Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп’ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня

магістра
(назва освітньо-кваліфікаційного рівня)

студента *Савостяненко Володимира Івановича*
(ПІБ)

академічної групи *121М-22-1*
(шифр)

спеціальності *121 Інженерія програмного забезпечення*
(код і назва спеціальності)

освітньої програми *«121 Інженерія програмного забезпечення»*
(назва освітньої програми)

на тему: *Технічний аналіз гри "Crypto Idle Transport Tycoon"*
оптимізація для покращення ігрового досвіду

_________________ *В.І. Савостяненко*

Керівники	**Прізвище, ініціали**	**Оцінка за шкалою**		**Підпис**
		рейтинговою	**інституційною**	
розділів кваліфікаційної роботи				
спеціальний	*Доц. Приходчекно С.Д.*			

Рецензент				

Нормоконтролер	*доц. Гуліна І.Г.*			

Дніпро
2023

Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

ЗАТВЕРДЖЕНО:
Завідувач кафедри
Програмного забезпечення комп'ютерних систем
(повна назва)

__________ М.О. Алексєєв
(підпис) (прізвище, ініціали)

« » 202 3 року

ЗАВДАННЯ
на виконання кваліфікаційної роботи

спеціальності *121 Інженерія програмного забезпечення*
(код і назва спеціальності)

студенту *121м-22-1* *Савостяненко Володимиру Івановичу*
(група) (прізвище та ініціали)

Тема кваліфікаційної роботи *Технічний аналіз гри “Crypto Idle Transport Tycoon” оптимізація для покращення ігрового досвіду.*

1 ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Наказ ректора НТУ «Дніпровська політехніка» від 09.10.2023 наказ №1227-с

2 МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень – гра "Crypto Idle Transport Tycoon" створена на платформі Unity.

Предмет досліджень – технічні аспекти гри, які впливають на ігровий досвід, включаючи продуктивність, графіку, оптимізацію ресурсів, а також інші технічні параметри.

Мета НДР – Розробити та виконати технічний аналіз гри "Crypto Idle Transport Tycoon" з метою ідентифікації можливостей оптимізації, щоб покращити загальний ігровий досвід для гравців. Метою є покращення продуктивності, зменшення завантаження ресурсів, поліпшення графічної якості та загального відчуття гри.

Вихідні дані для проведення роботи – включають теоретичні та експериментальні дослідження, методи оптимізації гри "Crypto Idle Transport Tycoon" з метою покращення ігрового досвіду на слабких пристроях. Однією з основних проблем, які варто врахувати, є негативний фідбек, які користувачі спостерігають під час гри на менш потужних девайсах.

3 ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Новизна запропонованих рішень результатів, що очікуються, цього дослідження можуть бути новаторськими завдяки використанню передових технічних підходів для оптимізації гри "Crypto Idle Transport Tycoon." Це включає в себе використання інтелектуальних алгоритмів адаптації, що дозволяють грі адаптуватися до характеристик конкретного пристрою або налаштувань гравця для досягнення оптимальної продуктивності. Крім того, новизна полягатиме у використанні аналізу поведінки гравців для покращення динаміки гри та інші інноваційні підходи, спрямовані на покращення ігрового досвіду.

Практична цінність результатів полягатиме в розробці рекомендацій для подальших вдосконалень гри та розширенні функціональності, яка сприятиме залученню нових гравців і підвищенню їх часу в грі. Крім того, оптимізація гри "Crypto Idle Transport Tycoon" може забезпечити більш стабільну гру для гравців у майбутньому.

4 ВИМОГИ ДО РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ

Результати досліджень мають бути подані у вигляді, що дозволяє їх безпосереднє використання на виробництві без необхідності внесення суттєвих змін до розробки гри "Crypto Idle Transport Tycoon."

5 ЕТАПИ ВИКОНАННЯ РОБІТ

Найменування етапів робіт	**Строки виконання робіт (початок – кінець**
Аналіз технічних характеристик гри "Crypto Idle Transport Tycoon" для ідентифікації можливостей оптимізації та покращення ігрового досвіду гравців.	12.09.2023-30.09.2023
Розробка технічних рекомендацій для оптимізації гри з метою поліпшення продуктивності та графічної якості.	01.10.2023-31.10.2023
Використання програми та аналіз отриманих результатів	01.11.2023-12.12.2023

6 РЕАЛІЗАЦІЯ РЕЗУЛЬТАТІВ ТА ЕФЕКТИВНІСТЬ

Економічний ефект від реалізації результатів роботи очікується позитивним дослідження та оптимізації гри "Crypto Idle Transport Tycoon" призведе до позитивного економічного ефекту. Покращений технічний стан гри дозволить розробникам привабити більше гравців та збільшити їхню активність. Збільшення гравців може призвести до зростання прибутку гри через покупку додаткових ігрових предметів або послуг. Економічний ефект також може бути видимим у покращенні репутації розробників, що може призвести до підвищення інвестицій та можливості розробляти нові ігри або оновлювати існуючі.

Соціальний ефект від реалізації результатів роботи очікується позитивним покращений ігровий досвід для гравців сприятиме задоволенню та розвагам. Оптимізація гри може зробити її більш доступною для гравців із різними технічними можливостями, що підвищить соціальну включеність в ігровій спільноті. Соціальний ефект також проявляється у покращенні умов праці розробників гри, оскільки оптимізація може спростити та полегшити їхню роботу.

Завдання видав	______________	*Приходченко С.Д*
	(підпис)	(прізвище, ініціали)
Завдання прийняв до виконання	______________	*Савостяненко В.І.*
	(підпис)	(прізвище, ініціали)

Дата видачі завдання: *12.09.2023 р.*

Термін подання кваліфікаційної роботи до ЕК *20.12.2023*

РЕФЕРАТ

Пояснювальна записка: 78 с., 2 додатки, 34 джерела.

Об’єкт досліджень – гра "Crypto Idle Transport Tycoon," розроблена на платформі Unity.

Предмет досліджень – технічні аспекти гри, які впливають на ігровий досвід, включаючи продуктивність, графіку, оптимізацію ресурсів, а також інші технічні параметри.

Мета роботи: розробити та виконати технічний аналіз гри "Crypto Idle Transport Tycoon" з метою ідентифікації можливостей оптимізації для поліпшення ігрового досвіду гравців. Метою є покращення продуктивності, зменшення завантаження ресурсів, поліпшення графічної якості та загального відчуття гри.

Методи дослідження включають аналіз літературних джерел, а також використання передових технічних підходів для оптимізації гри.

Наукова новизна отриманих результатів полягає в застосуванні інтелектуальних алгоритмів адаптації для досягнення оптимальної продуктивності гри та в аналізі поведінки гравців для покращення ігрового досвіду.

Практична цінність роботи полягатиме в розробці рекомендацій для подальших вдосконалень гри та розширенні її функціональності, що сприятиме залученню нових гравців та підвищенню їхнього часу в грі. Оптимізація гри може забезпечити більш стабільну гру для гравців у майбутньому.

Область застосування включає в себе ігрову індустрію та розробників геройських ігор.

Економічний ефект від реалізації результатів роботи очікується позитивним завдяки збільшенню кількості гравців та їхньої активності, що може призвести до зростання прибутку гри через покупку додаткових ігрових предметів або послуг. Економічний ефект також може бути видимим у покращенні репутації розробників, що може призвести до підвищення інвестицій та можливості розробляти нові ігри або оновлювати існуючі.

Соціальний ефект від реалізації результатів роботи проявляється у покращенні ігрового досвіду для гравців та підвищенні соціальної включеності в ігровій спільноті. Оптимізація гри може зробити її більш доступною для гравців із різними технічними можливостями та покращити умови праці розробників.

Значення роботи та висновки. Розробка системи технічного аналізу гри "Crypto Idle Transport Tycoon" з метою її оптимізації для покращення ігрового досвіду дозволить істотно поліпшити продуктивність та графічну якість гри. Використання передових технічних підходів та інтелектуальних алгоритмів допоможе зробити гру більш привабливою для гравців, зменшити навантаження на їхні ресурси та підвищити загальний рівень задоволення від гри. Прогнозні припущення про розвиток досліджень. У майбутньому можливим є розширення досліджень у галузі оптимізації ігрових досвідів, включаючи подальшу роботу

над інтелектуальними алгоритмами адаптації, що дозволять грі підлаштовуватися під конкретні умови гравця або пристрою для досягнення максимальної продуктивності. Список ключових слів: Технічний аналіз гри, оптимізація, ігровий досвід, продуктивність, графічна якість, інтелектуальні алгоритми, адаптація, програмне забезпечення.

Список ключових слів: ТЕХНІЧНИЙ АНАЛІЗ ГРИ, ОПТИМІЗАЦІЯ, ІГРОВИЙ ДОСВІД, ПРОДУКТИВНІСТЬ, ГРАФІЧНА ЯКІСТЬ, ІНТЕЛЕКТУАЛЬНІ АЛГОРИТМИ, АДАПТАЦІЯ, ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ.

ABSTRACT

Explanatory Note: 78 pages, 2 appendices, 34 sources.

Research Object – "Crypto Idle Transport Tycoon," a game developed on the Unity platform.

Research Subject – the technical aspects of the game that affect the player's experience, including performance, graphics, resource optimization, and other technical parameters.

The objective of the study is to conduct a technical analysis of the game "Crypto Idle Transport Tycoon" to identify opportunities for optimization to enhance the gaming experience for players. The goal is to improve performance, reduce resource load, enhance graphic quality, and overall game satisfaction. Research methods include the analysis of literature sources and the use of advanced technical approaches for game optimization.

The scientific novelty of the results lies in the application of intelligent adaptation algorithms to achieve optimal game performance and in the analysis of player behavior to enhance the gaming experience.

The practical value of the work lies in developing recommendations for further game improvements and expanding its functionality, attracting new players, and increasing their engagement. Game optimization can provide a more stable gaming experience for players in the future.

The scope of application includes the gaming industry and the developers of heroic games. The economic impact of implementing the research results is expected to be positive, leading to increased player numbers and their activity, potentially resulting in higher game revenue through the purchase of additional in-game items or services.

The economic impact can also be observed in the improvement of the developers' reputation, potentially leading to increased investments and opportunities for developing new games or updating existing ones.

The social impact of implementing the research results includes an improved gaming experience for players and increased social inclusion in the gaming community. Game optimization can make the game more accessible to players with varying technical capabilities and improve working conditions for developers.

Significance and Conclusions. The development of a technical analysis system for the game "Crypto Idle Transport Tycoon" with the goal of optimizing it to enhance the gaming experience will significantly improve game performance and graphic quality. The use of advanced technical approaches and intelligent algorithms will make the game more attractive to players, reduce resource load, and increase overall satisfaction.

Predictions for Research Development. In the future, there is potential for further research in the field of optimizing gaming experiences, including continued work on intelligent adaptation algorithms that allow the game to adapt to specific player or device conditions to achieve maximum performance.

Keywords: TECHNICAL GAME ANALYSIS, OPTIMIZATION, GAMING EXPERIENCE, PERFORMANCE, GRAPHIC QUALITY, INTELLIGENT ALGORITHMS, ADAPTATION, SOFTWARE

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ECS - Entity-Component-System

Unity - багатоплатформовий інструмент для розроблення відеоігор і застосунків, і рушій, на якому вони працюють.

Jobsystem - це механізм в Unity, призначений для оптимізації продуктивності за допомогою паралельного виконання завдань.

DOTS - Data Oriented Technology Stack

ПК – персональний комп'ютер.

ЗМІСТ

ВСТУП
РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ І ПОСТАНОВКА ЗАВДАННЯ. 11
1.1. Задача технічного аналізу гри "Crypto Idle Transport Tycoon" 13
1.2. Огляд існуючих рішень 14
1.3 Постановка задачі 16
1.4 Висновки 17
РОЗДІЛ 2. ДОСЛІДЖЕННЯ МЕТОДІВ ВИРІШЕННЯ ЗАДАЧІ
2.1. Розробка моделі, методів та програмного забезпечення розрахунку ефективності роботи мобільних застосунків 19
2.2. Ациклічний граф 23
2.3. C# Job API 25
2.4. Переваги операції порівняння та обміну (CAS) 36
2.5. Планувальник завдань 42
2.6. NativeArray 44
РОЗДІЛ 3. Використання методів та аналіз отриманих результатів
3.1. Batching 45
3.2 Використання DOTS 49
3.3 Використання Pool 54
3.4 Використання Task.Factory 57
Висновки 62
Додаток А 65
Додаток Б 79

ВСТУП

Ігрова індустрія є однією з найбільш динамічних індустрій у сучасному світі. Вона постійно еволюціонує, пропонуючи гравцям нові враження і можливості. Розробка і вдосконалення ігор стають складнішими завдяки високим технічним вимогам і різноманітним платформам.

Однією з таких ігор є "Crypto Idle Transport Tycoon," створена на платформі Unity. Ця гра пропонує гравцям можливість відчути себе власниками великого транспортного імперії в умовах криптовалютного світу. Проте технічні аспекти гри, такі як продуктивність, якість графіки, оптимізація ресурсів та інші технічні параметри, можуть впливати на загальний ігровий досвід гравців.

Мета даної роботи полягає в проведенні технічного аналізу гри "Crypto Idle Transport Tycoon" з метою ідентифікації можливостей оптимізації для покращення ігрового досвіду. Наша мета - покращити продуктивність гри, зменшити навантаження на ресурси гравців, підвищити якість графіки та зробити гру більш захопливою для геймерів.

Для досягнення цієї мети ми використовуємо аналіз літературних джерел і передових технічних підходів, які допоможуть нам оптимізувати гру. Ми також досліджуємо інтелектуальні алгоритми для адаптації гри під конкретні умови гравців, що підвищить задоволення від ігрового процесу.

Результати цієї роботи матимуть практичне значення для розробників ігор, які зможуть вдосконалити свої продукти та залучити більше гравців. Крім того, оптимізація гри може забезпечити більш стабільний геймплей для існуючих і майбутніх гравців.

Висновки цієї роботи можуть бути застосовані в ігровій індустрії та сприяти поліпшенню ігрового досвіду для гравців з різними технічними можливостями. Наша робота має значення для розвитку ігор та покращення їх соціальної включеності в ігровій спільноті.

Ця робота допоможе зрозуміти, як оптимізувати гру "Crypto Idle Transport Tycoon" та зробити її більш захопливою для гравців. У майбутньому, ми можемо розширити наші дослідження, працюючи над ще більш ефективними інтелектуальними алгоритмами адаптації.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ І ПОСТАНОВКА ЗАВДАННЯ
1.1 Задача технічного аналізу гри "Crypto Idle Transport Tycoon"

Проведення технічного аналізу гри "Crypto Idle Transport Tycoon" передбачає докладне вивчення існуючих літературних джерел та методологій, пов'язаних із схожими технічними аспектами в ігровій індустрії. Основною метою цього аналізу є виявлення кращих практик та підходів, які можуть бути застосовані для оптимізації гри та покращення ігрового досвіду гравців.

Вже відомо, що гра "Crypto Idle Transport Tycoon" включає в себе вражаючу кількість 165 країн, реалістичний глобус та 2312 аеропортів, розташованих по кординатах у реальному світі. Кожен аеропорт може бути з'єднаний з іншими аеропортами для створення маршрутів. Оскільки маршрути враховують обидва напрямки (з аеропорту A до B і з аеропорту B до A), нам потрібно поділити загальну кількість можливих пар аеропортів на 2. Таким чином, загальна кількість можливих маршрутів визначається за допомогою комбінаторики: Загальна кількість маршрутів = (2312 * 2311) / 2 = 1336356 можливих маршрутів. Це число включає в себе всі можливі комбінації маршрутів між аеропортами у грі.

Однак важливо врахувати, що для кожного маршруту можна додати 5 літаків. Загальна кількість літаків, які можуть бути в грі, обчислюється, враховуючи, що на кожному можливому маршруті можна додати 5 літаків. Ми вже розрахували, що загальна кількість можливих маршрутів становить 1336356. Тепер, щоб знайти загальну кількість літаків, ми просто множимо кількість можливих маршрутів на кількість літаків на кожному маршруті: Загальна кількість літаків 6681780 літаків. Отже, в грі може бути 6681780 літаків, якщо для кожного маршруту можна додати 5 літаків. Це вражаюча цифра, і для ефективного управління такою кількістю ресурсів та маршрутів потрібні оптимальні технічні рішення.

Логіка генерації пасажирів: Кожен аеропорт має враховувати певну логіку для генерації пасажирів.

Ця логіка може базуватися на різних факторах, таких як географічне розташування аеропорту, рейтинг аеропорту тощо. Необхідно створити механізм, який забезпечить реалістичну та викликаючу геймплей логіку генерації пасажирів

Плавний рух літаків: Потрібно буде використовувати анімацію та інтерполяцію для плавного переміщення літаків від одного аеропорту до іншого.

Оптимізація ресурсів: З огляду на велику кількість аеропортів та літаків у грі, важливо мати оптимізований код, щоб гра працювала ефективно та не споживала занадто багато ресурсів комп'ютера гравця.

Задача технічного аналізу гри "Crypto Idle Transport Tycoon" полягає в розробці та впровадженні технічних рішень, які дозволять оптимізувати гру для покращення ігрового досвіду гравців. Для досягнення цієї мети необхідно вивчити існуючі методології управління розподіленими ресурсами, розробки і оптимізації маршрутів, а також управлінням великим обсягом даних.

1.2 Огляд існуючих рішень

Unity - це потужний двигун для створення ігор та програм, проте робота з великою кількістю об'єктів може стати викликом для продуктивності. Паралельність та асинхронність: Unity Jobsystem дозволяє виконувати обчислення паралельно на багатьох ядрах процесора, що призводить до покращення продуктивності гри. Завдяки асинхронному виконанню коду в інших потоках, головний потік залишається вільним, що робить гру більш стійкою та відзначно покращує ігровий досвід.

Однією з ключових аспектів технічного аналізу гри "Crypto Idle Transport Tycoon" є огляд літератури та існуючих програм, спеціально орієнтованих на оптимізацію гри з використанням "Unity Jobsystem" та асинхронного виконання коду в інших потоках у рамках "Entity Component System" (ECS).

DOTS - це частина ECS, призначена для роботи з великою кількістю даних. Вона використовує різні оптимізації, такі як робота з пам'яттю та багатоядерна обробка, для підвищення продуктивності гри.

Важливо також розробити свої власні оптимізовані алгоритми для роботи з великою кількістю об'єктів, особливо якщо ECS або DOTS не підходять для конкретної задачі.

Unity Jobsystem - це технологія, яка дозволяє оптимізувати роботу гри за рахунок паралельного виконання коду на багатьох ядрах процесора. Ця технологія може бути використана для покращення продуктивності гри, зменшення навантаження на головний потік та підвищення ігрового досвіду гравців .

JobHandle - використовується для керування порядком виконання завдань і забезпечення збіжності в асинхронних операціях. Ви можете створити ланцюги завдань і залежності між ними за допомогою JobHandle.

Інтерфейс IJob є частиною Unity Job System і використовується для організації завдань, які виконуються в асинхронних потоках з метою оптимізації продуктивності в іграх та додатках. Цей інтерфейс дозволяє розпаралелювати обчислення та виконувати їх одночасно на багатьох потоках процесора. IJobParallelForTransform є частиною Unity Job System і використовується для паралельної обробки даних компонентів Transform в контексті Entity Component System (ECS).Цей інтерфейс зазвичай використовується для виконання паралельних операцій над компонентами сутностей, які містять компонент Transform.

Для досягнення оптимальної продуктивності ігри "Crypto Idle Transport Tycoon," важливо виконувати обчислення та операції асинхронно в окремих потоках. Це дозволяє уникнути блокувань головного потока та забезпечити плавну гру.

Level of Detail (LOD) - це техніка, що дозволяє зменшити деталізацію об'єктів на великих відстанях від гравця. Це допомагає знизити навантаження на графічний процесор та пам'ять, особливо у великих відкритих світах.

Графіка відіграє ключову роль у візуальному сприйнятті гри, але вона також може бути джерелом збільшеного навантаження на графічний процесор. В даному розділі розглядається оптимізація графічного вмісту гри з метою покращення продуктивності. Велика кількість високорозширених текстур може збільшити вимоги до пам'яті та обчислювальної потужності. Оптимізація роботи з великою кількістю об'єктів у Unity - це важлива задача для розробників ігор. Використання ECS, DOTS, розпаралелювання та інших технік може покращити продуктивність та допомогти створити більш зручний користувацький досвід.

Важливо розглянути існуючі підходи та програмні рішення для асинхронної роботи в інших потоках та їхню застосовність у контексті ECS. З моменту релізу Unity Jobsystem та ECS на ринку ігрової розробки з'явилися нові практики та підходи щодо оптимізації ігрового досвіду.

1.3 Постановка задачі

Для досягнення поставленої мети - оптимізації гри "Crypto Idle Transport Tycoon" з метою покращення ігрового досвіду гравців - необхідно вирішити ряд ключових завдань:

1. Провести аналіз літератури та існуючих програм, спрямованих на оптимізацію ігор на платформі Unity, зокрема з використанням Unity Jobsystem та Entity Component System (ECS).
2. Дослідити технічні параметри гри, які впливають на ігровий досвід, такі як продуктивність, графіка, оптимізація ресурсів та інші технічні характеристики. Розробити програмне забезпечення для оптимізації гри "Crypto Idle Transport Tycoon" з використанням сучасних технологій, зокрема Unity Jobsystem та ECS. Метою є покращення продуктивності гри, зменшення завантаження ресурсів, покращення графічної якості та загального ігрового досвіду гравців.

3. Асинхронна генерація пасажирів та летючих літаків: Враховуючи великий обсяг географічних об'єктів та маршрутів, важливо розглянути можливість асинхронної генерації пасажирів та керування літаками. Введення асинхронності допоможе запобігти блокуванням головного потока та забезпечити більш плавний ігровий процес.
4. Враховуючи, що в грі присутні 165 країн та 2,312 аеропортів, потрібно розглянути оптимізацію обробки даних про ці аеропорти, їхню взаємодію та вплив на ігровий процес. Важливо розглянути способи зменшення обчислювального навантаження при взаємодії з таким великим обсягом географічних об'єктів.
5. Оптимізація маршрутної системи: З урахуванням можливості побудувати 1336356 маршрутів у грі та можливості додавати 5 літаків на кожному маршруті, важливо розглянути оптимізацію обробки та керування такою великою кількістю маршрутів та літаків. Розгляд способів оптимізації обчислювального процесу для забезпечення плавності гри та запобігання збоїв.
6. Включити в розроблене програмне забезпечення інтелектуальні алгоритми адаптації з метою досягнення оптимальної продуктивності гри та покращення ігрового досвіду гравців.
7. Розробити рекомендації для подальших вдосконалень гри та розширення її функціональності з метою залучення нових гравців та підвищення їхнього часу в грі.

1.4 Висновки

Технічний аналіз гри "Crypto Idle Transport Tycoon" виявив ряд ключових аспектів, що потребують оптимізації для покращення ігрового досвіду гравців. Велика кількість аеропортів, маршрутів та літаків створює великі виклики для продуктивності гри. Важливо використовувати технології, такі як Unity Jobsystem та Entity Component System (ECS), для роботи з багатьма об'єктами

паралельно та асинхронно з метою покращення продуктивності та зменшення завантаження ресурсів. Логіка генерації пасажирів та літаків має бути ретельно розроблена для забезпечення реалістичного геймплею і покращення ігрового досвіду. Оптимізація ресурсів та графічного вмісту важлива для зменшення навантаження на графічний процесор та пам'ять гравця. Існуючі рішення, такі як Unity Jobsystem, DOTS, та асинхронна обробка, можуть бути використані для покращення продуктивності та створення оптимізованого коду. Робота з великою кількістю даних в грі вимагає ретельного планування та оптимізації для досягнення оптимальної продуктивності.

Технічний аналіз гри "Crypto Idle Transport Tycoon" виявив ряд ключових аспектів, що потребують оптимізації для покращення ігрового досвіду гравців. Велика кількість аеропортів, маршрутів та літаків створює великі виклики для продуктивності гри. Важливо використовувати технології, такі як Unity Jobsystem та Entity Component System (ECS), для роботи з багатьма об'єктами паралельно та асинхронно з метою покращення продуктивності та зменшення завантаження ресурсів. Логіка генерації пасажирів та літаків має бути ретельно розроблена для забезпечення реалістичного геймплею і покращення ігрового досвіду. Плавний рух літаків від одного аеропорту до іншого потребує використання анімації та інтерполяції для створення плавного візуального ефекту. Оптимізація ресурсів та графічного вмісту важлива для зменшення навантаження на графічний процесор та пам'ять гравця. Існуючі рішення, такі як Unity Jobsystem, DOTS, та асинхронна обробка, можуть бути використані для покращення продуктивності та створення оптимізованого коду. Робота з великою кількістю даних в грі вимагає ретельного планування та оптимізації для досягнення оптимальної продуктивності.

РОЗДІЛ 2. ДОСЛІДЖЕННЯ МЕТОДІВ ВИРІШЕННЯ ЗАДАЧІ

2.1.Розробка моделі, методів та програмного забезпечення розрахунку ефективності роботи мобільних застосунків

У випуску 2017.3 було введено загальнодоступний C# API для внутрішньої системи Unity з програмування на C#, що дозволяє користувачам писати невеликі функції, відомі як "задачі" (jobs), які виконуються асинхронно. Мета використання задач замість звичайних функцій полягає в створенні API, яке робить легким, безпечним та ефективним виклик коду, який інакше виконувався б на основному потоці, на "робочих" потоках процесу, ідеально паралельно. Це допомагає зменшити загальний час, який основний потік потребує для завершення симуляції гри. Використання системи завдань для обчислювальної роботи на процесорі може призвести до значних покращень продуктивності та дозволити грі масштабуватися природно з поліпшенням обладнання, на якому вона запускається.

Якщо уявляти обчислення як обмежений ресурс, одне ядро CPU може виконати обмежену кількість обчислювальної "роботи" за певний період часу. Наприклад, якщо грі з одним потоком потрібно, щоб її функція оновлення (Update()) займала не більше 16 мс, а зараз вона займає 24 мс, то у CPU занадто багато роботи – потрібно більше часу. Щоб досягти цілі в 16 мс, є лише дві альтернативи: зробити CPU швидше (наприклад, підвищити мінімальні технічні вимоги до гри – як правило, не найкращий вибір) або зменшити обсяг роботи.

В кінцевому підсумку, вам потрібно зменшити обчислювальну роботу на 8 мс. Зазвичай це означає вдосконалення алгоритмів, розподіл роботи підсистеми на кілька кадрів, видалення зайвої роботи, яка може накопичуватися під час розробки і т. д.

Якщо це все ще не допомагає досягти цілі продуктивності, можливо, вам доведеться зменшити складність симуляції гри, зменшивши вміст та геймплей,

наприклад, скорочення кількості ворогів, які можуть з'явитися одночасно – що безумовно не є ідеальним варіантом.

Але що, якщо, замість того, щоб усувати роботу, ми передамо роботу для виконання на інше ядро CPU? Зараз більшість CPU мають кілька ядер, що означає, що доступна обчислювальна потужність для одного потока може бути помножена на кількість ядер, які є в CPU. Якщо ми могли б розділити всю роботу, яка зараз в функції Update(), між двома ядрами CPU, роботу в 24 мс (рис 2.1) можна було б виконати у два одночасних блоки по 12 мс. Це дозволило б нам значно знизити час до цільових 16 мс. Більше того, якщо ми могли б розділити роботу на чотири паралельні блоки і виконати їх на чотирьох ядрах, тоді Update() займала б лише 6 мс!

Такий тип розподілу роботи та виконання на всіх доступних ядрах відомий як масштабування продуктивності. Якщо додати більше ядер, ви можете ідеально виконувати більше роботи паралельно, скорочуючи час виконання Update() без змін коду.

На жаль, це фантазія. Ніщо не розділить функцію Update() на частини і не виконає їх на окремих ядрах без допомоги. Навіть якщо ми перейдемо на ЦП з 128 ядрами, функція Update() з вище зазначеним часом в 24 мс все одно займе 24 мс, при умові, що обидва процесори мають однаковий тактовий частоту. Яке марнотратство потенціалу! Як тоді можна написати програми, щоб використовувати всі доступні ядра ЦП та збільшити паралельність?

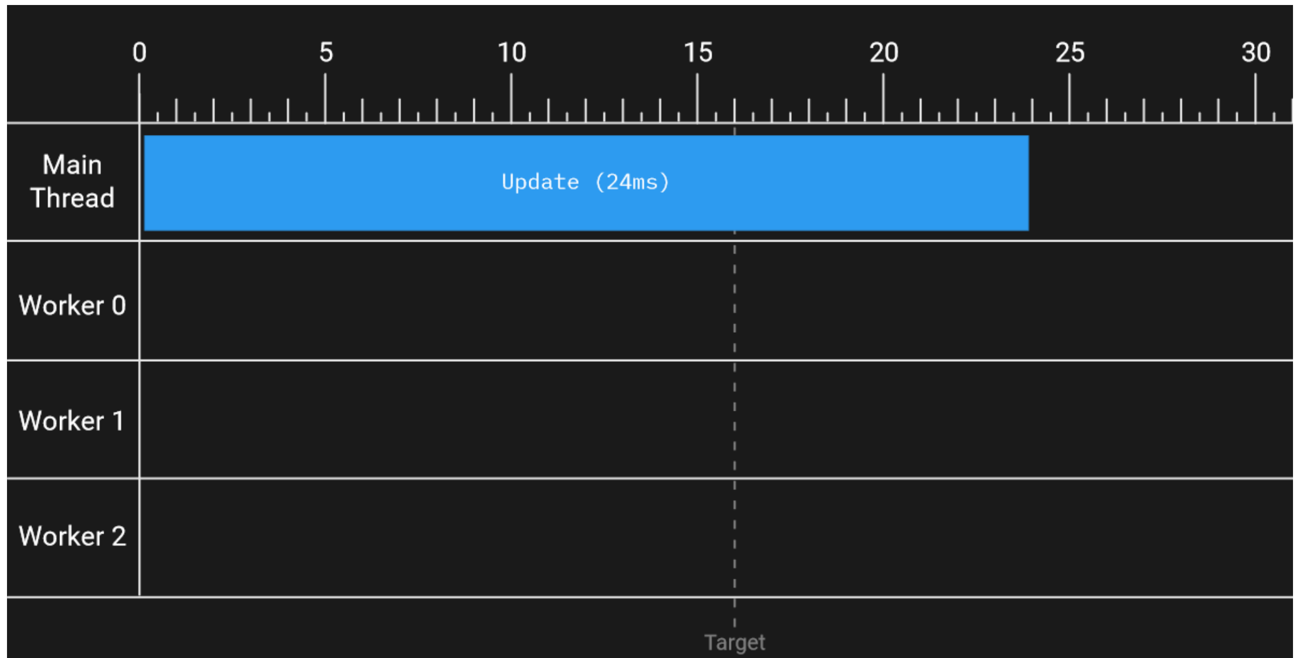

Рис. 2.1 Функція Update(), яка виконується протягом 24 мс на головному потоці

Один з підходів - це багатопотоковість. Тобто ваша програма створює потоки для виконання функції, яку операційна система планує виконувати за вас. Якщо у вашому ЦП є кілька ядер, то кілька потоків може працювати одночасно, кожен на своєму ядрі. Якщо потоків більше, ніж доступних ядер, операційна система визначає, який потік отримає можливість працювати на ядрі, і на скільки часу, переключаючись потім на інший потік - процес, відомий як перемикання контексту.

Проте багатопотокове програмування супроводжується кількома ускладненнями. У вищезазначеному казковому випадку функція Update() рівномірно розбивалася на чотири частини. Але насправді ви, ймовірно, не зможете зробити щось настільки просте. Оскільки потоки будуть працювати одночасно, потрібно бути обережними, коли вони читають і записують одні й ті ж дані в один і той же час, щоб уникнути взаємного впливу їхніх обчислень.

Це зазвичай включає використання синхронізаційних примітивів блокування, таких як м’ютекс чи семафор, для управління доступом до загального стану між потоками. Ці примітиви зазвичай обмежують рівень

паралелізму конкретних частин коду (зазвичай, вибираючи жодного зовсім), блокуючи інші потоки і заважаючи їм виконувати цю частину, поки тримач блоку не завершить роботу і не "розблокує" цю частину для будь-яких чекаючих потоків. Це зменшує ефективність використання кількох потоків, оскільки вони не працюють паралельно постійно, але це забезпечує коректність виконання програм. Ймовірно, також немає сенсу виконувати деякі частини вашого оновлення паралельно через залежності даних. Наприклад, майже у всіх іграх потрібно зчитати введення з контролера, зберегти це в буфер введення, а потім зчитати буфер введення і реагувати на значення.
Не має сенсу мати код, який читає буфер введення для вирішення того, чи має персонаж стрибати, виконуючись одночасно з кодом, який пише в буфер введення для оновлення кадру. Навіть якщо ви використовуєте м’ютекс, щоб забезпечити безпеку читання та запису в m_InputBuffer, ви завжди хочете, щоб m_InputBuffer спочатку було записано, а потім код читання m_InputBuffer запускався, так що ви знаєте, чи була натиснута кнопка стрибка для поточного кадру (і не для минулого). Такі залежності даних є звичайними і нормальними, але вони зменшують можливість паралелізму.

Існують різні підходи до написання багатопотокової програми. Ви можете використовувати платформоспецифічні API для прямого створення та управління потоками, або використовувати різні API, які надають абстракцію для управління деякими ускладненнями багатопотокового програмування.

Одним із таких відображень є система задач. Вона забезпечує можливість розбити частини вашого однопотокового коду на логічні блоки, визначити, які дані потрібні для цього коду, контролювати одночасний доступ до цих даних і виконувати якомога більше блоків коду паралельно, щоб використовувати всю обчислювальну потужність, доступну на ЦП за потреби.
Після створення екземпляра завдання його потрібно запланувати в системі завдань. Це робиться за допомогою методу .Schedule(), доданого до всіх типів завдань через механізм розширення C#. Для ідентифікації та відстеження запланованого завдання надається JobHandle.

Оскільки дескриптори завдань визначають заплановані завдання, їх можна використовувати для встановлення залежностей завдань. Залежності завдань гарантують, що заплановане завдання не почне виконуватися, доки його залежності не будуть завершені. Як прямий результат, вони також повідомляють нам, коли різні завдання можуть виконуватися паралельно, створюючи спрямований ациклічний графік завдань.

2.2. Ациклічний граф

В математиці, а саме в теорії графів та інформатиці, орієнтований ациклічний граф (DAG) - це граф, який має напрямлені зв'язки між вершинами і не містить замкнених циклів. Іншими словами, він складається з вершин і ребер (також відомих як дуги), де кожне ребро спрямоване від однієї вершини до іншої. Таким чином, відсутність циклів у графі гарантує відсутність повторюючихся шляхів через вершини.

Орієнтований граф вважається DAG, якщо його можна топологічно впорядкувати, тобто розташувати вершини у лінійному порядку, який узгоджений з напрямками ребер. DAG мають різні наукові та обчислювальні застосування, включаючи біологію (еволюція, генеалогічні дерева, епідеміологія), інформатику (мережі цитувань) та обчислення (планування).

Інколи орієнтовані ациклічні графи також називають ациклічними орієнтованими графами або ациклічними диграфами.

Щоб зрозуміти DAG, важливо зрозуміти визначення графа, який складається з вершин і ребер, де ребра мають напрямок від однієї вершини до іншої. Шлях у графі - це послідовність ребер, де кінцева вершина кожного ребра збігається з початковою вершиною наступного ребра (рис. 2.2). Контур утворює цикл, якщо початкова вершина першого ребра рівна кінцевій вершині останнього ребра. Орієнтований ациклічний граф - це такий граф, який не містить циклів.

Для визначення, що вершина v орієнтованого графа досяжна з іншої вершини u, використовується поняття шляху, який починається на u і

закінчується на v. Кожна вершина також вважається досяжною від самої себе, і якщо вершина може досягти самої себе через нетривіальний шлях, то граф має цикл. Орієнтовані ациклічні графи є графами, в яких немає циклів, і це може бути визначено тим, що жодна вершина не може досягти себе нетривіальним шляхом.

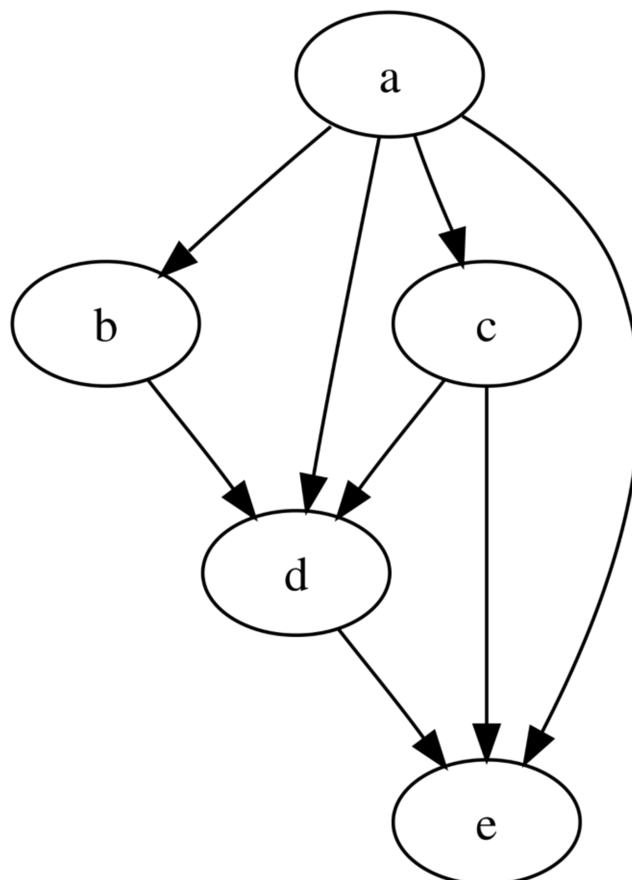

Рис. 2.2 Приклад орієнтованого ациклічного графа

Спрямовані ациклічні графи (DAG) знаходять широке застосування у плануванні систем задач з обмеженнями впорядкування. Основна ідея полягає в тому, щоб використовувати графічне представлення для керування послідовністю виконання завдань.

Задачі, такі як оновлення об'єктів, які взаємодіють між собою, наприклад, комірки електронної таблиці чи об'єктні файли в програмному забезпеченні, часто потребують планування відповідно до залежностей між ними. Граф залежностей утворюється, коли один об'єкт повинен бути оновлений перед

іншим. З циклічними залежностями у графі нам важко послідовно планувати завдання, тому такі графи повинні бути ациклічними (DAG).

Наприклад, при зміні однієї комірки електронної таблиці, потрібно перерахувати значення інших комірок, що залежать від зміненої. Планування оновлень комірок може бути виконано ефективно, використовуючи топологічне впорядкування графа залежностей.

Техніка оцінки та перегляду програм (PERT) використовує DAG для формулювання обмежень планування. У PERT вершини DAG представляють віхи проекту, а ребра - завдання, що з'єднують початок і завершення. Кожне ребро має вказану оцінку часу для виконання завдання. Задача полягає в плануванні віх проекту відповідно до найдовших шляхів, що визначають критичний шлях. Це дозволяє ефективно керувати часом проекту та заплановувати відповідні завдання.

Використання DAG також застосовується в інших областях, таких як компіляція програм і оптимізація інструкцій у комп'ютерних програмах.Спрямовані ациклічні графи є потужним інструментом для ефективного планування та визначення залежностей між завданнями. Їх використання в різних галузях віддзеркалює їхню універсальність та ефективність в управлінні послідовністю операцій.

2.3. C# Job API

Мета C# Job API полягає в наданні безпечного способу доступу до власної системи робіт. Хоча це є зв'язуючим шаром для переходу від C# до C++, це також шар, який дозволяє уникнути випадкового планування робіт C#, які можуть спричинити гонки або застоювання при доступі до NativeContainers з середини роботи.

Крім того, це розділення забезпечує більш багатий спосіб створення самих робіт. На рівні C++, роботи – це просто вказівник на якісь дані та вказівник на функцію. Але завдяки C# API зверху, ви можете налаштовувати типи робіт, які

ви плануєте, що дозволяє краще контролювати те, як дані роботи повинні бути розбиті та паралельно виконані для відповідності конкретним випадкам використання користувача.

При плануванні роботи зв'язуючий шар C# копіює структуру роботи в нерозміщену область пам'яті. Це дозволяє відокремити час життя структури роботи C# від часу життя роботи в системі робіт, оскільки це впливає на залежності роботи та загальне навантаження на платформу. Система робіт умовно виконує перевірки безпеки в режимі відтворення редактора, щоб забезпечити безпечний запуск роботи.

Ці кроки є важливими, але вони можуть бути вже не безкоштовними і сприяти накладенню завдань на систему. Оскільки розмір завдання може варіюватися, а також кількість NativeContainers та залежностей, які може мати завдання, вартість копіювання завдань та перевірки їх безпеки не є фіксованою. Через це важливо, щоб Unity утримувала витрати на мінімальному рівні та обмежувала їх до лінійної обчислювальної складності.

У Tech 2021.2 команда внесла значні поліпшення до системи безпеки завдань, кешуючі результат перевірки безпеки для окремих дескрипторів завдань. Це особливо важливо, оскільки системі безпеки потрібно розуміти всі ланцюги залежностей завдань та кожен вказівник на локальну пам'ять, яку містять всі завдання, щоб визначити, які можуть бути втрачені дані про залежності та до якого завдання слід додати залежність. Це може призвести до нелінійної кількості елементів для ітерації під час планування (тобто для кожного завдання та його залежностей перевіряється доступ на читання/запис для кожного NativeContainer, на який посилається завдання, і для будь-якого завдання, яке посилається на NativeContainers).

Однак Unity може скористатися тим, що завдання C# плануються лише по одному за раз, і перевірити їх безпеку під час цього планування. Замість повторного сканування всіх завдань при кожному плануванні, ми можемо швидко визначити, чи потрібно перевалідувати ланцюги залежностей завдань чи ні, що дозволяє пропустити велику кількість роботи. Навіть для невеликих

ланцюгів залежностей це сильно зменшує вартість перевірок безпеки завдань. Кожного разу, коли для виконання запланована дія C# або C++, вона проходить через планувальник завдань. Роль планувальника полягає в наступному:

- Відстеження завдань за допомогою дескрипторів завдань (task handles).
- Керування залежностями завдань, забезпечуючи початок виконання завдань лише тоді, коли всі залежності завершилися.
- Керування "робочими потоками" (worker threads), які виконують завдання.
- Забезпечення якнайшвидшого виконання завдань, що зазвичай означає їх паралельний запуск, якщо залежності дозволяють.

Крім того, хоча C# Job API дозволяє планувати завдання лише з основного потоку, планувальник завдань повинен підтримувати можливість одночасного планування завдань з багатьох потоків. Це тому, що базовий рушій Unity використовує багато потоків, які планують завдання, і може навіть планувати завдання зсередини інших завдань. Ця функціональність має свої переваги та недоліки, але вимагає більше уваги до правильності та додає вимогу, що планувальник завдань повинен бути потокобезпечним.

Безпека потоків - це концепція програмування, яка відноситься до багатопоточного коду. Потокобезпечний код ефективно маніпулює спільними структурами даних так, що забезпечує, що всі потоки ведуть себе належним чином та виконують свої конструкційні вимоги без випадкової взаємодії. Існують різні стратегії для створення потокобезпечних структур даних.

Програма може виконувати код в кількох потоках одночасно в спільному просторі пам'яті, де кожен з цих потоків має доступ до практично всієї пам'яті кожного іншого потоку. Потоковий захист - це властивість, яка дозволяє коду працювати в багатопотокових середовищах шляхом відновлення деяких відповідностей між реальним потоком управління та текстом програми за допомогою синхронізації.
Типові застосування використовують таку модель: коли завдання замовляють, вони ставляться в чергу у глобальну, без блокуючу, багато продюсерну, багато

споживачу чергу, яка представляє собою набір завдань, які будуть оброблені як робочі потоки, а потім використовується основним потоком для визначення семафора, що використовується для запуску робочого потока.

Кількість робочих потоків, які повинні прокинутися, залежить від запланованої роботи: для одиночних завдань, таких як IJob, прокидається лише один робочий потік, оскільки це завдання не розподіляє роботу між кількома робочими потоками, у той час як завдання IGobParallel, навпаки, представляють завдання для можливих паралельних запусків. Навіть у межах одного проекту кілька частин може бути корисними для одного чи всіх робітників одночасно. Алгоритм без блокування (Non-blocking algorithm) — це підхід у паралельному програмуванні на симетрично-багатопроцесорних системах, який відхиляється від традиційних примітивів блокування, таких як семафори, м'ютекси та події. Розподіл доступів між потоками відбувається за допомогою атомарних операцій і спеціально розроблених механізмів блокування, адаптованих до конкретної задачі.

Перевага алгоритмів без блокування полягає в кращій масштабованості за кількістю процесорів. Крім того, якщо операційна система призупинить один із фонових потоків, інші принаймні продовжили свою роботу без простою. В найкращому випадку вони можуть взяти на себе не виконану роботу. Lock-freedom дозволяє окремим потокам фрізитись, але гарантує загальну пропускну здатність системи. Алгоритм є безблокуючим, якщо, коли програмні потоки працюють достатньо довго, принаймні один із потоків зробить прогрес (за розумним визначенням прогресу). Усі безчекові алгоритми є без блокуючими.

Зокрема, якщо один потік призупинений, то без блокуючий алгоритм гарантує, що інші потоки можуть продовжувати роботу. Таким чином, якщо два потоки можуть конкурувати за той самий м'ютекс або спілок, то алгоритм не є без блокуючим. (Якщо ми припинимо один потік, який утримує блокування, то другий потік блокується.)

Алгоритм є безблокуючим, якщо нескінченно часто операції деяких процесорів завершаться за скінченну кількість кроків. Наприклад, якщо процесори намагаються виконати операцію, деякі процеси успішно завершать операцію за скінченну кількість кроків, інші можуть зазнавати невдач і повторювати її.

Відмінність між безчековим і безблокуючим полягає в тому, що без перевірки операція кожним процесом гарантує успіх за скінченну кількість кроків, незалежно від інших процесорів.

Взагалі без блокуючий алгоритм може працювати у чотирьох фазах: завершення власної операції, допомога заважаючій операції, скасування заважаючої операції та очікування. Завершення власної операції ускладнюється можливістю конкурентної допомоги та скасування, але це завжди найшвидший шлях до завершення.Вирішення, коли допомагати, скасовувати або чекати при зустрічі завади, є відповідальністю менеджера конкуренції. Це може бути дуже просто (допомагати операціям з вищим пріоритетом, скасовувати ті, що мають нижчий пріоритет), або може бути оптимізовано для досягнення кращої пропускної здатності або зменшення латентності операцій з визначеним пріоритетом.

Правильна конкурентна допомога зазвичай є найскладнішою частиною без блокуючого алгоритму і часто коштує дорого виконати: не тільки допомагаючий потік уповільнюється, але завдяки механізмам спільної пам'яті уповільнюється і потік, який отримує допомогу, якщо він все ще працює. Перевага накладання сигналізації потоку.

Існують кілька проблем щодо можливостей у вищенаведеному підході до планування завдань, які можуть призвести до зайвого навантаження на систему задач. Розглянемо деякі приклади.

- Основний потік планує IJob (непаралельне завдання) без залежностей
- Завдання додається до черги, і сигналізується робочий потік для прокидання.
- Робочий потік прокидається.

– Робітник виконує завдання.

– Робітник перевіряє наявність ще завдань для виконання.

– Робітник іде спати, оскільки більше завдань немає.

Після того, як основний потік сигналізує, використовуючи семафор планувальника завдань, один із сплячих робочих потоків (не обов'язково робітник з номером 0) прокинеться. Прокидання та перемикання контексту займає певний час на ядрі робітника. Це тому, що, поки робітник спить, ЦП-ядро, на якому він потім буде працювати, ймовірно, вже виконувало якісь дії – можливо, обробляло інший потік, запущений грою, або якусь іншу процедуру на системі, яка використовувала цей потік.

Щоб дозволити потокам призупиняти та відновлювати в подальшому, потрібно зберігати реєстраційний стан потоку, чистити конвеєрні трубопроводи і відновлювати стан потоку, на який переключаються. Навіть сигналізація потока займає час на ядрі основного потоку, оскільки повідомлення про те, який потік прокидати, обробляється операційною системою. Зрештою, це все означає, що виконується робота на ядрі основного потоку та ядрі робочого потоку, яка не стосується нашої роботи, і, отже, це додатковий навантаження, яке ми хочемо зменшити (рис 2.3).

Рис. 2.3 Приклад виконання задач

Задача запланована на основному потоці і в кінцевому підсумку виконується на робочому потоці 0. Виконання задачі затримується витратами на сигналізацію робочому потоку 0 прокинутися на основному потоці, часом перемикання контексту на робочому потоці 0 і часом, який система завдань витрачає на пошук задачі для виконання.

Наскільки швидко в потоці можуть бути повідомлені і скільки часу займає виконання окремої задачі, також може впливати на систему. Наприклад, якщо взяти вищезазначений випадок використання, але запланувати дві задачі замість однієї:

- Завдання додається до черги, і робочий потік сигналізується для прокидання.
- Друге завдання додається до черги, і робочий потік сигналізується для прокидання.
- Робочий потік прокидається.
- Робітник виконує завдання.
- Робітник перевіряє наявність ще завдань для виконання.
- Потік іде спати, оскільки більше завдань немає.

Якщо часу вистачає, у вас є два потоки, які працюють паралельно над завданням.

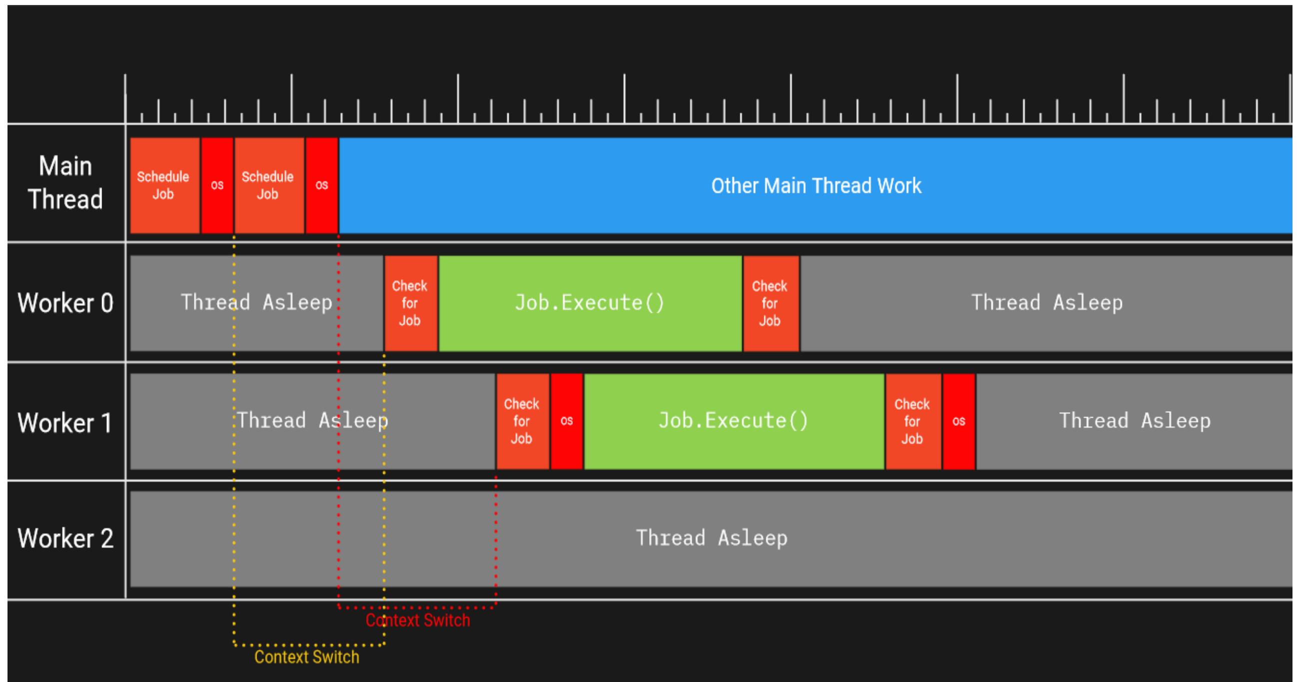

Рис. 2.4 Приклад використання Context Switch

Паралельне завдання заплановано на основному потоці і в кінцевому підсумку виконується одночасно на робочому потоці 0 і робочому потоці 1. Однак, якщо одне з завдань є занадто маленьким або занадто довго сигналізується та прокидається обидва робітники, один робітник може вкрасти всю роботу в черзі, і в результаті ми сигналізуємо робітнику без причини.

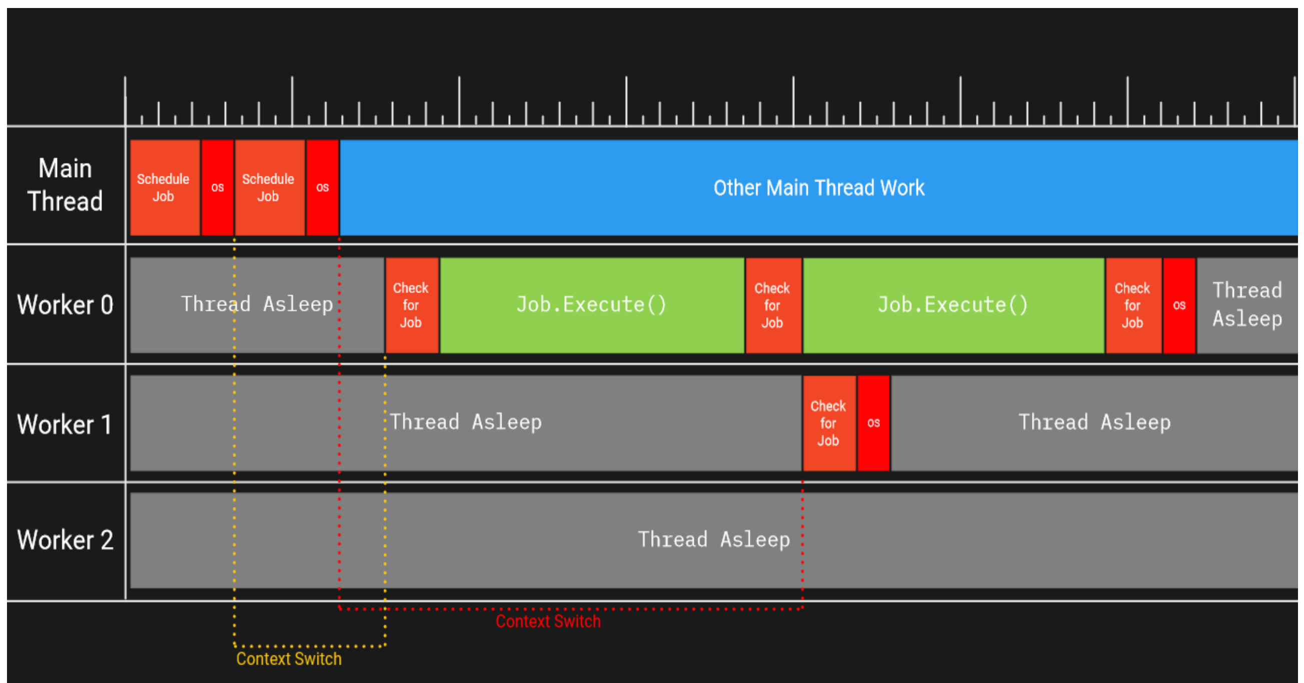

Рис. 2.5 Приклад використання Context Switch доповнення

Два завдання заплановані на основному потоці, але обидва виконуються на робочому потоці 0 через те, що Робітник 1 не прокидається до того, як Робітник 0 витрачає всі завдання у черзі. Можливо, у системі занадто багато не робочих потоків, які займають ядра процесора, або завдання занадто маленькі, щоб надати робочим потокам достатньо часу для прокидання в середньому (рис 2.5). Цей тип голодування завдань та циклу прокидання сну може виявитися досить витратним і обмежити кількість паралелізму, який пропонує система завдань.

Ви, можливо, думаєте: "Чи не є наклади від сигналізації потоків та перемикання контексту вартістю бізнесу при роботі з потоками взагалі?" Ви, праві. Але, хоча у вас немає прямого контролю над тим, наскільки витратна сигналізація чи прокидання потоків, ви можете контролювати те, як часто вони відбуваються.

Одним з рішень для уникнення прокидання робітників без причини є прокидання їх лише тоді, коли ви підозрюєте, що в черзі є багато робочих елементів, які варто витрачати витрати на прокидання. Це може бути досягнуто за допомогою пакетування: Замість сигналізації робітників, як тільки ви

заплановуєте завдання, додайте завдання до списку та, в конкретні моменти, вивільніть цей пакет завдань у систему завдань, прокидаючи відповідну кількість робітників одночасно.

Рис. 2.6 Два завдання в одному в одному пакеті

Два завдання заплановані в один пакет, і потім весь пакет вивільнюється, прокидаючи двох робітників майже одночасно. Цей метод пакетування покращує ймовірність того, що обидва робітники знайдуть роботу, коли вони прокинуться (рис 2.6).
Є все ще ризик того, що сам процес прокидання займає занадто багато часу, завдання в пакеті дуже маленькі або кількість завдань у пакеті просто не дуже велика. Загалом, чим більше завдань ви включаєте в пакет, тим більше ймовірності уникнення наклад від прокидання потоків без причини. Unity підтримує глобальний пакет, який вивільнюється кожного разу, коли викликається JobHandle.Complete(). Так що якщо вам потрібно явно чекати на завершення завдання, намагайтеся робити це якнайпізніше та якнайрідше, і загалом віддавайте перевагу плануванню завдань залежностей для найкращого контролю за безпечним доступом до даних.

Ви, можливо, також питаєте себе: "Якщо сигналізація потоків та очікування їх прокидання/призупинення - це виключно накладні витрати, чому б ми не тримаємо наші потоки постійно прокинутими, шукаючи роботу?" Коли у черзі є достатньо багато завдань, це може відбуватися природно. Якщо операційна система не вважає робочий потік менш пріоритетним, ніж якась інша робота (або якщо йому явно надано часовий відрізок і його слід перемикати, щоб дати іншим потокам чесну частку часу ЦП - це залежить від вашої платформи), робочі потоки будуть щасливо продовжувати працювати.

Однак, як із функціями PartialUpdateA та PartialUpdateB, які ми бачили в першій частині, не всі завдання можна паралелізувати і позбавити від залежностей від даних. Таким чином, зазвичай вам потрібно чекати, поки завершиться деяка підмножина завдань, перед тим як ви зможете запустити інші. У результаті ми бачимо затори в паралелізмі графа завдань, коли кількість завдань, які можна запустити (завдань без відкритих залежностей), менше, ніж робітничі потоки, що призводить до того, що деякі робітники залишаються безпродуктивними.

Якщо ви ніколи не дозволяєте робітничим потокам спати, ви можете зіткнутися з кількома проблемами. Коли робітничі потоки постійно перевіряють нові завдання і не можуть їх знайти, це вважається "активним очікуванням" або роботою, яка є марнотратною і не сприяє прогресу програми. Підтримка усіх ядер у роботі з максимальним паралелізмом, але без прогресу гри, споживає енергію акумулятора. Більше того, якщо ядро не має вільного часу, без достатнього охолодження температура ЦП підніметься, що призведе до зниження тактової частоти - сповільнення для уникнення пошкодження від перегріву. Фактично, на мобільних платформах не рідкісне явище тимчасово вимикати цілі ядра ЦП, якщо вони занадто гарячі. Для системи завдань дуже важливо ефективно використовувати ядра, тому існує баланс між тим, щоб робітники були в режимі сну, і тим, щоб вони постійно перевіряли нові завдання.

2.4. Переваги операції порівняння та обміну (CAS)

Ще однією областю, яка може генерувати надмірні витрати в конструкції вище, є безблокуюча черга та стек. В цих структур даних, але одна загальна риса без блокуючих реалізацій - використання циклу порівняння та обміну (CAS). Алгоритми без блокування не використовують синхронізаційні примітиви блокування для забезпечення безпечного доступу до спільного стану, а замість цього використовують атомарні інструкції для обережного створення атомарних операцій вищого порядку, таких як вставка елемента в чергу в потоко безпечний спосіб. Однак, можливо, не інтуїтивно, алгоритми без блокування все ще можуть заважати одному потоку продовжувати виконання, поки інший не завершиться. Вони також можуть мати вторинні ефекти на інструкції ЦП та пам'ять, що погіршує масштабованість продуктивності. ("безочікувальні" алгоритми дозволяють всім потокам завжди просуватися, але це не завжди забезпечує найкращу загальну продуктивність на практиці.)

В інформатиці операція порівняння та обміну (CAS) є атомарною інструкцією, яку використовують у багатопотокових середовищах для досягнення синхронізації. Вона порівнює вміст пам'яті з заданим значенням і, лише якщо вони однакові, змінює вміст цього місця пам'яті на нове задане значення. Це виконується як єдина атомарна операція. Атомарність гарантує, що нове значення обчислюється на основі актуальної інформації; якщо значення було оновлено іншим потоком протягом цього часу, запис буде невдалим. Результат операції повинен вказувати, чи вона виконала заміну; це може бути зроблено або звичайною булевою відповіддю (цей варіант часто називається порівняй та встанови), або повертаючи значення, прочитане з місця пам'яті (а не значення, записане в нього).
Ця операція використовується для реалізації синхронізаційних примітивів, таких як семафори і м'ютексы, а також більш вдосконалених алгоритмів без блокування та чекання. Моріс Герліхі (1991) довів, що CAS може реалізувати більше таких алгоритмів, ніж атомарне читання, запис або отримання-та-

додавання, і припускаючи достатньо велику [потрібно уточнення] кількість пам'яті, він може реалізувати всі їх. CAS еквівалентно, в тому сенсі, що постійна кількість викликів будь-якого з цих примітивів може бути використана для реалізації іншого в режимі чекання.

Алгоритми, побудовані навколо CAS, зазвичай читають певне ключове місце в пам'яті і запам'ятовують старе значення. На основі цього старого значення вони обчислюють нове значення. Потім вони намагаються замінити нове значення за допомогою CAS, де порівняння перевіряє, чи місце все ще дорівнює старому значенню. Якщо CAS вказує, що спроба не вдалася, її слід повторити знову: місце повторно читається, нове значення переобчислюється і CAS знову випробовується. Замість негайної повторної спроби після невдачі операції CAS, дослідники виявили, що загальна продуктивність системи може бути покращена в багатопроцесорних системах — де багато потоків постійно оновлюють певну спільну змінну — якщо потоки, які бачать, що їхній CAS не вдається, використовують експоненціальне затримання перед повторною спробою. CAS та інші атомарні інструкції іноді вважаються зайвими в однопроцесорних системах, оскільки атомарність будь-якої послідовності інструкцій може бути досягнута шляхом вимкнення переривань під час її виконання. Проте вимкнення переривань має численні недоліки. Наприклад, код, якому це дозволяється, повинен бути надійним і не бути зловживаним, монополізуючи ЦП, а також бути коректним і не випадково викликати зависання машини у нескінченному циклі або викликати сторінкові помилки. Крім того, вимкнення переривань часто вважається занадто витратним, щоб бути практичним. Таким чином, навіть програми, які призначені для використання тільки на багатопроцесорних машинах, користуються атомарними інструкціями, як у випадку futex у Linux.

У багатопроцесорних системах, як правило, неможливо вимкнути переривання на всіх процесорах одночасно. Навіть якщо це було б можливо, два або більше процесорів можуть намагатися отримати доступ до одного й того ж регістру семафора одночасно, і тому атомарність не буде досягнута. Інструкція

порівняння та обміну дозволяє будь-якому процесору атомарно тестувати і модифікувати область пам'яті, запобігаючи таким колізіям між багатьма процесорами.

На серверних багатопроцесорних архітектурах 2010-х років порівняння та обмін вважається дешевшим порівняно з простим завантаженням, яке не обслуговується з кешу. У дослідженні 2013 року вказується, що CAS всього лише у 1,15 рази дорожчий за не кешоване завантаження на процесорі Intel Xeon (Westmere-EX) і у 1,35 рази на AMD Opteron (Magny-Cours). Приклад додавання числа до змінної-члена, m_Sum, з використанням циклу CAS:

```
public int Add(int val)
{
    int newSum;
    do
    {
        // Завантажити поточне значення, яке ми хочемо
оновити
        var oldSum = m_Sum;
        // Обчислити нове значення, яке ми хочемо зберегти
        newSum = oldSum + val;
        // Спробувати записати нове значення.
CompareExchange повертає
        // значення, побачене всередині m_Sum при записі
newSum в m_Sum.
        // Якщо newSum не відповідає oldSum, ми повторимо
цикл,
```

```
        // оскільки це означає, що інший потік записав у
пам'ять перед нами.
        // Якщо ми записали б наше значення без цієї
перевірки, можливо,
        // ми записали б неправильне значення
    }while (oldSum != Interlocked.CompareExchange(ref
m_Sum, newSum, oldSum));
    return newSum ;
}
```

Цикли CAS залежать від інструкції порівняння і обміну (тут ми використовуємо бібліотеку C# Interlocked, абстрагуючи платформозалежні деталі), яка "порівнює два значення на рівність і, якщо вони рівні, замінює перше значення". Оскільки ми хочемо, щоб користувачі функції Add() не мали додаткових турбот щодо можливого невдачі цієї функції, використовується цикл для повторного виклику, якщо він невдалося через те, що якийсь інший потік випередив нас у оновленні m_Sum.

Цей цикл повторення, в суті, є циклом "завантаження за зайнятістю". Це має неприємний вплив на масштабування продуктивності: якщо кілька потоків входять в цикл CAS одночасно, тільки один з них буде виходити в кожний момент часу, узгоджуючи операції, які виконуються кожним потоком. На щастя, цикли CAS, як правило, роблять намірено мало роботи, але вони все ще можуть має великий негативний вплив на продуктивність. Зі збільшенням кількості ядер, які виконують цей цикл паралельно, кожному потоку буде потрібно більше часу для завершення циклу, коли потоки конкурують між собою.

Крім того, оскільки цикли CAS залежать від атомарного читання та запису в спільну пам'ять, кожен потік, як правило, потребує анулювання своїх ліній кешу на кожній ітерації, що викликає додаткові накладні витрати. Ці накладні

витрати можуть бути дуже дорогими порівняно з вартістю повторення обчислень всередині циклу CAS (у вищезазначеному випадку - повторне виконання операції додавання двох чисел). Таким чином, вартість може бути неочевидною на перший погляд.

Під керуванням планувальника завдань, коли робочі потоки не виконували завдання, вони шукали роботу у спільному, безблоковому стеку або черзі. Обидві ці структури даних використовували принаймні один цикл CAS для вилучення роботи зі структури даних. Таким чином, зі збільшенням кількості ядер вартість взяття роботи зі стеку чи черги збільшувалася при конфліктах у структурах даних. Зокрема, коли завдання були невеликими, робочі потоки відносно більше часу проводили у пошуку роботи в черзі або стеці.

У невеликому проекті я створив детерміновані графіки завдань, які може мати типова гра під час оновлення кадру. Граф нижче складається з окремих завдань та паралельних завдань (кожне паралелізується на 1–100 паралельних завдань), де кожне завдання може мати від 0 до 10 залежностей та основний потік має іноді явні точки синхронізації, де він повинен чекати, поки певні завдання завершаться, перш ніж планувати ще більше. Якщо я створюю 500 завдань у графі завдань і призначаю кожному фіксовану кількість часу для виконання (кожна частина паралельного завдання також займає цей час), можна побачити, що при використанні більше ядер накладні витрати в системі завдань зростають.

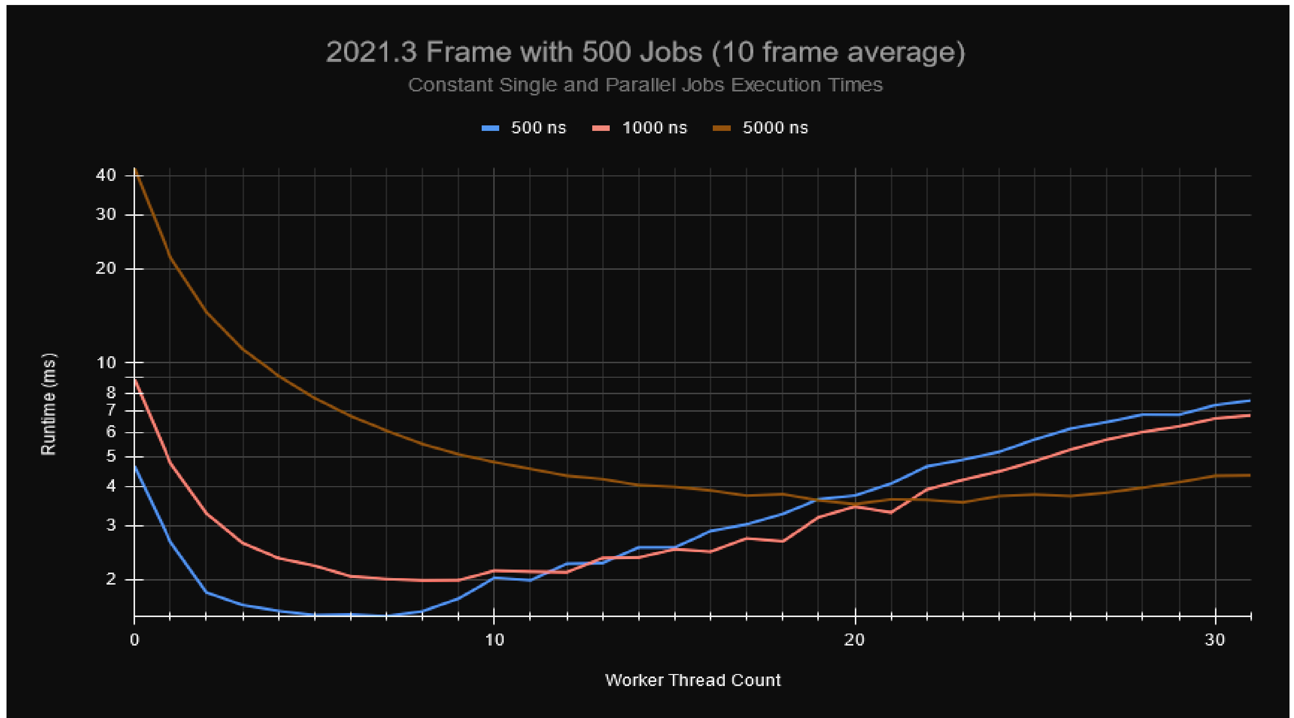

Рис. 2.8 Графік Frame with 500 Jobs 2021.3

Для задач, які займають 0.5 мкс, коли є 20 робочих потоків, оновлення кадру відбувається так само швидко, якщо взагалі не використовувати систему задач, і працює майже вдвічі повільніше при використанні всіх ядер на моєму комп'ютері (рис. 2.8). За замовчуванням в Unity використовуються всі ядра, тому для завдань тривалістю 1 мкс практично немає покращення продуктивності, навіть при використанні 31 робочого потоку. Це безпосередньо пов'язано з високою конкуренцією у безблокованій черзі та стеку. На щастя, завдання користувача зазвичай є більшими за розміром і можуть приховати це навантаження. Проте проблема з масштабуванням існує, а невеликі завдання все ще досить поширені (особливо для паралельних задач). Навіть при використанні більших завдань ваші патерни планування та синхронізації робочих потоків можуть призводити до значного збільшення навантаження через конфлікт з глобальною безблокованою чергою та стеком в планувальнику завдань.

2.5. Планувальник завдань

Так само, якщо робочий потік потребує планування завдання (наприклад, завдання планує завдання в своєму виконанні), це завдання планується у власній черзі робітника, а не в черзі основного потока. Це зменшує трафік пам'яті, оскільки робітники зменшують частоту інвалідації кеш-строк, коли вони записують у чергу. Таким чином, робітники не читають/писати до всіх різних черг з однаковою частотою.

Петля робочого потоку має такий вигляд:

```
while(!scheduler.isQuitting)
{
    Job* pJob = m_worker_queue[m_workerId].dequeue();
    if(pJob == nullptr) {
        pJob = StealFromOtherQueues()
    }
    if(pJob) {
        WakeWorkers();
        ExecuteJob(pJob);
    }
    else if(ShouldSleep())
    {
        m_semaphores[m_workerId].Wait(1);
    }
}
```

Робітники перевіряють свою власну чергу на роботу та дивляться на черги інших робітників лише у випадку, якщо їхня черга порожня. Оскільки робітники

віддають перевагу своїм чергам для вилучення та вставлення роботи, кількість конфліктів на будь-якій окремій черзі зменшується.

Ще однією відмінністю є те, як потоки сигналізуються для пробудження. Тепер робочі потоки відповідають за пробудження інших робочих потоків, і головний потік відповідає за те, щоб принаймні один робочий потік був пробуджений, коли він планує роботу.

Ця зміна відповідальності дозволяє головному потоку усунути зайвий наклад, оскільки тепер йому не потрібно виключно відповідати за пробудження потоків під час відправлення паралельних завдань. Замість цього система завдань виконує відстеження, щоб знати, чи потрібно прокидати які-небудь робітники взагалі. Головний потік може забезпечити, що робітник завжди прокинутий для продовження роботи (рис. 2.9), і коли робітники прокидаються та знаходять завдання в своїй власній черзі чи в чужій, вони можуть сигналізувати іншим робітникам прокинутися та допомогти видалити чергу, якщо це потрібно.

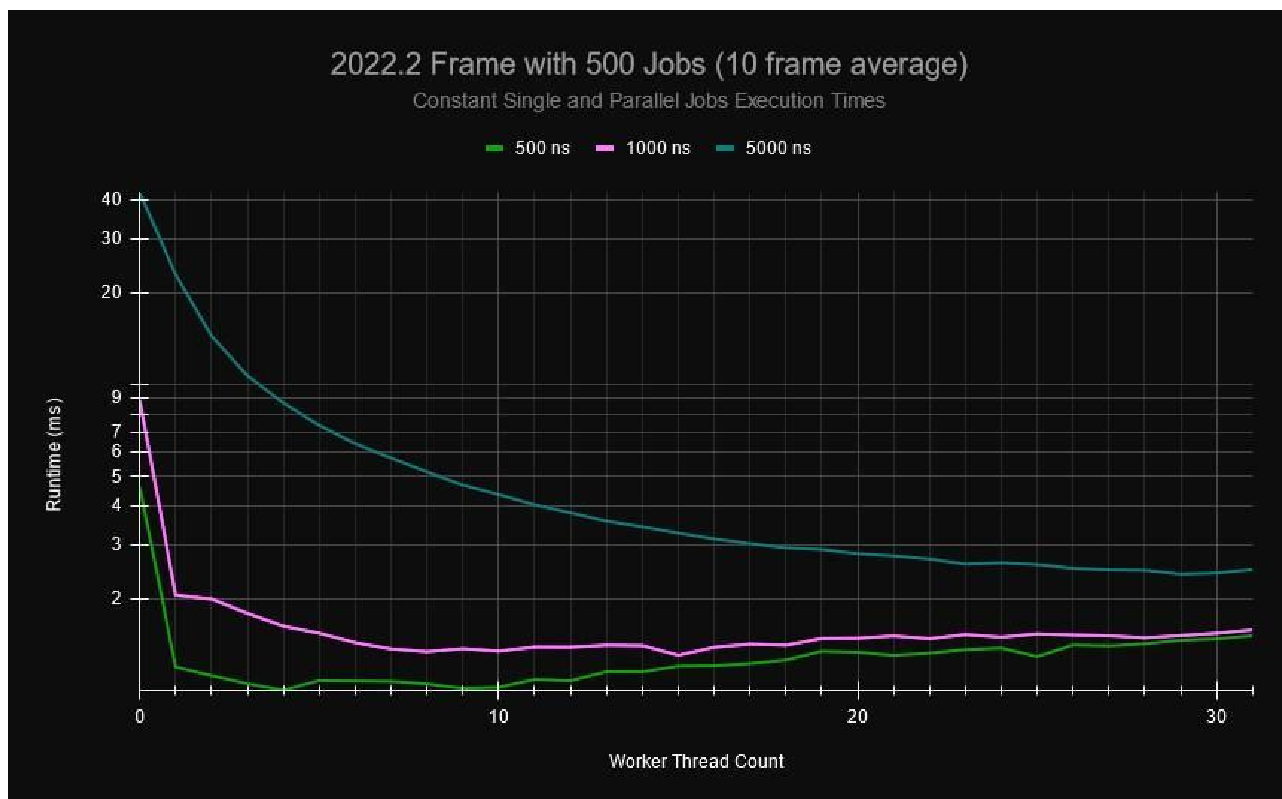

Рис. 2.9 Графік Frame with 500 Jobs 2022.3

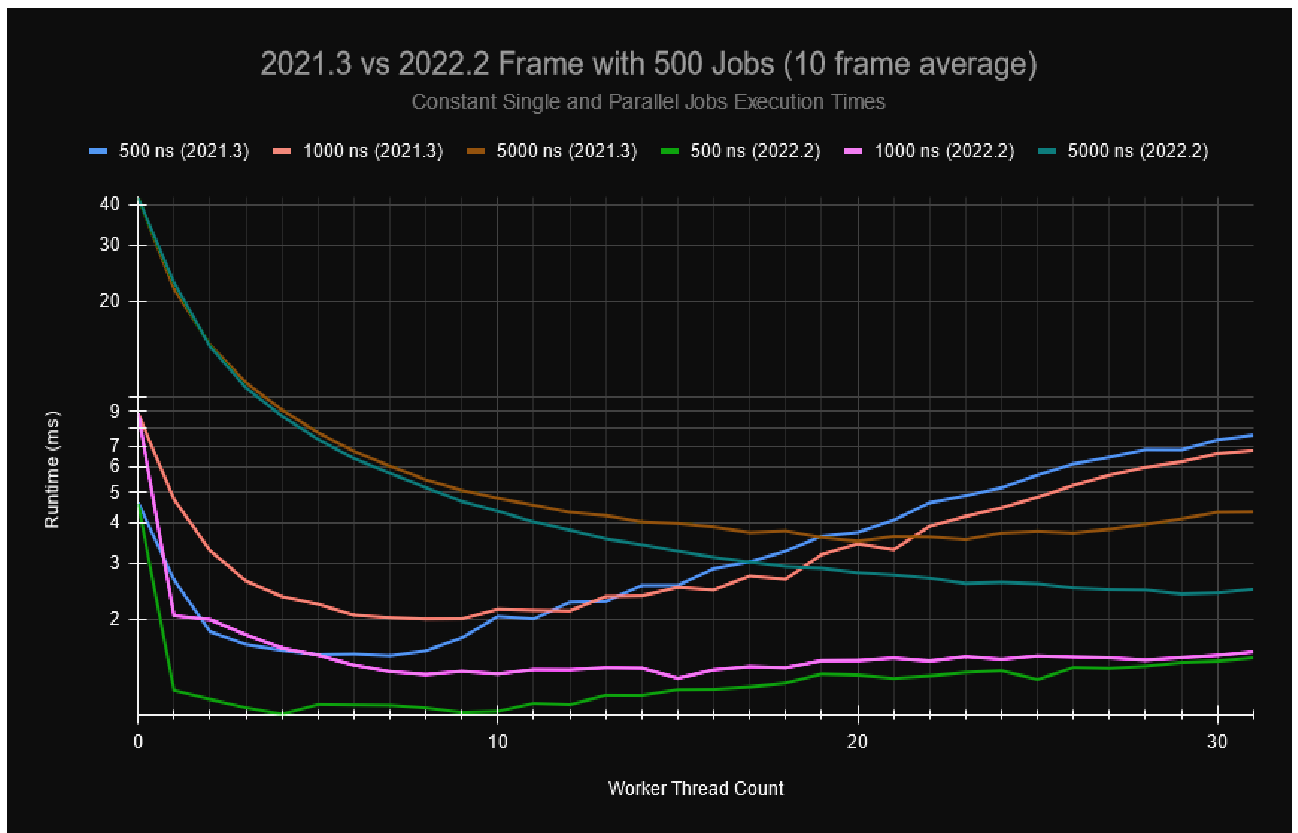

Рис. 2.10 Графік Frame with 500 Jobs 2021.3 vs 2022.2

У версії 2022.2 користувачі повинні помітити зниження витрат на головному потоці (рис 2.10) для пробудження робочих потоків та покращення продуктивності робіт на робочих потоках, незалежно від кількості ядер на їхньому платформі. Крім того, хоча Unity не внесла розділення черг до версії 2021.3 LTS, повернули зміну дизайну, щоб робочі потоки були відповідальні за сигналізацію одне одному, а не тільки головний потік. Високий наклад системи завдань на головному потоці через сигналізацію глобального семафора більше не повинен бути проблемою з версії 2021.3.14f1.

2.6. NativeArray

NativeArray<T> - це новий тип, який недавно був представлений у Unity. Це схоже на List<T>, за винятком того, що воно базується на некерованому масиві, а не на керованому. Крім того, це структура, а не клас. NativeArray надає доступ до буфера нативної пам'яті для управління кодом, що дозволяє

обмінювати даними між управліним та нативним кодом без витрат на сортування.
NativeArray - це структура даних, яка зберігає масив елементів в пам’яті. Вона є частиною Unity Job System, який дозволяє виконувати обчислення паралельно на різних потоках. NativeArray є незмінною структурою даних, тобто після створення її елементи не можуть бути змінені. Це дозволяє забезпечити безпеку під час паралельного виконання обчислень.

РОЗДІЛ 3. Використання методів та аналіз отриманих результатів

3.1 Batching

Щоб намалювати об'єкт на екрані у вашій грі, Unity повинна викликати команду "Draw" до графічного API. Ця дія суттєво називається "Draw Call" (виклик малювання). Але перед цим Unity також повинна встановити всі стани GPU, необхідні для малювання цього об'єкта: меш, шейдер, текстури, налаштування злиття та інші властивості шейдера. Команди зміни стану, плюс одна або кілька команд малювання, - це те, що ми називаємо Batch.

Те, що робить пакет повільним, - це команди зміни стану GPU, в той час як команди малювання фактично досить дешеві. Тому Unity намагається об'єднати кілька об'єктів, які малюються за допомогою одного і того ж самого стану GPU, в один пакет. Цей процес називається Batching.

У Unity існують три типи пакетування: статичне пакетування (static batching), динамічне пакетування (dynamic batching) та інстансування на GPU (GPU instancing).

Статичне пакетування об'єднує статичні меші в один або кілька великих мешів на етапі збірки, а під час виконання рендерить їх як один пакет на меш.

Динамічне пакетування бере кілька невеликих мешів кожен кадр, трансформує їх вершини на CPU, групує багато схожих вершин разом і малює їх усі в одному проході.

Інстансування на GPU малює багато ідентичних об'єктів з різними положеннями, обертаннями та іншими властивостями шейдера за менше кількість викликів малювання.

Спрайт-атлас у Unity представляє собою об'єднання графічних ресурсів, таких як зображення, текстури, чи інші графічні елементи, у одному файлі. В основі його концепції лежить ідея об'єднання невеликих фрагментів графіки в один ресурс для ефективного управління та оптимізації використання пам'яті. Основні переваги використання спрайт-атласів:

- Зменшення кількості звернень до пам'яті: Об'єднання зображень в одному файлі дозволяє зменшити кількість окремих звернень до пам'яті, що може покращити швидкість завантаження та ефективність роботи гри.
- Покращення продуктивності: Зменшення кількості файлів сприяє покращенню продуктивності гри, особливо на пристроях з обмеженими ресурсами, таких як мобільні пристрої.
- Ефективне використання текстурних ресурсів: Спрайт-атласи оптимізують використання текстурних ресурсів, дозволяючи об'єднувати багато малих текстур в одному файлі.

Unity надає зручні інструменти для створення, управління та використання спрайт-атласів. Редактор Unity дозволяє створювати атласи, встановлювати розміри та параметри кожного спрайта, а також здійснювати анімацію та інші ефекти. Інтеграція з кодом Unity також дозволяє динамічно керувати спрайт-атласами під час виконання гри.

Використання спрайт-атласів у розробці гри в Unity дозволяє досягти оптимального використання ресурсів та поліпшити продуктивність гри.

Звернімо увагу на деякі конкретні аспекти та деталі щодо типів компресії та їх використання в спрайт-атласах Unity.

Default Compression:

- Unity автоматично вибирає формат компресії в залежності від платформи. Найбільш поширеними форматами є ETC (Android), ASTC (iOS), DXT (Windows).
- Цей варіант зазвичай найбільш універсальний, оскільки Unity самостійно визначає оптимальний формат для конкретного пристрою.

High Quality Compression (Високоякісна компресія):

- Використовується для елементів, які потребують високої якості зображення, наприклад, головних персонажів або важливих об'єктів.
- Цей тип може використовувати більш потужні та менш агресивні формати компресії, такі як ASTC, що забезпечує більшу деталізацію при збереженні відмінної якості.

Low Quality Compression (Низькоякісна компресія):

- Використовується для елементів, які можуть компенсувати деяку втрату якості для зменшення розміру файлу.
- Може використовувати більш агресивні формати, такі як ETC чи DXT, що забезпечує зменшення розміру за рахунок меншої деталізації.

No Compression:

- Зберігає зображення без зжимання, у його оригінальному вигляді.
- Зазвичай використовується для тих випадків, коли максимальна якість є найважливішою, і розмір файлу не є критичним.

Alpha Split:

- Дозволяє окремо компресувати альфа-канал та кольорову інформацію.
- Використовується, коли важлива збереження високої якості альфа-каналу, щоб уникнути артефактів на гранях зображень з прозорістю.

Під час розробки важливо ретельно тестувати різні налаштування компресії на конкретних пристроях та платформах, а також враховувати бажані характеристики якості та продуктивності для кожного конкретного випадку

використання. Також слід враховувати, що формати компресії можуть відрізнятися для різних платформ, і розробникам слід адаптувати свої налаштування відповідно до потреб конкретного проекту.

Оскільки наша гра розроблена для мобільних платформ, для нас необхідні типи компресії, такі як ETC (Android) та ASTC (iOS). Unity дозволяє створювати різні версії спрайт-атласів для різних роздільностей екрану та пристроїв, що підтримує мультирезолюційність. Це важливо для забезпечення оптимальної якості графіки на різних пристроях. Спрайт-атласи є потужним інструментом для оптимізації та управління графікою в Unity, і їх використання може значно полегшити розробку гри, покращити продуктивність та зменшити використання ресурсів.

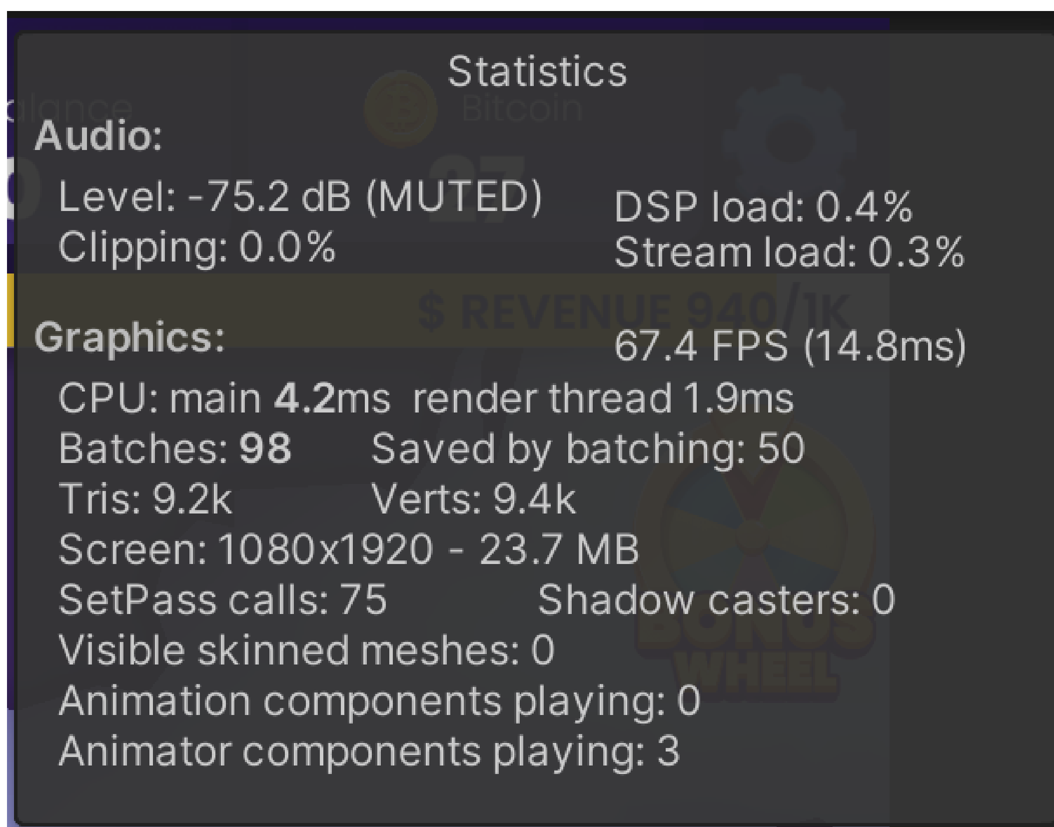

Рис. 3.1 Показники статистики гри до оптимізації батчингу

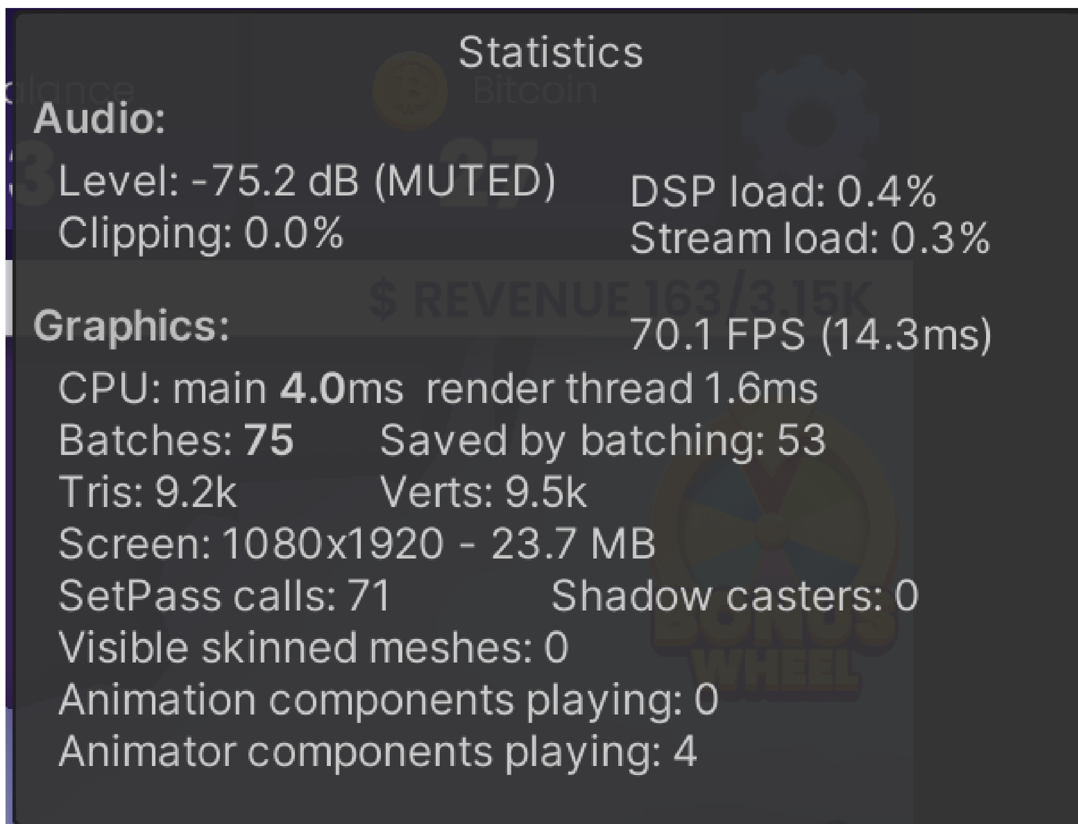

Рис. 3.2 Показники статистики гри після оптимізації батчингу

3.2 Використання DOTS

Parallel-for-transform дозволяють вам виконувати ту саму незалежну операцію для кожного положення, обертання та масштабу всіх переданих в роботу.

У випадку гри яку ми оптимізуємо мені потрібно створити структуру ось такого вигляду :

```
using UnityEngine;


public struct AirplaneMoveJobData

{

    public Vector3 TargetPosition;
```

```
    public float AirplaneSpeed;

    public bool DoNotProcessAirplaneStatus;

    public bool IsNearTargetResult;

}
```

Структура даних яка використовується в джобі переміщення літака Раніше до прийняття оптимізації кожен літак був окремим об'єктом.

Що створювала величезні проблеми для перформансу гри. Було прийняте рішення створювати окремий клас для керування авіалініями який вже мав посилання на літаки а літаки мали посилання на об'єкт в грі.

Ось приклад використаної IJobParallelForTransform :

```
using Unity.Burst;

using Unity.Collections;

using UnityEngine;

using UnityEngine.Jobs;


[BurstCompile]

public struct AirplaneMoveJob : IJobParallelForTransform

{

    public NativeArray<AirplaneMoveJobData>
AirplaneMoveJobDatas;

    public float DeltaTime;

    public float GameSpeed;

```

```
    public void Execute(int index, TransformAccess
transform)
    {
        var airplaneMoveData = AirplaneMoveJobDatas[index];
        if (airplaneMoveData.DoNotProcessAirplaneStatus)
        {
            airplaneMoveData.IsNearTargetResult = false;
            AirplaneMoveJobDatas[index] = airplaneMoveData;
            return;
        }

        var targetPosition =
airplaneMoveData.TargetPosition;
        var speed = airplaneMoveData.AirplaneSpeed;
        float step = speed * DeltaTime * GameSpeed;
        var position = transform.position;
        position = Vector3.MoveTowards(position,
targetPosition, step);
        transform.position = position;
        airplaneMoveData.IsNearTargetResult =
Vector2.Distance(position, targetPosition) <= 0.01f;
        AirplaneMoveJobDatas[index] = airplaneMoveData;
    }
}
```

Ось сам клас який робить переміщення літаків з одного міста в інше. Маршрут записується і генеруются точки при створення самої авіалінії. В цьому скрипті ми бачимо використання атрибуту BurstCompile.

Burst - це технологія компіляції наперед, яку можна використовувати для прискорення продуктивності проектів Unity, створених за допомогою нового стеку технологій, орієнтованих на дані (DOTS), та системи робіт Unity. Burst працює шляхом компіляції підмножини мови C#, відомої як High-Performance C# (HPC#), щоб ефективно використовувати потужність пристрою за допомогою впровадження вдосконалених оптимізацій на основі компіляторного каркасу LLVM.

Burst ідеально підходить для використання прихованого паралелізму в ваших додатках. Використання Burst у проекті DOTS - це просто, і це може відкрити величезні можливості для покращення продуктивності в алгоритмах, що обчислюються на центральних процесорах. У цьому відео ви можете побачити порівняння роботи в режимі скрипта в демонстраційному середовищі з активованим та вимкненим Burst.

Burst перетворює код HPC# в LLVM IR - проміжну мову, яку використовує компіляторний каркас LLVM. Це дозволяє компілятору повністю використовувати підтримку LLVM для генерації коду для архітектури Arm з метою створення ефективного машинного коду, оптимізованого навколо потоку даних вашої програми (рис. 3.3).

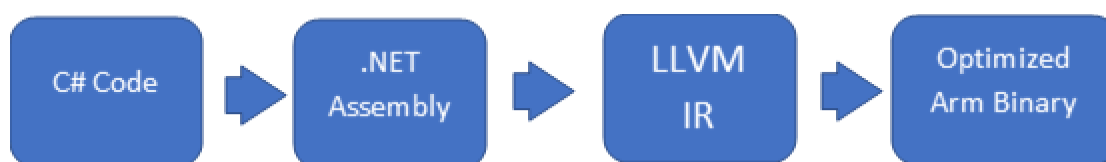

Рис. 3.3 Burst перетворює код HPC# в LLVM IR

Майк Ектон виступав з лекцією під назвою "Орієнтований на дані дизайн і C++", в якій висловлено ключовий принцип: "знай свій апарат, знай свої дані" як засіб

досягнення максимальної продуктивності. Burst працює добре, оскільки він надає видимість обмеженням на псевдонімізацію масивів, які гарантуються мовою HPC# та фреймворком DOTS, і може використовувати знання LLVM про архітектуру вашого апарату. Це дозволяє Burst робити трансформації, орієнтовані на конкретну мету, на основі властивостей скриптів, написаних на основі API Unity.

Entity Component System (ECS) в Unity - це інноваційний підхід до розробки ігор, спрямований на підвищення продуктивності та оптимізацію використання ресурсів.

ECS визначає структуру гри через три ключові концепції - сутності, компоненти і системи. Ентіті представляють об'єкти в грі. Кожна сутність є контейнером для компонентів і не має жодної логіки. Наприклад, якщо у вас є персонаж у грі, його можна представити як ентіті. Компоненти визначають дані та властивості сутностей. Кожен компонент відповідає за конкретний аспект об'єкта в грі. Наприклад, компонент позиції, компонент візуального вигляду, компонент фізики тощо.

Системи обробляють і маніпулюють групами сутностей, які мають конкретний набір компонентів. Вони виконують операції над цими групами для забезпечення ефективної обробки даних.

Burst Compiler - це ключовий компонент DOTS, який підтримує оптимізацію виконання коду ECS. Burst Compiler використовується для генерації виконавчого коду, який оптимізований для конкретної архітектури процесора. Це дозволяє вам використовувати мову C# для написання високопродуктивного коду. Також, використовує техніки низькорівневої оптимізації, такі як векторизація і інлайнінг, для покращення швидкодії коду.

Job System дозволяє виконувати обчислення паралельно на різних ядрах процесора. Замість використання традиційних потоків, Job System розподіляє завдання між доступними ядрами процесора, що дозволяє вам використовувати повністю потенціал вашого обладнання. Job System вбудований в ECS і

допомагає уникнути конфліктів даних, що може виникнути при паралельному виконанні коду.

Загальною метою DOTS (Data-Oriented Technology Stack) в Unity є забезпечення високої продуктивності та ефективності в розробці ігор. DOTS визначається не лише ECS, Burst Compiler та Job System, але й іншими технологіями, які допомагають оптимізувати роботу з даними та забезпечують більш паралельну та швидку обробку ігрової логіки. DOTS пропонує різні моделі розміщення пам'яті, такі як Entity Data, Component Data Array, і Buffer Elements, які спрощують і прискорюють доступ до даних, зменшуючи фрагментацію пам'яті та покращуючи кешування. DOTS також охоплює Hybrid Rendering, що дозволяє поєднувати переваги нових технологій з традиційним рендерингом. Це може полегшити міграцію від старих систем до нових, забезпечуючи гнучкість та швидкість рендерингу. Включає оптимізовані системи фізики та штучного інтелекту, що дозволяють обробляти складні фізичні взаємодії та приймати ефективні рішення в грі. Він створений для того, щоб відповідати потребам розробників у створенні великих ігор з високим рівнем деталізації та взаємодії. Використання цього технологічного стеку може покращити продуктивність, зменшити витрати ресурсів та зробити розробку більш масштабованою та швидкою.

DOTS, як цілісна система, спрямована на оптимізацію роботи з даними в ігровій розробці, і використання цих технологій може допомогти покращити продуктивність ігор та ефективність використання ресурсів.

3.3 Використання Pool

Пул об'єктів - це відмінний спосіб оптимізувати ваші проекти та зменшити навантаження, яке розміщується на ЦП при необхідності швидко створювати та знищувати ігрові об'єкти. Це є хорошою практикою та шаблоном дизайну, які варто мати на увазі, щоб допомогти звільнити обчислювальні ресурси процесора для обробки більш важливих завдань і уникнути перевантаження від

повторюваних викликів створення та знищення. В проекті “Crypto Idle Transport Tycoon” тому було прийнято рішення використовувати UnityEngine.Pool.ListPool. У випадках, коли ви стикаєтеся з потребою часто створювати та видаляти ігрові об'єкти, такі як кулі зброї, ефективне використання пулів може суттєво покращити продуктивність вашої гри. Пули можуть бути передвідомою одиницею ігрового об'єкта, і використання пулу для їх створення та повторного використання може виявитися значно більш ефективним за традиційний підхід. Зокрема, якщо у вас є сцена із великою кількістю взаємодіючих об'єктів, таких як масові вибухи чи стріляння, пули можуть стати важливим елементом оптимізації. Використання пулів дозволяє уникнути частого виділення та звільнення пам'яті, зменшуючи таким чином переваги, пов'язані зі стандартними операціями створення та знищення об'єктів. Також, враховуючи, що велика кількість алокацій може призводити до фрагментації пам'яті, що ускладнює розташування вільних блоків пам'яті, використання пулів може допомогти управляти цією проблемою та підтримувати стабільну ефективність гри. Такий підхід може бути особливо корисним у великих проектах, де оптимізація роботи з пам'яттю грає важливу роль.Використання ListPool в Unity дозволяє ефективно управляти пам'яттю та оптимізувати роботу зі списками в геймдев-проектах. Давайте розглянемо, чому це може бути корисно та як воно сприяє поліпшенню продуктивності. Однією з основних переваг використання ListPool є уникнення постійного створення та видалення списків у пам'яті. Коли ви викликаєте ListPool<T>.Get(), ви отримуєте екземпляр списку, який може бути вже створений та готовий до використання. Це дозволяє уникнути затрат, пов'язаних із створенням нових об'єктів у пам'яті, та спрощує управління ресурсами.

Коли ви завершуєте роботу зі списком, ви викликаєте ListPool<T>.Release(myList), і цей список повертається назад у пул. Таким чином, ми уникдаємо зайвого утримання пам'яті, оскільки пул доглядає за чисткою та підготовкою списків для подальшого використання.

Використання ListPool особливо корисне в сценаріях, де маємо справу із великою кількістю списків, наприклад, при роботі з подіями, коллекціями гри або системами штучного інтелекту. Це дозволяє гнучко та ефективно керувати пам'яттю та ресурсами, зменшуючи навантаження на систему управління пам'яттю та забезпечуючи більш плавну роботу гри.

Загалом, ListPool - це інструмент, який допомагає розробникам Unity підвищити ефективність своїх проектів, поліпшити використання пам'яті та забезпечити оптимізований геймплей.

Використання пулінгу об'єктів в геймдеві може сприяти більш ефективному управлінню ресурсами та оптимізації використання пам'яті. Однак, разом із перевагами, виникають і деякі обмеження та питання, які важливо враховувати. Важливо відзначити, що, додавши логіку пулінгу об'єктів, може збільшитися складність коду, особливо у великих проектах. Потрібно вивчати та враховувати взаємодію з об'єктами у пулі, щоб правильно керувати їхнім життєвим циклом та уникнути непередбачених ситуацій. Крім того, використання пулінгу може вимагати додаткового обсягу коду для кожного класу чи об'єкта, який потрібно пуліти, що може призвести до зростання складності коду. Також важливо належно налаштовувати розмір пулу, щоб уникнути зайвого використання пам'яті чи недостатку об'єктів у пулі. Застосовуючи пулінг об'єктів в геймдеві, слід також враховувати, що пулінг найбільше ефективний у випадках, де існує велика кількість створення та знищення об'єктів. У деяких сценаріях, де об'єкти зберігаються тривалий час чи їх кількість невелика, пулінг може виявитися зайвим.

Використання пулів об'єктів вносить свої проблеми:

- Забуті повернення об'єктів у пул може спровокувати надмірне навантаження сбірника сміття.
- Невиконане скидання стану може призвести до збереження старих даних у екземплярах, отриманих із пула, роблячи їхній стан неактуальним.
- Повернення об'єкта в пул слід виконувати лише після завершення всіх його використань для уникнення конфліктів та непередбачених ситуацій.

- Важливо усвідомлювати, що управління пам'яттю для колекцій може бути складним, оскільки припущення щодо їхніх розмірів впливає на ефективність використання пам'яті.
- Пули не гарантують потокобезпеки за замовчуванням, і введення механізмів потокобезпеки може призвести до додаткових витрат процесора.

Незважаючи на ці обмеження, використання пулінгу об'єктів залишається важливим інструментом для оптимізації продуктивності в геймдеві. Правильно враховуючи переваги та обмеження, розробники можуть забезпечити більш ефективне та стабільне використання пам'яті у своїх проектах.

3.4 Використання Task.Factory

В геймдеві на платформі Unity, ефективне використання асинхронної роботи є ключовим елементом для забезпечення плавності гри та відмінного користувацького досвіду. Однією з потужних конструкцій для цього є Task.Factory з бібліотеки Task Parallel Library (TPL), яка надає зручний спосіб працювати з асинхронним кодом та багатозадачністю. Використання задач було застосовано під час генерації пасажирів.

Ось приклад використання Task.Factory:

```
private async void TryAddPassengers(float deltaTime)
{
   _currentTime += deltaTime;

   if (_currentTime > _globalGameConfig.PassengerAddTime / GameSpeed)
   {
```

```
        if (_addPassengersInProgress)
        {
            Debug.LogWarning("TryAddPassengers return");
            return;
        }

        _addPassengersInProgress = true;

        _currentTime = 0;

        var task = Task.Factory.StartNew(AddPassenger);

        await task;
        if (CurrentOpenedAirport != null)
            CurrentOpenedAirport.UpdateAllPassengers();

        _addPassengersInProgress = false;
        task.Dispose();
    }
}

private void AddPassenger()
{
    for (int i = 0; i < AirportControllers.Count; i++)
```

```
    {
        AirportControllers[i].CalculateAddPassenger();
    }
}
```

Метод Task.Factory.StartNew з бібліотеки Task Parallel Library (TPL) в C# використовується для створення та запуску асинхронних задач. Цей метод дозволяє виконувати обчислення чи операції паралельно в окремому потоці виконання. При використанні StartNew:

- Створюється новий об'єкт Task, який представляє асинхронну операцію чи обчислення.
- Виконується код, переданий в делегаті методу, а цей код виконується в окремому потоці чи робочій одиниці.
- Можна передавати різні параметри для використання у ваших обчисленнях.
- Є можливість налаштовувати різні параметри для керування потоком виконання, такі як вибір потоку виконання чи планувальник задач.
- Можна визначити продовження для обробки результату або виконання дій після завершення асинхронної операції.

Цей підхід полегшує створення та виконання асинхронних завдань в C#, що є важливим для оптимізації продуктивності та ефективного використання ресурсів.

Метод Dispose() в C# використовується для вивільнення ресурсів, зайнятих об'єктом. Проте, в контексті Task, необхідність в явному вивільненні ресурсів рідко виникає, оскільки Task автоматично виконує вивільнення ресурсів після завершення своєї роботи.

Зазвичай, ви не потребуєте вручну викликати Dispose() для об'єктів типу Task. Замість цього, ви можете скористатися іншими конструкціями, такими як конструкція using для ресурсів, які підтримують інтерфейс IDisposable.

Зауважте, що вивільнення ресурсів та керування життєвим циклом об'єктів може бути складнішим, коли мова йде про використання асинхронних операцій і Task у контексті геймдеву. Зазвичай, система управління пам'яттю .NET сама керує вивільненням ресурсів для завершених задач, і ви не повинні вручну викликати Dispose() для об'єктів типу Task. При дослідженні було виявлено, що в деяких випадках відсутність виклику методу Dispose() може призводити до витоків пам'яті. Особливо це може статися, коли ви маєте справу з класами, які володіють зовнішніми ресурсами, і ви не викликаєте Dispose() явно. Застосування Dispose() стає важливим кроком для ефективного вивільнення ресурсів та попередження витоків пам'яті у ваших програмах. Краще викликати Dispose() для вивільнення ресурсів, особливо якщо ви впевнені, що ресурси не будуть автоматично вивільнені при виході з області видимості або завершенні роботи з об'єктом.

Використання Task.Factory.StartNew в Unity може призвести до проблем, таких як неправильний доступ до UI-елементів та інших об'єктів, пов'язаних з головним потоком, оскільки виконання коду не завжди відбувається на головному потоці Unity (рис 3.4). Працювати з Task.Factory.StartNew також може бути непрактичним, оскільки монобехи, які є ключовим елементом Unity, повинні взаємодіяти тільки на головному потоці. Крім того, використання Task.Factory.StartNew ускладнює управління потоками та може викликати неявні витрати ресурсів. Замість цього, Unity надає свої власні засоби для асинхронної роботи, такі як корутини та асинхронні методи, які інтегруються краще з цим середовищем.

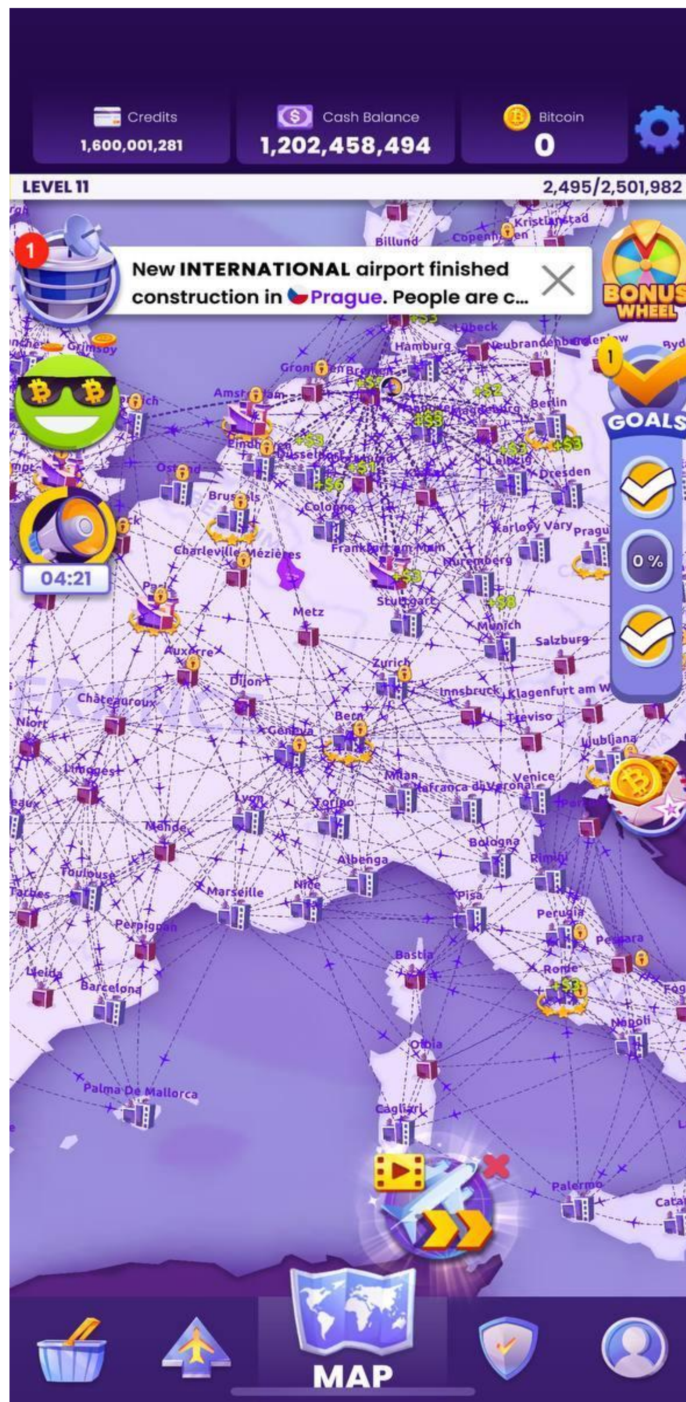

Рис. 3.4 Графічні дизайн гри

ВИСНОВКИ

В результаті виконання кваліфікаційної роботи було проведено Дослідженням ефективності впровадження методів оптимізації гри «Crypto Idle Transport Tycoon». Графічні представлення результатів дозволяють візуалізувати покращення та показати, як використання вказаних методів призвело до оптимізації гри.

Для розв'язання поставлених задач використані: DOTS, ECS, Job System, Task, Pool, Burst Compiler, Batching, NativeArray.

Отримані результати не лише підтверджують ефективність використання вказаних технологій у геймдеві, але і вказують на можливості їх застосування в оптимізації складних проектів загалом. Враховуючи широкий спектр технік, використаних у роботі, можна зазначити, що їх комбінація виявляється особливо ефективною в управлінні ресурсами та оптимізації швидкодії програм.

Отже, дані отримані в результаті кваліфікаційної роботи можуть бути використані як підґрунтя для розробки оптимізаційних стратегій у сфері геймдеву та інших високонавантажених програмних проектах. Подальші дослідження в цій області можуть спрямовуватися на розширення набору використовуваних технік, а також на адаптацію результатів для конкретних умов застосування, що дозволить розробникам вибирати оптимальні стратегії оптимізації залежно від конкретних вимог їхніх проектів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Unity Game Development Cookbook: Essentials for Every Game 1st Edition Автори: Джон Меннінг Тім, Ньюджент Периіс, Батфілд Еддісон
2. Unity in Action: Multiplatform Game Development in C# with Unity 5 Автори: Джо Хокінг Джессі Шелл
3. Introduction to Game Design, Prototyping, and Development Автор: Джеремі Гібсон Бонд
4. Nice Touch documenation https://nice-touch-docs.moremountains.com/API/
5. Nice Vibration documentation https://nice-vibrations-docs.moremountains.com/#does-it-work-everywhere-
6. Multiplayer and Networking documentation documentation https://docs.unity3d.com/Manual/UNet.html
7. Atributs in Unity documentation https://docs.unity3d.com/ScriptReference/TooltipAttribute.html
8. Microsft quickstart in Unity documentation https://docs.microsoft.com/en-us/visualstudio/gamedev/unity/get-started/getting-started-with-visual-studio-tools-for-unity?view=vs-2022&pivots=windows
9. C# in Unity documentation https://docs.microsoft.com/en-us/archive/msdn-magazine/2014/august/unity-developing-your-first-game-with-unity-and-csharp
10. Microsft quickstart in Photon documentation https://docs.microsoft.com/en-us/gaming/playfab/sdks/photon/quickstart
11. Photon introduction documentation https://doc.photonengine.com/en-us/pun/current/getting-started/pun-intro
12. Photon Features overview documentation https://doc.photonengine.com/en-us/pun/current/getting-started/feature-overview
13. Photon setup and connect documentation https://doc.photonengine.com/en-us/pun/current/getting-started/initial-setup
14. Photon C# callbacks documentation https://doc.photonengine.com/en-us/pun/current/getting-started/dotnet-callbacks
15. Photon migration node documentation https://doc.photonengine.com/en-us/pun/current/getting-started/migration-notes
16. Photon app and lobby stats documentation https://doc.photonengine.com/en-us/pun/current/lobby-and-matchmaking/appandlobbystats
17. Photon regions documentation https://doc.photonengine.com/en-us/pun/current/connection-and-authentication/regions
18. Quickstart to TextMesh Pro https://learn.unity.com/tutorial/working-with-textmesh-pro#:~:text=TextMesh%20Pro%20is%20an%20easy,to%20any%20project%27s%20user%20interface.

19. TextMesh Pro settings https://docs.lunalabs.io/docs/playable/code/unity-plugins/tmp
20. Unity Editor guide https://learn.unity.com/tutorial/get-started-in-the-unity-editor
21. Create user UI in Unity documentation https://docs.unity3d.com/Manual/UIToolkits.html
22. Canvas in Unity documentation https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/UICanvas.html
23. Rect Transform in Unity documentation https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/class-RectTransform.html
24. Animation in Unity documentation https://docs.unity3d.com/Manual/AnimationSection.html
25. Unity quick search https://docs.unity3d.com/Packages/com.unity.quicksearch@1.0/manual/index.html
26. Tag component https://gametorrahod.com/tag-component/
27. Game object coversioin https://gametorrahod.com/gameobject-conversion-ecosystem-code-tour/
28. TextMesh Pro anatomy https://gametorrahod.com/textmeshpro-anatomy/
29. Unity Event serialization https://gametorrahod.com/unityevent-serialization-research/
30. Quality in Unity documentation https://docs.unity3d.com/Manual/class-QualitySettings.html
31. Integrated development environment (IDE) support https://docs.unity3d.com/2020.2/Documentation/Manual/ScriptingToolsIDEs.html
32. Draw call batching documentation https://docs.unity3d.com/2019.4/Documentation/Manual/DrawCallBatching.html
33. DOU https://dou.ua/?utm_source=google&utm_medium=cpc&utm_campaign=search-brand&gclid=CjwKCAjwqauVBhBGEiwAXOepkT0MwxHh446-SO3crkQwq9LdD0FqNhAFx4IzbssoJV-a_ldFX1ThRBoCJyYQAvD_BwE
34. Metanit https://metanit.com/sharp/tutorial/3.7.php

ДОДАТОК А

КОД ПРОГРАМИ

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Eu.Cryptotycoon.State;
using Google.Protobuf.Collections;
using PopupManager;
using UnityEngine;
using UnityEngine.Pool;
using Random = UnityEngine.Random;

public class AirPlane : IDisposable
{
    public
RepeatedField<Eu.Cryptotycoon.State.PassengerData>
Passengers = new
RepeatedField<Eu.Cryptotycoon.State.PassengerData>();
    public AirportController CurrentDestination;
    public AirportController PreviousDestination;
    public AirPlaneLevel AirPlaneLevel;
    public float TotalPassengersInPlane;
    public string PassengersGuid;
    public bool IsFlight;
    public AirLine AirLine;
    public int AvailableProfit;
    public int PathFlyIndex;
    public bool IsBusy;
    public GameObject GameObject;
    public Transform MyTransform;
    public MeshRenderer PlaneMeshRenderer;
```

```
    public AirplaneType AirplaneType;

    private MeshFilter PlaneMesh;
    private AirplaneCollider PlaneCollider;
    private AirPlaneLevelConfig _airPlaneLevelConfig;
    private GlobalGameConfig _globalGameConfig;
    private RemoteConfigController _remoteConfigController;
    private GameController _gameController;
    private EventsController _eventsController;
    private GoalSystem _goalSystem;
    private SoundManager _soundManager;
    private Action _onWaitFinish;
    private int direction = 1;
    private IPopupManager _popupManager;
    private TutorialController _tutorialController;
    private AirlinesState _airlinesState;
    private PassengersState _passengersState;
    private AirlineInfo _airlineInfo;

    public void
AddPassengersToFlight(RepeatedField<Eu.Cryptotycoon.State.P
assengerData> passengers, AirportController destination,
float totalPassengers)
    {
        float tempProfit = AirLine.AirlineDistance *
totalPassengers *
_remoteConfigController.BalanceConfig.PassengerIncome /
1000;
        AvailableProfit = Mathf.CeilToInt(tempProfit);
        TotalPassengersInPlane = totalPassengers;
        CurrentDestination = destination;
        Passengers.Clear();
```

```
        Passengers =
_passengersState.GetPassengerById(PassengersGuid);

        Passengers.AddRange(passengers);

        IsFlight = true;

    }


    private Quaternion LookRotation(Vector3 forward, Vector3
upDirection)

    {

        return QuaternionExtensions.LookRotation(forward,
upDirection);

    }


    private void OperationCost()

    {

        int cost = Mathf.RoundToInt(-AirLine.AirlineDistance
* AirPlaneLevel.PlaneOperationCost);

        _gameController.AddCoins(cost);

    }


    public void ChangeViewVisibility(bool isVisible)

    {

        if (PlaneMeshRenderer.enabled == isVisible)

            return;


        PlaneMeshRenderer.enabled = isVisible;

        PlaneCollider.ChangeAvailability(isVisible);

    }


    public void InitPlane(AirPlaneLevel airPlaneLevel,
AirLine airLine, AirPlaneLevelConfig airPlaneLevelConfig,
```

```
        GlobalGameConfig globalGameConfig,
RemoteConfigController remoteConfigController,
GameController gameController,

        EventsController eventsController, GoalSystem
goalSystem, SoundManager soundManager, GameObject
gameObject,

        IPopupManager popupManager, TutorialController
tutorialController, AirlinesState airlinesState,
PassengersState passengersState, AirplaneType airplaneType
= AirplaneType.Default,

        AirportController sourceAirportController = null)

    {

        _passengersState = passengersState;

        _airlinesState = airlinesState;

        AirLine = airLine;

        _tutorialController = tutorialController;

        _popupManager = popupManager;

        GameObject = gameObject;

        _soundManager = soundManager;

        _goalSystem = goalSystem;

        _eventsController = eventsController;

        _gameController = gameController;

        _remoteConfigController = remoteConfigController;

        AirplaneType = airplaneType;

        PreviousDestination = airLine.FirstAirport;

        CurrentDestination = airLine.SecondAirport;

        _globalGameConfig = globalGameConfig;

        MyTransform = gameObject.transform;

        _airPlaneLevelConfig = airPlaneLevelConfig;

        AirPlaneLevel = airPlaneLevel.Clone();

        AirPlaneLevel.PlaneOperationCost =
_remoteConfigController.BalanceConfig.GetPlaneOperationCost
ByLevel(AirPlaneLevel.Level);
```

```
        PathFlyIndex = 0;
        PassengersGuid = Guid.NewGuid().ToString();

        if (sourceAirportController != null)
        {
            if (airLine.FirstAirport ==
sourceAirportController)
            {
                PreviousDestination = airLine.FirstAirport;
                CurrentDestination = airLine.SecondAirport;
                PathFlyIndex = 0;
            }
            else if (airLine.SecondAirport ==
sourceAirportController)
            {
                PreviousDestination = airLine.SecondAirport;
                CurrentDestination = airLine.FirstAirport;
                PathFlyIndex = AirLine.Path.Count - 1;
            }
        }


_soundManager.PlaySingleSound(SoundType.NewPlaneOnRoute);
        MyTransform.position = AirLine.Path[PathFlyIndex];
        MyTransform.LookAt(AirLine.Path[PathFlyIndex]);
        PlaneMesh =
MyTransform.GetComponentInChildren<MeshFilter>();
        PlaneMesh.sharedMesh =
_airPlaneLevelConfig.GetAirPlaneMeshDataByLevel(AirPlaneLev
el.Level, AirplaneType).Mesh;
```

```
        PlaneMeshRenderer =
MyTransform.GetComponentInChildren<MeshRenderer>();
        PlaneCollider =
MyTransform.GetComponentInChildren<AirplaneCollider>();
        PlaneCollider.OnPlaneClick += HandlePlaneClick;
        MyTransform.rotation =
LookRotation(AirLine.Path[PathFlyIndex],
MyTransform.forward);


        GameObject.SetActive(true);


        if (AirLine.SecondAirport.IsEventEnable ||
AirLine.FirstAirport.IsEventEnable)
        {
            DisableFly(() =>
            {

RepeatedField<Eu.Cryptotycoon.State.PassengerData>
passengers =

CurrentDestination.GetPassengersToConnectedAirport(Previous
Destination.Airport, this, out float totalPassengers);


                AddPassengersToFlight(passengers,
CurrentDestination, totalPassengers);
            });
        }
    }


    public void LvlUp(AirPlaneLevel nextLevel)
    {

_soundManager.PlaySingleSound(SoundType.AirplaneUpgrated);
```

```
        AirPlaneLevel = nextLevel.Clone();
        PlaneMesh.sharedMesh =
_airPlaneLevelConfig.GetAirPlaneMeshDataByLevel(AirPlaneLev
el.Level).Mesh;
        AirPlaneLevel.PlaneOperationCost =
_remoteConfigController.BalanceConfig.GetPlaneOperationCost
ByLevel(AirPlaneLevel.Level);
        AirLine.TotalSpendMoney += AirPlaneLevel.Price;

_goalSystem.UpdateGoalExecution(Eu.Cryptotycoon.State.GoalT
ype.UpgradeXairplanesToYlevel, AirPlaneLevel.Level);
        _airlinesState.UpdatePlanesOnAirLine(AirLine);
    }


    public void InitPlaneBySave(AirLine airLine,
AirPlaneLevelConfig airPlaneLevelConfig, GlobalGameConfig
globalGameConfig,
        PlaneDate planeData, List<AirportController>
airportControllers, RemoteConfigController
remoteConfigController,
        GameController gameController, EventsController
eventsController, GoalSystem goalSystem, SoundManager
soundManager,
        GameObject gameObject, IPopupManager popupManager,
TutorialController tutorialController, AirlinesState
airlinesState, PassengersState passengersState)
    {
        _passengersState = passengersState;
        _airlinesState = airlinesState;
        _tutorialController = tutorialController;
        _popupManager = popupManager;
        GameObject = gameObject;
        _soundManager = soundManager;
        _goalSystem = goalSystem;
```

```
        _eventsController = eventsController;
        _gameController = gameController;
        _remoteConfigController = remoteConfigController;
        AirplaneType =
ConverterExtensions.ConvertAirPlaneType(planeData.AirplaneT
ype);


        PassengersGuid = planeData.PassengersGuid;


        CurrentDestination =
airLine.FirstAirport.Airport.AirportName ==
planeData.CurrentDestination
            ? airLine.FirstAirport
            : airLine.SecondAirport;


        PreviousDestination =
airLine.FirstAirport.Airport.AirportName ==
planeData.PreviousDestination
            ? airLine.FirstAirport
            : airLine.SecondAirport;


        Passengers =
passengersState.GetPassengerById(PassengersGuid);


        _globalGameConfig = globalGameConfig;
        MyTransform = gameObject.transform;
        _airPlaneLevelConfig = airPlaneLevelConfig;
        AirLine = airLine;
        AirPlaneLevel =
airPlaneLevelConfig.GetAirPlaneLevelByOrder(planeData.Plane
Level,
ConverterExtensions.ConvertAirPlaneOrder(planeData.AirPlane
Order)).Clone();
```

```
        AirPlaneLevel.PlaneOperationCost =
_remoteConfigController.BalanceConfig.GetPlaneOperationCost
ByLevel(AirPlaneLevel.Level);

        PlaneMesh =
GameObject.GetComponentInChildren<MeshFilter>();

        PlaneMesh.sharedMesh =
_airPlaneLevelConfig.GetAirPlaneMeshDataByLevel(AirPlaneLev
el.Level).Mesh;

        PlaneMeshRenderer =
MyTransform.GetComponentInChildren<MeshRenderer>();

        PlaneCollider =
MyTransform.GetComponentInChildren<AirplaneCollider>();

        PlaneCollider.OnPlaneClick += HandlePlaneClick;

        PathFlyIndex = Mathf.Clamp(planeData.PathFlyIndex -
1, 0, AirLine.Path.Count - 2);

        MyTransform.position = AirLine.Path[PathFlyIndex];

        MyTransform.rotation =
LookRotation(AirLine.Path[PathFlyIndex] - Vector3.zero,
MyTransform.forward);

        TotalPassengersInPlane = planeData.TotalPassengers;

        AvailableProfit = planeData.AvailableProfit;


        IsFlight = true;


        GameObject.SetActive(true);


        if (AirLine.SecondAirport.IsEventEnable ||
AirLine.FirstAirport.IsEventEnable)

        {

            DisableFly(() =>

            {


RepeatedField<Eu.Cryptotycoon.State.PassengerData>
passengers =
```

```
PreviousDestination.GetPassengersToConnectedAirport(PreviousDestination.Airport, this, out float totalPassengers);

                AddPassengersToFlight(passengers,
CurrentDestination, totalPassengers);
            });
        }
    }

    public void TickUpdate()
    {
        if (CurrentDestination == AirLine.SecondAirport)
        {
            PathFlyIndex++;

            if (PathFlyIndex == AirLine.Path.Count)
            {
                IsFlight = false;
                PathFlyIndex = AirLine.Path.Count - 1;
                (PreviousDestination, CurrentDestination) =
(CurrentDestination, PreviousDestination);
                OperationCost();
                AirLine.FinishFlight(PreviousDestination,
this);
                OnFlightFinish?.Invoke();
            }
            if(_airlineInfo != null)

_airlinesState.UpdatePlanePositionIndex(AirLine,
_airlineInfo);
        }
```

```
        else
        {
            PathFlyIndex--;

            if (PathFlyIndex < 0)
            {
                IsFlight = false;
                PathFlyIndex = 0;
                (PreviousDestination, CurrentDestination) =
(CurrentDestination, PreviousDestination);
                OperationCost();
                AirLine.FinishFlight(PreviousDestination,
this);
                OnFlightFinish?.Invoke();
            }
            if(_airlineInfo != null)

_airlinesState.UpdatePlanePositionIndex(AirLine,
_airlineInfo);
        }


        MyTransform.rotation =
LookRotation(AirLine.Path[PathFlyIndex] -
MyTransform.position, MyTransform.forward);
    }


    private void OnFinishFly()
    {
        AirLine.FinishFlight(PreviousDestination, this);
    }
```

```
    public void UpdateToTargetLevel(int targetNextLevel,
AirPlaneOrder airPlaneOrder)
    {
        AirPlaneLevel nextLevel =
_airPlaneLevelConfig.GetAirPlaneLevelByOrder(targetNextLeve
l, airPlaneOrder);
        AirPlaneLevel = nextLevel.Clone();
        PlaneMesh.sharedMesh =
_airPlaneLevelConfig.GetAirPlaneMeshDataByLevel(AirPlaneLev
el.Level).Mesh;
        AirLine.TotalSpendMoney += AirPlaneLevel.Price;
        AirPlaneLevel.PlaneOperationCost =
_remoteConfigController.BalanceConfig.GetPlaneOperationCost
ByLevel(AirPlaneLevel.Level);

_goalSystem.UpdateGoalExecution(Eu.Cryptotycoon.State.GoalT
ype.UpgradeXairplanesToYlevel, AirPlaneLevel.Level);
        _airlinesState.UpdatePlanesOnAirLine(AirLine);
    }

    public void DisableFly(Action onWaitFinish = null)
    {
        IsBusy = true;
        MyTransform.gameObject.SetActive(false);
        _onWaitFinish = onWaitFinish;
        _eventsController.OnEventCompleted += EnableFlight;
    }

    private async void EnableFlight(EventModel eventModel)
    {
        if (IsEventActive())
            return;
```

```
        if
(!_gameController.CountryControllers.Contains(AirLine.First
Airport.MyCountry))
        {
            _eventsController.OnEventCompleted -=
EnableFlight;
            IsBusy = false;
            return;
        }

        await Task.Delay(Random.Range(1000,6000));
        _eventsController.OnEventCompleted -= EnableFlight;

        if (GameObject == null)
            return;

        IsBusy = false;
        _onWaitFinish?.Invoke();
        GameObject.SetActive(true);
    }

    private void HandlePlaneClick()
    {
        if (_tutorialController.IsTutorialActive ||
AirplaneType == AirplaneType.ExtraByEvent)
            return;


_popupManager.ShowPopup(PopupConstants.AirplaneInfoPopupPat
h, new ShowOptions
        {
            InParameter = new AirplaneInfoPopup.Params
```

```
            {
                AirPlane = this
            },
        });
    }

    private bool IsEventActive()
    {
        return AirLine.SecondAirport.IsEventEnable ||
AirLine.FirstAirport.IsEventEnable;
    }

    public event Action OnFlightFinish;
    public void Dispose()
    {
        _eventsController.OnEventCompleted -= EnableFlight;
    }

    public void SetAirlineInfo(AirlineInfo airlineInfo)
    {
        _airlineInfo = airlineInfo;
    }
}
```

ДОДАТОК Б

ПЕРЕЛІК ДОКУМЕНТІВ НА ОПТИЧНОМУ НОСІЇ

Ім'я файла	**Опис**
Пояснювальні документи	
Диплом_ Савостяненко.doc	Пояснювальна записка роботи. Документ Word.
Диплом_ Савостяненко.pdf	Пояснювальна записка роботи в форматі PDF
Програма	
Program.rar	Архів. Містить коди програми і откомпільовану програму
Презентація	
Презентація_ Савостяненко.ppt	Презентація роботи