця
10	420	100	4.2
15	630	150	4.2
20	840	200	4.2
50	2100	500	4.2
100	4200	1000	4.2

Іншими словами, майже в більшості випадків, структура буде приблизно в 4 рази менше займати пам’яті ніж клас.

Наступним правилом є використання реально необхідних даних. Будь-яке довільне поле структури повинно використовувати реально необхідні ресурси. Це також стосується типу переліку. За цією нормою, необхідно порівнювати область допустимих значень поля та діапазон можливих значень кожного типу. Наприклад, якщо в грі використовується близько 40 тисяч унікальних екземплярів об'єктів, то для їх ідентифікації доречніше буде використовувати тип Short або UShort. Такі самі дії необхідно провести з іншими полями.

Підбиваючи підсумки частини дослідження, можна сказати, що у випадку роботи з масивами даних, необхідно використовувати структури. Це дозволить легко та швидко отримувати доступ, обробляти їх та зменшить кількість необхідного об’єму пам’яті. Класи в свою чергу слід використовувати здебільшого для створення якоїсь поведінки об’єкта. Також необхідно пам’ятати про ефективність типу полів.

3.3.2 Використання атрибутів організації даних

Наступною частиною дослідження є вплив використання вбудованих атрибутів для організації та оптимізації даних. Уявімо проблему того, що автоматична організація даних створила певні вирівнювання елементів. Наприклад, якщо перший елемент був byte, а другий int. В такому випадку може відбутись процес вирівнювання елементів в пам’яті, що додатково використає байти інформації.

Щоб уникнути цієї проблеми та вручну керувати усіма процесами існує два атрибути: StructLayout та FieldOffset. Перший використовується для того, щоб чітко вказати за яким алгоритмом відбувається розміщення полів об’єкту. В нього є кілька режимів роботи: Sequential (використовує значення пам’яті в тому порядку в якому елементи створенні), Explicit (використовує попередньо налаштовані позиції елементів) та Auto (автоматично розраховує положення елементів у пам’яті та додає вирівнювання). Також

використовуючи цей атрибут, можна чітко вказати розмір вирівнювання. Робиться це за допомогою параметра Pack. Має він кілька варіацій, проте для найбільшої оптимізації варто вказати значення 1 (лістинг 3.8).

Лістинг 3.8 – Використання атрибуту

```
[StructLayout(LayoutKind.Sequential, Pack = 1)]
public struct PackedStruct
{
    public byte A;
    public int B;
}
```

Говорячи про вирівнювання, його можна описати як процес розбиття усієї необхідної кількості пам’яті на сегменти, розмір яких визначений в параметрі Pack. У випадку коли трапляється ситуація, що спочатку йде 1 байт, а потім 4 байти, то для першого буде виконано вирівнювання та додано додатковий байт інформації (рисунок 3.9-3.10).

Рисунок 3.9 – Вирівнювання з розміром 2

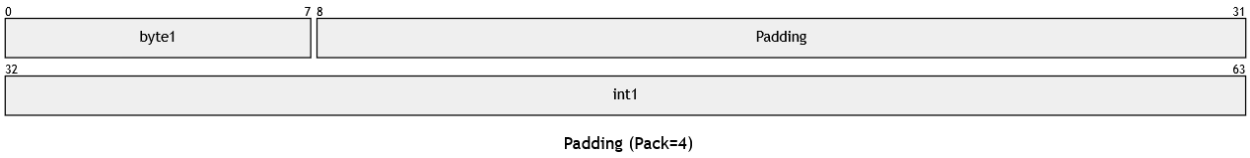

Рисунок 3.10 – Вирівнювання з розміром 4

Інший атрибут керування FieldOffset дозволяє якраз таки налаштувати розміщення елементів у сегменті. Також, якщо ситуація вимагає певного розділення елементу на дві або більше частин без додаткового виділення пам’яті – можна застосувати механізм об’єднання. Робота такого механізму доволі проста. Для двох елементів різної величини ставиться однакове

значення зсуву. Наприклад, якщо взяти ushort, його можна умовно поділити на 2 байти (лістинг 3.9). Також для простоти розуміння, роботу цього атрибуту можна легко представити графічно (рисунок 3.11).

Лістинг 3.9 – Розміщення полів у пам’яті

```
[StructLayout(LayoutKind.Explicit)]
public struct DataStruct
{
    [FieldOffset(0)]
    public ushort Indexes;

    [FieldOffset(0)]
    public byte FirstIndex;

    [FieldOffset(1)]
    public byte SecondIndex;
}
```

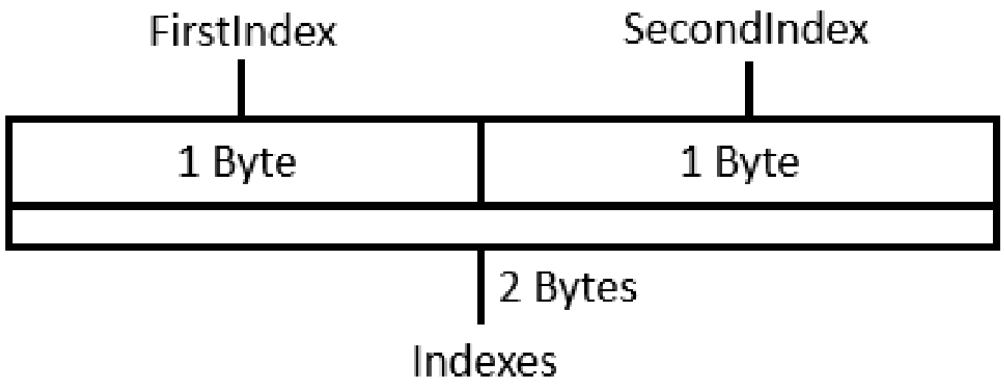

Рисунок 3.11 – Графічне представлення роботи FieldOffset

3.3.3 Перевикористання даних

Останнім правилом оптимізації є механізм пулінгу елементів та кешування. Робота першого алгоритму доволі проста. На початку процесу створюється невеликий пул об’єктів. Якщо якомусь елементу системи необхідно створити об’єкт, він бере перший вільний в пулі. Якщо ж такого немає, то створюється новий, у разі якщо немає суворого обмеження розміру пулу. Використання такого механізму здебільшого дозволяє уникнути

зайвого виділення пам’яті. Структура такої моделі проста: стек (найшвидша колекція у порівнянні з іншими) та кілька методів для маніпуляцій (лістинг 3.10).

Лістинг 3.10 – Приклад пулу об’єктів

```
public class ObjectsPool<T>
{
    private readonly Stack<T> _pool;
    private readonly Func<T> _createInstance;

    public ObjectsPool(Func<T> createInstance, int starterSize =
0)
    {
        _pool = new Stack<T>();
        _createInstance = createInstance;
        if (starterSize > 0)
        {
            for (int i = 0; i < starterSize; i++)
                _pool.Push(createInstance.Invoke());
        }
    }

    public T Get()
    {
        return _pool.Count == 0 ? _createInstance.Invoke() :
_pool.Pop();
    }

    public void Release(T obj)
    {
        _pool.Push(obj);
    }
}
```

Іншим гарним прикладом перевикористання є кешування. Якщо подібний об’єкт вже був створений, немає необхідності створювати інший. Іншими словами кешування дозволяє зберегти усі унікальні об’єкти, для того щоб уникнути зайвого перестворювання. Проте з кешуванням необхідно бути обережним, адже зміна такого об’єкту може створити кілька проблем. Тому загальною рекомендацією є використовувати кешування тільки у випадку, коли дані є незмінними.

3.4 Автоматизація процесів розробки гри

Нерідко перед розробниками постає проблема недостатньої автоматизації деяких процесів. Це змушує витрачати багато часу на монотонний однаковий процес – створення нового контенту. Деякі ігрові рушії мають більш-менш оптимізовану систему роботи. Проте, якщо цього недостатньо – необхідно створювати свої рішення [4]: створення особистого рушія або вдосконалення існуючого.

3.4.1 Створення власних редакторів

Першим таким рішенням є створення власних редакторів об’єктів. Уявімо ситуацію, коли інформація об’єкта складається з десятків різних параметрів. Вбудовані інструменти дозволять відобразити їх для подальшої роботи, але це буде неструктурована та монолітна система елементів (рисунок 3.12).

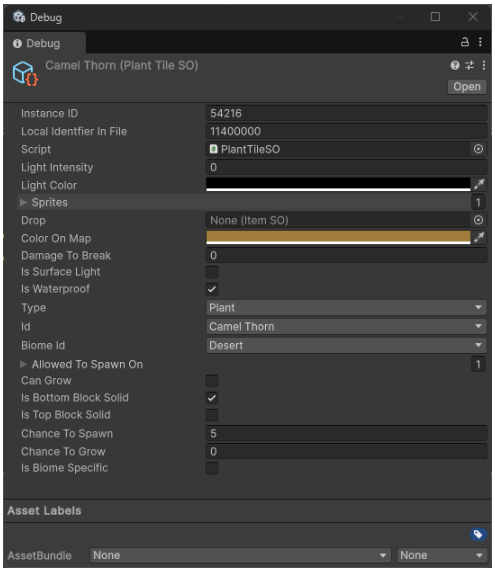

Рисунок 3.12 – Приклад обробки об’єкту із використанням вбудованих редакторів

Для вирішення цієї проблеми, необхідно розглянути можливість

модифікації ігрового рушія. Так от наприклад Unity надає розробникам певні інструменти для розширення функціоналу. Для цього є два розвитку подій – використання старих елементів керування, або ж використання більш сучасного інструменту – UIBuilder. В рамках цієї моделі було використано другий варіант роботи.

Для побудови нового редактора використовується успадкування від класу Editor та явне вказування для якого класу це буде використано. Unity сам потім автоматично знайде усю інформацію та використає необхідний варіант редактора.

Наприклад, для прискорення роботи з об’єктами було створено свій редактор інтерактивного налаштування властивостей елемента(рисунок 3.13). Таке рішення дозволяє структурувати інформацію за критеріями перетворивши комплексну послідовність в ієрархію властивостей. Також за потреби можна доповнити функціонал іншими елементами, що покращать розуміння про об’єкт. Одним таким елементом стало прев’ю об’єкта. Це невелике зображення, що ілюструє вигляд об’єкта в грі.

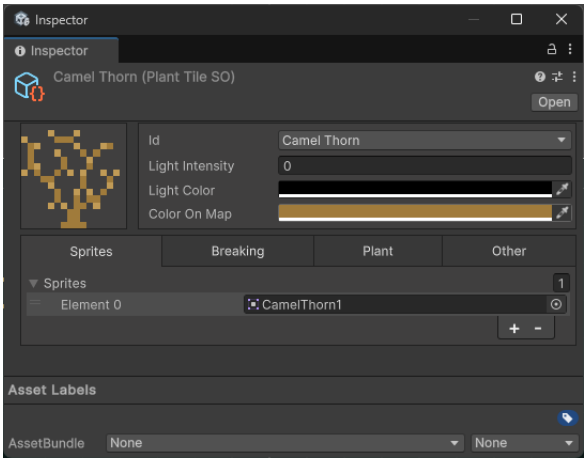

Рисунок 3.13 – Власна версія редактору блоків

Використання такого підходу оптимізації є не лише для вузьких цілей. Його область використання є майже нескінченною. Усе залежить лише від потреб розробників.

3.4.2 Створення власних вікон редактору

Іншим прикладом модифікації рушія є створення власних вікон. Це по суті створення нової частини програмного забезпечення, що є інтеграцією нових можливостей у процес розробки.

Наприклад в Unity надано можливість створювати власні вікна для різного роду операцій. Їх можна використовувати для автоматизації рутинних задач, інтерактивного управління контентом, або ж використовувати як інструмент для розробників.

В рамках цієї роботи було створено кілька вікон для автоматизації певних процесів:

- швидке створення нового контенту з можливістю кастомізації та конфігурації об’єктів;
- візуалізація моделі штучного інтелекту неігрових персонажів.

Говорячи про першу ситуацію, для ігрового додатку в якості приклада було створено систему менеджменту усіх об’єктів у застосунку. Це вікно (рисунок 3.14) використовує також власні версії редакторів, що дозволяє не тільки прискорити процес створення контенту, але й надає можливість зручної конфігурації даних. Для швидкої роботи було інтегровано систему пошуку (рисунок 3.15), класифікацію об’єктів за призначенням (рисунок 3.14) та за типом (рисунок 3.16). Така модифікація дозволяє легко та швидко маніпулювати усіма об’єктами в системі в одному місці.

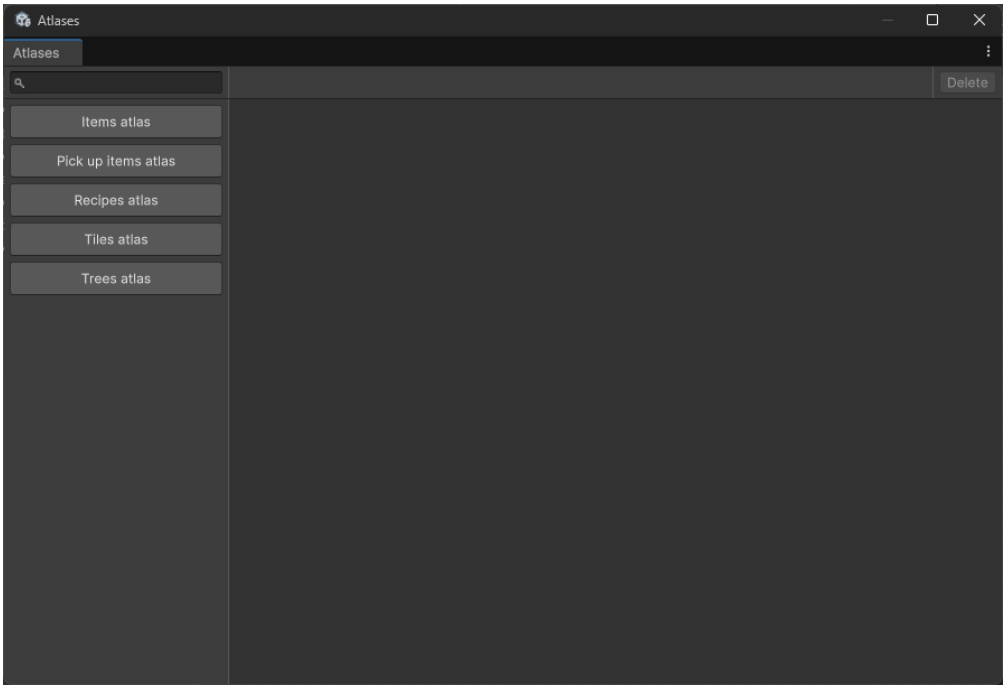

Рисунок 3.14 – Початковий вигляд вікна менеджера об’єктів

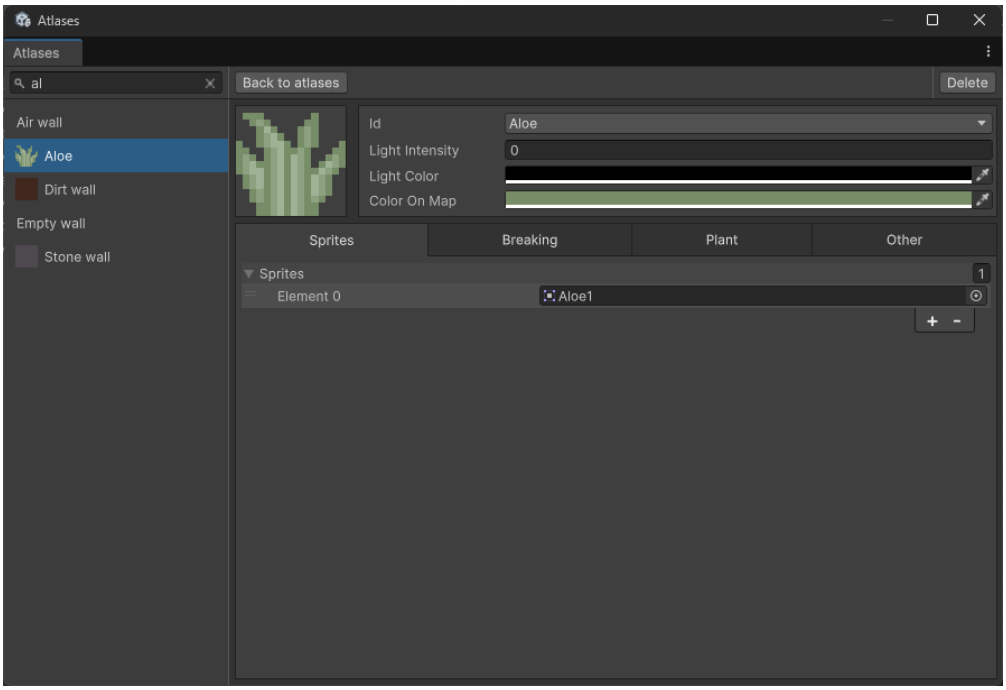

Рисунок 3.15 – Система пошуку об’єктів

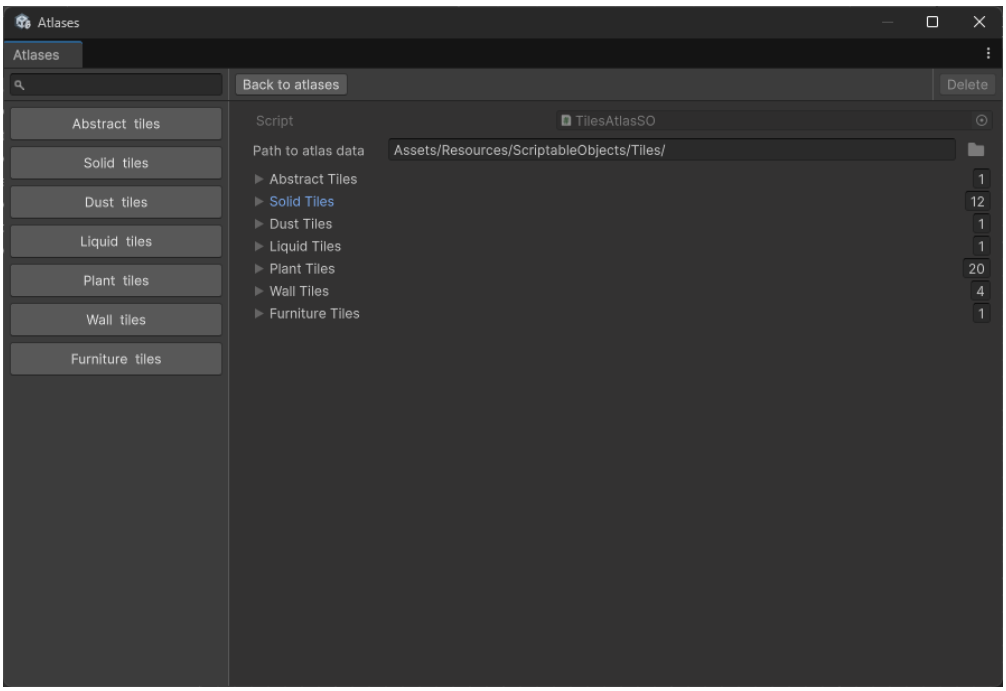

Рисунок 3.16 – Класифікація за типом

Іншим прикладом вікна є інструмент для роботи з моделлю Finite State Machine. Так як, на момент цієї роботи, в ігровому застосунку використовувалась лише ця модель, було прийнято рішення у створенні спеціального інструменту для полегшеної маніпуляції станами. Створена модель, може використовуватися в будь-якому проєкті, адже головна мета – інтерактивне створення екземпляру моделі FSM.

Спираючись на можливості Unity, візуальна частина вікна поділена на дві частини (рисунок 3.17): панель для редагування стану та сітка з усіма станами. Додатково створено функціонал пошуку усіх можливих дій у системі з можливістю подальшого використання (рисунок 3.18). Також, для прискорення конфігурації умов переходу між станами, було створено додатковий контент, що дозволяє вручну обрати необхідну властивість (рисунок 3.19) з автоматичним пошуком усіх доступних умов (рисунок 3.20).

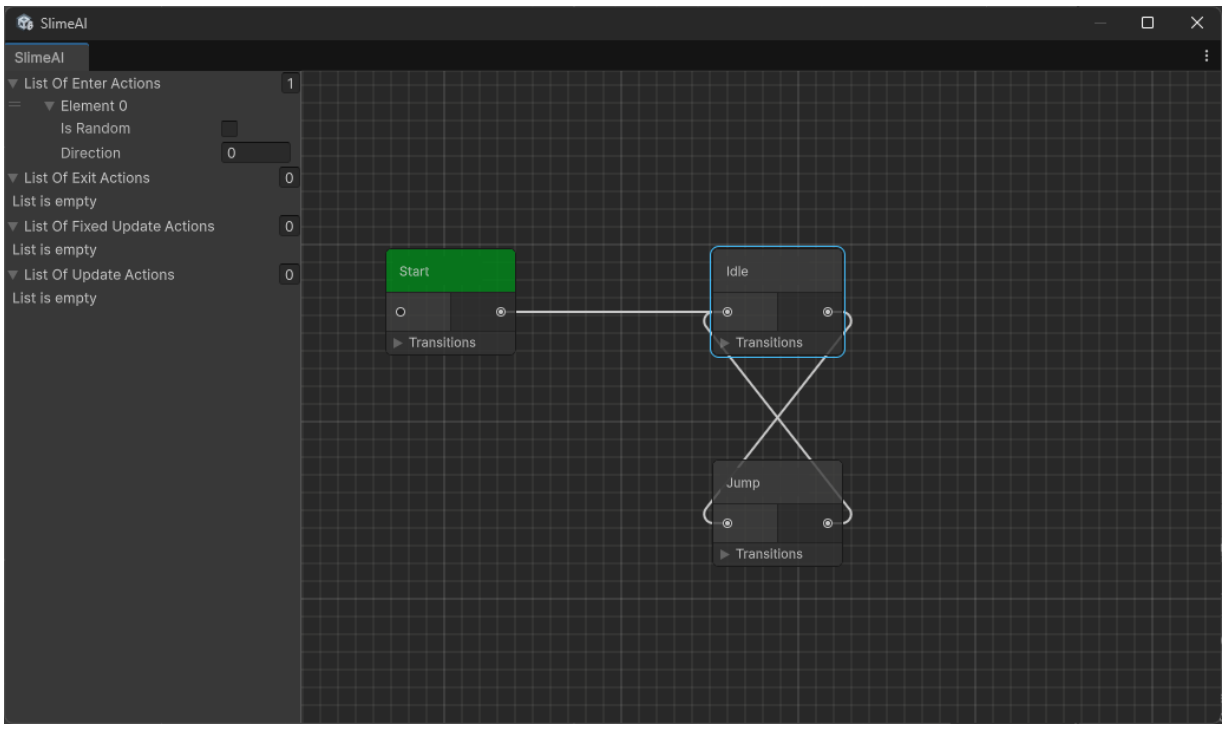

Рисунок 3.17 – Основна частина інструменту FSM

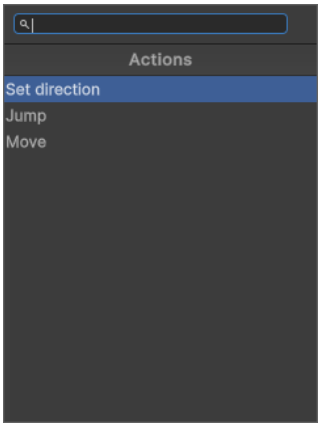

Рисунок 3.18 – Результат пошуку дій

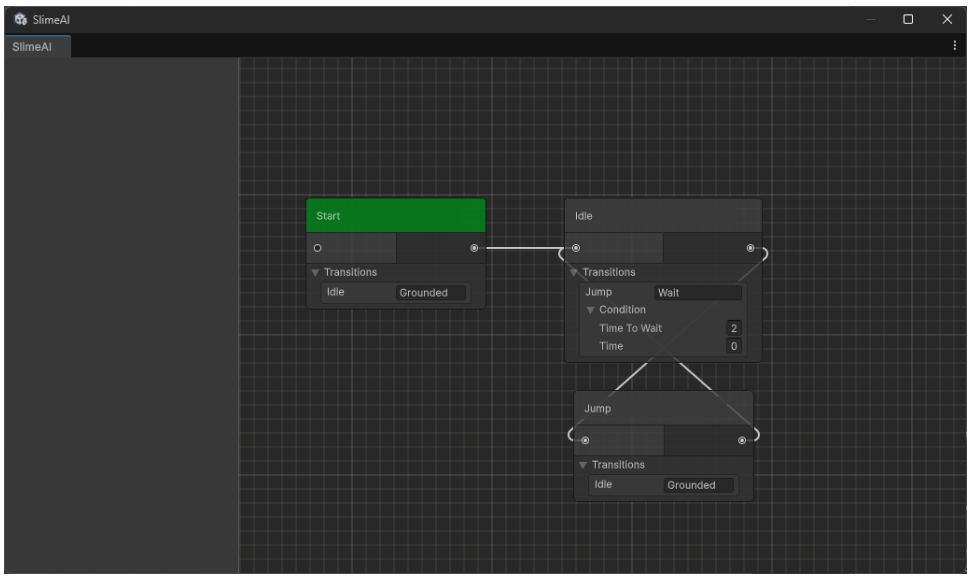

Рисунок 3.19 – Стани з детальною інформацією

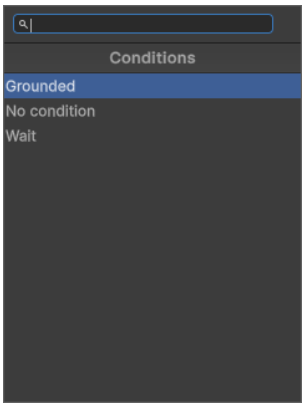

Рисунок 3.20 – Результат пошуку усіх умов

3.4.3 Створення власних атрибутів

Інколи бувають ситуації, коли створювати новий редактор для вирішення лише невеликої проблеми буває недоречно. Для цього на допомогу можна використати можливість створення власних атрибутів. Такі атрибути будуть використовуватися лише як показники того, який шаблон необхідно застосувати для конкретного поля.

Загальний принцип створення таких компонентів полягає у тому, що необхідно створити власне атрибут, створити PropertyDrawer (компонент, що дозволяє налаштувати зовнішній вигляд властивості) та централізовану

систему збору та обробки таких атрибутів.

В рамках цієї роботи було створено два компоненти: візуалізація метода як кнопки (рисунок 3.21) та використання рядка як шлях до об’єкту. Перший використовується для того, щоб пришвидшити тестування деяких функцій. Адже простіше натиснути на кнопку, щоб викликати її та мати цілковитий контроль над нею, аніж використовувати події або інші механізми виклику.

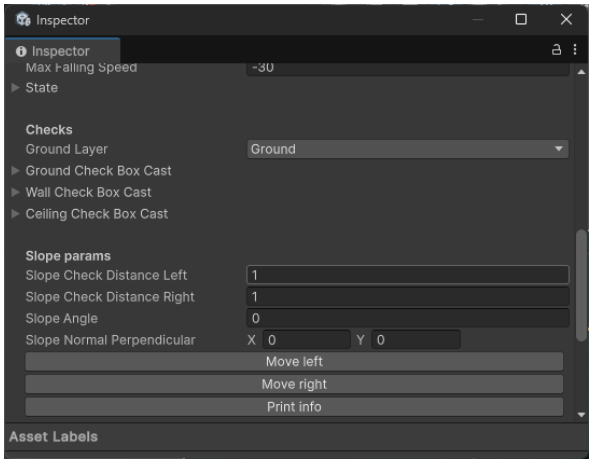

Рисунок 3.21 – Результат створення методів як кнопок

Другий атрибут націлений на створення зручної взаємодії з рядком для пошуку та збереження обраного шляху до об’єкту або папки (рисунок 3.22). Його створення пов’язано з тим, що в деяких випадках простіше самому знайти за допомогою провідника певний об’єкт (рисунок 3.23). Якщо ж цього не використовувати то розробнику доведеться переходити самостійно до папки проєкту, шукати файл, копіювати значення та вставляти його до змінної.

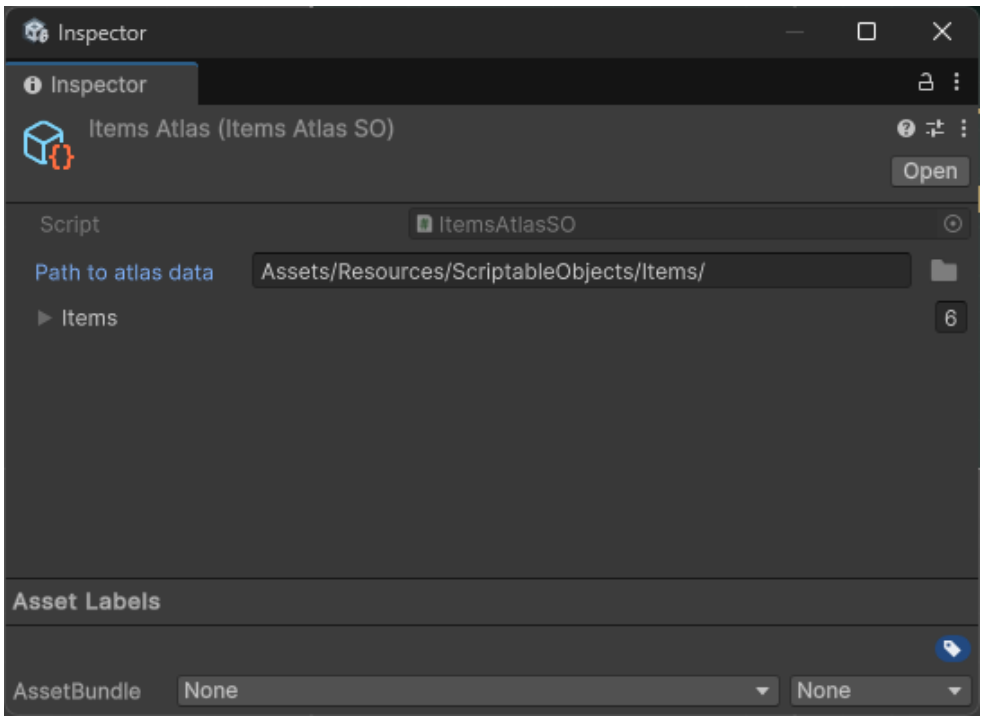

Рисунок 3.22 – Результат роботи атрибута для пошуку файлу

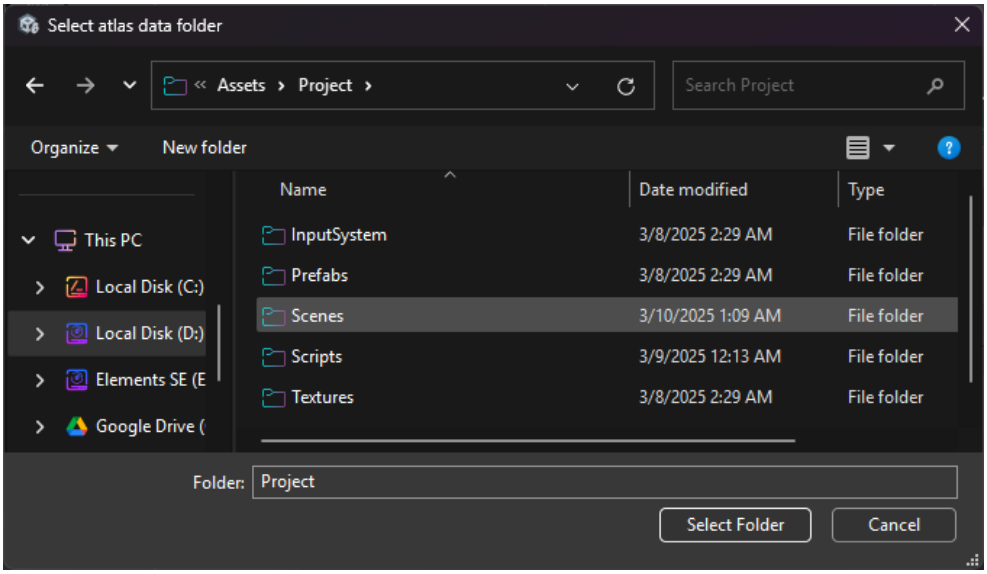

Рисунок 3.23 – Вікно пошуку файлу

4 ТЕСТУВАННЯ ПРОДУКТИВНОСТІ ТА АНАЛІЗ ЕФЕКТИВНОСТІ ОПТИМІЗАЦІЇ

4.1 Методика тестування продуктивності

4.1.1 Методика тестування моделі для паралельної обробки освітлення

Для того, щоб зрозуміти наскільки розроблені системи є ефективними, необхідно чітко визначитись із системою тестування цих елементів. Для цього було визначено кілька головних тестів.

Першим є тестування системи освітлення. Критеріями оцінки цього тестування є середній показник FPS та CPU time, або ж середній час обробки кадру на процесорі у мілісекундах. Також, так як система використовує графічний процесор для обробки певного кластеру інформації, додатково можна розглянути GPU time, або середній час рендерингу кадру на графічному процесорі у мілісекундах. Вхідними параметрами тестування є:

- розмір мапи освітлення – її довжина та ширина. Чим більший розмір тим більше інформації буде взято для обробки;
- режим кольору. Так як система підтримує два режими кольоровості: монохромний та кольоровий – необхідно перевірити ці два режими з різними розмірами;
- кількість джерел освітлення. Цей параметр відповідає за перевірку стійкості системи та залежність результуючих аргументів від кількості джерел світла.

Для тестування необхідно використовувати вбудовану технологію профілювання від Unity (рисунок 4.1). Цей інструмент дозволяє швидко та майже в режимі реального часу досліджувати усі головні параметри системи. Використовуючи це вікно, можна дізнатися про поточну кількість кадрів на секунду, час роботи CPU та час, що витрачається на рендеринг одного кадру.

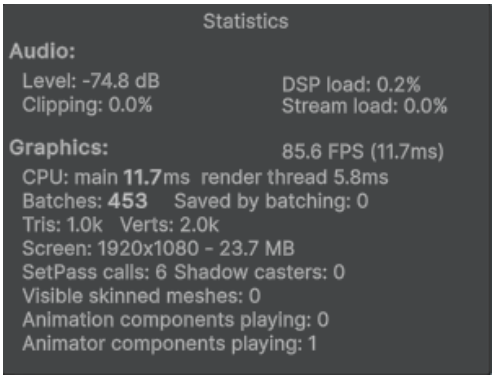

Рисунок 4.1 – Вбудована технологія профілювання роботи ігрового застосунку

Також додатково можна використовувати більш детальний профайлер (рисунок 4.2) для дослідження частин коду, що займають найбільше всього ресурсів центрального процесора. Такий інструмент дозволяє детально розглянути різні фази побудови світла та виміряти необхідний час на їх обробку. Також бонусом є зручне та інтуїтивно зрозуміле графічне супроводження з графіками роботи, використовуючи які, можна побачити в які моменти відбуваються непередбачувані скачки продуктивності.

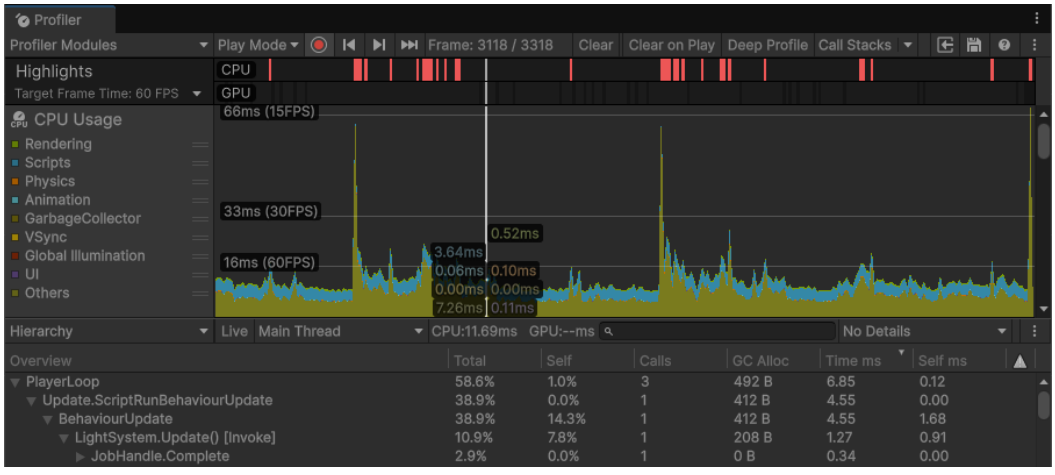

Рисунок 4.2 – Детальний аналізатор продуктивності

4.1.2 Алгоритм тестування моделі оптимізації структури об’єктів

Наступним тестом є перевірка продуктивності розробленого алгоритму оптимізації пам’яті в ігровій системі. Для цього вихідним параметром буде лише кількість витраченої оперативної пам’яті для роботи системи.

У якості вхідних параметрів є:

- розмір ігрової мапи. Це є довжина та ширина ігрової сцени. За замовчуванням вона дорівнює 8400 тайлів на 2400. Загальна сума інформації про блоки складає близько 20 мільйонів;
- рівень оптимізації структури інформації. Цей параметр відповідає за те, наскільки добре оптимізований клас або структура. Більш оптимізовані об’єкти займають менше пам’яті. Використання цього параметру дозволить дослідити залежність між різними кроками алгоритму.

Для тестування необхідно використовувати або вбудований в Unity інструмент профілювання пам’яті (рисунок 4.3), сторонній інструмент, що є майже офіційним (рисунок 4.4) та диспетчер задач (рисунок 4.5), що також дозволяє проаналізувати кількість пам’яті що витрачається на один процес.

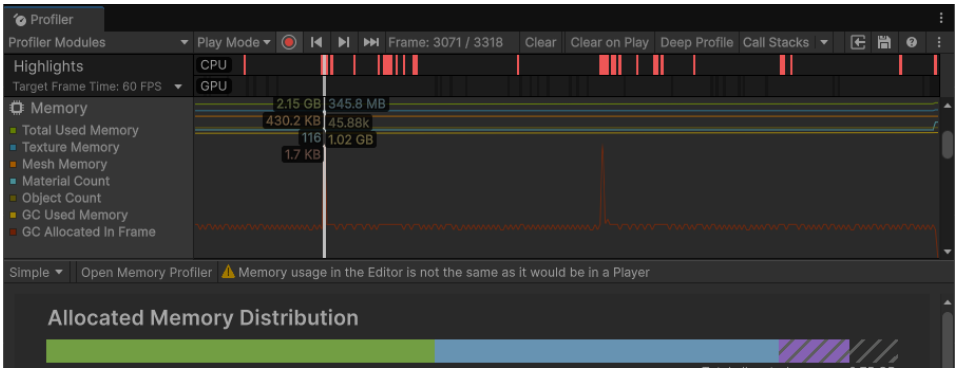

Рисунок 4.3 – Вбудований аналізатор пам’яті

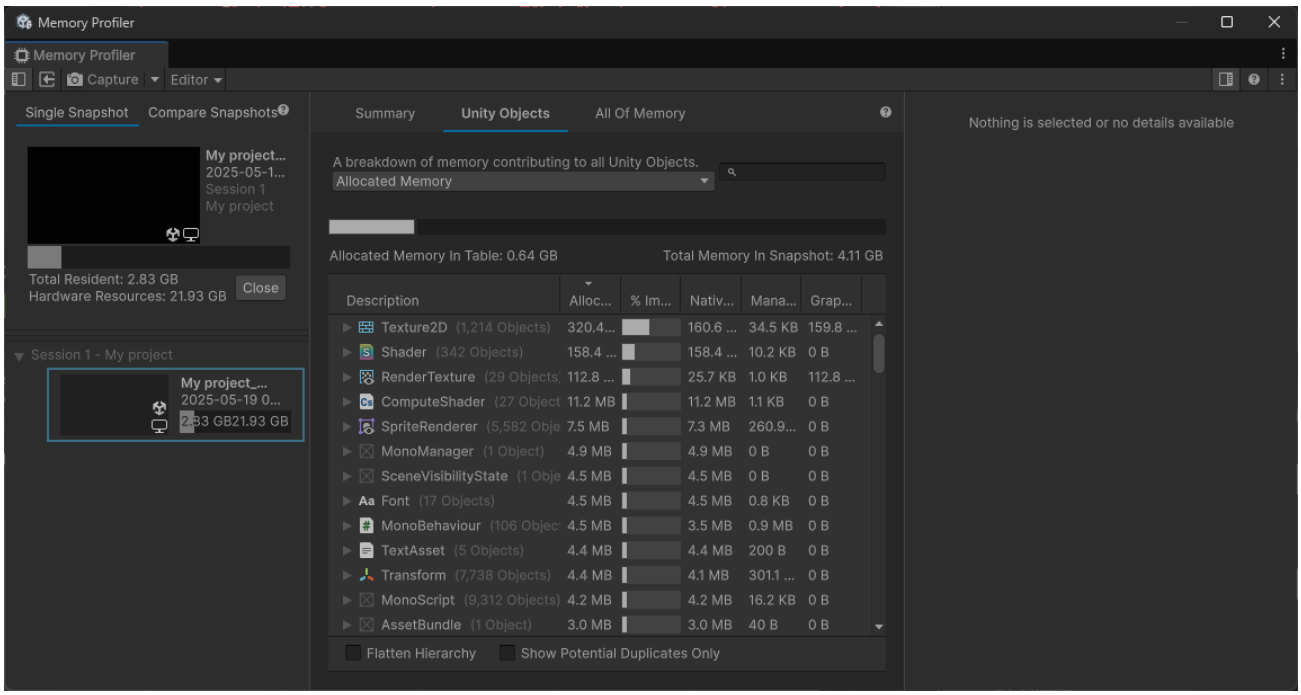

Рисунок 4.4 – Сторонній аналізатор пам’яті

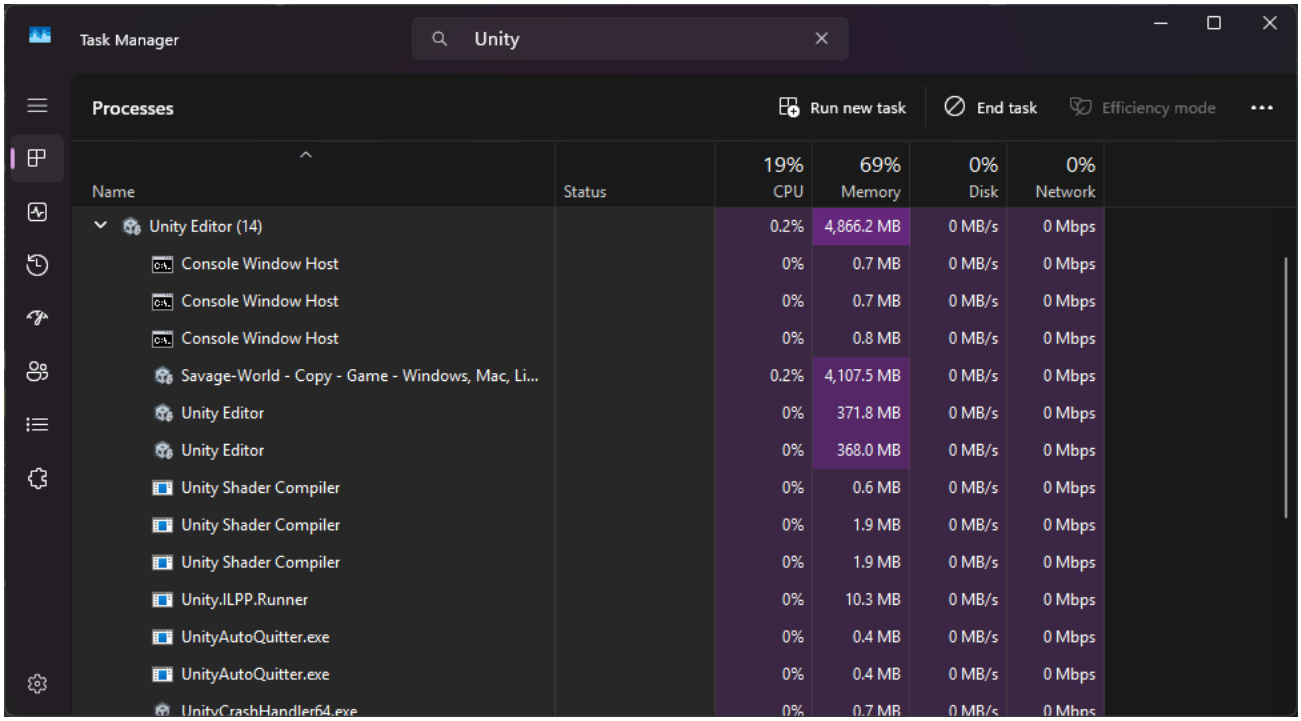

Рисунок 4.5 – Диспетчер задач

4.1.3 Метод тестування автоматизації

Тестування ефективності автоматизованої системи є доволі простим процесом. Результатом такого тестування є час, що необхідно витратити на розробку одного об’єкта системи. Так як в системі є різні рівні автоматизації: редактори, вікна та атрибути – кожен такий аспект буде перевірятись незалежно один від одного.

В якості вхідних параметрів для тестування вікон є ручне створення та налаштування без автоматизованих систем, та ідентичні дії із використанням створених моделей. Для розрахунку часу, що необхідно витратити на конфігурацію об’єктів буде використано два шляхи: використання вбудованих редакторів та створених. Для тестування атрибутів, необхідно також виконати ручну обробку та автоматичну.

4.2 Результати тестування

Для того, щоб перевірити рівень оптимізації методів та моделей, необхідно застосувати кожну вищеописану методику. Результати такого практичного експерименту можна подати у вигляді як таблиць так і графіків. Використовуючи такі графічні елементи, можна доволі легко виконати порівняння роботи системи за різних умов та при різному навантаженні.

4.2.1 Тестування моделі для паралельної обробки освітлення

Першим таким експериментом є тестування моделі для обробки освітлення із використанням як центрального процесора так і графічного. Як зазначалося вище, вхідними параметрами до тестування є розмір мапи освітлення (кількість стовпців та рядків у двовимірному масиві даних), режим кольору (використання монохромного режиму, або ж із використанням різних варіацій RGB) та кількість джерел світла (кількість

ігрових об’єктів, що генерують світлові ефекти).

Вихідні ж параметри в цьому випадку – це середня кількість кадрів на секунду (цей параметр дозволить просто та легко побачити, наскільки система змінює основний критерій оцінки продуктивності), CPU Time (частіш за все використовується при розрахунку кількості FPS) та GPU Time (це час, за який один кадр генерується). Сукупність усіх цих даних, дозволить легко та просто виконати компаративний аналіз у майбутньому.

Для отримання найбільш реально-приближених результатів, тестування буде відбуватись за наступними правилами. По-перше, для отримання CPU та GPU Time, необхідно використовувати вбудований інструмент профілювання – Profiler. Ця технологія дозволить швидко, та головне точно, отримати необхідні показники. По-друге, для отримання стабільної кількості FPS, система буде тестуватись від 15 до 30 секунд, із замірами показників. Після цього буде взято середній показник.

За дотриманням усіх умов, було проведено 3 практичних тести. Перший – залежність вихідних параметрів від розміру світлової карти (таблиця 4.1).

Таблиця 4.1 – Результати тестування залежності FPS, CPU Time та GPU Time від розміру світлової мапи

Вхідні параметри		Вихідні парамтери		
Ширина	Довжина	Середній FPS	CPU Time, ms	GPU Time, ms
100	100	130	0.528	0.905
150	150	115	0.602	1.231
300	300	80	2.125	5.365
500	500	50	6.254	14.826

За результатами можна побачити, що середній рівень кількості кадрів та час, що витрачається на обробку світлових ефектів змінюється нелінійно. Це означає, що система готова витримати більшість основних критичних

ситуацій без різкого падіння продуктивності. Додатково результати наведені у вигляді графіків (рисунки 4.6-4.7).

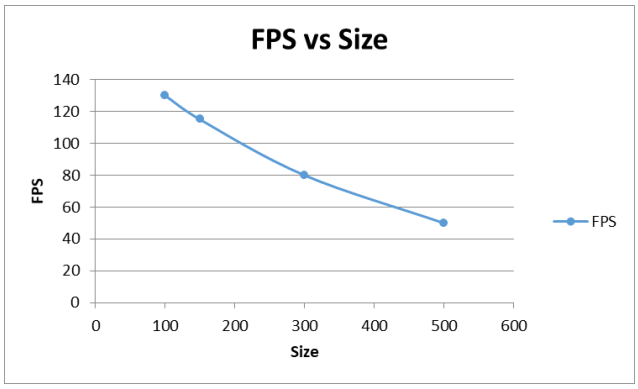

Рисунок 4.6 – Графік залежності FPS від розміру мапи

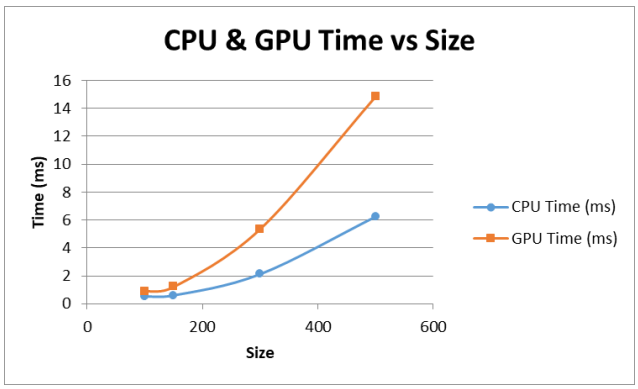

Рисунок 4.7 – Графік залежності CPU та GPU Time від розміру мапи

Другий тест – перевірка залежності вихідних параметрів від режиму кольору (таблиця 4.2).

Таблиця 4.2 – Результати тестування залежності FPS, CPU Time та GPU Time від розміру світлової мапи та режиму кольору

Вхідні параметри			Вихідні парамтери		
Ширина	Довжина	Режим кольору	Середній FPS	CPU Time, ms	GPU Time, ms
1	2	3	4	5	6
100	100	Монохром	152	0.512	0.634

1	2	3	4	5	6
100	100	Кольоровий	147	0.605	0.705
300	300	Монохром	85	3.025	6.542
300	300	Кольоровий	77	3.254	6.853

За результатами можна побачити, що середні показники кількості кадрів та часу не піддаються впливу режиму кольору. Основна залежність – це розмір світлової мапи. З результатів також було побудовано графіки (рисунки 4.8-4.9)

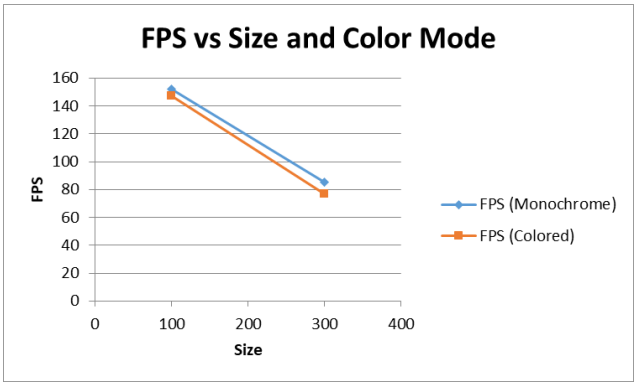

Рисунок 4.8 – Графік залежності FPS від розміру мапи та режиму кольору

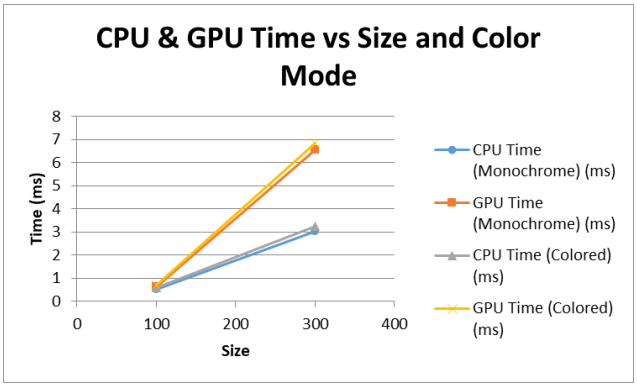

Рисунок 4.9 – Графік залежності CPU та GPU Time від розміру мапи та режиму кольору

Третій тест – залежність вихідних параметрів від кількості джерел світла (таблиця 4.3).

Таблиця 4.3 – Результати тестування залежності FPS, CPU Time та GPU Time від розміру світлової мапи та кількості джерел світла

Вхідні параметри			Вихідні парамтери		
Ширина	Довжина	Кількість джерел світла	Середній FPS	CPU Time, ms	GPU Time, ms
100	100	10	152	0.512	0.634
100	100	50	150	0.554	0.677
100	100	100	146	0.571	0.628
100	100	500	151	0.567	0.682

За результатами можна побачити, що кількість джерел освітлення ніяк не впливає на кінцевий результат. Пов’язано це з логікою обробки таких елементів. З результатів було побудовано графіки (рисунки 4.10-4.11)

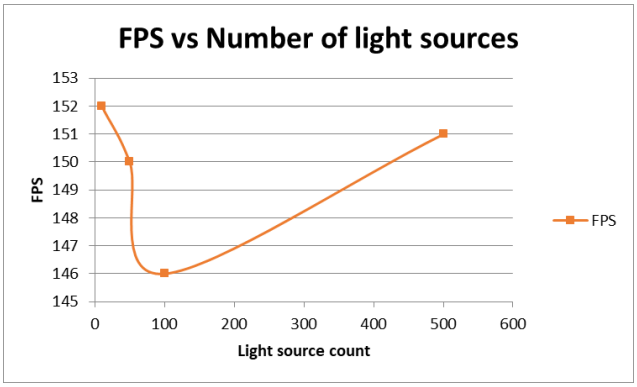

Рисунок 4.10 – Графік залежності FPS від кількості джерел світла

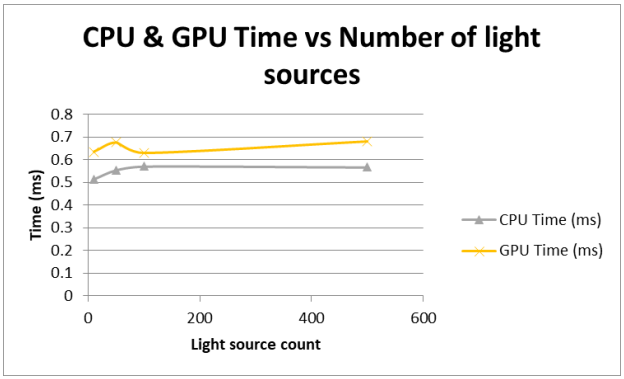

Рисунок 4.11 – Графік залежності CPU та GPU Time від кількості джерел світла

4.2.2 Тестування моделі оптимізації структури об’єктів

Наступним необхідним експериментом є тестування пам’яті що витрачається при використанні різного роду підходів зі створеного алгоритму оптимізації структури об’єктів.

Для цього кількість вхідних даних змінюються від тесту до тесту. Загалом буде тип розташування, розмір буферної області та тип даних. Вихідними ж даними є загальний розмір та розмір одного елемента.

Для того, щоб протестувати різні підходи необхідно виконати низку невеликих тестів. Умови доволі прості: розмір ідеального ігрового об’єкта дорівнює 13 байт, а кількість елементів у матриці 8400 на 2400, що дорівнює 20 16 000 елементів. Усього буде проведено два тести. Перший на залежність використаної пам’яті від типу розташування полів у структурі, результати якого представлені у вигляді таблиці 4.4.

Таблиця 4.4 – Результати тестування залежності витраченої пам’яті від типу розміщення полів у пам’яті

Вхідні параметри		Вихідні парамтери	
LayoutKind	Pack	Розмір структури, біт	Загальна пам’ять, Мб
Sequential	1	13	262
Sequential	2	14	282.24
Sequential	4	16	322.56
Auto (за замовчуванням)	-	-	322.56

За результатами, можна побачити пряму залежність кількості використаної пам’яті від розміру вирівнювання, що налаштовується параметром Pack. Чим менше значення – тим краще результат. Також для наглядності результати подані у вигляді скріншотів з профайлера Visual Studio (рисунки 4.12-4.15).

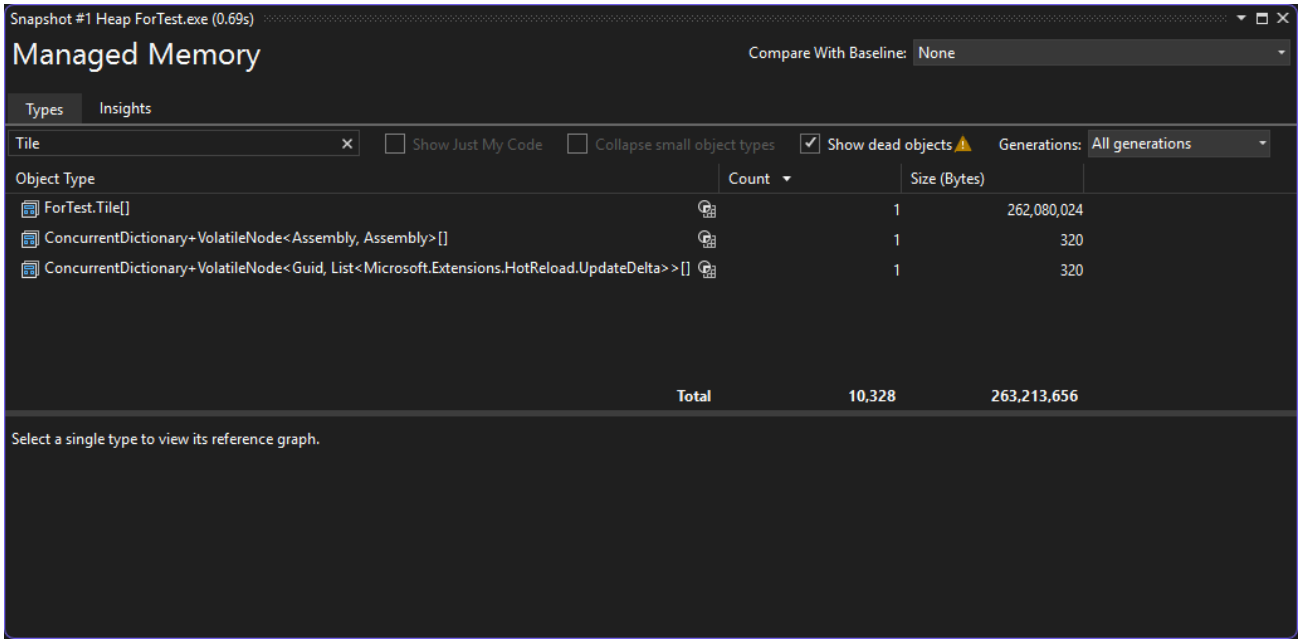

Рисунок 4.12 – Загальна пам’ять при використанні Pack = 1

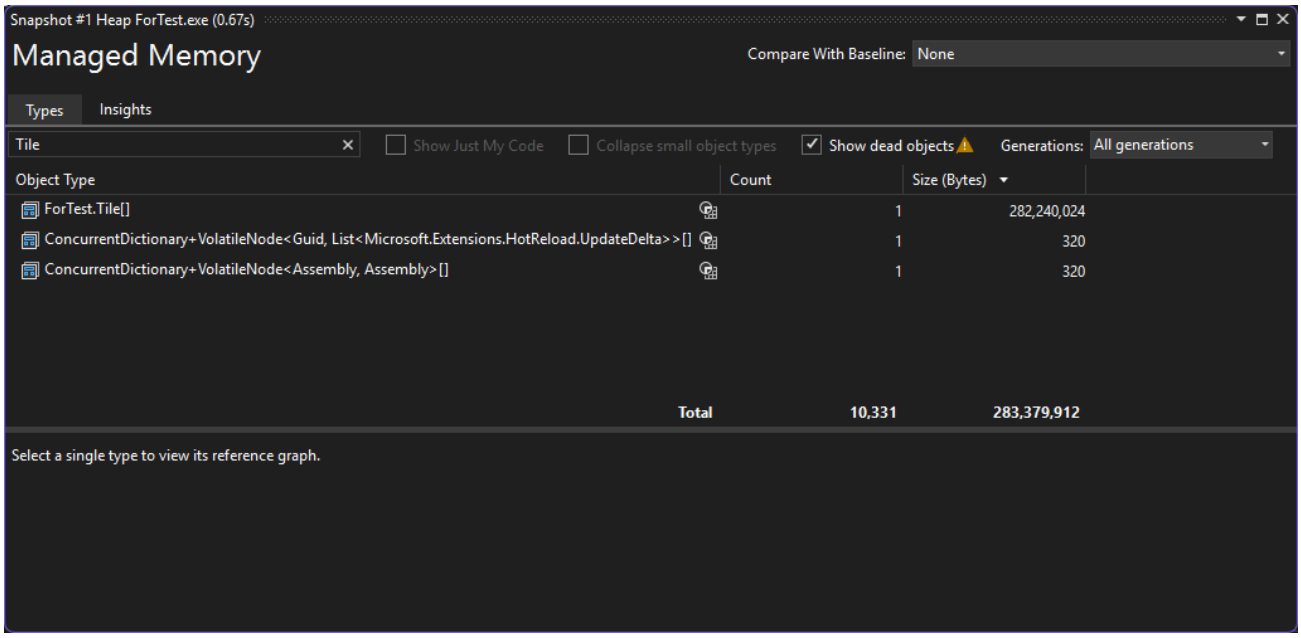

Рисунок 4.13 – Загальна пам’ять при використанні Pack = 2

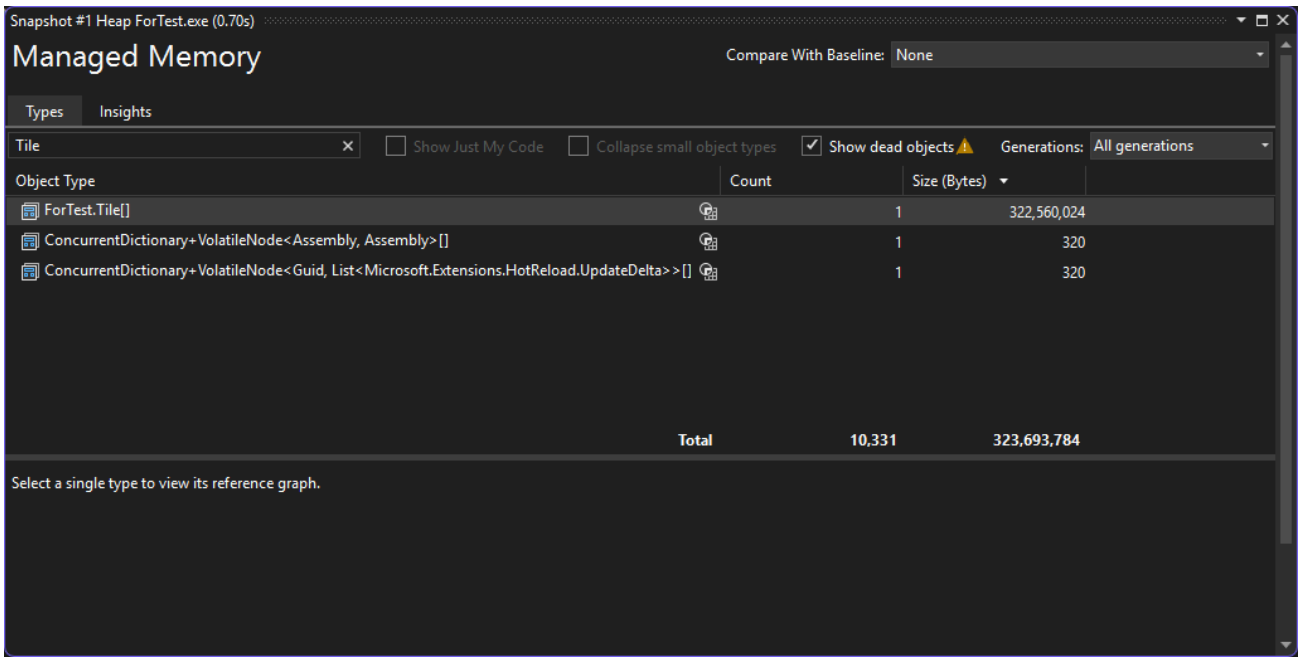

Рисунок 4.14 – Загальна пам’ять при використанні Pack = 4

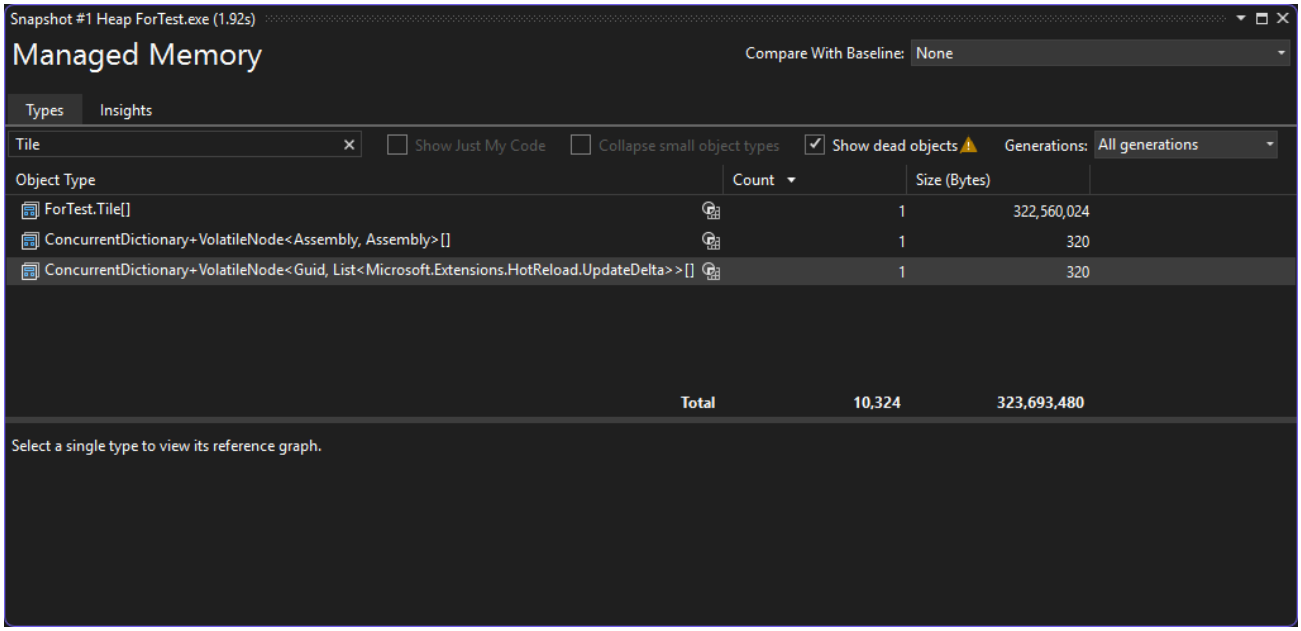

Рисунок 4.15 – Загальна пам’ять при використанні LayoutKind = Auto

Другий тест – перевірка залежності використаної пам’яті від типу даних (клас чи структура) (таблиця 4.4).

Таблиця 4.5 – Результати тестування залежності витраченої пам’яті від типу даних

Вхідні параметри	Вихідні парамтери	
Тип даних	Розмір структури, біт	Загальна пам’ять, Мб
Struct	13	262
Class	13	806.4

За результатами можна побачити, що використання структури замість класу є кращим. Кількість пам’яті, що зменшилась приблизно 2.5 рази. Усі результати також наведені у вигляді скріншотів (рисунки 4.16-4.17)

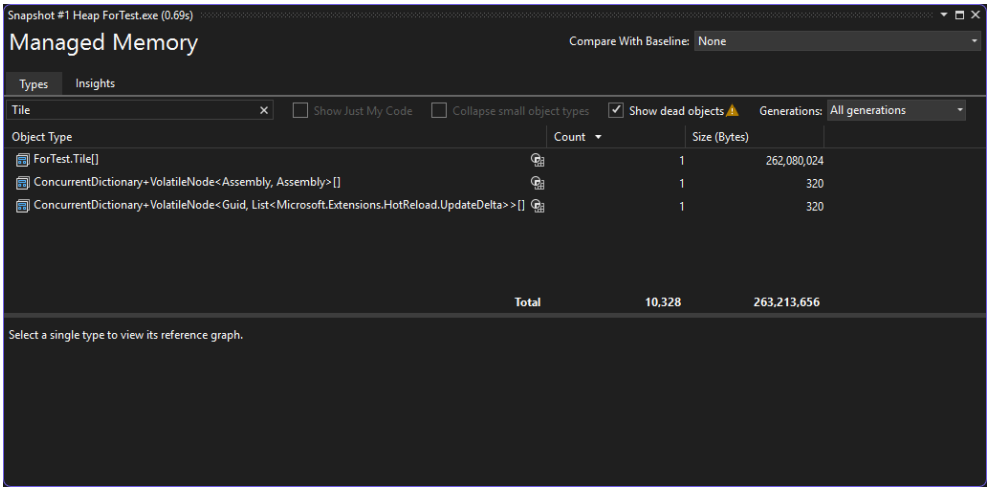

Рисунок 4.16 – Загальна пам’ять при використанні struct

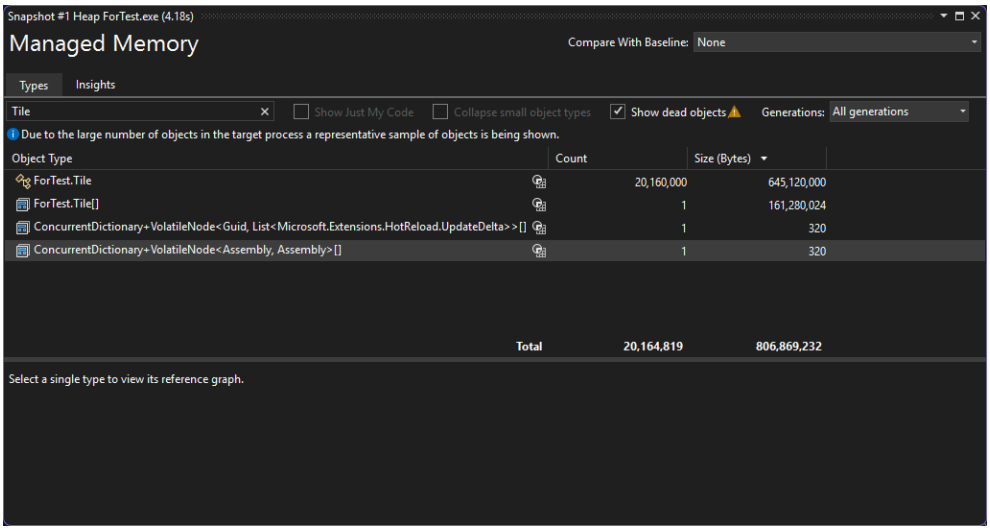

Рисунок 4.17 – Загальна пам’ять при використанні class

4.2.3 Тестування автоматизації

Тестування автоматизації можна вважати найпростішим, адже по суті це банальне вимірювання часу секундоміром.

В якості вхідних параметрів є 3 набори. Перший підходить до тестування редакторів. До нього входить варіація редактору (власний та за замовчуванням). Другий набір складається з варіації вікон. Третій з варіацій підходу обробки функції.

Усього в системі буде 3 тести: робота з редактором, вікнами та атрибутами. Перший тест складається з 2 кроків: відкрити об’єкт та відредагувати необхідні поля. Результатами такого тесту є часова таблиця 4.6.

Таблиця 4.6 – Результати тестування швидкості конфігурації об’єкта

Вхідні параметри	Вихідні парамтери
Тип редактору	Час, с
Вбудований	15
Власний	9

За результатами можна побачити різницю у часі між налаштуванням певних об’єктів із використанням вбудованого рішення та власного. Ця різниця може варіюватися в залежності від швидкості дій розробників та відклику системи в якому створюється гра. Проте частіш за все, показник залишається однаковим. Результати також подані у вигляді діаграми для наглядності (рисунок 4.18).

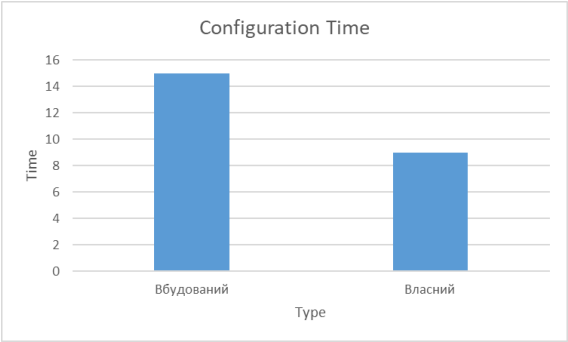

Рисунок 4.18 – Діаграма часу конфігурації об’єкта за різних редакторів

Другий тест містить 3 кроки: створити об’єкт, відредагувати його та зберегти. Також для цього тесту є часова таблиця 4.7.

Таблиця 4.7 – Результати тестування швидкості створення та конфігурації об’єкта

Вхідні параметри	Вихідні парамтери
Тип вікна	Час, с
Вбудований	25
Власний	18

За результатами також можна спостерігати значний спад часу створення нових елементів. Завдяки додатковій інтеграції покращених інтерфейсів цей час зменшився в кілька разів. Додатково створено діаграму (рисунок 4.19)

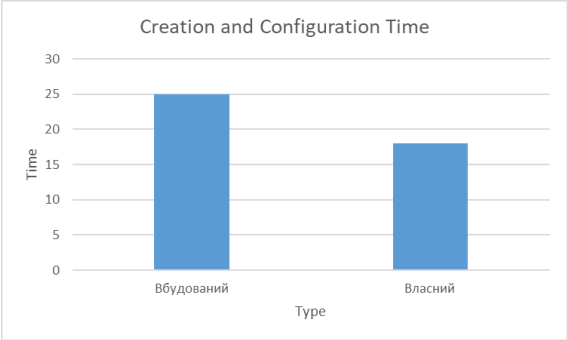

Рисунок 4.19 – Діаграма часу створення та налаштування об'єктів

Останнім тестом є перевірка швидкості виконання функції. Тут буде використовуватись інтерактивна кнопка, та вбудований в Unity функціонал. Результати також подані у вигляді таблиці 4.8.

Таблиця 4.8 – Результати тестування швидкості обробки функцій

Вхідні параметри	Вихідні парамтери
Тип кнопки	Час, с
Вбудований	4
Власний	2

За результатами, можна зробити висновок, що час тестування нової функції збільшився вдвічі. Сюди також не враховується той час, що витрачався на додавання атрибуту, адже він теж доволі великий. Результати подані також у вигляді діаграми (рисунок 4.20).

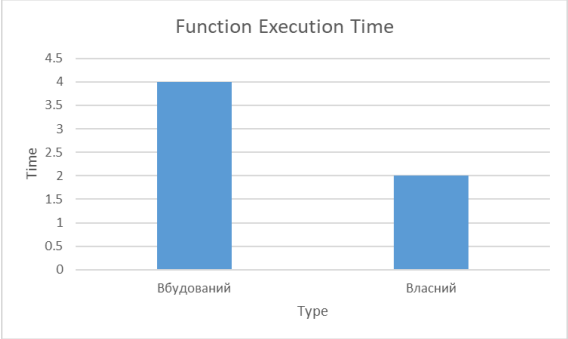

Рисунок 4.20 – Діаграма часу виклику функції

4.3 Аналіз ефективності методів

4.3.1 Аналіз результатів тестування моделі для паралельної обробки освітлення

Після усіх тестувань, збору інформації та її подання у вигляді таблиць та графіків, необхідно підбити підсумки та проаналізувати наскільки

розроблені методи та моделі є ефективними.

Для початку проаналізовано модель паралелізації. Після проведення низки експериментів, можна сказати, що система працює стабільно та в деяких випадках значного збільшення на навантаження не створюється. Розбираючи перший тест, можна побачити, що хоч система має розмір мапи 100 на 100, що в результаті дорівнює 10 000 елементів для обробки, хоч система має розмір 500 на 500, що в свою чергу дорівнює 250 000 елементів, система звісно збільшила навантаження, проте не в 25 разів, а приблизно в 12. Такий результат насправді є доволі непоганим, адже це говорить про стабільність системи та її непоганий рівень оптимізації. Як додатково, можна розглянути механізми кешування та обробки лише кластерів даних, для додаткового зменшення навантаження. Такий підхід може ще сильніше зменшити навантаження, адже замість постійної обробки великої кількості інформації, модель буде оброблювати лише частини цього набору даних. Результати другого тестування показують, що режим кольоровості майже ніяк не впливає на систему. Якщо користувач буде грати з кольоровим режимом або ж монохромним, явних просадок у FPS помітно не буде. Порівнюючи час обробки одного кадру ЦП та ГП, можна зробити висновок, що система стабільна за різних умов якості. Третій тест показує, що хоч система і працює з генерацією світлових ефектів, вона розглядає усі джерела як одну сукупність, оброблюючи їх разом та не маючи великого навантаження на систему. Це означає, що кількість джерел світла майже не впливає на кінцевий результат. Також це говорить про достатній рівень оптимізації.

4.3.2 Аналіз результатів тестування алгоритму оптимізації структур інформації

Наступним кроком йде аналіз результатів тестів у роботі з оптимізацією структур даних. Усього було проведено два тести. Результати

першого доказують пряму залежність розміру структури від положення її полів у пам’яті. Якщо між полями, додатково виконується вирівнювання (що часто застосовується при стандартній роботі структур та класів), то за замовчуванням цей параметр використовує значення 4. Це означає, що всі або майже всі поля будуть модифіковані, додаванням до кінцевої точки додаткові байти інформації. Це є неоптимізованим рішенням, особливо беручи до уваги ситуації, коли об’єктів в системі зустрічаються мільйони разів. Використання LayoutKind явно показують різницю у результатах, з чого можна зробити висновок, що найбільш оптимальним рішенням є використання Sequential параметру. Якщо ж використовувати атрибут за замовчуванням або ж взагалі його не використовувати, то по стандарту до полів застосовується вирівнювання де параметр Pack = 4.

Другий тест показує залежність пам’яті від обраного типу для збереження даних. При використанні масиву структур та масиву класів, було досліджено, що на виділення пам’яті під масив структур, в пам’яті напряму для кожного елемента створюється екземпляр структури, що в свою чергу забезпечує нас тим, що додатково пам’ять на посилання виділятися не буде. У такого рішення є як плюси: гарний рівень оптимізації та швидкість обробки інформації – так і мінуси: незмінність інформації, додаткове клонування при спробі призначити структуру до іншої змінної. Мінуси можуть здатися надто великими, проте вони оправдані. Адже за результатами, можна побачити, що різниця у використанні пам’яті між структурами та класами становить майже 4 рази. Уявімо, що система використовує більше полів. Наприклад для кожного блоку, необхідно розробити динамічне пошкодження його частин. Якщо блок використовує текстури розміром 8 на 8, то це можуть бути додаткові 8 байт інформації, що в свою чергу збільшить розмір майже в 2 рази.

4.3.3 Аналіз результатів тестування автоматизації

Найпростіші тести – перевірка автоматизації. Це по суті звичайна перевірка часу ручної роботи. В цьому контексті це використовувалось для створення об’єктів, їх налаштування та обробки функцій як інтерактивних об’єктів.

За результатами першого тесту, можна побачити, що час на налаштування об’єктів значно зменшився. Це означає, що при частому створенні та налаштуванні це буде не тільки швидко, але й зручно. Система налаштована під конкретну область, проте її також можна модифікувати та легко оновлювати. Це робить таку систему гнучкою та ефективною.

Другий тест, показує, що використання редакторів плюс інтеграція користувацьких вікон – гарна та ефективна комбінація. Це не тільки швидка система, але й ефективна зі сторони користувацького інтерфейсу. Табличні результати показують значне прискорення. Проте це не єдиний плюс. Адже завдяки додатковим функціям, система автоматично знає куди зберігати необхідний файл, що робить її ще більш зручною.

Останній експеримент показує ефективність тестування різних функцій. В Unity є певний обмежений варіант такої розробки, проте візуально він незручний, адже не додає ніякого елементу інтерфейсу. Поточне ж рішення якраз таки додає, що робить його кращим, адже в майбутньому можна налаштувати також положення цієї кнопки та певну поведінку.

ВИСНОВКИ

У ході кваліфікаційної роботи було проведено аналіз загальних помилок при розробці ігрових застосунків. Такий аналіз дозволив визначити основні проблеми надмірного споживання ресурсів комп’ютерної системи та отримати нові знання в області їх практичного вирішення. Розроблені технології дозволяють покращити оптимізацію та збільшити продуктивність при створенні ігрових застосунків.

Було розглянуто систему освітлення від компанії Unity – URP. Створено власну модель, що націлена на вирішення проблем, які виникають при інтеграції URP до проєкту з 2D перспективою. Використовуючи ці дві системи було зроблено компаративний аналіз характеристик та результатів роботи. За підсумками цих досліджень встановлено, що створена система є більш оптимізованою у порівнянні з існуючою, що робить її унікальною для конкретної предметної області та задачі.

Для вирішення проблеми надмірного використання оперативної пам’яті в ігрових застосунках, було виконано дослідження впливу різних факторів на ступінь навантаженості системи. У ході цієї частини роботи проаналізовано існуючі методи оптимізації структури даних. На основі цих моделей та методів створено алгоритм дій, що націлений на вирішення проблеми, яка нерідко може виникнути у початківців. Для створення цього алгоритму було проведено дослідження збереження різного роду інформації в пам’яті, вплив вбудованих інструментів, використання різних мехінізмів перевикористовування. За підсумками цієї частини роботи встановлено, що дотримання основних вимог та правил цього алгоритму довзолить зменшити навантаження на оперативну пам’ять, що в свою чергу може позитивно вплинути на роботу інших компонентів системи.

Для оптимізації людського ресурсу, а саме часу, було використано можливості ігрового рушія Unity. Завдяки цьому, було розроблено кілька

систем, що дозволяють полегшити та автоматизувати процес менеджменту різних компонентів системи. Завдяки такій розробці, графічно показано різницю між вбудованими можливостями та набутими. Проаналізовано результат, з чого встановлено, що системи мають більш інтуїтивно зрозумілий інтерфейс, полегшену систему управління та зручну систему пошуку. Для більш швидкої побудови користувацього інтерфейсу створено кілька спеціальних атрибутів. Така інтеграція дозволить розробникам швидше налаштовувати систему та керувати нею.

Більшість отриманих результатів може бути використана при розробці будь-яких ігрових систем. Їх впровадження до проєкту надасть змогу оптимізувати більшість компонентів та знизити навантаження на них. Також з поміж них можна виділити одну технологію, що націлена на розв’язок конкретної проблеми – система освітлення. Цю модель доречно використовувати лише для 2D перспективи з умовою наявності динамічної зміни ігрового світу. Під такі критерії ідеально можуть підійти застосунки жанру «Sandbox» та усі можливі похідні від цього проєкти. Інші ж технології можна дослідити та впровадити до проєкту незалежно від умов.

Подальша робота в цьому напрямку може включати детальне дослідження роботи шейдерів, написання власних графічних програм. Також додатково можна розглянути створення інших атрибутів, що націлені на пришвидшення побудови користувацького інтерфейсу.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Гейминг : скільки пам'яті потрібно? URL: https://www.kingston.com/ua/blog/gaming/how-much-memory-for-gaming (дата звернення: 06.05.2025).

2. Єнько Ю. Unity Job System на практиці. Як ми підвищили FPS з 15 до 70 у нашій грі. URL: https://gamedev.dou.ua/blogs/unity-job-system (дата звернення: 11.05.2025).

3. Сергеєв Д. В., Долгополов О. М., Фесенко Т. Г. Застосування rle алгоритму для створення ігрового контенту. *27-й Міжнародний молодіжний форум «Радіоелектроніка та молодь у XXI столітті»* : Міжнар. молодіж. форум, м. Харків. Харків, 2023. С. 129–130.

4. Сергеєв Д. В., Долгополов О. М., Фесенко Т. Г. Особливості генерації структур 2d ігр на платформі unity. *27-й Міжнародний молодіжний форум «Радіоелектроніка та молодь у XXI столітті»* : Міжнар. молодіж. форум, м. Харків. Харків, 2023. С. 131–132.

5. Що таке GPU й чому він важливий? URL: https://blog.acer.com/ua/discussion/2279/scho-take-gpu-y-chomu-vin-vazhliviy (дата звернення: 02.05.2025).

6. Що таке рендеринг та фіналізація (rendering and finalizing)? URL: https://tseivo.com/b/MovieFreak/t/bgpqarx1kr (дата звернення: 10.05.2025).

7. Як вибрати процесор? URL: https://www.itbox.ua/ua/blog/Yak-vibrati-procesor/ (дата звернення: 03.05.2025).

8. Burst user guide. URL: https://docs.unity3d.com/Packages/com.unity.burst@0.2/manual/index.html (дата звернення: 20.05.2025).

9. Evans C. Tips for optimizing performance and reducing memory usage in unity games. URL: https://faun.pub/tips-for-optimizing-performance-and-reducing-memory-usage-in-unity-games-1d48f5b6b86c (дата звернення:

13.05.2025).

10. Dev N. S. Learn unity editor scripting: component inspector editors (part 3). URL: https://nosuchstudio.medium.com/learn-unity-editor-scripting-component-inspector-editors-part-3-ae4707ec3e96 (дата звернення: 21.05.2025).

11. NVIDIA nsight systems. URL: https://developer.nvidia.com/nsight-systems (дата звернення: 04.05.2025).

12. Optimization techniques in game development. URL: https://codefinity.com/blog/Optimization-Techniques-in-Game-Development (дата звернення: 16.05.2025).

13. Optimize performance and quality. URL: https://unity.com/features/profiling (дата звернення: 16.05.2025).

14. Oteir N. CPU або GPU: зробіть правильний вибір. URL: https://introserv.com/ua/blog/cpu-abo-gpu-zrobit-pravilnij-vibir/ (дата звернення: 02.05.2025).

15. Oteir N. Обчислення на GPU: сфери застосування. URL: https://introserv.com/ua/blog/obchislennya-na-gpu-sferi-zastosuvannya/ (дата звернення: 04.05.2025).

16. RAM і VRAM: у чому між ними різниця? URL: https://gsminfo.com.ua/58752-ram-i-vram-u-chomu-mizh-nymy-riznyczya.html (дата звернення: 07.05.2025).

17. RenderDoc. URL: https://renderdoc.org/ (дата звернення: 09.05.2025).

18. Unity – Manual: create a compute shader. URL: https://docs.unity3d.com/Manual/class-ComputeShader-create.html (дата звернення: 15.05.2025).

19. Unity – manual: frame debugger window reference. URL: https://docs.unity3d.com/Manual/frame-debugger-window.html (дата звернення: 17.05.2025).

20. Unity – manual: occlusion culling. URL: https://docs.unity3d.com/Manual/OcclusionCulling.html (дата звернення: 10.05.2025).

21. Unity – manual: optimize shaders. URL: https://docs.unity3d.com/Manual/SL-ShaderPerformance.html (дата звернення: 08.05.2025).

22. Unreal Insights в Unreal Engine. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-insights-in-unreal-engine (дата звернення: 15.05.2025).

23. Venkat A. Unity job system and burst compiler: getting started. URL: https://www.kodeco.com/7880445-unity-job-system-and-burst-compiler-getting-started (дата звернення: 17.05.2025).

24. West C. Why occlusion culling improves performance. URL: https://gamedevchris.medium.com/why-occlusion-culling-improves-performance-887ae292e2a1 (дата звернення: 13.05.2025).

25. Фесенко Т.Г. Долгополов О.М., Сергеєв Д.В., Сергородцев І.Д., Жук М.В. Інформаційні технології для створення музично-ігрових проектів: бібліометричний аналіз. Збірник наукових праць. Системи управління, навігації та зв’язку, 2025, Том 2, №80.