ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інженерії програмного забезпечення

Пояснювальна записка
до магістерської роботи
на ступінь вищої освіти магістр

на тему: «Застосування інформаційно-орієнтованих технологій Unity DOTS для підвищення продуктивності мобільних додатків»

Виконав: студент 6 курсу, групи ПДМ-61
спеціальності:
121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

Нагорний Є.О.
(прізвище та ініціали)

Керівник Бондарчук А.П.
(прізвище та ініціали)

Рецензент_____
(прізвище та ініціали)

Київ – 2022

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ТЕЛЕКОМУНІКАЦІЙ ТА ІНФОРМАТИЗАЦІЇ

Кафедра Інженерії програмного забезпечення
Ступінь вищої освіти - «Магістр»
Спеціальність – 121 Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри
інженерії програмного забезпечення
Негоденко О.В.

"____" _____2022 року

ЗАВДАННЯ
НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Нагорному Євгену Олеговичу
(прізвище, ім'я, по батькові)

1. Тема магістерської роботи: «Застосування інформаційно-орієнтованих технологій Unity DOTS для підвищення продуктивності мобільних додатків»

Керівник роботи Бондарчук Андрій Петрович, директор ННІІТ, професор, доктор технічних наук
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом закладу вищої оствіти від "11" жовтня 2022 року № 170

2. Строк подання студентом роботи

3. Вихідні дані до роботи:
Науково-технічна література з питань, пов'язаних з побудовою архітектури додатків та технологій орієнтованих на дані.

4.Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
4.1 Аналіз наявних засобів та технологій для розробки оптимізованої архітектури додатків за допомогою архітектурного патерну проектування ECS.
4.2 Дослідження інформаційно-орієнтованих технологій.
4.3 Реалізація оптимізованої архітектури мобільних додатків з використанням інформаційно-орієнтованих технологій.

5. Перелік графічного матеріалу
1.Титульний слайд
2.Мета, об’єкт дослідження, предмет дослідження
3.Актуальність роботи.
4.Аналіз існуючих рішень
5.Модель архітектурного патерну проектування ECS
5.Технології Unity DOTS: Job system, Burst Compiler
5.Результати тестування
6. Висновки

6. Дата видачі завдання

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
1.	Вибір теми магістерської роботи	05.10.2021р.	
2.	Вивчення літературних джерел, збирання та обробка інформації	10.10.2021р.	
3.	Складання плану роботи	15.10.2021р.	
4.	Узгодження плану роботи та списску використаних джерел з науковим керівником	25.10.2021р.	
5.	Аналіз існуючих рішень	20.09.2021р.	
6.	Дослідження інформаційно-орієнтованих технологій Unity DOTS	10.10.2021р.	
7.	Розробка оптимізованої архітектури додатку	24.10.2021р.	
8.	Вступ, висновки, реферат	05.12.2021р.	
9.	Розробка обов’язкових демонстраційних матеріалів	13.12.2021р.	
10.	Попередній захист роботи		
11.	Здача роботи в деканат		

Студент _____ Нагорний Є.О.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Бондарчук А.П.
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Текстова частина магістерської роботи: 70 сторінок, 31 рисунків, 19 джерел.

Об'єкт дослідження - процес розробки архітектури мобільних додатків з використанням інформаційно-орієнтованих технологій Unity DOTS.

Предметом дослідження – архітектурні патерни проектування та інформаційно орієнтовані технології Unity DOTS.

Методи дослідження: аналіз та порівняння.

Метою даної роботи є вдосконалення архітектури мобільних додатків для підвищення продуктивності на багато поточних процесорах з використанням інформаційно-орієнтованих технологій Unity DOTS.

У роботі проведено аналіз існуючих ECS фреймворків, таких як Entitas, LeoECS та ін. Ці фреймворки використовують архітектурний патерн для проектування ECS, він потрібен для того щоб вирішити проблему продуктивності у класичній компонентній архітектурі Unity3d.

Загальною проблемою цих фреймворків є те, що вони не використовують потенціал процесорів, які б могли покращити продуктивність у рази.

Головною особливістю у використанні технологій Unity DOTS є те, що вони дозволяють чітко організувати дані у пам’яті та розпаралелити процеси пов’язані з використання цих даних, для того щоб використати багато поточність процесора.

Дослідження полягає у розробленні теоретико-методичних та практичних рекомендацій які дозволяють розробити архітектурне рішення для зменшення енергоспоживання батареї мобільного пристрою та збільшення продуктивності на багатопоточних процесорах.

Запропановані заходи удосконалення можуть бути використанні у процесі розробки архітектури мобільних додатків.

КЛЮЧОВІ СЛОВА: UNITY3D, СТЕК ТЕХНОЛОГІЙ ОРІЄНТОВАНИХ НА ДАНІ, UNITY DOTS, ENTITY COMPONENT SYSTEM, ПІДВИЩЕННЯ ПРОДУКТИВНОСТІ МОБІЛЬНИХ ДОДАТКІВ.

ЗМІСТ

Стор.

ВСТУП

Обгрунтування вибору теми та її актуальність: у сучасному світі запит на використання передової графіки та сучасних технологій зростає, продуктивність ЦП і пам'яті не збільшується відповідно до потреб додатків, саме тому програмна архітектура, яка може якнайкраще використовувати характеристики та обмеження мобільних пристроїв, забезпечує життєздатне рішення для задоволення цих потреб, саме тут орієнтований на дані підхід до кодування в Unity може надати підвищення продуктивності. Unity створений для цілого ряду апаратних цілей, але є загальні проблеми для всього обладнання та області, які ми можемо вирішити. Простіше кажучи, це такі проблеми:

- Швидкість обробки процесора знижується. Темпи зростання, які ми спостерігали за останні 20 років, навряд чи повторюватимуться знову;
- Для багатьох апаратних систем швидкість доступу до пам'яті не збільшується пропорційно швидкості обробки. Unity потрібна надзвичайно висока швидкість доступу до пам'яті. Для тих платформ, які мають високу швидкість доступу до пам'яті, вам потрібен метод організації та доступу до даних у пам'яті, який може ефективно використовувати цю швидкість;
- Графічні процесори забезпечують рішення підвищення продуктивності за допомогою використання паралельної обробки, але без спеціальних інструментів їх використання є відносно невикористаним ресурсом.

Програмне забезпечення є ключем до вирішення таких апаратних обмежень підвищення продуктивності. На практиці це означає менше енергоспоживання та збільшений термін служби батареї.

Сутність вивчення проблеми: на даний момент існує не багато фреймворків які дозволяють вирішити проблему з низькою продуктивністю додатків при використанням класичної архітектури компонентів Unity3d. Існують фреймворки, які основані на архітектурному патерні проектування ECS (цей патерн проектування дозволяє дуже швидко звертатися до пам’яті), але всі ці фреймворки

які існують на сьогоднішній день, не дозволяють використовувати потенціал процесорів.

Метою даної роботи є вдосконалення архітектури мобільних додатків для підвищення продуктивності на багато поточних процесорах з використанням інформаційно-орієнтованих технологій Unity DOTS.

Для реалізації мети було сформовано та вирішено наступні завдання:

1. Провести аналіз існуючих рішень.
2. Дослідити інформаційно-орієнтовані технології Unity DOTS.
3. Розробити оптимізовану архітектуру додатка за допомогою новітніх технологій.
4. Проаналізувати отримані показники продуктивності.

Об'єкт дослідження - процес розробки архітектури мобільних додатків з використанням інформаційно-орієнтованих технологій Unity DOTS.

Предмет дослідження – архітектурні патерни проектування та інформаційно орієнтовані технології Unity DOTS.

Методи дослідження: аналіз та порівняння.

Наукова новизна роботи: Дослідження полягає у розробленні теоретико-методичних та практичних рекомендацій які дозволяють розробити архітектурне рішення для зменшення енергоспоживання батареї мобільного пристрою та збільшення продуктивності.

Практична значущість результатів: Запропановані заходи удосконалення можуть бути використанні у процесі розробки архітектури мобільних додатків.

1 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

1.1 Опис існуючих ECS фреймворків

ECS (Entity-Component-System) – це шаблон проектування, який ставить в основу дані, а не об'єкти, і тим самим перевертає звичайне уявлення про програмування серед цінителів ООП. Подібний підхід розвиває ідею композиції над успадкуванням і дозволяє легко адаптуватися до динамічних потреб гейм-дизайнера.

Цей патерн в основному використовується в іграх, він забезпечує величезну гнучкість у проектуванні загальної архітектури програмного забезпечення. Такі великі компанії, як Unity, Epic або Crytek використовують цей шаблон у своїх фреймворках, щоб надати розробникам дуже багатий на можливості інструмент, за допомогою якого вони можуть розробляти власне ПЗ.

Сьогодні потенціал ЦП пристроїв кінцевих користувачів може відрізнятися. Як правило, розробники визначають мінімальні специфікації ЦП та використовують цю мету продуктивності для реалізації симуляції та ігрових систем. В результаті багато потенційно доступних ядр і функцій, вбудованих в сучасні основні процесори, простоюють. Підхід до проектування ECS в Unity не тільки дозволяє легко використовувати ресурси ЦП, що раніше не використовувались, але й допомагають більш ефективно виконувати весь ігровий код. Потім ви можете використовувати ці додаткові ресурси процесора, щоб додати стільки обь’єктів на сцені скільки вам потрібно.

Цей шаблон проектування використовується для вирішення двох важливих проблем продуктивності у обчисленнях ігрового двигуна Unity. Перша проблема, яку він вирішує, не ефективне розташування даних, Entity Component System (ECS) покращує зберігання та управління даних та забезпечує високопродуктивні обчислення на цих структурах. Друге питання це відсутність високопродуктивної мови завдань та векторизації SIMD, які можуть працювати з упорядкованими даними.

Коли потрібно використати патерн проектування ECS у розробці архітектури додатку, зазвичай використовують два найпопулярнішних фреймворка, це Entitas та Leo ECS.

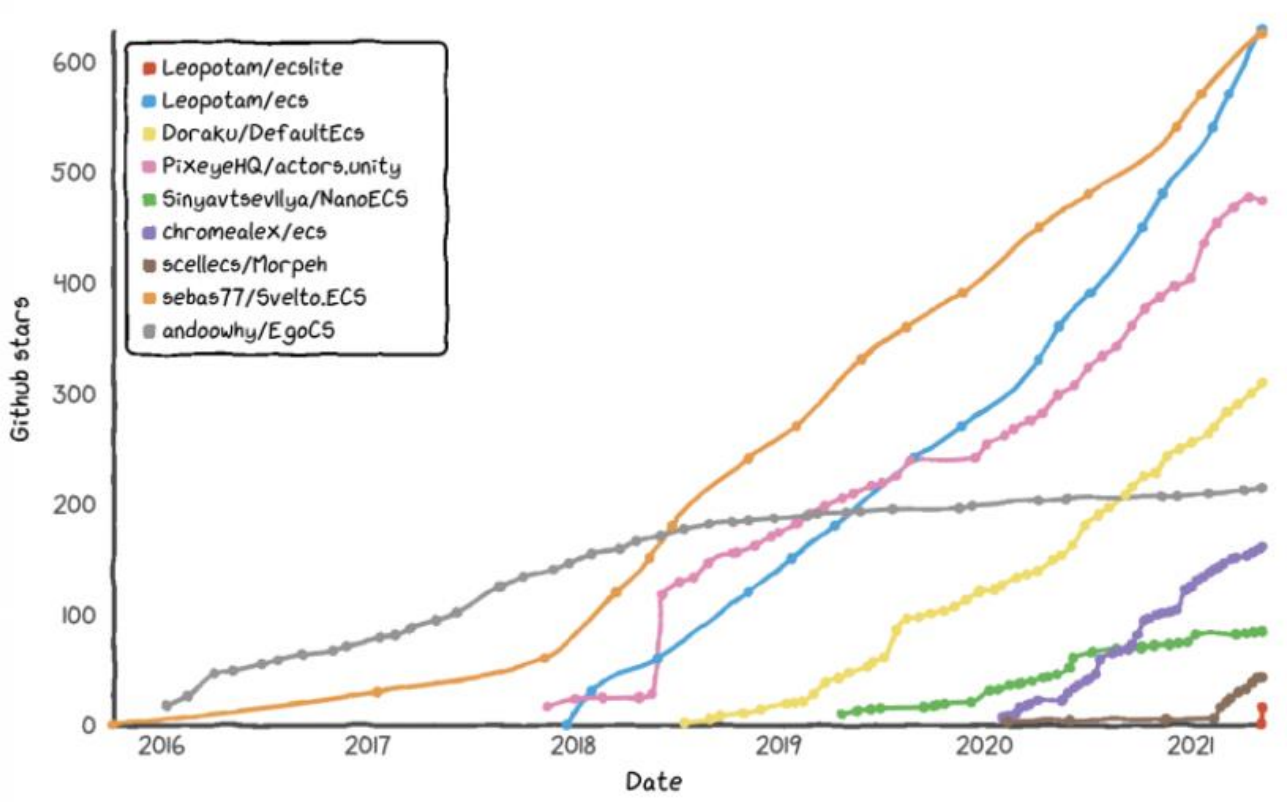


Рисунок 1.1 – Діаграма використання ECS фреймворків.

Entitas – це надшвидкий Entity Component System (ECS), спеціально створений для C# та Unity. Це рішення має свої плюси і зручні можливості, яких у Unity ECS поки немає, наприклад, реактивні системи. Реактивні системи — це такий підвид систем, які дозволяють обробляти ті сутності із заданими компонентами, дані яких змінилися з попереднього запуску системи. Але, незважаючи на те, що цей фреймворк є досить продуктивним рішенням, щодо швидкості тягатися з Unity ECS він, звичайно, не може. Однак він є production ready рішенням. Внутрішнє кешування та неймовірно швидкий доступ до компонентів роблять його неперевершеним. В ньому було прийнято кілька проектних рішень для оптимальної роботи в середовищі зі складанням сміття та для спрощення роботи зі збирачем сміття. Entitas поставляється з додатковим генератором коду,

він генерує для вас класи та методи, щоб ви могли зосередитись на виконанні своєї роботи. Це радикально скорочує обсяг коду, який вам потрібно написати, та значно покращує читаність. Це робить ваш код менш схильні до помилок, забезпечуючи при цьому кращу продуктивність.

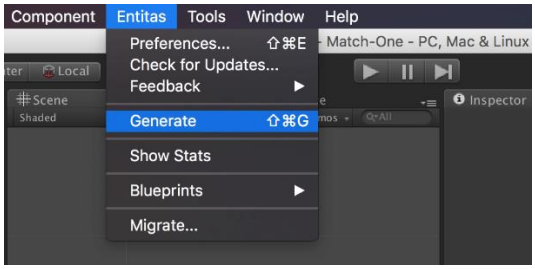


Рисунок 1.2 –Вікно для генерації коду.

Додатковий модуль Unity чудово інтегрує Entitas у Unity та надає потужні розширення редактора для перевірки та налагодження контекстів, груп, сутностей, компонентів та систем.

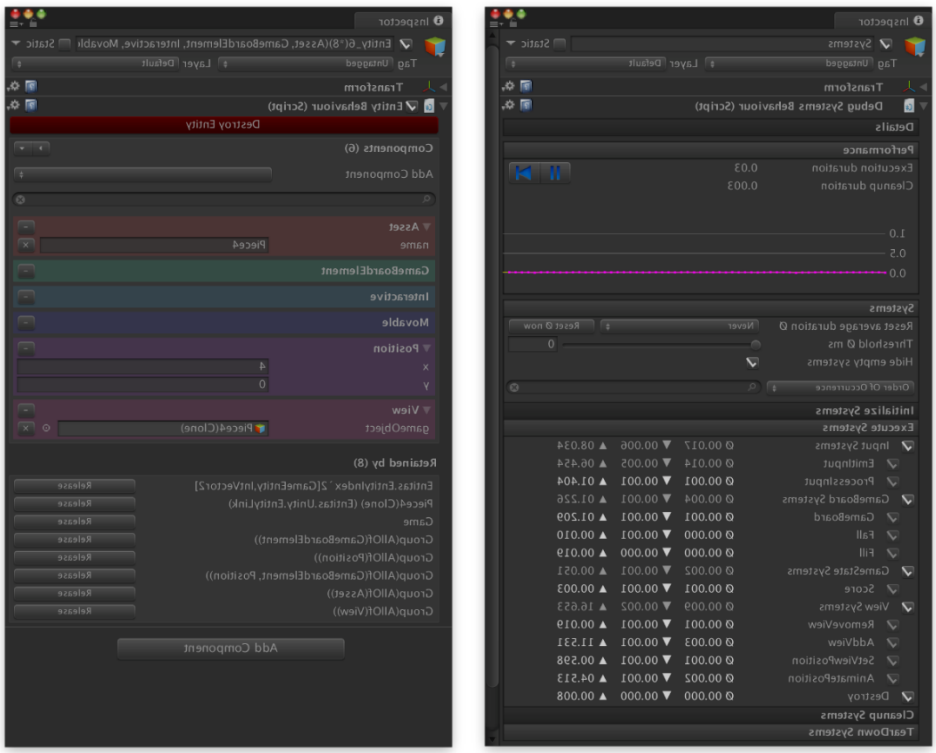


Рисунок 1.3 – Розширення редактора під час використання Entitas.

Апетитний фреймворк із найбільшим ком'юніті, але вся його проблема в тому, що він працює на класах. Додавати цей інгредієнт зараз не модно, кухарі вважають за краще використовувати структури на компонентах. Класи майже в 2 рази повільніші за структури в синтетичних тестах, тому що зберігають у пам'яті посилання. А дані тести демонструють відставання за швидкістю готування від 10 до 20 разів. Entitas працює швидко, легко і позбавляється непотрібних складнощів. Є менше, ніж кілька класів, які вам потрібно знати, щоб запустити гру або програму: entity, context, group, entity collector.

Leo ECS – Проста та легка структура системи компонентів сутностей C#, дуже легкий та швидкий ECS фреймворк, який не потребує інтеграції з Unity Engine. Оперує структурами, а не класами що в свою чергу прискорює роботу. Крім цього, на даний момент Leo ECS найпопулярніший ECS фреймворк після Entitas.

Як працює фреймворк LeoECS? Для відповіді на це питання ми розглянемо приклад реалізації функціоналу для пересування, розберемо базові моменти при використанні фреймворку LeoECS, для того щоб на практиці розуміти як працює архітектурний патерн ECS.

Для того щоб встановити репозиторій з github в свій проект, спочатку потрібно скопіювати посилання на репозиторій з github, далі відкрити unity engine і зайти в package manager та вставити силку на репозиторій.

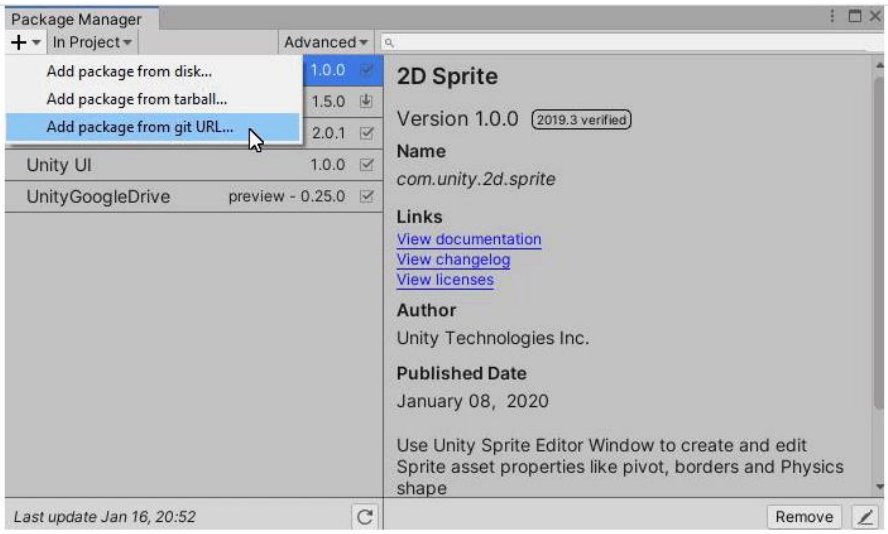


Рисунок 1.4 – Встановлення розширення з репозиторія github.

Під час завантаження з репозиторія розробника завжди буде використовуватиметься остання випущена версія.

Для початку нам потрібно зв'язати ECS з Monobihavior, для цього ми створимо эдиный monobihavior скрипт, який буде вхідною точкою для ECS. В цьому скрипті потрібно оголосити приватні змінні, такі як: ECS World та ECS System. World – це класс, який містить у собі всі сутності (Entity), які знаходяться у ньому. Дуже важливо визвати метод EcsWorld.Destroy(), коли екземпляр більше не використовується. Для наглядності представимо що world це сцена в unity зі всіма об'єктами, цих сцен може бути кілька, но на практиці частіше всього використовують один world. System зберігає у собі всі системи для обробки даних, також містить загальні методи для виклику всіх систем, таких як init, preInit, run, destroy, postDestroy.

```
class UserSystem : IEcsPreInitSystem, IEcsInitSystem, IEcsRunSystem, IEcsDestroySystem, IEcsPostDestroySystem {
    public void PreInit () {
        // Will be called once during EcsSystems.Init() call and before IEcsInitSystem.Init.
    }

    public void Init () {
        // Will be called once during EcsSystems.Init() call.
    }

    public void Run () {
        // Will be called on each EcsSystems.Run() call.
    }

    public void Destroy () {
        // Will be called once during EcsSystems.Destroy() call.
    }

    public void PostDestroy () {
        // Will be called once during EcsSystems.Destroy() call and after IEcsDestroySystem.Destroy.
    }
}
```

Рисунок 1.5 – інтерфейси систем LeoECS.

На рисунку 1.5 ми можемо побачити подібність у функціоналі інтерфейсах, вони дуже схожі з методами unity, такими як Start, Awake та Update.

Далі у методі Start() ініціалізуємо оголошені класи та в ECS System передаємо world. Використовуємо метод system.convertScene який необхідний для роботи юні

лео. Потім прописуємо виклик ініціалізації всіх систем в update виклику метода system.run(). Прописуємо у методі OnDestroy() видалення всіх систем та світів. Тепер для систем та компонентів пропишемо методи, такі як: AddSystems(), AddInjections(), AddOneFrames().

Для реалізації функціоналу пересування потрібно також написати систему яка буде оперувати логікою над нашими компонентами з даними. Для початку напишемо компонент який буде зберігати посилання на контролер та змінну швидкості, він буде називатися MovableComponent.

```
struct WeaponComponent {
    public int Ammo;
    public string GunName;
}
```

Рисунок 1.6 – Приклад створення компонентів в LeoECS.

Для цього компоненту напишемо систему пересування, яка буде називатися MovementSystem. Модифікатор sealed - говорить нам що неможливо буде успадковуватися від класу. Реалізуємо в системі інтерфейс IEcsRunSystem – це аналог методу update в unity. В системах існують базові речі які потрібно буде частіше за все реалізовувати, це світ та фільтри. В змінних типу world треба присвоїти значення null, тому що згодом вони будуть автоматично підключені так само як і фільтри. Здебільшого оголошувати світи треба в тих випадках коли треба створити нову сутність (Entity) або відправити повідомлення у світ.

Система повина розуміти з якими сутностями вона повина працювати, для цього існують фільтри. Під час використання фільтрів треба вказати ті компоненти які повині бути присутніми на об’єкті при пошуку.

Для системи переміщування потрібно створити компонент та систему під назвами DirectionComponent та InputSystem. В системі InputSystem треба вказати фільтр на компонент DirectionComponent.

Окрім компонентів також існують теги, вони складаються зі структури, частіше за все використовуються для пошуку у фільтрі.

В системах логіка повина розміщуватися у методі run(), для того щоб отфільтрувати компоненти треба у цьому методі пройтись по циклу foreach заздалегідь всіх відфільтрованих сутностей.

Для того щоб оголошувати локальні змінні всередині foreach з неявною типізацією потрібно використати ключове слово var, разом з цим потрібно використати ключове слово ref яке вказує, що значення будуть передаватися за посиланням, це значить що упаковка перетворення відбуватися не буде.

В середені циклу foreach ми створюємо локальну змінну directionComponent, для отримання даних цього компоненту ми використаэмо метод Get2. Всього існує шість методів Get, вони створені для того щоб брати з фільтру конкретний компонент. Фільтр може містити у собі кілька компонентів для більш чіткого пошуку сутностей.

```
// Creates new entity in world context.
EcsEntity entity = _world.NewEntity ();

// Get() returns component on entity. If component not exists - it will be added.
ref Component1 c1 = ref entity.Get<Component1> ();
ref Component2 c2 = ref entity.Get<Component2> ();

// Del() removes component from entity. If it was last component - entity will be removed automatically too.
entity.Del<Component2> ();

// Component can be replaced with new instance of component. If component not exist - it will be added.
var weapon = new WeaponComponent () { Ammo = 10, GunName = "Handgun" };
entity.Replace (weapon);

// With Replace() you can chain component's creation:
var entity2 = world.NewEntity ();
entity2.Replace (new Component1 { Id = 10 }).Replace (new Component2 { Name = "Username" });

// Any entity can be copied with all components:
var entity2Copy = entity2.Copy ();

// Any entity can be merged / "moved" to another entity (source will be destroyed):
var newEntity = world.NewEntity ();
entity2Copy.MoveTo (newEntity); // all components from entity2Copy moved to newEntity, entity2Copy destroyed.

// Any entity can be destroyed. All component will be removed first, then entity will be destroyed.
entity.Destroy ();
```

Рисунок 1.7 – створення сутностей (Entities) та їх методи.

Дуже важливо те що ми не повинні використовувати ref модифікатор для будь-яких даних фільтра поза циклом foreach, це може порушити цілісність пам'яті.

Для системи InputSystem потрібно зчитувати введення з клавіатури, для цього створимо приватний метод в якому помістимо стандартні методи Input.GetAxis. Цей метод будемо викликати в методі run().

Як ми вже бачимо використовувати ECS легко, принцип дуже легкий, потрібно лише створити компоненти з даними та системами, в системах ми повинні відфільтрувати компоненти, також в цих системах ми пишемо логіку, але це ще не все, нам ще потрібно запустити ці системи.

Для того щоб запустити системи спочатку потрібно створити метод який будет додавати всі нащі системи, раніше ми створювали змінну systems типу ECSSystem, цей класс містить у собі метод для того щоб додавати системи. Дуже важливо слідкувати за порядком систем.

Викликає різонне питання, де і як створювати сутності (Entity), як добавити на них потрібні компоненти та передати контролер для пересування якщо сама ECS абстрагована від сцени в unity. Для вирішення цієї проблеми розробники LeoECS розробили розширення під назвою UniLeo, це такий собі провайдер для компонентів.

Для того щоб підключити UniLeo спочатку потрібно підключити бібліотеку Voody.UniLeo, далі створити класс для конкретного компонента (MovableProvider) та успадкуватися від MonoProvider, та передати потрібний нам компонент (<MovableComponent>). Також треба написати для конкретного компонента атрибут [Serializable]. Далі відкриваємо unity та додаємо до потрібного gameObject тільки що створений провайдер, до цього об’єкту також автоматично добавиться скрипт під назвою Convert to Entity, він знайде всі провайдери на об’єкті та створить entity передав до нього потрібні компоненти.

Із фільтрів можливо брати не тільки компоненти, а також сутності (Entity), які містять ці компоненти, для цього існує метод GetEntity(). Для того щоб отримати конкретний компонент в системі треба використати метод Get() вже відфільтрованої сутності. Якщо в entity існує цей компонент цей метод його отримає, якщо ні створить новий компонент.

В LeoECS також існують події, по суті своїй це структура яка нічого в собі не містить. Запит або request це структура яка і є подією, яка може щось передавати. Для того щоб створити подію в LeoECS спочатку потрібно створити порожню структуру, далі в системі прописати фільтр на посилання цієї порожньої структури, а також тег, та добавити цю подію у методі AddOneFrames – це метод який буде існувати на сутності протягом одного кадру, існує ще одна система яка буде відправляти цю подію наприклад при натисканні на кнопку, подія додається на ігрока або другий gameObject завдяки методу сутність.Get<івент>().

Окрім додовання до фільтру певних компонентів також їх можливо видаляти, для цього треба після переліку компонентів додати через крапку до фільтра виклик методу Exclude та перерахувати ті компоненти які ми хочемо виключити. Таким чином ми можемо додати до системи певні умови, наприклад гравець не може швидко пересуватися, якщо на ньому є певні компоненти. В методі exclude в у відмінності від методу include можливо додати лише два компонента, у відмінності від include де можливо додати цілих шість компонентів.

Всі компоненти з Include у фільтрі можуть бути швидко доступні через EcsFilter.Get1() та EcsFilter.Get2() і так далі у тому порядку, в якому вони використовувалися при оголошенні типу фільтра.

При знищені ігрового об’єкту, обов’язково потрібно знищувати entity методом entity.destroy(), але якщо ми створюємо об’єкт для пулу видаляти нічого не потрібно, коли потрібно його сховати, викликаємо метод entity.despawn(), потім накидуємо на сутність тег, який буде говорити що цей об’єк не є активним, також потрібно не забути виключити із фільтра системи цей тег за допомогою exclude, для того щоб вони зайвий раз не обробляли цей об’єкт.

Також існують не настільки популярні ECS фрейморки, але у них є свої особливості. Наприклад ECS.ME - приголомшливий фреймворк ECS для Unity, приготовлений для мережних ігор з елементом відкату (RollBack). Такий собі некомерційний аналог Quantum від Exit Games. Рецепт дуже крутий, але щоб rollback не дав тріщину, потрібно чітко дотримуватися правил даного фреймворку.

EgoECS - ECS фреймворк із найсильнішою інтеграцією в Unity. З його допомогою можна готувати MonoBehavior як компоненти. Якщо вам не важлива швидкість роботи, EgoECS виглядає дуже хорошим рецептом. Кожна система – це лише 1 фільтр, комусь такий підхід може скручувати руки під час написання коду. Сутності" (Entity) – об'єкти-контейнери, що не мають властивостей, але є сховищами для "Компонентів".

Svelto ECS – дуже амбітний рецепт, з дуже складним API. Але я мав його згадати. Швидше за все, він не підходить новачкам, але якщо ви зацікавилися, то на habr існує переклад Wiki даного фреймворку.

Actors - спочатку розроблявся як набір утиліт. У рецепті є шина подій (глобальні івенти), яка може порушити порядок функцій і ніяк не відноситься до ECS. Реактивщина в ECS може збити вас з пантелику при пошуку багів / помилок у вашій грі.

Отже, ми розглянули основні фреймворки які використовують архітектурний патерн проеткування ECS. Ми переконалися, наскільки приємним є рефакторинг коду написаного з використанням архітектурного рішення ECS від фреймворку LeoECS. Також на прикладах були розглянуті шляхи використання цього фреймворку. Тим не менш, жодне зі спеціалізованих рішень ECS не може надати Unity великих змін у продуктивності, ніж дозволяє сам двигун, але все ж таки приріст є в порівнянні з класичною архітектурою розробки у Unity.

1.2 Основні відмінності у проектуванні під час використання ESC

Для того щоб виявити основні відміності у проктуванні під час використання архітектурного патерну ECS ми повині розглянути та відповісти на такі питання:

1. Які проблеми та задачі вирішує використання ECS.
2. Плюси і мінуси двох підходів до проектування.

Почнемо з того, що на сьогоднішній день воістину переважна більшість як компаній, що добре зарекомендували себе, так і аматорських розробників ігор та інді-команд використовують середовище розробки Unity3d для розробки ігор,

додатків, моделювання, візуалізації і т.д. Однією з причин такої популярності є досить низький поріг входу в порівнянні з іншими. Це дозволяє новачкам створювати свої перші прототипи через кілька днів навчання.

Звичайно, така простота виглядає дуже привабливо. Адже справді, досить кинути асети на ігрову сцену, повісити на них пару вбудованих компонентів та кілька скриптів, і все це запрацює. Unity подбає практично про все самостійно.

Такий підхід дає результати, але лише до певного моменту. Таким чином ви можете створити кілька швидких та простих ігрових прототипів, які не вимагають реалізації складних взаємодій між об'єктами та виконання багатьох умов. Але якщо ви хочете створити справді якісний продукт, ви обов'язково зіткнетеся з проблемою – такі проекти не масштабуються.

Якщо наші амбіції виходять за рамки звичайної реалізації «клікера», слід зосередити увагу на проблемі розширюваності ігрової архітектури. Чим складніша і різноманітніша ігрова механіка, різноманітність контенту та можливі взаємодії, тим складніше правильно працювати та уникати так званого «спагетті-коду».

Зв'язки між класами та модулями стають надзвичайно тісними. Взаємодія класів сильно переплетена. Тому зміна будь-якого з існуючих механізмів або додавання нового стає неможливим або неймовірно складним, оскільки все починає залежати від усього, і що ще гірше стає все важче і важче знаходити ці зв'язки.

Навіть найменша зміна (яка, як ми думали, не повинна нічого зламати) в одній частині гри вносить хаос у непередбачувані місця. Спроби додати нову функцію перетворюються на довгий шлях встановлення тісних зв'язків із великою кількістю залежних модулів та виправлення низки непередбачених помилок. Звичайно, використання принципів та патернів гнучкої розробки під час створення ігор, безсумнівно, спростить ситуацію.

Але є ще один цікавий підхід, який дуже зручно застосовувати при розробці ігор – патерн Entity-Component-System (ECS). На відміну від стандартного підходу, при реалізації цього шаблону проектування ми маємо справу не з конкретними об'єктами, поведінка яких описується відповідними класами, а з «сутностями», які

власними силами не несуть жодної логіки і є скоріше контейнерами для «компонентів». Останні – це сховища даних, які визначають властивості сутностей із такими компонентами.

«Системи» відповідають за обробку сутностей та його компонентів. Це класи, які найчастіше обробляють сутності з аналогічним набором компонентів (і отже, з аналогічними властивостями). Вони є центральною ланкою усієї логіки.

У цьому підході примітно те, що для системи немає значення, з якими сутностями працювати. Всі вони будуть оброблятися відповідно до чітко визначених правил незалежно від того, який тип ігрового об'єкта їх представляє. Наприклад, є система, яка переміщає всі об'єкти за допомогою компонента «Рухливий». Неважливо, що саме викликає рух – чи то гравець чи снаряд, випущений зі зброї – кожен із них буде оброблений однаково.

Сутності можуть відрізнятися швидкістю або, наприклад, значенням обмеження швидкості всередині рухомого компонента (тип даних і кількість заздалегідь визначаються, тому початкові значення встановлюються під час створення компонента).

У нас може бути стільки систем, скільки нам потрібно, і кожна з них відповідатиме лише за певну ігрову логіку та оброблятиме лише ті сутності, які відповідають заданим фільтрам компонентів. Це дозволяє нам створювати архітектуру, що складається із сильно розділених модулів.

Які ж переваги існують у ECS? Коли ми вперше починаємо працювати з підходом ECS, все здається надто складним. Потрібно буквально перебудувати спосіб мислення та почати пошук схожих властивостей об'єктів, групуючи їх у компоненти та створюючи системи для їх обробки. Зробивши це, ми почнемо по-іншому бачити архітектуру гри.

Використання ECS дозволяє відокремити логіку від представлення. Це також допомагає зробити логіку неймовірно гнучкою та настроюваною через відсутність міцних зв'язків між сутностями. Таким чином, нові функції можуть бути легко додані або видалені, не побоюючись існуючих порушень.

Незважаючи на те, що на початку розробки гри потрібно багато часу приділяти реалізації підходу ECS, ми, безумовно, виграємо час на середній та пізній фазі розробки за рахунок гнучкості архітектури. Це в кілька разів краще, ніж стандартний підхід, використовуючи тільки Unity MonoBehaviours.

До недавна найбільш цінними перевагами використання ECS у Unity були гнучкість та можливість масштабування. Підвищення продуктивності було менш помітною перевагою.

Однак продуктивність була покращена за рахунок зменшення кількості активних MonoBehaviour. За стандартом вони мають безліч вбудованих функцій, тому час виконання для системи, що обробляє пару сотень сутностей, буде меншим у порівнянні з MonoBehaviour, кожна з яких має власний метод оновлення.

Як ми вже з'ясували існує два варіанти робочого процесу:

1. Робочий процес, орієнтований на MonoBehaviour, який працює із повністю розробленими інструментами Unity, робочим процесом та API-інтерфейсами MonoBehaviour.
2. Робочий процес ECS, який повністю використовує нову парадигму розробки ECS і пов'язані з нею поліпшення продуктивності.

Але перш ніж ми розглянемо всі деталі, важливо пам'ятати, що ці робочі процеси не виключають один одного. Більшість Unity розробників, починають використовувати робочий процес, орієнтований на MonoBehaviour, а потім оптимізують свою гру за допомогою ECS.

1.2.1 Приклад роботи ECS

Розглянемо приклад, у нас є завдання від дизайнера: "треба зробити переміщення гравця і завантаження наступного рівня, коли він доходить до точки X".

Розбиваємо завдання на кілька підзадач, по одному на "систему":

- UserInputSystem - введення користувача;
- MovePlayerSystem – переміщення гравця на основі введення;
- CheckPointSystem – перевірка досягнення точки гравцем;

- LoadLevelSystem - завантаження рівня в потрібний момент.

Визначаємо компоненти:

- UserInputEvent — подія про наявність введення користувача з даними про нього. Так, події – це також компоненти;
- Player – збереження поточної позиції гравця та його швидкості;
- CheckPoint – точка взаємодії на карті;
- LoadLevelEvent — подія про необхідність завантаження нового рівня.

І ось як це все працює:

1. Завантажується сцена та ініціалізуються всі системи у зазначеній вище послідовності. Так, порядок обробки систем можна контролювати без складних рухів тіла - це ще один приємний бонус.
2. Створюються сутності гравця (з додаванням на нього компонента Player) та сутності контрольної точки (з додаванням до неї компонента CheckPoint).
3. Далі стартує основний цикл обробки систем - насправді аналог способу MonoBehaviour.Update.
4. UserInputSystem перевіряє введення користувача через стандартне Unity-api і створює нову сутність з компонентом UserInputEvent і даними про введення (якщо він був).
5. MovePlayerSystem перевіряє, чи є сутності з компонентом UserInputEvent і чи є сутності з компонентом Player. Якщо введення користувача є - обробляємо всіх знайдених "гравців" (навіть якщо він один) з отриманими даними, а сутність з компонентом UserInputEvent видаляємо повністю. Так, це працює дуже швидко, не викликає роботи збирача сміття (garbage collector) – все йде у внутрішній пул для подальшого перевикористання.
6. CheckPointSystem перевіряє, чи є сутності з компонентом CheckPoint і чи є сутності з компонентом Player. Якщо є те й те — у циклі перевіряє дистанції між кожним гравцем та точкою. Якщо один із "гравців" знаходиться досить близько для спрацьовування - створює нову сутність з компонентом LoadLevelEvent.

7. LoadLevelSystem перевіряє, чи є сутності з компонентом LoadLevelEvent і виконує завантаження нової сцени за наявності. Усі сутності з таким компонентом видаляються перед цим.
8. Повторюємо основний цикл обробки систем.

1.2.2 Переваги ECS

Виходячи з прикладу, можна одразу винести основні особливості ECS по відношенню до компонентної моделі Unity. Серед істотних переваг при використанні архітектурного патерну ECS слід виділити наступні:

- гнучкість і масштабність (додавання нових, видалення старих систем та компонентів);
- ефективне використання пам'яті (особливість реалізації, ми можемо перевикористовувати інстанси "чистих" C#-класів як завгодно, з monobehaviour цього зробити неможливо);
- зрозумілий поділ логіки та даних;
- спрощеність тестування через легке відтворення тестового оточення (легко тестувати);
- можливість використання логіки на сервері без Unity (немає залежностей від самого unity engine);
- один Update на цілу сцену;
- контроль порядку виклику систем. Можна відключати одну систему, при цьому інші системи спокійно працюватимуть і надалі;
- зручний доступ до об'єктів (вибірка, фільтрація без втрати швидкості та алокацій пам'яті), чого компонентний підхід у Unity не має.
- пошук сутностей по фільтру в рази дешевше за продуктивністю ніж у звичайного методу GetComponent у monobihavior;
- single responsibility (Багато маленьких систем і компонентів, які відповідають за щось своє).

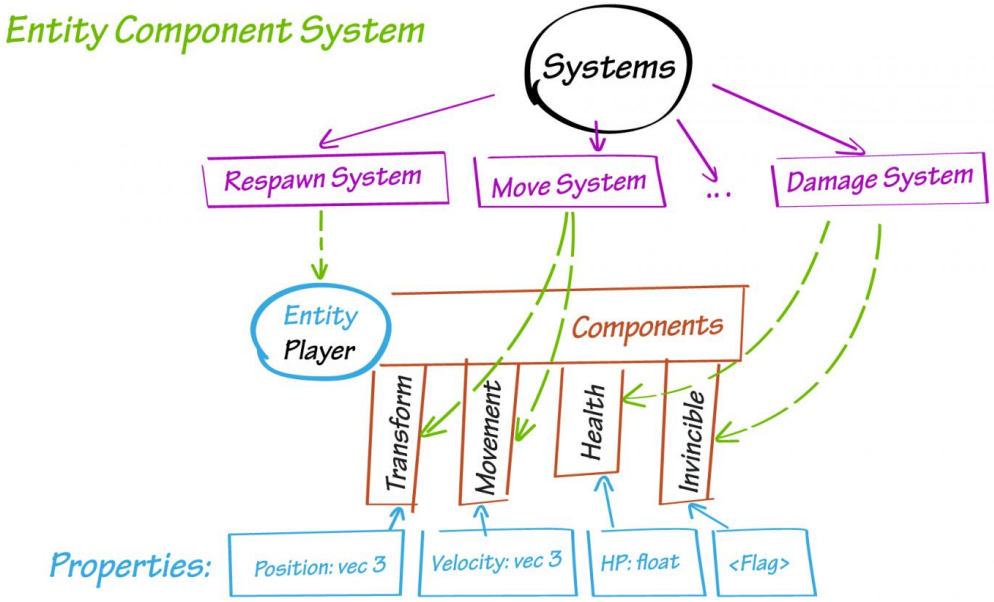


Рисунок 1.8 – Схема взаємодії систем, сутностей та компонентів.

На рисунку 1.8 ми бачемо структуру ECS, яка сприяє більш чистій базі. Його модульна конструкція дозволяє писати модульний код, який легко читати, повторно використовувати та рефакторити. Також структура пам'яті створена ECS дозволяє додатково оптимізувати код, використовуючи попередню вибірку з кеша, процес використовуваний вашим процесором для забезпечення завантаження наступного біта даних, вже завантаженого в кеш, на той час, коли він його потребує. Парадигма ECS робить це за допомогою однакових за розміром частин компонентів, які можна перебирати лінійно. Замість того, щоб мати справу з об'єктами різних розмірів, що випадково зберігаються, CPU має справу тільки з цими частинами однакового розміру, які лінійно зберігаються в одній і тій же області пам'яті.

З іншого боку, якщо розглядати цю структуру з точки зору людини яка вперше бачить що таке ECS, можна подумати що це виглядає як надмірне ускладнення коду в порівнянні з одним «MonoBehaviour» класом у десяток рядків, але це дозволяє отримати такі переваги як:

- Дозволяє відокремити введення від решти логіки. Ми можемо змінити модель введення з клавіатури на мишу, контролер, тачскрін та інший код не зламається;
- Дозволяє розширювати поведінку обробки гравця новими способами без ламання поточних. Наприклад, ми можемо додати зони уповільнення або прискорення на карті шляхом додавання ще однієї або кількох систем та зміною параметра швидкості у компоненті Player для певних сутностей;
- Дозволяє мати на карті скільки завгодно контрольних точок, а не лише одну, як просив дизайнер;
- Дозволяє навіть мати кілька гравців, які управляються одним способом. Теж може бути частиною ігрової механіки.

Ми перерахували основні переваги ECS щодо продуктивності. Давайте більш детально розглянемо, як досягається кожна з цих переваг у продуктивності.

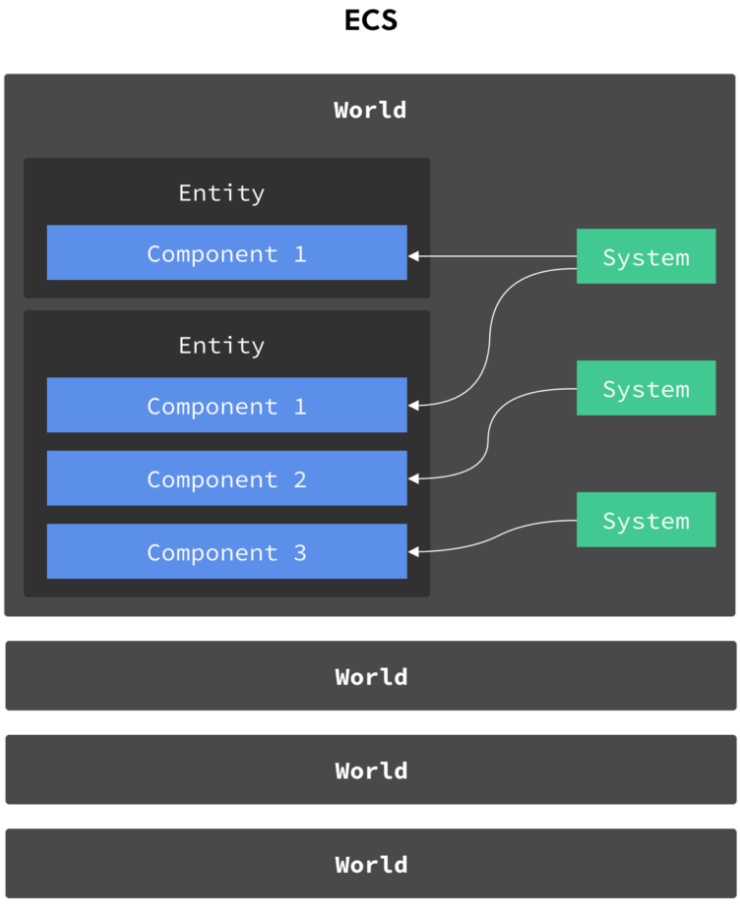


Рисунок 1.9 – Структура робочого процесу орієнтованого на ECS.

На рисунку 1.9 ми бачемо структуру ECS, вона поділяє дані (об'єкти і компоненти) і логіку (системи). ECS також виробляє більш модульний код, що легко читається і є більш ефективним ніж структура monobihavior.

«Код, згенерований ECS, виконує значно менше переходів між адресами пам'яті, ніж код, згенерований MonoBehaviour».

Таким чином ми можемо уникнути стрибків між адресами пам'яті, зберігаючи байти, з якими ми хочемо працювати, безперервно в тому самому фрагменті пам'яті. Як приклад того, як Unity ECS створює макет пам'яті, який дозволяє досягти цього, давайте представимо систему працездатності, яка віднімає значення компонента пошкодження з компонента здоров'я в кожному циклі оновлення:

ECS Unity поділяє кожну унікальну комбінацію типів компонентів, які з'являються по суті на архетипи. ECS динамічно визначає архетипи під час виконання, щоб класифікувати нові комбінації компонентів у міру їхньої появи. Одна унікальна комбінація, яка буде визначена як архетип, - це здоров'я та пошкодження. Інший - це, наприклад, здоров'я, пошкодження та мана.

Після класифікації в якості архетипу всі компоненти здоров'я та пошкодження, що існують на об'єктах без інших компонентів, зберігатимуться в тому самому блоці. Так само всі ізольовані екземпляри здоров'я, пошкодження і мани зберігатимуться в одному блоці. Єдиний виняток із цього - якщо сутностей більше, ніж може вмістити блок, і в цьому випадку вони будуть переповнюватися в наступний блок.

Коли ECS виконує ітерацію компонентів, доступ до пам'яті компонентів усередині блоку завжди повністю лінійний. Це призводить до того, що всі компоненти, з якими наша система охорони здоров'я має працювати, зберігаються у відносно невеликій кількості блоків та зберігаються безперервно всередині цих блоків. Це вимагає меншої кількості стрибків пам'яті.

Крім того, структура пам'яті, викликана ECS, дозволяє проводити подальшу оптимізацію коду з використанням попередньої вибірки з кешу пам’яті.

Чанки, створені Unity ECS, містять щільно упаковані та безперервні дані. Цей макет, на відміну від пам'яті, що рідко виділяється, купи, що генерується робочим процесом, орієнтованим на MonoBehaviour, спрощує апаратному пристрої попередньої вибірки ЦП передбачати і витягувати потрібні об'єкти в очікуванні необхідності в них.

Unity ECS може викликати таке розташування пам'яті, тому що компоненти представлені як неперетворювані типи. Перетворювані типи - це типи даних, які мають ідентичне уявлення (тобто вони займають однакову кількість байтів) у пам'яті як керованого, так некерованого коду. Використовуючи їх, ECS може записувати неперетворювані типи блок, поки цей блок не заповниться, а потім відразу ж переходити до наступного, щільно упаковуючи кожен блок.

Роз'єднуючи дані (об'єкти та компоненти) та логіку (системи), ECS також створює більш модульний та простий для читання код.

Отже, підсумовуючи можна сказати те, що у цьому підході коли ми використовуємо системи, немає значення з якими сутностями ми будемо парцювати. Всі вони будуть оброблятися згідно з чітко визначеними правилами, незалежно від того, який тип ігрового об'єкта їх представляє.

У нас може бути стільки систем, скільки нам потрібно, і кожна з них відповідатиме лише за певну ігрову логіку та оброблятиме лише ті сутності, які відповідають заданим фільтрам компонентів. Це дозволяє нам створювати архітектуру, що складається із сильно розділених модулів і це є головною перевагою.

1.2.3 Недоліки ECS

Серед недоліків можна виділити наступні:

- значно вищий поріг входу;
- для використання Unity API необхідно прокидати їх в ECS оточення через monoBehaviour обертки;
- в середньому значно більше за код (хоча і код сам по собі, зазвичай, простіше);

- доведеться писати системи та компоненти на кожен функціонал у додатку;
- потрібно стежити за порядком виклику систем;
- можуть бути складнощі у налагодженні;
- прев'ю та нестабільність API (тимчасово).

Хоча ці зміни відкривають перераховані вище переваги, Unity в даний час розробляється з урахуванням робочого процесу, орієнтованого на GameObject та MonoBehavior. Це означає, що численні функції, які раніше працювали з коробки (фізика, аудіо та системи рендерингу), вимагають додаткових зусиль, якщо ви використовуєте тільки ECS. Так буде до тих пір, поки Unity не адаптує або не замінить ці системи для використання з ECS.

Для написання додатків з використанням архітектурного паттерну ECS, потрібен час, щоб звикнути до «data oriented design».

Порядок виклику або виконання систем дуже важливий. Наприклад, є дві системи, одна отримує дані з клавіатури для переміщення, а інша рухає гравця, якщо поставити першу після другої, гравець не рухатиме, за порядком систем потрібно завжди стежити.

Отже, підсумовуючи вищесказане можна зробити висновок, що ECS дозволяє отримати підвищення ефективності тестування, скорочення числа ітерацій і зниження витрат на усунення проблем і помилок, але використання цього патерну потребує зміни парадигми програмування, вимагає повного розуміння і стає набагато складнішим у використанні ніж звичайний класичний підхід до проектування.

1.2.4 Робочий процес, орієнтований на MonoBehaviour

Використовуючи об'єктно-орієнтоване програмування, ви створюєте «GameObject» для представлення ігрового об’єкту, а потім приєднуєте MonoBehaviour-сценарії для надання функціональності цих об’єктів, наприклад таких як контролер гравця, атрибути здоров'я або фізики. Ці MonoBehaviour компоненти містять як дані (наприклад показники життя), так і код, який модифікує ці дані (наприклад, зменшення очок життя, коли гравець отримує пошкодження).

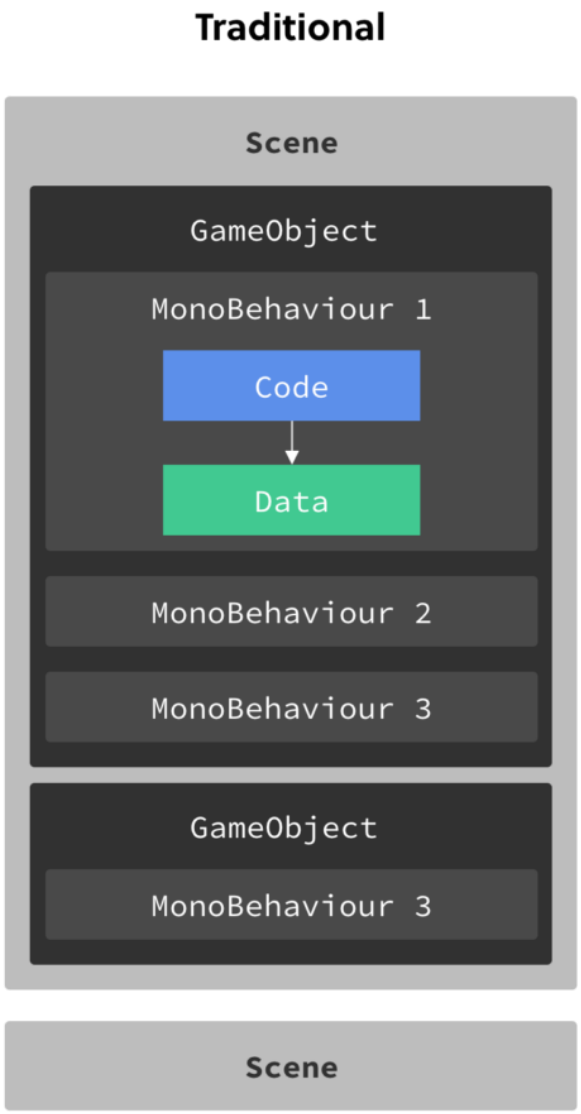


Рисунок 1.10 – Структура робочого процесу орієнтованого на monobihavior.

Основна перевага робочого процесу, орієнтованого на MonoBehaviour, полягає в тому, що ви як розробник Unity вже знаєте, як його використовувати. При використанні цього підходу ви працюєте з повністю розробленими Unity інструментами, робочим процесом та API-інтерфейсами MonoBehaviour. Ви можете використовувати такі вбудовані системи як наприклад фізику, звук та рендеринг, до яких звикли.

Головний недолік робочого процесу, орієнтованого на MonoBehaviour, полягає в тому, що створюваний ним код поступається продуктивністю ECS коду. Unity дивовижна, тому що дозволяє розробникам створювати ігри та програми за допомогою дійсно простої та зручної у використанні системи після всього лише

кількох годин навчання. Проблема з цією простотою полягає у тому, що у простих ігрових проектах ця система сяє, і ви можете дуже легко створити дійсно чудову гру, не створюючи складних систем та взаємодій, але якщо ви хочете створити проект з амбітними деталями та якістю, тоді потрібна система, яка здатна передати такі деталі у великому масштабі. По мірі розвитку проекту доводиться регулярно переробляти та покращувати його структуру. Ми можемо спостерігати поступове зростання обсягу класів MonoBehavior. Класам MonoBehaviour потрібно мати доступ до всіх даних, тому що вони виступають обгорткою для всієї логіки. Гарний підхід коли логіка відокремленна від даних, принцип полягає в тому, що ні відправник, ні одержувач не знають про існування один одного, але обидва знають про існування системи або шини повідомлень. В ECS дані та логіка розділені, це прискорює виконання завдань та зменшує залежності чого не можна сказати про monobihavior. Ще однією проблемою такого підходу є те що дуже важко відслідити залежності при тестуванні коду.

Підсумовуючи вище сказане можно зробити декілька висновків, а саме головні відмінності між підходомами до проектування на Monobihavior і ECS, в Monobihavior ми використовуємо ООП, об'єктно орієнтоване програмування і кожен об'єкт зберігає в собі свої дані і сам себе обробляє. ECS має інший підхід - «data oriented design», де ECS оперує даними, а не об’єктами, має окремі системи які відповідають за певну логіку виконання, а також вони управляють своїми відповідними сутностями окремо.

Висновок до розділу 1

В даному розділі ми зробили порівняльну характеристику існуючих фреймворків, які використовують патерн для проектування архітектури додатку Entity Component System (ECS). Розглянули які проблеми та задачі вирішує ECS. Виявили недоліки класичного підходу до проектування та переваги ECS. Порівняли швидкість роботи ECS та класичної monobihavior реалізації та зробили певні висновки.

2 ДОСЛІДЖЕННЯ ІНФОРМАЦІЙНО-ОРІЄНТОВАНИХ ТЕХНОЛОГІЙ

2.1 Технології Unity DOTS

Для використання технологій Unity DOTS спочатку розглянемо основну мотивацію для переходу від класичного проектування в Unity до підходу орієнтованого на дані, ключову термінологію та концепції, пов'язані з DOTS.

DOTS знаменує собою фундаментальну зміну напряму архітектури Unity. DOTS - це комбінація нових для Unity технологій, яка вимагає нового способу роботи з Unity та іншого підходу до роздумів про код та дані з використанням орієнтованого на дані дизайну (DoD), на відміну від об'єктно-орієнтованого підходу Unity.

DOTS дозволяє використовувати переваги багатоядерних процесорів для розпаралелювання обробки даних і підвищення продуктивності ваших проектів в Unity.

На даний час DOTS співіснує в гібридному форматі і працює разом з Unity MonoBehaviour, що не належать до DOTS. Зрештою Unity перейде на реалізацію повної DOTS.

DOTS - це новий продукт, який все ще перебуває у зародковому стані. Оскільки він перебуває у постійному розвитку, ми можемо спостерігати, як API буде розвиватися і вдосконалюватися.

На даний момент ми можемо думати про DOTS як про розширення для двигуна Unity. DOTS все ще знаходиться на стадії розброки, але ми вже можемо почати вивчати цей стек технологій та використовувати його у своїх проектах. Ми можемо створити новий проект з нуля, використовуючи DOTS (і MonoBehavior, якщо потрібно), або ми можемо перетворити елементи існуючого проекту на DOTS.

Хоча елементи DOTS та їх API зміняться, фундаментальні принципи DoD та підходи до розробки проектів із використанням DOTS залишаться незмінними.

DOTS – це технологічний стек Unity, орієнтований на дані, це є комбінацією технологій, які працюють разом, щоб забезпечити орієнтований на дані підхід до кодування в Unity. Він дозволяє створювати проекти, які краще підходять для вашого цільового обладнання, отже, більш продуктивні.

DOTS складається з наступних елементів:

1. Система компонентів сутності (ECS), що забезпечує основу для кодування з використанням підходу, орієнтованого на дані. Це розширення можна отримати через пакет Entities, який ви можете додати до редактора через диспетчер пакетів.
2. Система завдань C# (Job System), що забезпечує простий метод генерації багатопотокового коду. Це розширення можна отримати через пакет Jobs.
3. Компілятор Burst (Burst Compiler), який генерує швидкий та оптимізований власний код. Розповсюджується через пакет Burst, доступний у редакторі через диспетчер пакетів.
4. Власні контейнери, що є структурою даних ECS, що забезпечують контроль над пам'яттю.

Елементи 1–3 часто називають трьома стовпами DOTS та доступні у вигляді окремих пакетів Unity. Разом вони утворюють основу надання ефективних рішень, орієнтованих на дані.

Всі три компоненти підсистеми DOTS спрямовані на отримання максимально можливої (на даний момент у Unity) продуктивності, при цьому не обмежують розробника, що не тільки не ускладнює його життя, а й у багатьох аспектах робить його простіше.

Unity має багато додаткових систем, які вимагають оновлення для роботи з трьома стовпами DOTS, перш ніж DOTS стане приємним середовищем для роботи.

Загальний шлюз для використання DOTS - це Entity Component System (ECS). ECS - це спосіб структурування та написання коду, що дозволяє відокремити інформацію (наприклад, поведінку) від даних. Це дозволяє впорядковувати дані логічним чином, що знижує кількість промахів у кещі ЦП, отже, підвищує

швидкість доступу даних з пам'яті. У багатьох сценаріях це може дати значне підвищення продуктивності, яке було б неможливим без DOTS.

ECS дозволяє легко визначити, як дані організовані у пам'яті та як до них звертається ЦП. Це великий відхід від структури, яка використовується в Unity без DOTS. Працюючи з лінійними даними та системами, що обробляють набори даних замість окремих сценаріїв MonoBehavior, ми можемо забезпечити більш масштабовану основу на майбутнє. Іншими словами, з ECS ми переходимо від об'єктно-орієнтованого підходу до проектування, орієнтованого на дані.

Одним з плюсів DOTS є те, що він дозволяє використовувати усі можливості сучасних багатоядерних процесорів без потреби у складних програмних алгоритмах.

DOTS дозволяє творцям контенту розробляти знакові можливості, великі та маленькі, які розширюють наше розуміння того, що можливо у розвагах та симуляторах у реальному часі.

Що таке дизайн, орієнтований на дані та чим він відрізняється від об'єктно-орієнтованого дизайну. Дизайн, орієнтований на дані, наголошує на кодуванні на вирішенні проблем шляхом визначення пріоритетів та організації даних, щоб зробити доступ до них у пам'яті якомога ефективнішим. Це суперечить об'єктно-орієнтованому принципу, згідно з яким при розробці коду має керуватися модель світу, яку ви створюєте.

До Unity DOTS працювала виключно на об'єктно-орієнтованій моделі програмування. Типовий об'єктно-орієнтований робочий процес полягає в наступному:

1. Створити GameObject.
2. Додати до нього компоненти.
3. Написати сценарії MonoBehaviour, які змінюють властивості цих компонентів.

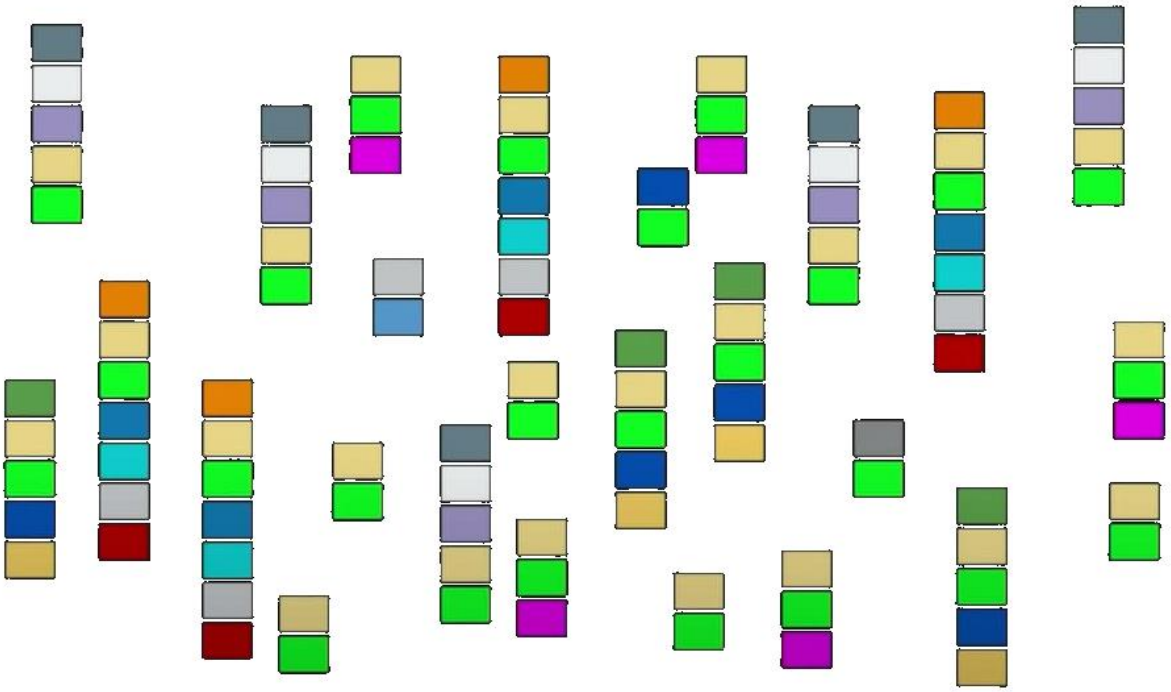


Рисунок 2.1 – Приклад класичного вигляду пам’яті у проекті.

Під час виконання GameObject залежить від посилань на компоненти. Коли сценарії MonoBehaviour шукають дані компонентів, вони розкидані по пам'яті, доступу до яких потрібен час.

При застосуванні об'єктно-орієнтованого підходу код зазвичай структурується на основі конструкції речей (об'єктів, побудованих як класи), їх складу (які об'єкти даних мають), а потім визначень того, що речі роблять і як вони взаємодіють із іншими речами. Це майже природне продовження того, як ми створюємо речі у реальному світі. Але з погляду структурування інформації, що міститься в об'єктах, цей підхід додає рівень абстракції поверх базових даних, що може спотворити організацію даних та уповільнити доступ до них. Наприклад, якщо ви хочете, щоб функція працювала з іншим фрагментом даних в об'єкті, функція повинна успадковуватися від батьківського класу, або її потрібно переписати.

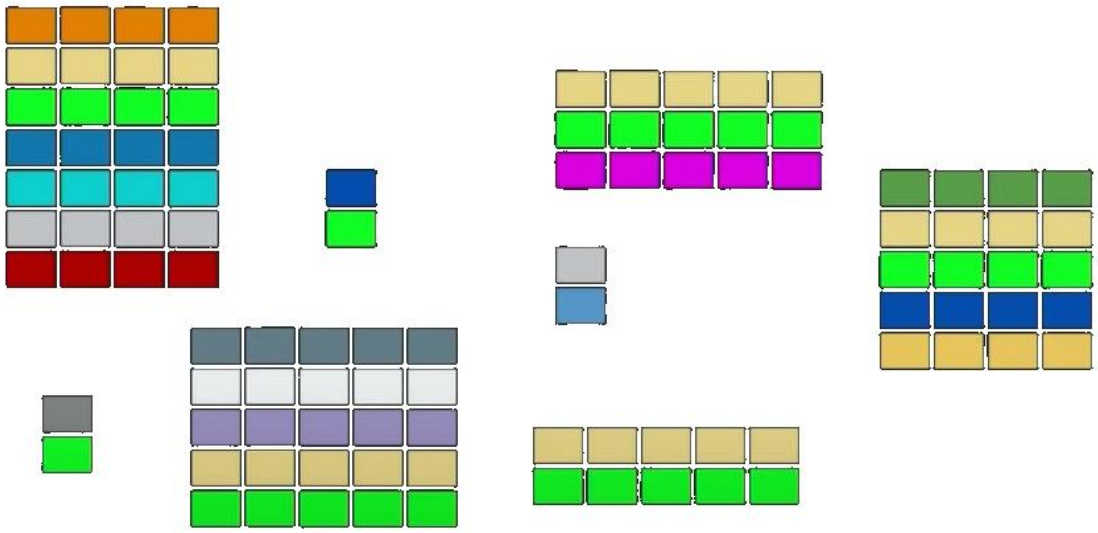


Рисунок 2.2 – Вигляд пам’яті при застосуванні ECS.

Під час використання ECS всі дані в оперативній пам'яті знаходяться на невеликих відстанях одна від одної і overhead процесора знижується в рази. При використанні класичної компонентної системи, при будь-яких операціях з об'єктами всі дані розкидаються по пам'яті, процесор витрачає більше часу на виділення, пошук або видалення інформації.

У дизайні, орієнтованому на дані, потрібно розглядати все як дані, а не як об'єкти. Це забезпечує легкий доступ до всіх даних та їх використання без обмежень ієрархічного класу.

Використання Unity ECS гарантує, що всі дані зберігаються у сховищі у лінійному режимі, що означає, що система матиме доступ до фізичних компонентів.

ECS замінює об'єктно-орієнтоване кодування орієнтованим на дані підходом, що найбільше підходить для багатьох частин ігрових додатків. Він відокремлює дані від логіки, усуваючи величезну кількість розкиданих даних.

ECS вирішує проблему неефективного розташування даних, покращує управління зберіганням даних та забезпечує високопродуктивного обчислення.

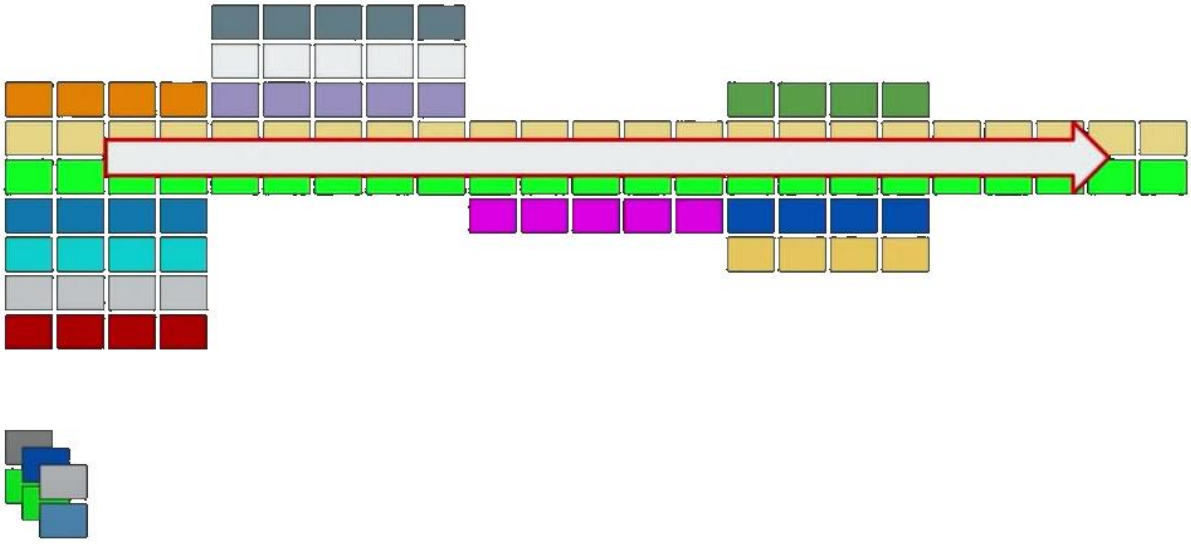


Рисунок 2.3 – Візуальне представлення шкидкої роботи ECS.

При підході, орієнтованому на дані, типовий робочий процес починається з визначення та подальшої організації даних, що лежать в основі найпоширеніших завдань, які ви хочете реалізувати. Ваші найпоширеніші проблеми – це ті, які вам потрібно буде вирішувати найчастіше під час виконання, тому їхня пріоритетність при розробці є ключовим моментом. Організація коду навколо даних і потоку цих даних означає, що доступ до них під час виконання може бути більш ефективним, ніж вибірка даних через кілька класів.

ECS дозволяє правильно зберігати дані, тому що відокремлює дані від логіки. ECS зберігає дані компонентів лінійно в пам'яті, пробігаючи по цих компонентах системою, ми використовуємо всі можливості процесора.

Для прикладу уявіть собі два об’єкта, один з них гравець інший супротивник, кожен з цих об’єктів складається з різних компонентів та мають своє посилання наприклад на положення або стан здоров’я. Сутності та компоненти – це чисті дані без додаткових функцій. Системи застосовують функції до об’єктів. Наприклад система руху до сутностей у яких є набір потрібних компонентів. Такий поділ даних і функцій дозволяє Unity створювати завдання, які можна обробляти паралельно кількох ядрах.

Рисунок 2.4 – Візуальне представлення використання ECS та Job system (паралельні обчислення).

Поєднуючи архітектурний патерн ECS та систему завдань C# від Unity, ми можемо використовувати багатоядерні обчислення та багатопоточність. Під час розробки коду з використанням багатопоточності є свої недоліки, частіше за все код обробляються безладно, що призводить до непередбачуваних результатів, накладаються певні обмеження, а постійне перемикання контексту є неефективним. Тим не менш, система завдань вирішує всі перераховані вище проблеми, дозволяючи розробникам зосередитися на ігровому коді. Завдання (Jobs) дозволяють запускати системи в кілька потоків, використовуючи різні ядра ЦП. Раніше це було дуже складно реалізувати. ECS та система завдань (Job System) дозволяють використовувати кілька ядер на всі сто відсотків, що в свою чергу збільшує продуктивність.

Завдяки ECS та Job System ми можемо написати безліч систем та сутностей, як ми вже знаємо, системам не потрібно знати з якими сутностями вони працюють, одна система може відповідати за безліч сутностей. З використанням багатопоточності ми отримуємо багато процесів які можуть виконуватися

одночасно. Job System забезпечує справжню багатопоточність, оскільки події більше не потрібно обробляти послідовно.

На сьогодні масові процесори мають до 4-6 фізичних та 8-12 логічних ядер, у той час як процесори для ентузіастів мають до 16 фізичних та 32 логічних ядра. Більшість із них не використовуються, але з використанням DOTS ми можемо повною мірою використати їх усі.

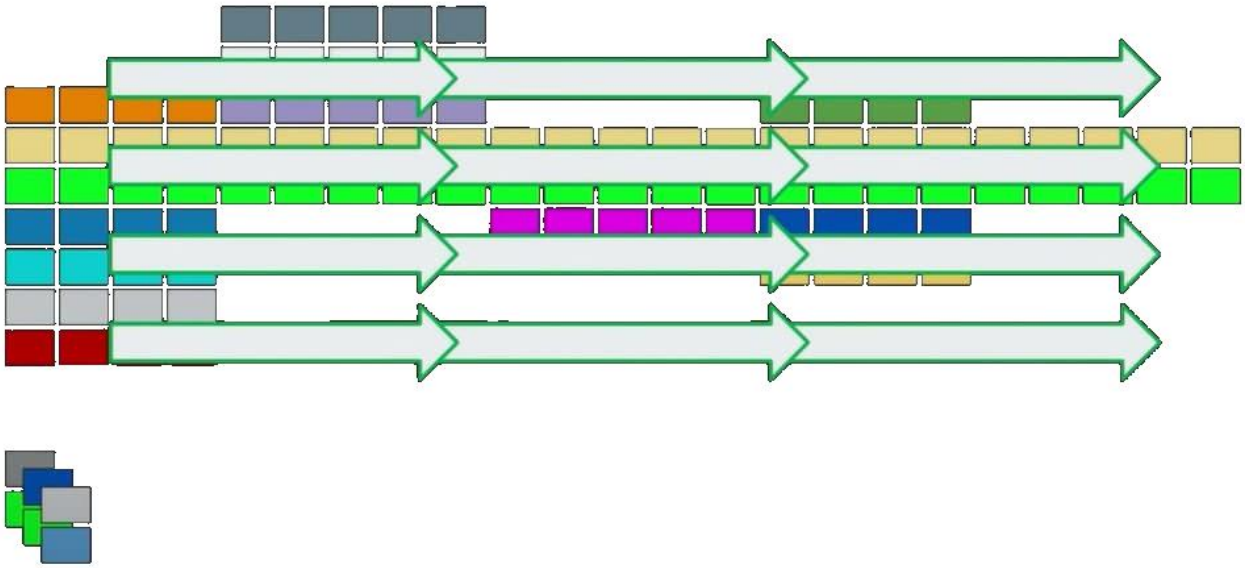


Рисунок 2.5 – Візуальне представлення використання ECS, Job system та Burst Compiler.

Новий компілятор Burst заснований на технологіях LLVM та принципах математично-орієнтованого компілятора. Він використовує завдання C# для створення оптимізованого машинного коду з урахуванням можливостей цільової платформи.

Цей компілятор працює шляхом компіляції підмножини мови програмування C#, відомого як High-Performance C# (HPC#), щоб ефективно використовувати потужність пристрою за рахунок розгортання розширених оптимізацій, створених на основі компілятора LLVM.

За допомогою Burst Compiler ми позбавляємося від необхідності у вивчені та переробці складного низькорівневого коду для усунення проблем продуктивності.

На рисунку 2.5 ми бачимо як Burst Compiler використовує завдання C# (Job System) для створення оптимізованого машинного коду з урахуванням можливостей цільової платформи.

Burst чудово підходить для використання паралельного обчислення. Він може відкрити великі переваги у підвищенні продуктивності алгоритмах, прив'язаних до ЦП.

Компілятор Burst дає переваги оптимізованого асемблерного коду для безлічі платформ без необхідності вручну переробляти його, при взаємодії великих систем компілятор дозволяє уникнути значної кількості помилок інтеграції, які зазвичай траплялися у процесі контролю якості.

Наприклад на практиці цей компілятор дозволяє прискорити обробку циклів майже у 30 разів, особливо якщо вони виконувались кілька разів на одному кадрі.

Головне питання яке може виникнути, чи підходить DOTS для мене, моєї команди або мого проекту. Якщо ми негайно не прагнемо підвищення продуктивності в короткостроковій або середньостроковій перспективі, може бути важко визначити, чи слід або коли переходити на DOTS.

DOTS швидше за все забезпечить деякий рівень підвищення продуктивності майже в кожному додатку. Сфери включають продуктивність, час автономної роботи, ітерацію та масштабованість проекту. Ми не побачимо жодного зниження продуктивності під час переходу на DOTS, але оцінка вартості переходу на DOTS має вирішальне значення, особливо для проектів, у яких досягаються лише невеликі покращення.

Для всіх програм DOTS підходить для роботи з великими обсягами даних, такими як середовища відкритого світу або складні структури, в яких використовуються великі обсяги одних і тих самих матеріалів. DOTS також підходить для елементів, що повторюються, розділяючи загальні дані між примірниками, щоб зменшити доступ до пам'яті.

Важливо враховувати, що DOTS допоможе у майбутньому розробляти високоякісний контент, який Unity без використання DOTS не зможе надати. Наприклад, стандартні ігри та проекти Unity сьогодні були іграми AAA минулого.

Unity розробникам потрібно буде прийняти DOTS, щоб залишатися конкурентоспроможним у майбутньому.

Для різних вертикалей DOTS може підходити для різних рішень:

1. Для додатків AEC. DOTS підходить для роботи з великими наборами даних та забезпечення масштабованості контенту. DOTS ідеально підходить для великих інтерактивних карт та середовищ з великою кількістю моделей та повторюваним вмістом, таким як будівлі та дороги. DOTS підходить для складних інженерних візуалізацій, які імітують реальні текстові середовища у великому масштабі. Наприклад, для DOTS ідеально підходять проектування фабрики та інфраструктури у дрібному масштабі.
2. Для автомобільного застосування. Моделювання та візуалізація для автономного водіння. DOTS ідеально підходить для моделювання великого руху та пішоходів, що потребують реалістичного руху та взаємодії тисяч агентів.
3. Для інді-розробників та фрілансерів. DOTS може допомогти вам розвантажити деякі дорогі операції у вашій грі і підвищити продуктивність, особливо для процесів, що повторюються. Багато легких ігор, наприклад, для мобільних пристроїв, не дозволяють досягти максимальної продуктивності обладнання. Навіть для тих, хто це робить, це може бути основною проблемою. Однак, оскільки ігри продовжують розвиватися та збільшувати вимоги до обладнання, розумно підготуватися до використання DOTS у майбутньому. У Unity є проект під назвою «Project Tiny», він надає рішення для розробки невеликих додатків та ігор з використанням DOTS. У тих випадках, коли у вас може не бути негайної необхідності починати роботу з DOTS, це чудова ідея - випередити криву та підвищити свої навички в DOTS, щоб ви були готові до того, коли DOTS стане стандартною практикою у розробці Unity.
4. Для ігрових студій. DOTS у його поточному форматі може допомогти почати досягати масштабування та продуктивності, яких ми не могли досягти раніше. Зокрема, ключовими перевагами є збільшений час автономної роботи

та температурний контроль, а також можливість повторного використання коду DOTS. Розширена продуктивність у цих областях також дозволяє розробляти дешевші пристрої, особливо поза західних ринків, які у іншому мають жорсткі апаратні обмеження. Якщо групи НДДКР почнуть працювати в DOTS, це допоможе вам почати розуміти, які підходи ви можете використовувати в майбутньому, і поінформує вас про поточні функції та галузі, які дадуть найбільші переваги у продуктивності та вплив на розробку. DOTS не ставить собі за мету підмінити роль груп розробників, але дає інженерам можливість вводити нововведення у власних галузях знань, таких як тіні або шейдери.

Можливості та ризики використання DOTS. DOTS надає величезний потенціал для підвищення продуктивності ваших проектів Unity. Однак, використовуючи DOTS, необхідно враховувати деякі аспекти, які вплинуть на графік, бюджет та команди розробників вашого проекту. Це те, що вам потрібно порівняти та протиставити пріоритетам вашого проекту. Ці міркування можна розділити на ризики та можливості.

Можливості:

1. Підвищена продуктивність. Ми часто використовуємо термін «За замовчуванням» для опису DOTS. Що ми маємо на увазі під цим? Завдяки орієнтованому на дані дизайну та багатопоточності DOTS може забезпечити експонентне зростання пам'яті, часу роботи та продуктивності акумулятора. Потенціал покращення продуктивності збільшується зі збільшенням кількості елементів, що відображаються у вашій грі. І навпаки, ви побачите менш значне зростання продуктивності для ігор із невеликою кількістю елементів.
2. Кодовий контроль. DOTS забезпечує найкращий контроль складності коду у міру зростання проекту. Код, написаний для DOTS, зазвичай краще розділяє завдання. Завдяки цьому спрощується рефакторинг коду, написання модульних тестів та розподіл роботи між розробниками під час роботи в DOTS.

Ризики:

1. Якщо ви не знайомі з DoD, DOTS має криву навчання . Хоча DoD добре зарекомендував себе в інформатиці і існує вже багато років, а підхід DoD значно відрізняється від підходу ООП, DoD за своєю суттю не є складнішим за ООП. Це означає, що ECS є архітектурою структурування коду, відмінною від існуючого в Unity моноповедінкового підходу, для вивчення якого знадобиться час. В даний час ми припускаємо, що середньому професійному розробнику Unity потрібно в середньому 1 місяць стати професійним фахівцем з використання DOTS. Цей час виконання може бути компенсований покращенням якості коду та продуктивності при використанні DOTS. Це залежить від проекту.
2. Обмежена підтримка. DOTS зараз сумісний лише з обмеженим набором функцій у Unity. Зрештою, DOTS буде повністю сумісним з усіма функціями Unity, але в даний час ми не маємо графіка для повної сумісності. Однак DOTS дозволяє використовувати як ігрові об'єкти, так і DOTS в одному проекті, тому можна використовувати DOTS для найчастіших завдань обробки та не-DOTS Unity для інших.

Підготовка до використання DOTS. Перехід на DOTS потребує навчання. Працювати з DOTS буде складно тільки якщо концепції для звичайного користувача Unity будуть нові. В даний час немає простого готового рішення для впровадження DOTS. Перед тим, як почати працювати з DOTS, важливо зрозуміти, що таке DOTS і визначити області, в яких він принесе необхідні переваги. Потрібно виявити, що саме нам потрібно, перетворити тільки певні аспекти проектів для використання DOTS, або розпочати нові проекти, використовуючи виключно підхід DOTS у багатьох ключових сферах.

Підготовка до DOTS включає два аспекти: вивчення дизайну, орієнтованого на дані, та вивчення того, як реалізувати DOTS за допомогою API. Перед тим, як пачати працювати з DOTS, потрібно підготуватися до ряду речей.

Для всіх: розвивайте своє розуміння підходу до кодування, орієнтованого на дані. Перехід від об'єктно-орієнтованого проектування до орієнтованого на дані -

це не те саме, що вивчення нової мови програмування або нового способу програмування. Навпаки, це зрушення в підході до кодування і представлення інформації, яку ви створюєте за допомогою коду.

Навіть для досвідчених розробників перехід до мислення, орієнтованого на дані, може бути складним завданням. Це пов'язано з тим, що методи розробки коду та способи абстрагування поведінки, яка стала другою натурою, необхідно переосмислити. При підході, орієнтованому на дані, ми змушені думати про код по-іншому, скорочуючи об'єкти, класи та поведінку до необхідних даних та ефективно організовуючи ці дані, щоб їх можна було перетворити на вирішення загальних проблем насамперед. На практиці це нетривіальне завдання, що вимагає часу, оскільки ми удосконалюємося і зосереджуємось як на розумінні даних, так і на вивченні того, як використовувати DOTS для кращої роботи з даними.

Для програмістів: якщо ви вмієте писати код, ви можете працювати з DOTS. Дизайн, орієнтований на дані, - це підхід, якщо у вас є солідний досвід програмування в Unity, вам обов'язково потрібно буде змінити не свої навички програмування, а те, як ви застосовуєте їх до підходу, орієнтованого на дані.

Є кілька ключових змін у способі написання коду в Unity, які нам потрібно буде внести, щоб досягти такого приросту продуктивності. По-перше, як ми організуємо свої дані. Надання ЦП чистих лінійних масивів даних для читання замість вилучення даних із кількох місць у пам'яті під час обчислень дозволяє значно підвищити продуктивність. Беручи активну участь у керуванні пам'яттю, ми гарантуємо, що управління пам'яттю буде оптимальним для продуктивності. В API Unity додано набір нових інструментів, які дозволяють керувати макетом даних та спосіб управління пам'яттю явним та більш докладним чином.

Вам не потрібно переписувати весь свій код, тому що ви можете переписати тільки ті області коду, які чутливі до продуктивності, при написанні коду C # так, як ви завжди це робите в Unity для інших.

DOTS – це майбутнє Unity. Архітектура Unity рухається до прийняття DOTS у всіх аспектах. Це відбувається повільно, але якщо дізнатись про DOTS зараз,

перехід буде простіше, оскільки Unity планує впровадити DOTS у всі функції двигуна.

Дизайн, орієнтований на дані (DoD) – це майбутнє індустрії 3D у реальному часі. Спостерігається зростаюча тенденція використання принципів та реалізацій DoD для вирішення проблем та розробки комплексних рішень, особливо в іграх.

Наприклад, шведська ігрова студія Far North Entertainment використовувала DOTS для вирішення своїх проблем із продуктивністю, в проекті була задача створення шутера від третьої особи за допомогою DOTS, результат був дивовижний.

Використання DoD навіть вивчається іншими ігровими двигунами, такими як Frostbite і Unreal. Люди які ппроваджують DOTS в свої проекти, таким чином використовують найкращі практики розробки AAA ігор, на даний час ця технологія стала доступною для більш широкої аудиторії.

Приклади ігор, які не стосуються Unity, в яких повідомлялося про використання DoD, це OverWatch від Blizzard та The Witcher 3 від CD Projekt Red.

Отже, ми розглянули що таке DOTS і зрозуміли, чи підходить нам такий підхід до проектування, орієнтованого на дані для поточних або майбутніх проектів. Насамперед можна сказати що Unity переходить до повної реалізації DOTS, але Unity стверджує що і далі буде підтримувати класичний робочий процес не пов'язаний з DOTS, до тих пір поки їхні користувачі будуть потребувати цього. Це дає достатньо часу для підготовки до DOTS, оскільки зрештою вони очікують, що всім користувачам в якийсь момент потрібно буде перейти на DOTS.

2.2 Архітектурний патерн проектування Entity Component System

Entity Component System - це архітектурний патерн, на чолі якого стоять принципи композиції, а не спадкування. І хоча за назвою ця парадигма дуже схожа на стандартну парадигму Unity (Component System), концептуально вона досить сильно від неї відрізняється. ECS в Unity, написана з урахуванням data-oriented design. Звідси пішла назва DOTS, це є однією з причин високої продуктивності.

Незважаючи на плюси, які пропонує ECS, є і свої складнощі, а саме: необхідність перебудови мислення (уникнення класичного ООП та компонентно-орієнтованого програмування Unity). І хоча в цьому нічого особливо складного немає, але якийсь час це все ж таки займе.

Розробка під ECS значно відрізняється від звичного в Unity компонентного підходу. На відміну від компонента поведінки, який містить дані і логіку роботи з цими даними, ECS дані і логіка суворо розділені. ECS складається з трьох основних частин: сутностей (Entity), компонентів (Component) та систем (System). Цей підхід дає змогу реалізувати декомпозицію коду, тобто явно відокремити дані від поведінки, що (теоретично) сприятливо позначається на якості коду, швидкості розробки, продуктивності (саме собою) і перевикористовуваності коду.

Сутності чимось схожі на стандартні ігрові об'єкти в Unity - по суті вони є об'єктами-контейнерами для компонентів. На кожну суть можна навішувати різні компоненти. Причому, на відміну ігрового об'єкта, навісити кілька однотипних компонентів однією сутність не можна. Зазвичай це велика проблема, т.к. можна створити стільки сутностей, скільки потрібно, і навісити за потрібним компонентом на кожну з них. Компоненти є блоками даних, і лише даних. Саме сукупності цих даних і визначають, що собою представлятиме конкретна сутність.

Системи - це класи які потрібні для реалізації певної поведінки. Системи отримують на вхід список сутностей для обробки та виконують з ними деякі дії. Системи можуть обробляти як всі сутності, так і ті, що містять певні компоненти (що найчастіше і буває). Чітке відокремлення даних від логіки виконання має свої переваги. А саме: можна досить легко змінювати логіку виконання (прямо на льоту) шляхом додавання або видалення систем, зміни порядку виконання або тимчасового відключення систем, не ламаючи дані.

Оскільки самі дані відокремлені та представлені у вигляді компонентів, то з ними легко проводити деякі маніпуляції, такі як серіалізація та десеріалізація, пересилання через мережу. Це ускладнює роботу з ними, т.к. системам не має значення, звідки дані взялися (згенеровані іншою системою, прочитані з диска, передані по мережі). Усі вони обробляться однаково та прозоро.

ECS – це абревіатура трьох окремих слів, трьох сутностей, які представляють роботу всієї архітектури, зокрема у нас є Entity, кажучи простою мовою він є деякою сутністю, яка існує у світі гри. Якщо проводити аналогію з Unity, то це можна назвати як GameObject, але в ECS сутність це звичайний вказівник на набір деяких компонентів, які зв'язуються до якоїсь конкретної сутності, а компоненти це набір даних. Наприклад, гравітація містить у собі прискорення, позиція буде містити у собі координати x та y. Вся логіка буде проходити в іншому місці, в системах. Система, говорячи простою мовою, це клас який виконує виключно одне завдання, але виконує її особливим чином, для початку вона звертається до іншого простору, в якому зберігаються всі нащі сутності і каже нам наприклад видати всі сутності які містять у собі компонент гравітації. Можна провести аналогію з SQL базами даних, коли ми формуємо деякий запит і беремо набір потрібних нам даних і вони отримують їх назад. Після цього, система отримує набір потрібних для нас сутностей (Entity), проводить різні математичні операції які цікавлять конкретну систему, наприклад зміщення по позиції у гравітації і після цього передає вже інщій системі керування. Таким чином весь ігровий цикл зав'язаний на тому, що ми передаємо управління від одної системи до іншої, поступово передаючи їй управління. Ми можемо чітко простежити зверху вниз як обробляються нащі дані. Системи між собою можуть зовсім не взаємодіяти один з одним і не знати один одного. Вони можуть спілкуватися один з одним за допомогою різних івентів, або компонентів, які передаються в загальний контейнер нашого світу, завдяки цьому вони можуть спілкуватись один з одним. В різних реалізаціях ECS існують свої тонкощі та нюанси. Як ми же з'ясували, головною перевагою ECS є низька зв'язаність коду, тобто коли системи не знають один про одного і можуть просто брати та відключати, якщо цього потребує логіка гри. Наприклад нас цікавить система яка буде вбивати гравця, коли в нього закінчується здоров'я, ми можемо взяти її та вимкнути, гра буде продовжувати працювати без цієї системи, можна важати що це такий собі чит код. Таким чином, ми можемо підключати та відключати різні ігрові механіки. Якщо система працює, вона реалізує конкретну ігрову механіку, її можна як увімкнути, так і вимкнути. При правильному підході

до розробки архітектури можна абсолютно безпечно вимикати любу потрібну систему.

Друга достатньо важлива перевага, саме чому ECS дуже часто використовують у розробці ігор, це можливість комбінування. Якщо говорити простою мовою, у класичному поданні ООП, при написанні конкретної гри, ми будемо наслідувати один клас за іншим. При нарощуванні дерава з класів ми можемо зіткнутись з проблемами, коли структура наслідування нас не зовсім влаштовує. В Entity Component System достатньо накинути потрібний обробник логіки та компонент на об’єкт, таким чином геймдизайнери можуть отримати велику варіативність та можливість комбінувати любі потрібні нам сутності.

Отже, якщо провести підсумок, у ECS можна назвати такі переваги як:

1. Низька зв’язаність коду (коли ми можемо вмикати та вимикати різні системи).
2. Простіше писати код для розподілених обчислень (ми можемо змусити розпаралелити кожну окрему систему на обчислення окремих великих груп сутностей).
3. Можливість комбінування властивостей сутностей (для створення не передбачуваного результату з точки зору геймдизайну).

Використання технологій Unity DOTS у купі з ECS потребує переходу від об'єктно-орієнтованого підходу до інформаційно-орієнтованого.

ECS в Unity дозволяє використовувати систему задач C# (Job System) та Burst Compiler, розкриваючи весь потенціал сучасних багатоядерних процесорів та їх кеша.

Інформаційно-орієнтований підхід означає великі можливості щодо повторного використання коду та полегшення його розуміння та доповнення іншими членами команди, це дозволяє швидко та зручно масштабувати різного типу проекти.

При підході, орієнтованому на дані, структури даних організовані таким чином, щоб уникнути промахів у кеші, які згодом роблять доступ до ваших даних більш ефективним та швидким. Оскільки об'єктно-орієнтований дизайн не

орієнтований на цю організацію даних, промахи в кеші є звичайним явищем і це уповільнює доступ ЦП до даних, оскільки йому доводиться частіше повертатися до доступу до даних в основній пам'яті.

У поточному робочому процесі Unity ми спочатку створюємо об’єкт, додаємо до нього компонент, щоб забезпечити необхідними властивостями, наприклад фізикою з навколишнім середовищем чи іншими властивостями, створюємо сценарії MonoBehaviour та додаємо їх до об'єктів для того щоб контролювати зміни стану цих компонентів під час виконання.

Ми називаємо це класичним робочим процесом Unity. Цей підхід має деякі вагомі недоліки які пов’язані з продуктивністю. По-перше, дані та обробка тісно пов'язані, це означає що повторне використання коду буде відбуватися рідше, оскільки обробка інформації пов'язана з набором даних. На додаток до цього типові системи дуже залежать від типів посилань.

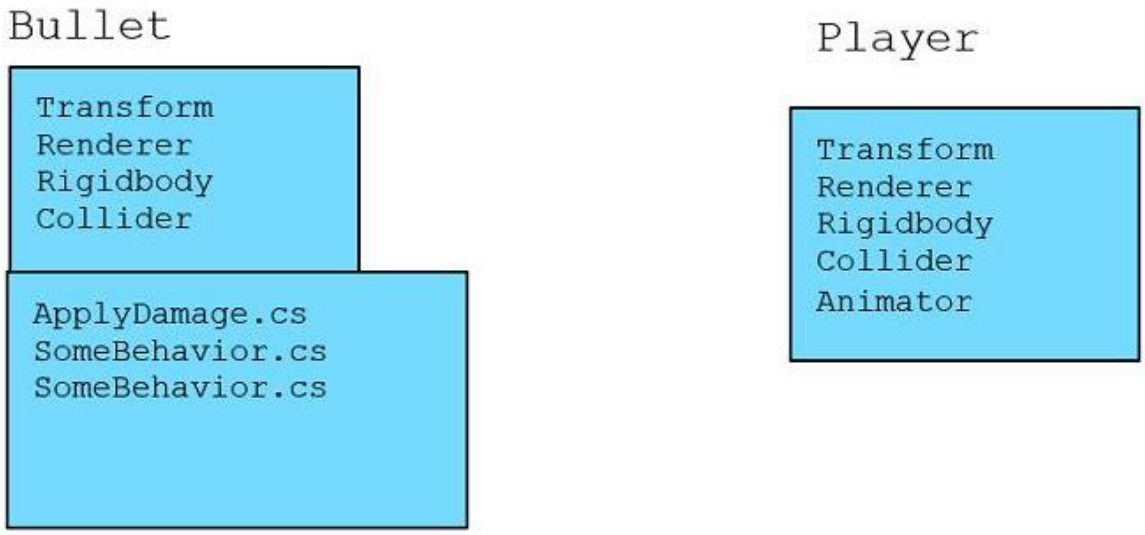


Рисунок 2.6 – Список класичних ігрових об'єктів та компонентів.

У наведеному вище прикладі класичного обєкту і його компонентів покладається посилання Transform, Renderer, Rigidbody та Collider. Об'єкти, на які посилаються ці скрипти, критичні до продуктивності, розкидані в купі пам'яті.

Отже, дані не перетворюються на форму, якою може маніпулювати швидший векторний блок SIMD.

Збільшення швидкості із попередньою вибіркою з кешу. Доступ до даних із системної пам'яті набагато повільніший, ніж вибірка даних із сусідніх кешів. Це де попередня вибірка входить у гру. Попередня вибірка кеша відноситься до апаратного забезпечення комп'ютера, яке прогнозує, до яких даних звертатися далі, і потім превентивно витягує їх з повільнішої початкової пам'яті в швидшу пам'ять, щоб її можна було нагрівати та готувати за необхідності. Використовуючи цю функцію, апаратне забезпечення може збільшити продуктивність інтелектуальних обчислень. При виконанні ітерації масиву апаратний блок попередньої вибірки може навчитися видобувати великі обсяги даних із системної пам'яті в кеш. Коли процесор працює на наступній частині масиву, необхідні дані знаходяться у кеші та готові. Для щільно упакованих безперервних даних, як у масиві, апаратний попередній вибірник може легко передбачити та отримати правильний об'єкт. Коли багато різних ігрових об'єктів рідко виділяються в купі пам'яті, засіб попередньої вибірки не може виконати завдання, змушуючи його отримувати марні дані.

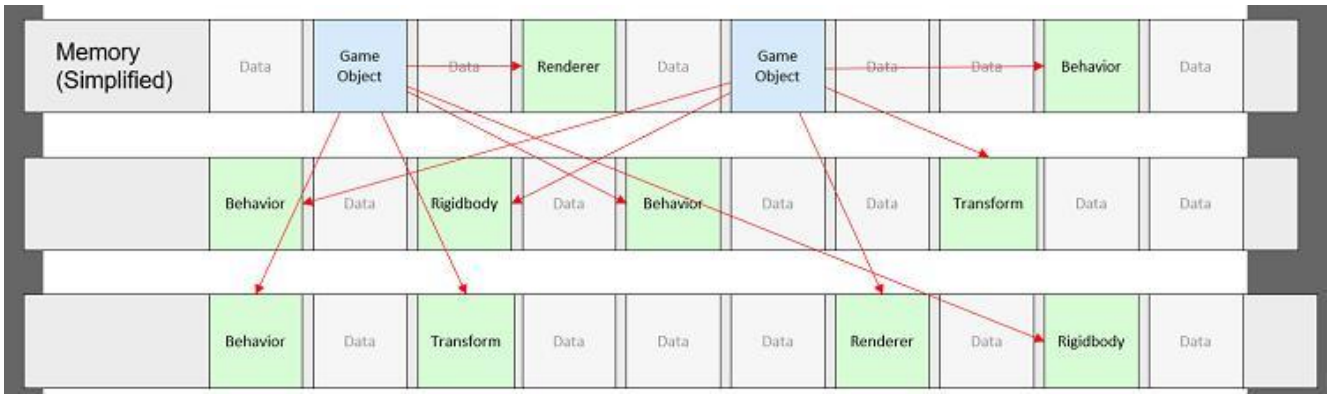


Рисунок 2.7 – Розкидані посилання у пам'яті між ігровими об'єктами, їхньою поведінкою та їх компонентами.

На рисунку 2.7 показано випадкову природу цього методу зберігання даних. У сценарії, показаному вище, кожне посилання (стрілка), навіть якщо вона кешується як змінна-член, може бути повністю вилучена із системної пам'яті.

Класична сцена Unity GameObject дозволяє створити прототип вашої гри та запустити її в дуже короткий час, але вона не є ідеальною для моделювання продуктивності та ігор. Щоб поглибити цю проблему, кожен тип посилань містить багато додаткових даних, до яких може не знадобитися доступ. Ці елементи, що не використовуються, також займають цінний простір в кеші процесора. Якщо нам потрібно тільки вибрати невелику кількість змінних-членів існуючого компонента, ми можемо розглядати інше як марну трату, як показано на діаграмі «Непотрібний простір» нижче:

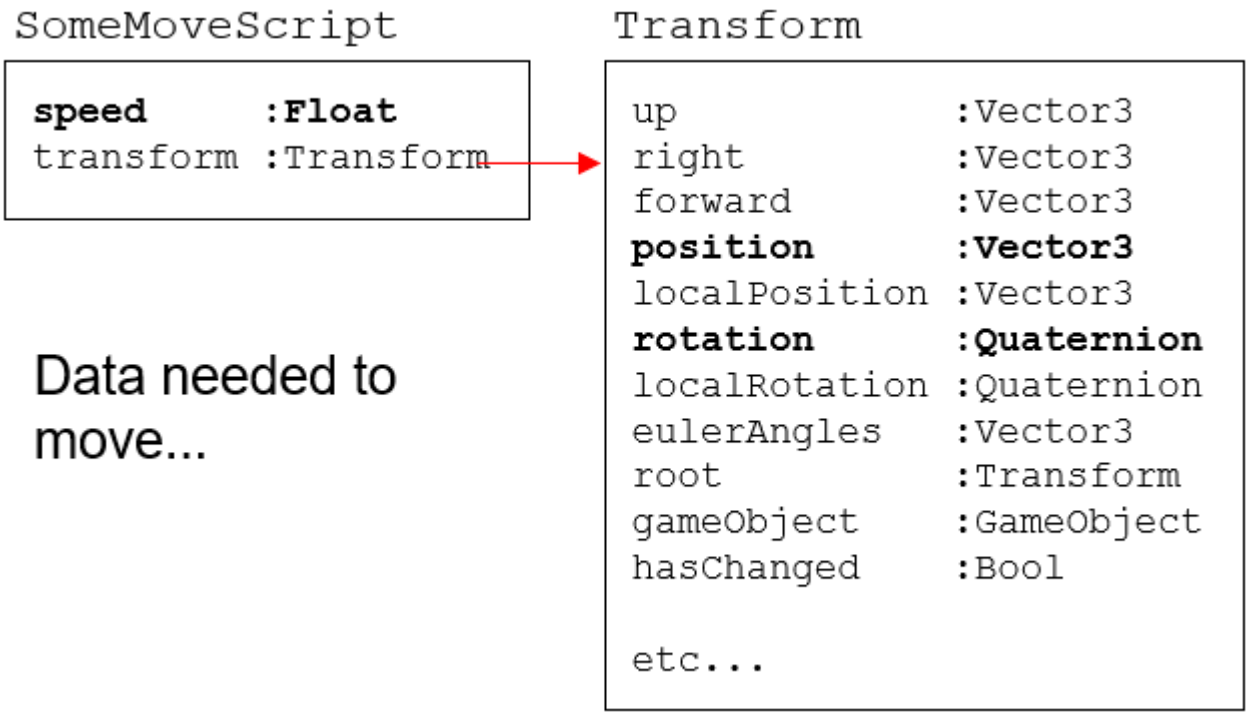


Рисунок 2.8 – Жирним шрифтом виділено елементи, що фактично використовуються для мобільних операцій, решта - марна трата місця в пам’яті.

Сценарій повинен перемістити Transform. Компоненти отримують доступ до даних положення та обертання. Коли обладнання отримує дані з пам'яті, рядок кешу заповнюється потенційно непотрібними даними. Якщо нам потрібно буде переместити тільки GameObjects хіба це не добре для створення масиву тільки з

елементами позиції та обертання? Це дозволить виконувати звичайні операції за короткий термін.

Ведденя системи фізичних компонентів. Нова система компонентів сутності Unity допоможе усунути неефективні посилання на об'єкти. Давайте розглянемо об'єкт, що містить лише ті дані, які йому потрібні, а не розглянемо об'єкт із власною колекцією компонентів об’єкта.

Потрібно звернути увагу, що в наступній системі компонентів, до сутності Bullet не прикріплен компонент Transform або Rigidbody. Пульовані об'єкти - це необроблені дані, необхідні для явного запуску процедури оновлення. За допомогою цієї нової системи можна повністю відокремити обробку від кожного типу об'єкта, та позбавитися від не потрібних компонентів.

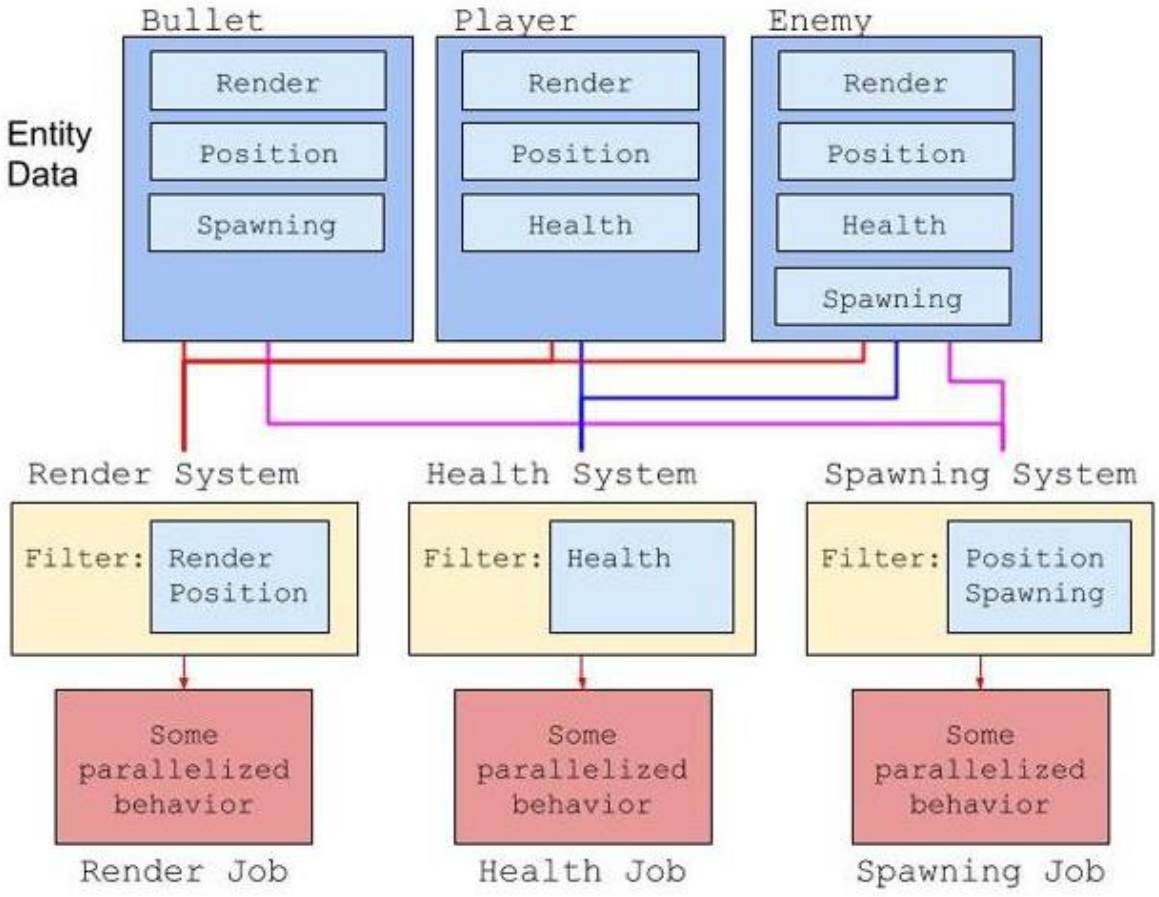


Рисунок 2.9 – Система компонентів об'єкта зі схемою операцій.

Звичайно, це не тільки система руху, яка приносить користь. Іншим поширеним компонентом у багатьох іграх є складніша система здоров'я, побудована серед різних ворогів та союзників. Для різних типів об'єктів ці системи зазвичай практично не відрізняються, тому вони є ще однією відмінною альтернативою для використання переваг нової системи. Сутність - це дескриптор, використовуваний для індексації своєї колекції різних типів даних (прототип ComponentDataGroups).

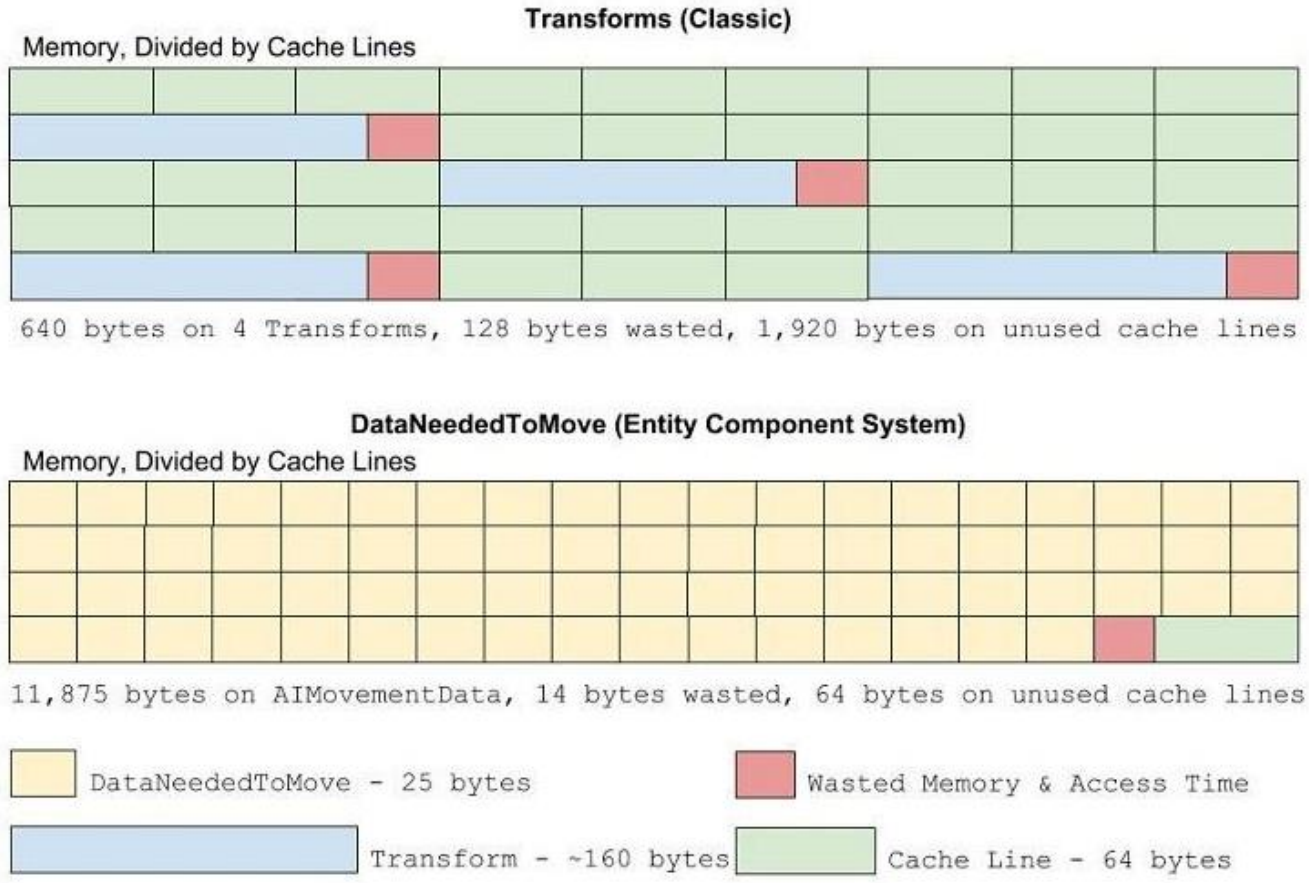


Рисунок 2.10 – Фрагментація при зберіганні рядків кешу та втрати простору, створюваного типовими системами.

Якщо порівняти ці дві операції, можна помітити, що під час застосування класичної архітектури з використанням Monobihavior класів, втрачений час доступу до пам’яті є більшим.

На рисунку 2.10 ми можемо помітити що ECS групує дані які однакові за розміром, таким чином ми можемо набагато швидше виконувати пошук для резервування місця або використання даних, які зберігаються блоками однакових за розміром частин компонентів, які можна перебирати лінійно. Замість того, щоб мати справу з об'єктами різних розмірів, що випадково зберігаються, CPU має справу тільки з тими частинами, які однакового розміру і лінійно зберігаються в одній і тій же області пам'яті. Це полегшує роботу процесора та робить пошук набагато швидшим.

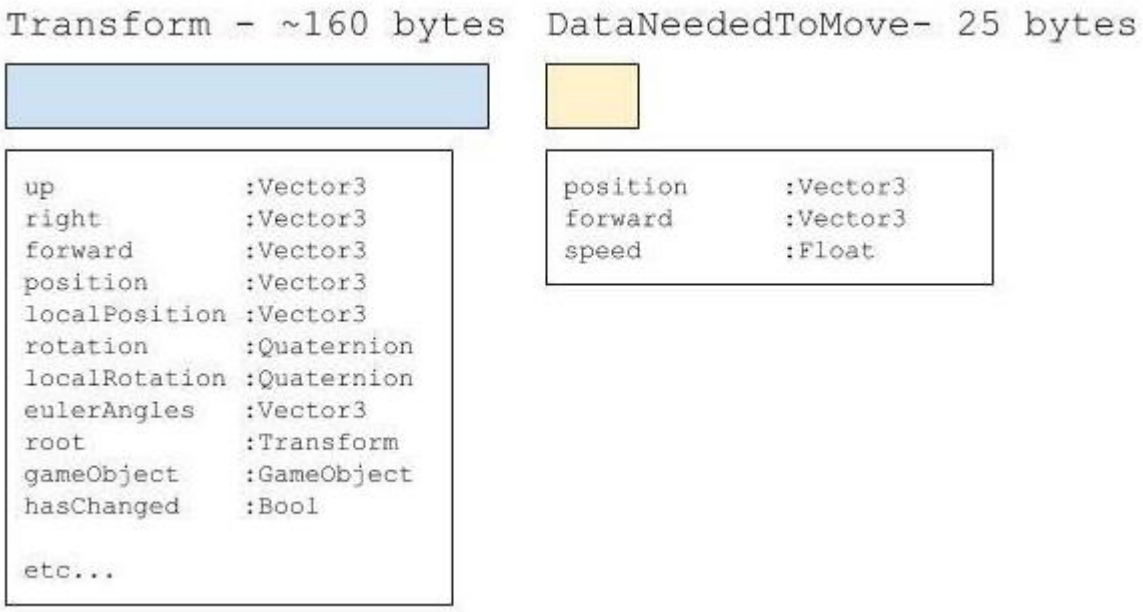


Рисунок 2.11 – Даталі операції переміщення.

Отже, порівнюючи простір пам'яті, який пов'язаний з двома операціями переміщення, які досягають однієї і тієї ж мети з використанням ECS та Monobihavior підходами, ми можемо отримати наступні результати. Класичний Monobihavior підхід в порівнянні з ECS при мереміщені об’єктів резервує набагато більше не потрібної пам’яті, а це означає що буде більше використано оперативної пам’ять ніж це потрібно.

У масштабних проектах, де використовуються дуже багато об’єктів, це може бути дуже небезпечним аспектом, який дасть замітного зниження продуктивності.

Окрім того що в пам’яті знаходится не використаний кеш, затрачений час на пошук об’єктів буде набагато більшим, саме тому що об’єкти в пам’яті будуть розкидані випадковим чином.

2.2.1 Інструмент налагодження Entity Debugger

Entity Debuger вирішує головну проблему коли ми в перший раз починаємо використовувати ECS. ECS виробляє дані і зберігає їх у зовсім іншому місці від звичайного Unity. Зазвичай при розробці гри у вас є об’єкти, компоненти та всі інщі речі. ECS, якщо можна так сказати із зовсім іншого світу, за стандартом редактор Unity не зможе показати нічого з цього світу, це означає що, якщо ми не налаштували все правильно ми не зможемо перевірити, так як у нас нічого не буде відображається у вікні для відображення гри або в ієрархії. Саме тому це не дуже гарний спосіб для розробки додатків.

Ще одна проблема пов’язана з ECS полягаэ в тому, що за стандартом, сутності не містять ніякої інформації. У редакторі ви можете побачити безліч інформації про компоненти об’єкта які на ньому знаходятся, але насправді це і є основною проблемою об’єктів monobihavior, вони дуже роздуті. У звичаного об’єкта monobihavior є ім’я, слої, теги, а також є transform об’єкта, його координати, кут повороту, або нахилу і т.д.. Ці всі речі при розробці високопродуктивного додатку зовсім не потрібні, саме тому сутності (Entity) являють собою звичайні індекси і за стандартом в них не зберігається ніякі дані. Насамперед хочу сказати що знаючи це, дуже важко інтегрувати сутності та компоненти в редактор Unity, для цього потрібні будуть будівельні ліса та гарантія того, що конкретна інформація буде доступною та сутності будуть правильно відображатися в редакторі.

Таким чином нам потрібен спосіб відображення даних ECS, як ми вже зрозуміли при використанні ECS ми не зможемо побачити об'єкт у вікні Game View або Hierarchy редактора Unity. Це дуже незручна ситуація для тих, хто раніше створював GameObjects у редакторі Unity та працював, перевіряючи інформацію візуально. За допомогою інформаційно-орієнтованого стеку технологій Unity

DOTS, який містить у собі інструмент під назвою Entity Debuger ми зможемо побачити Entity у вікні Entity Debugger.

Після завантаження технологій DOTS через package manager, потрібно відкрити його зверху у вікні. Якщо ви знайдете Entity Debugger у вікні редактора Unity і відкриєте його, з'явиться наступний екран.

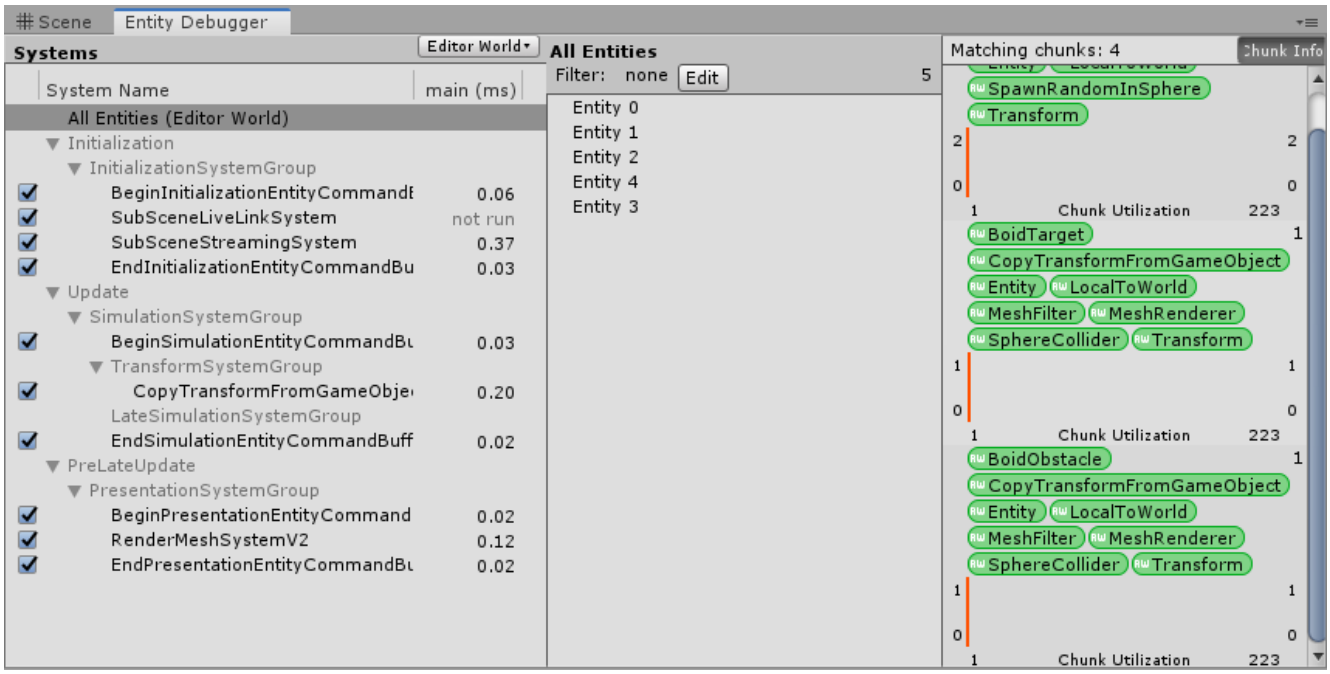


Рисунок 2.12 – Вікно Entity Debuger.

Entity Debugger дозволяє візуалізувати ваші сутності, системи та компоненти. На вкладці «система» зліва відображається список усіх систем, що працюють у грі. Список називається сортуванням Player Loop, а не сортуванням за абеткою. Кожну систему можна налагодити, вимкнувши та ввімкнувши її за допомогою прапорця увімкнення/вимкнення. В основному поле (мс) у правій частині вкладки «система» відображається час виконання кадру кожної системи в основному потоці.

Якщо ми оберемо конкретну систему, докладна інформація про неї відображається праворуч.

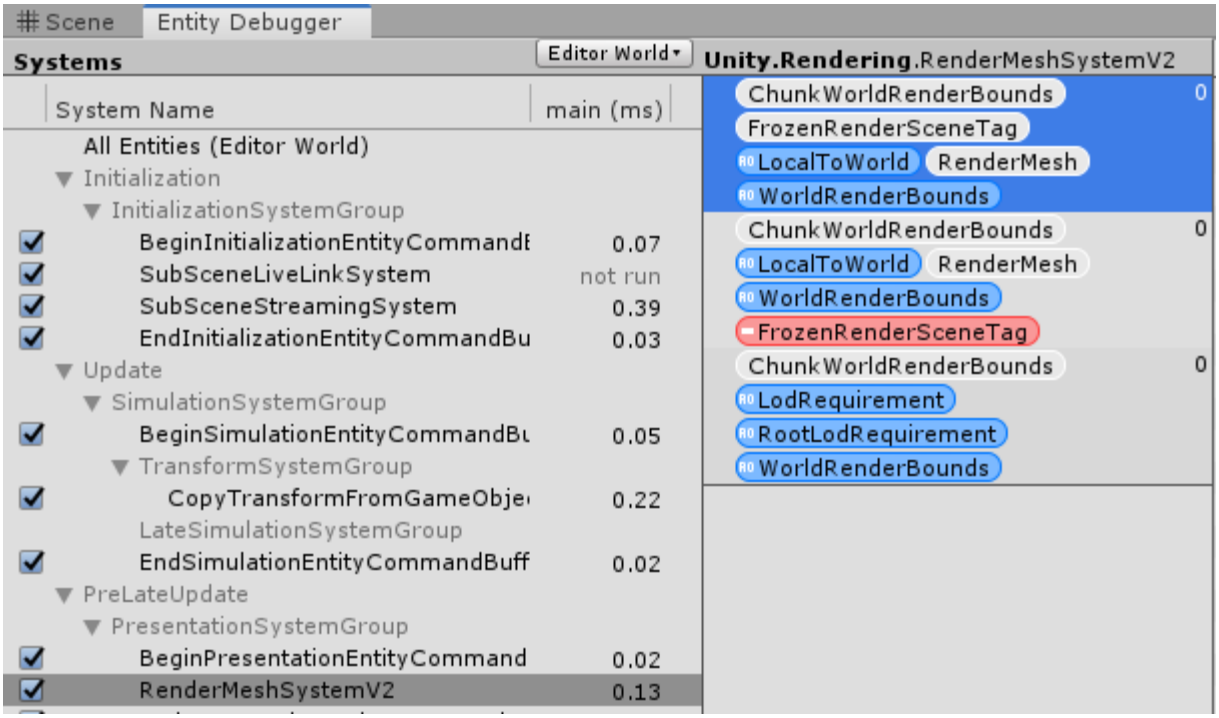


Рисунок 2.13 – Відображення систем у Entity Debuger.

Кожен елемент означає запит (називається групою компонентів) для системи отримання даних або виконання будь-якої операції. На малюнку вище до системи RenderMeshSystemV2 прикріплено три запити. Компоненти, що становлять один елемент групи компонентів, можуть легко ідентифікувати властивості за кольором. Білий колір – це властивість ReadWrite, синій колір – тільки для читання, а червоний колір – це властивість Subtractive, властивість, яка може виключати небажані компоненти.

Якщо ми обираємо певну групу компонентів, використання пам'яті фрагментом відображається у вікні інформації про фрагмент.

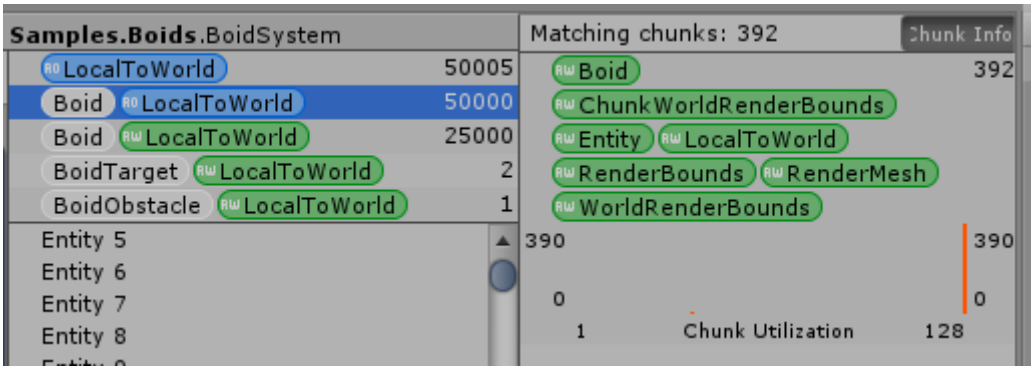


Рисунок 2.14 – Відображення фрагменту.

Знизу вікна Chunk Info відображається гістограма у вигляді червоної стовпчастої діаграми. Гістограма показує, як добре ми використовуємо свої фрагменти. Сенс рисунка вище полягає у наступному. 50000 об'єктів зберігаються в 392 блоках, що означає, що 390 блоків заповнені. 128 означає максимальну кількість об'єктів, які можуть бути включені до кожного блоку. Це досить бездоганне використання пам'яті. Якщо блоки заповнені так, ми можемо правильно побачити ефект підвищення продуктивності від ECS.

2.2.2 Інструмент конвертації об’єктів Unity DOTS Conversion Workflow

В Unity DOTS є дуже зручний інструмент під назвою Conversion Workflow, який одним натиском перетворює ігрові об'єкти на сутності, дозволяючи вам працювати з DOTS та використовувати звичайні ігрові об’єкти (GameObject).

DOTS Conversion workflow – спрощує процес створення сутностей. Для того щоб провести конвертацію об’єкта у стуність під час виконання, треба додати до ігрового об’єкта (GameObject) компонент (сценарій), який називається «Convert to Entity». У цього сценарія є два режими конвертації. Перший варіант використання «Convert and destroy», конвертація об’єкту відбувається автоматично, вона знищує ігровий об’єкт, який був до конвертації та створює сутність зі всіма компонентами.

Таким чином ми зможемо навіть використовувати цей обєкт в якості префабу, наприклад для створення більшої кількості сутностей.

Якщо подивитися на сутність яку ми конвертували у Entity Debuger, в результаті перетворення ми можемо побачити що архітектура ігрового об’єкта майже така сама, за виключенням того, що у нас є додатковий тег типу даних відбраковування для кожного екземпляру.

Насправді, це набагато простіший спосіб створення сутностей (Entities), наприклад, якщо ви створюєте навколишне середовище, ви би могли додати до кожного обєкту в ієрархії сценарій «convert to entity» і далі працювати, як ви це звичайно і робите з ігровими об’єктами. Кожен об’єкт автоматично перетвориться у сутність, під час виконання, потім виходичи з режиму «Play mode» всі ващі сутності знову стають ігровими об’єктами, з якими так легко працювати. Це

безпрограшний варіант, коли у вас є інтерактивність ігрових об'єктів у редакторі та продуктивність під час виконання.

Також, у цього сценарія є інший варіант використання «Convert and inject game object». Це означає що ви отримаєте в режимі «play mode» як сутність, в Entity Debugger, так і ігровий об'єкт (GameObject) в ієрархії. Це користно в деяких випадках, але так як Unity DOTS знаходится ще на стадії розробки, деяких фунцій ще немає і вам може знадобитися ігровий об'єкт для роботи з частиною вашого додатку та сутностями.

Після конвертації та створення сутностей ми можемо спостерігати що під час виконання в ієрархії проекту немає жодного ігрового об'єкта. За допомогою Entity Debuger ми можемо подивитися відображення кожної сутності, яку ми створили, а також всіх її компонентів.

Перетворення на сутності дуже добре підходить для будь-яких об'єктів, які можна заздалегіть налаштувати. Використовуючи Unity DOTS, вам в будь-якому випадку знадобиться якийсь механізм для конвертації сутностей під час виконання, з використанням цього робочого процесу перетворення, ви дуже легко зможете це зробити.

Сутності можна створювати відразу у скрипті. Це вам може знадобиться, наприклад під час створення механіки гри, де потрібно стріляти кулями зі зброї або генерації ворогів у середені ігрового процесу.

Конвертація сутностей працює в декілька етапів. Уявімо що об'єкти які знаходятся в ієрахії - це класиччний світ. Все що конвертується в сутності відправляється в проміжний світ, в якому двигун unity дивиться на кожен ігровий об'єкт та розбиває його на дані, він намагається створити сутність та передати їй ті компоненти з яких був складений ігровий об'єкт. Після цього конвертуючого процесу, ігровий об'єкт видаляється з ієрархії, а сутність з'являється у світі за своїм призначенням. Всі сутності ми можемо побачити у вікні Entity Debuger.

Конвертація ігрового об'єкта через код виконується за допомогою статичного методу «GameObjectConversionUtility.ConvertGameObjectHierarchy()». Всі його дочірні елементи також будуть конвертовані. Цей метод приймає два

аргумента, в перший ми передаємо префаб на ігровий об’єкт, в другий налаштування конвертації. Для того щоб отримати та передати налаштування, потрібно використати статичний метод «GameObjectConversionSettings.FromWorld(defultWorld, null)». У результаті, ми можемо повторно використовувати сутність яку ми створили з префабу ігрового об’єкта для створення езмепляру.

2.3 Система завдань Job System

Разом з новим пакетним компілятором Burst та Entity Component System (ECS) система завдань утворює високопродуктивну багатопотокову систему, яка дозволить іграм повністю використовувати доступні на сьогодні багатоядерні процесори.

Система завдань C# (Job System) - дозволить симуляції гри використовувати усі доступні ядра ЦП. Майже всі сучасні процесори мають кілька ядер, і ця тенденція посилюється. Тим не менш, багато ігор та програм використовують тільки одне ядро. Коли ми поділяємо обробку даних на кілька менших фрагментів і запускаєте їх на кількох ядрах, ми можемо обробляти одночасно паралельно, а не один за одним. Це ефективніше використовує ємність ядер і, отже, значно покращує продуктивність. Або, якщо бути більш конкретним, використання всіх доступних ядер змушує процес моделювання використовувати менше часу на стіні (час від початку моделювання до його завершення) без оптимізації часу потоку (кількість інструкцій ЦП, витрачених на обчислення результату).

Найпростіший спосіб скоротити витрачений час на обробку певного завдання, за допомогою системи завдань - використовувати завдання ParallelFor. Завдання ParallelFor використовується при обробці великого набору значень. По суті, система завдань обробляє кожен елемент у масиві індивідуально за допомогою завдання - це означає, що ці елементи можуть оброблятися паралельно одне одному з допомогою кількох ядер ЦП, якщо вони доступні. На практиці кількість завдань насправді набагато менша, ніж одне завдання на елемент у масиві,

на кожне ядро ЦП припадає одне завдання, і кожне з них отримує парну кількість елементів для обробки. Оскільки деякі робітники завершують свою роботу швидше, ніж інші, ми використовуємо так звану крадіжку роботи, щоб зрівняти час, витрачений на кожне ядро. Коли працівник завершує свою роботу, він переглядає чергу інших працівників і намагається обробити деякі елементи, призначені іншому виконавцю.

Якщо у нас є дуже важкі системи, що містять багато схожих елементів, ParallelFor чудово підійде. Але навіть якщо у нас є кілька речей кожного типу, ми можемо скористатися системою завдань. На високому рівні структура системи завдань у тому, щоб розділити всі додатки завдання на невеликі автономні одиниці роботи. Кожне ядро ЦП має свій власний потік, який виконує ці завдання, завдяки чому всі завдання виконуються паралельно один до одного.

Отже, поки різні елементи не залежать один від одного, все, що нам потрібно зробити, це запланувати для них завдання, не чекаючи на інші завдання, і вони будуть виконуватися паралельно з іншими завданнями.

Коли ми говоримо про систему завдань, часто пропонуємо концепцію раннього планування і пізнього очікування. Ціль цього шаблону - переконатися, що основний потік не повинен чекати завершення завдання. На той час, коли основним потоком будуть потрібні результати завдання, в ідеалі він має вже завершити виконання. Дуже поширене питання, на яке немає простої відповіді: який етап оновлення є «раннім» та «пізнім»? Коли ми говоримо «планувати раніше» та «чекати пізно», ми маємо на увазі, що повинні дати роботі якнайбільше часу для виконання. Не має великого значення, в якій частині кадру ми плануємо і чекаємо, якщо вони знаходяться якнайдалі один від одного. Якщо затримка одного кадру прийнятна, ми можемо дочекатися виконання завдання в наступному кадрі. Щоразу, коли ми бачимо "очікування" в основному потоці у profiler.

Система завдань не призначена для вирішення завдань, що довго виконуються з низьким пріоритетом, і вона не призначена для операцій, що очікують замість використання ресурсів ЦП, таких як введення-виведення. Це все

ще можливо, але це не є основною метою системи завдань, а це означає, що вони мають деякі обмеження, про які нам потрібно знати.

Кожен робочий потік у JobSystem прив'язаний до фізичного чи віртуального ядра ЦП. Як тільки один із цих потоків почне виконання завдання, завдання буде виконане до кінця без будь-яких перерв. Якщо ми хочемо поділитися ядром процесора з чимось ще, нам потрібно вручну поступитися, і єдиний спосіб зробити це – розділити нашу роботу на два завдання із залежностями між ними. Оскільки система ніколи не перемикає контекст за нас, завдання буде займати одне повне ядро ЦП, навіть якщо ми насправді не робимо нічого важливого.

Використання системи завдань C# має безліч наслідків, і взагалі, цей підхід має призвести до підвищення продуктивності в усіх напрямках. Це особливо вірно, оскільки в гру вступають нові функції Unity, такі як Entity Component System та технологія компілятора Burst. Система Entity Component System спрямована на скорочення часу потоку, необхідного для обчислення результату, за рахунок організації даних дуже зручним для кешування способом. Burst фокусується на скороченні часу потоку за рахунок кращої оптимізації коду, коли він виконується у системі завдань. Ціль всіх цих систем - збільшити те, що в принципі можливо в Unity з точки зору продуктивності, при цьому підтримуючи існуючі робочі процеси та полегшуючи перехід.

Сучасна апаратна архітектура оснащена та має тенденцію до використання кількох ядер. Тим не менш, багато процесів покладаються на використання тільки одного ядра. Запускаючи кілька процесів на кількох ядрах, ми можемо запускати їх одночасно паралельно, а не один за одним, тим самим ефективно використовуючи ємність ядер і домагаючись значного підвищення продуктивності.

Нова система завдань C# Job System використовує переваги кількох ядер простим та безпечним способом. Легко, оскільки він розроблений, щоб відкрити цей підхід для наших сценаріїв і дозволити нам писати швидко налаштовуваний код, і безпечний, оскільки забезпечує захист від деяких пасток багатопоточності, таких як умови гонки.

Ми можемо використовувати нові багатопоточні системи для створення ігор, що працюють на різноманітному обладнанні. Ми також можемо повністю використовувати переваги зрісту продуктивності, для того щоб створювати більш багаті ігрові світи з великою кількістю юнітів і складнішими симуляціями.

2.4 Компілятор Burst Compiler

Burst Compiler - це компілятор, він перетворює байт-код IL/.NET на високооптимізований власний код з використанням LLVM для додатків Unity. Він випущений як єдиний пакет та інтегрований у Unity за допомогою Unity Package Manager.

Нова технологія заснована на LLVM обчислювань бекенд-компіляції, яка перетворює завдання C# (Job System) на глибоко оптимізований машинний код.

Високооптимізований власний код означає, що ми отримаємо найкращу продуктивність. У деяких кейсах продуктивність набагато краща. Ось чому Burst грає велику роль в продуктивності Unity DOTS.

Компілятор Burst призначений в першу чергу для роботи з системою завдань C# Job System, Набагато важливіше те, що burst – це компілятор не спільного призначення, він являється частиною Unity та створений для того щоб додатки на Unity працювали швидше.

Він підтримає широкий спектр платформ, від автономних, настільних коп'ютерів, до консолей, і саме головне мобільні пристрої (IOS та Android).

Розглянемо саме чому burst compiler є швидким. Якщо задуматись, може виникнути питання – Чому просто не використати LLVM та дозволити створювати йому власний код? По-перше, LLVM на даний час не підтримує C#, але C# використовують unity по ряду причин. Він здатний вирішити безліч проблем та в той же час надати максимальну продуктивність. Однак в Unity вже існує технологія яка перетворює код C# у C++, який згодом компілюється в машиний код, результат швидше ніж при запуку Mono (IL2CPP). Навіщо ж потрібен ще один проміжний крок. IL2CPP – це більш широке поняття, а Burst – більш глибоке. IL2CPP має більш

велику підтримку функцій C#, в тот час як Burst виконує більший аналіз оптимізації меншої частини коду.

Burst має необхідні знання у API інтерфейсів Unity, які дозволяють виконати оптимізацію, звичайний компілятор не зможе цього зробити. Наприклад Burst знає що власні масиви не перекриваются в пам'яті чи те що вони не мають псевдонімів, що дозволяє векторизувати код.

Burst також знає ваше обладнання, для оптимальних результатів вам потрібно вказати цільове ЦП, потім Unity надає можливість вручну написати свій асемблерний код, якщо по якійсь причині ви не задоволені результатом. Для цьому ми відкриваємо внутрішні апаратні засоби. Знаючи своє обладнаня та дані з якими працює Burst, він спроможний отримати максимальний приріст продуктивності.

Ще одна відмініть між Burst та IL2CPP, Burst більш орієнтований на дизайн та дані, в той час як IL2CPP краще підтримує об'єктно-орієнтовані методи.

Розглянемо більш детально роботу Burst Compiler. Наприклад ви пишите код на C# потім натискаєте кнопку «build» і ваш код компілюється завдяки .NET Assembly який має розширення DLL, ця збірка складається з коду IL або як ще кажуть проміжної мови .NET і може бути запущена на віртуальній машині MonoVM. Пакетний аналіз та маніпулювання цим кодом переводить його на проміжну мову представлення LLVM або LLVM IR. Добре те що ми можемо легко порівняти IL с IR, не дивлячись на те що вони засновані на різних технологіях. Тут вступає знання про API-інтерфейсів Unity від Burst, потім для компіляції та оптимізації цього коду, використовується налаштована збірка LLVM. Посилання на платформу використовується для створення вихідної бібліотеки. Якщо ми кажимо про Android це називається Lib. Це конвеер компіляції або те що називается AOT компіляцією, або іншими словами попередня компіляція використовувана при розробці гри. В період виконання виклику пакетних функцій будуть направлятися в цю власну бібліотеку, а не просто запускати код .NET із початкової збірки.

Burst також працює в редакторі Unity коли ви входите в режим відтворення. Це називается своєчасна компіляція. Він використовує аналогічний конвеер з декількома функціями які є спецефічними для редактора Unity.

Що потрібно для того щоб почати працювати з Burst Compiler? Ми починаємо писати свій код навколо системи завдання C# Job System і завершує цей процес в системі компонентів ECS.

Рисунок 2.15 – Приклад використання Burst Compiler в проекті.

Як ми бачимо на рисунку 2.15, нам потрібно лише додати один рядок коду, а саме атрибут [BurstCompile] і все готово. Зверніть увагу, що якщо функція не відзначена атрибутом, то вона не буде перетворена на машиний код і виконається за стандартом без використання компілятора.

Burst – це безкоштовна продутивність, але не зовсім, існує ряд обмежень:

- Працює тільки з high-perfomance C# (value types, nativeArray, no reference types);
- Займає багато часу на компіляцію коду;
- Потрібен власний компонувальник (Android NDK);
- Відлагодження.

Для того щоб позбавитися від збирача сміття (garbage collector) ми не можемо використовувати типи за посиланням, а саме класи.

Для компіляції потрібен час, тому що чим більше ми оптимізуємо тим повільнішим є час компіляції. На даний час, це все ще залишається серйозною

проблемою над якою ще працюють, впроваджуючи різноманітні рівні кешування та інщі методи.

Однією з переваг цього компілятору є забезпечення майбутньої готовності до новітніх процесорів. Якщо вийде абсолютно нова лінійка процесорів, що містить кілька дивовижних нових функцій, Unity може зробити всю тяжку роботу за вас у фоновому режимі. Потрібно буде просто оновити компілятор, для того щоб отримати ці переваги. Компілятор базується на пакетах і може бути оновлений без необхідності оновлення редактора Unity. Оскільки пакетний пакет буде оновлюватися відповідно до ваших темпів, ви зможете скористатися останніми покращеннями та функціями апаратної архітектури, не чекаючи на введення коду в наступній версії редактора.

Також варто виділити те, що для програміста потрібно менше написанного коду власноруч. Ми з легкістю можемо користуватись перевагами налаштованого вручну асемблерним кодом на різних платформах.

Пакетний компілятор виробляє високо оптимізований код, який використовує платформи апаратних засобів, які ви збираєте.

2.4.1 Бібліотека Unity Mathematics

Unity Mathematics – це математична бібліотека C#, що надає векторні типи та математичні функції із синтаксисом, подібним до шейдера. Використовується компілятором Burst для компіляції C# / IL у високоефективний машинний код.

Основна мета цієї бібліотеки забезпечити дружній Math API , знайомий SIMD і графічними розробниками шейдерів, використовуючи добре відомі типи даних float4, float3 і т.д. з усіма вбудованими функціями, що надаються статичним класом «math» який можна легко імпортувати в C# підключивши бібліотеку наступним чином - «using static Unity.Mathematics.math;».

Unity Mathematics реалізує дуже прості векторні та матричні типи, такі як Vector3, Vector2 і т.д. , вони представляються як FloatN, тобто як float2, float3 і т.д., а також кватерніони та матричні типи, такі як float3x3, float4x4.

Також традиціонно, unity mathematics надає елементарні математичні функції, такі як min(), max(), abs(), тригерні функції, нормалізуючі векторні функції і т.д.

Окрім цього, компілятор Burst може розпізнавати оптимізований тип SIMD для працюючого ЦП на всіх підтримуваних платформах (x64, ARMv7a ... і т.д.).

Unity Mathematics потрібно використовувати, коли вже використовується стек технологій Unity DOTS орієнтований на дані, тому що пакет «Entities» та пакет «Physics» дуже часто використовують математичні функції, крім цього Burst Compiler знає про Unity Mathematics і це дуже важливий момент, тому що всі функції обчислення які знаходятся у цієї бібліотеці вже оптимізовані под Burst Compiler і вони перевершують за продуктивністю, класичні функції та типи даних у бібліотеці C#. Таким чином, на виході ми отримуємо зріст продуктивності при використанні математичних функцій.

Навіщо потрібно розробляти іншу математичну бібліотеку замість існуючої Unity Vector3... і т.д.? Надання API, ближчого до того, як розробники графіки використовують математичні бібліотеки, має краще сприяти його прийняттю та простоті використання. Математична бібліотека HLSL/GLSL – це дуже добре спроектована та зрозуміла математична бібліотека, що забезпечує велику узгодженість.

Також може виникнути питання, чому б не використовувати System.Numerics.Vectors. System.Numerics.Vectors багато в чому схожа на попередню векторну бібліотеку (більше об'єктно-орієнтована, ніж орієнтована на графічне програмування). Той факт, що компілятор Burst може розпізнавати набагато більше шаблонів для типів SIMD та математичних функцій, спрощує роботу з виділеним API, що відображає цю здатність.

Угода про іменування C#, «int» та «float» вважаються вбудованими типами. Burst розширює цей набір типів, включаючи вектори, матриці і кватерніони. Ці типи численні в тому сенсі, що Burst знає про них і може згенерувати кращий код, використовуючи ці типи, ніж те, що було б можливо з еквівалентним кодом з використанням типів, що налаштовуються.

Щоб позначити, що ці типи є множиною, їх імена типів пишуться малими літерами. Оператори цих типів bultin, знайдені в бібліотеці «Unity.Mathematics.math», вважаються внутрішніми і тому записуються в нижньому регістрі.

Немає планів розширювати набір внутрішніх типів за межі поточного набору векторів (typeN), матриць (typeNxN) та кватерніонів (quaternion). Ця угода має додаткову перевагу, оскільки робить бібліотеку дуже сумісною з кодом шейдера і робить перенесення чи спільне використання коду між ними майже без проблем.

Висновок до розділу 2

У цьому розділі ми розглянули набір технологій Unity DOTS, який орієнтований на дані. Він містить у собі патерн проектування ECS для побудови інформаційно-орієнтованої архітектури додатків, систему завдянь C# Job System, для розпаралелювання завдань на багатопоточних процесорах та компілятор Burst Compiler, який конвертує написаний C# код в оптимізований машинний. Також були розглянуті інструменти для конвертації ігрових об’єктів та корисні бібліотеки для використання цих технологій.

3 ПРАКТИЧНА ЧАСТИНА

3.1 Опис розробки архітектури додатку

У другому розділі ми дослідили інформаційно-орієнтовані технології Unity DOTS та розібрали патерн їх використання. Тепер ми застосуємо отримані знання на практиці для того щоб ми могли оптимізувати додаток на мобільних пристроях.

Для цього ми створимо проект в середовищі розробки Unity3d, щоб впершу чергу порівняти продуктивність при використанні різних підходів до проектування за допомогою стрес-тесту, а також у процесі використати інформаційно-орієнтовані технології Unity DOTS.

Загалом архітектура проекта побудована наступним чином, для отримання результатів продуктивності та їх порівняння ми написали менеджер, який відповідає за налаштування у виборі тієї чи іншої архітектури. Цей сценарій порівнює продуктивність між «класичним» підходом (генерування та переміщення простих об'єктів з моноповедінками) та підхідоми ECS (Entities Components Systems) та технологіями Unity DOTS (C# Job System, Burst Compiler).

Якщо розглядати більш детально, для побудови класичної архітектури ми написали сценарій «Movement», який успадковується від Monobihavior, це система яка відповідальна за логіку переміщення префабів кубів у просторі.

Для побудови архітектури заснованої на ECS ми написали систему під назвою «Movement System», а також компонент «MoveForwardComponent», який зберігає дані про кожну сутність.

Для використання технологій Unity DOTS були написані два сценарія «MovementSystemJobs» та «MovementSystemJobsBurst». Кожен з цих сценаріїв використовує окремо ту чи іншу технологію Unity DOTS (Job System або Burst Compiler).

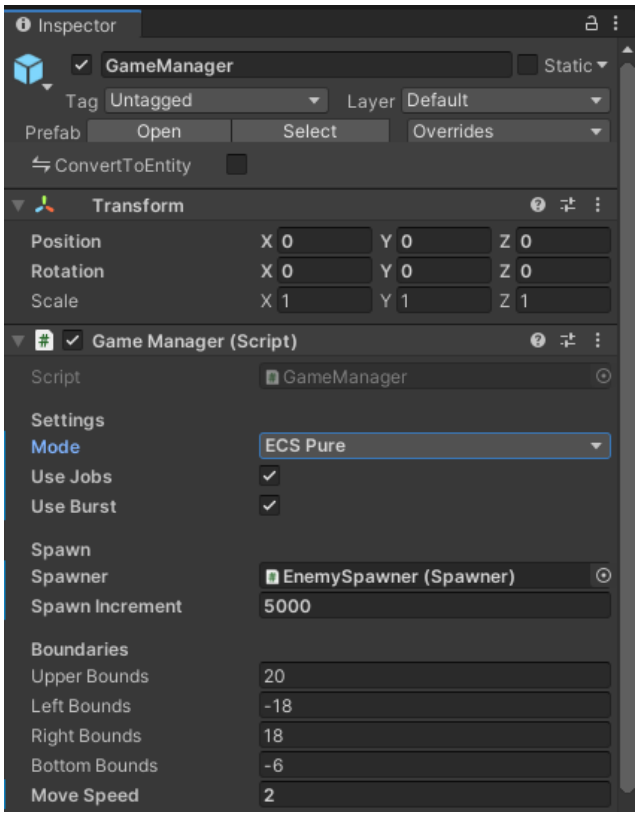


Рисунок 3.1 – Сценарій відповідаючий за логіку налаштування.

На рисунку 3.1 ми бачимо менеджер який відповідає за логіку налаштування процесу тестування продуктивності.

Цей менеджер оснащений перемикачем режиму, є три варіанти використання:

1. Classic (Monobihavior workflow).
2. ECS pure (Entity Component System).
3. ECS conversion (Conversion workflow).

Під час використання класичного робочого процесу, ми використовуємо префаби об’єктів. На цих префабах є сценарій відповідаючий за просте переміщення у просторі. Кожен куб переміщує себе по осі Z, і коли куб заходить дуже далеко за рамки екрану, він переміщує себе у початок. Таким чином об’єкти завжди знаходятся активними на екрані і не зникають.

Для використання ECS з технологіями Unity DOTS, у менеджері є прапорці для вибору використання тієї чи іншої технології.

Серед інших налаштувань ми маємо опцію для вибору кількості об'єктів, які ми можемо додати на сцену за одне натискання кнопки.

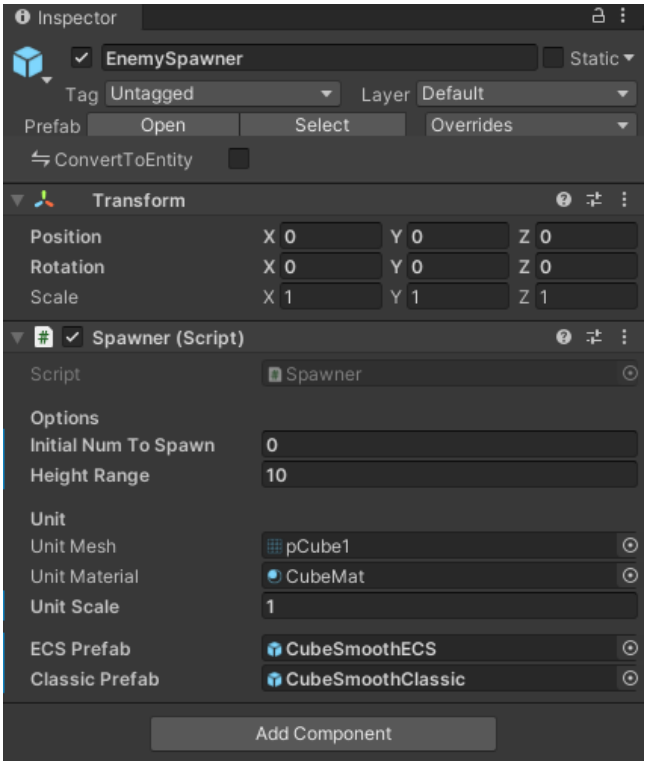


Рисунок 3.2 – Сценарій відповідальний за створення об’єктів.

Також у нас є сценарій відповідаючий за генерацію об’єктів після вибору режима тестування.

Якщо вибраний класиччний режим, тоді ми будемо генерувати об’єкти за допомогою метода «Object.Instantiate».

Під час використання режиму ECS pure, створення об'єктів буде проходити безпосередньо з коду.

Також є варіант з використанням Conversion workflow, це коли ми використовуэмо збірний GameObject як шаблон для наших Entities. Якщо ми переключимо режим у менеджері на «ECS Conversion» (режим використання ECS та конвертації об’єктів), ми можемо виконати деяке налаштування ігрових обєктів (GameObjects), а потім використати сутності (Entyties) під час виконання програми. Цей процес розробки є дуже зручним, а також продуктивним, головною перевагою

є те, що для цього потрібно менше коду, а кількість кубів на сцені такаж велика, як при використанні режиму «ECS Pure».

3.2 Порівняння отриманих результаті

Після того як ми створили робочий проект, ми можемо провести-стрес теставання. Стрес-тестування - це одна з форм тестування, яка використовується для визначення стійкості системи або як в нашому випадку архітектурних рішень та технологій, в умовах перевищення меж нормального функціонування.

Основним показником продуктивності є частота оновлення. Частота оновлення, що вимірюється в Гц (герцах) – це число оновлень зображення на дисплеї за 1 секунду. У більшості мобільних телефонів частота оновлення – 60 Гц, тобто зображення оновлюється 60 разів на секунду.

Для того щоб підбити підсумок, та виявити користь використання цих технологій, порівняємо отримані результати.

Рисунок 3.3 – Показники продуктивності при використанні класичної архітектури.

При використанні класичної архітектури ми можемо отримати десять тисяч кубів на комп'ютері і тримати показники продуктивності близько 25 кадрів у секунду. Ці показники стануть відправною точкою у порівнянні.

Рисунок 3.4 – Показники продуктивності при використані чистого ECS.

Ми можемо змінити режим з класичного на чистий ECS, де ми генеруємо об'єкти разом. Тепер повністю за допомогою коду написаного на ECS, використовуючи ту саму сцену. Як ми бачимо на рисунку 3.4 ми можемо отримати близько 45 кадрів у секунду, при використанні десяти тисяч кубів.

Це означає, що ми можемо збільшити кількість кубів і отримати приблизно таку ж кількість кадрів у секунду як і у класичній архітектурі, що в свою чергу приблизно перевищує в два рази кількість кубів на екрані і це досить непогані результати, при тому що ми просто змінили підхід до проектування, натиснувши на одну кнопку.

Тепер ми реалізуємо зовсім іншу архітектуру для переміщення кубів замість Monobihavior класів ми використовуємо дуже легкі структури для даних, такі як «MoveForwardComponent» для швидкості переміщення кубів. Також систему для переміщення кубів «MovementSystem». Тепер самі куби представлені у вигляді не

ігрових об’єктів (GameObject), а сутностей (Entities), саме тому ієрархія дивовижно пуста, але на екрані ми все ще отримуємо в два рази більше об’єктів.

Рисунок 3.5 – Показники продуктивності при використанні ECS та Job System.

На Рисунку 3.1 ми бачимо в інспекторі, що серед функцій менеджера є можливість підключити технологію системи завдань C# Job System.

Система завдань C# Job System допоможе використати переваги обладнання, якщо у вас є кілька ядер процесора, ви можете використати їх для того щоб розділити частину роботи, за допомогою цих додаткових потоків ми можемо отримати ще більше продуктивності, а також отримати більше кубів на екрані.

Як ми бачимо на рисунку 3.5 показники продуктивності зросли на 10 кадрів у секунду. У порівнянні з класичною архітектурою це майже 60 стабільних кадрів у секунду, для мобільних пристроїв цей показник вважається дуже гарним, так як для більшості мобільних пристроїв це максимальна частота оновлення зображення.

Рисунок 3.6 – Показники продуктивності при використанні ECS, Job System та Burst Compiler.

У випадку коли Burst Compiler не використовує Job System показники не є дуже вражаючими, але коли ми поєднуємо ці дві технології ми отримуємо всі переваги пакетного компілятора. Як тільки ми вмикаємо обидві технології, ми можемо отримати в три рази більше активних об'єктів ніж при використанні класичної архітектури. Також показники продуктивності становлять в районі 60 – 70 кідрів у секунду.

Для того щоб отримати справжнє враження від інформаційно-орієнтованих технологій Unity DOTS, потрібно зібрати проект в Unity та запустити його на цільовому мобільному пристрої.

Показники продуктивності можуть вразити, не дивлячись на те що ми використовуємо мобільний процесор, а також враховуючи те, що у нас немає додаткового навантаження від середовища розробки, ми можемо ще більше навантажити систему, додаючи на екрані ще більше об'єктів, а показники продуктивності будуть такими ж.

Отже, головна ідея полягає в тому, що якщо ви можете перебудувати вашу архітектуру і виникає питання як саме ви можете це зробити, Unity DOTS може

надати багато продуктивності, використовуючи теж саме обладнання, але при цьому використовуючи додаткові ядра процесора. Навіть якщо вам не потрібні тисячи, або навіть сотні тисяч об’єктів на екрані, переваги продуктивності відображаються у оптимізованому використанні енергоспоживанні батареї та ефективності її використання. Окрім цього, якщо ви створюєте мобільний додаток, ваш додаток буде споживати значно менше енергії, а також виділяти менше тепла і це завжди добре для кінцевих споживачів, загалом це безпрограшний варіант, але нажаль це зовсім інший спосіб роботи, до якого ще потрібно звикнути.

Висновок до розділу 3

В останьому розділі ми створили проект в середовищі розробки Unity3d, для проведення синтетичних тестів. Ми отримали та порівняли результати продуктивності в різних умовах та з використанням різних підходів до проеткування архітектури проекту. Використали архітектурний патерн проектування ECS та набір технологій орієнтованих на дані Unity DOTS.

ВИСНОВКИ

В ході даної магістерської роботи були проаналізовані та виявлені недоліки сучасних архітектурних фреймворків, які використовують архітектурний патерн проектування ECS. Для вдосконалення цього рішення був досліджений набір технологій Unity DOTS, який орієнтований на дані. Також був розроблений проект в середовищі розробки Unity3d, для проведення синтетичного тесту, щоб порівняти продуктивності двох підходів до проектування. Виходячи з результатів дослідження ми можемо впевнено сказати, що новий високопродуктивний багатопоточний стек інформаційно-орієнтованих технологій Unity DOTS дозволив використати сучасні багатоядерні процесори на повну потужність. Завдяки DOTS ми отримали збільшення продуктивності, при цьому не розробляючи складних алгоритмів для паралельних обчислень. DOTS - це зручна пісочниця для розробки безпечного багатопоточного коду, що підвищує продуктивність, оптимізує тепловиділення та енергоспоживання мобільних пристроїв. Крім того, перехід від об'єктно-орієнтованого до інформаційно-орієнтованого підходу полегшив багаторазове використання коду, що в свою чергу дало зріст продуктивності за рахунок зменшення кількості активних MonoBehaviour сценаріїв. Таким чином, Entity Component System (ECS) дозволив розробити високопродуктивний написаний код C#, що забезпечив величезну гнучкість у проектуванні загальної архітектури програмного забезпечення. Система задач C# (Job System) безпечно використала переваги багатоядерних процесорів та дала змогу розпаралелити задачі між ядрами процесора. Компілятор Burst Compiler перетворив написаний нами C# код на високооптимізований машинний код, таким чином ми отримали найкращу продуктивність.

ПЕРЕЛІК ПОСИЛАНЬ

1. Unity Learn Premium [Електронний ресурс] : [Інтернет-портал]. - Електронні дані. – Unity C# Survival Guide. – Режим доступу: https://learn.unity.com/course/unity-c-survival-guide?language=en

2. Unity3d documentation. [Електронний ресурс] : [Інтернет-портал]. - Електронні дані. – Американська організація «Unity Technologies». – Режим доступу: https://docs.unity3d.com/2019.3/Documentation/Manual/UnityManual.html

3. Unity3d Best practice guides. [Електронний ресурс] : [Інтернет-портал]. - Електронні дані. – Американська організація «Unity Technologies». – Режим доступу: https://docs.unity3d.com/2019.3/Documentation/Manual/BestPracticeGuides.html

4. Unity Learn Premium [Електронний ресурс]. - Електронні дані. – Entity Component System. – Режим доступу: https://learn.unity.com/tutorial/entity-component-system

5. Unity Learn Premium [Електронний ресурс] : [Інтернет-портал]. - Електронні дані. – ECS Survival guide . – Режим доступу: https://learn.unity.com/tutorial/ecs-survival-guide

6. Unity Learn Premium. [Електронний ресурс] : [Інтернет-портал]. - Електронні дані. – What is DOTS and why it’s important. – Режим доступу: https://learn.unity.com/tutorial/what-is-dots-and-why-is-it-important

7. Unity3d documantation [Електронний ресурс] : [Інтернет-портал]. - Електронні дані. – ESC documentation. – Режим доступу: https://docs.unity3d.com/Packages/com.unity.entities@0.17/manual/index.html

8. Unity3d documantation [Електронний ресурс] : [Інтернет-портал]. - Електронні дані. – Job System documantation. – Режим доступу: https://docs.unity3d.com/Manual/JobSystem.html

9. Unity3d documantation [Електронний ресурс] : [Інтернет-портал]. - Електронні дані. – Burst Compiler documantation. – Режим доступу: https://docs.unity3d.com/Packages/com.unity.burst@0.2/manual/index.html

10. Chris Dickinson , Unity 5 Game Optimization: Master performance optimization for Unity3D applications with tips and techniques that cover every aspect of the Unity3D Engine / Burningham B3 2PB, UK. – K., 2015. – 306 c.

11. Jonathon Manning, Paris Buttfield-Addison , Mobile Game Development with Unity / O'Reilly Media, Inc., Shroff – K., 2017 – 457 c.

12. Unity Learn Premium [Електронний ресурс] : [Інтернет-портал]. - Електронні дані. – Building for Mobile. – Режим доступу: https://learn.unity.com/tutorial/building-for-mobile#

13. Unity3d documantation [Електронний ресурс] : [Інтернет-портал]. - Електронні дані. – Unity Mathematics documantation. – Режим доступу: https://docs.unity3d.com/Packages/com.unity.mathematics@1.0/manual/index.html

14. Unity3d documantation [Електронний ресурс] : [Інтернет-портал]. - Електронні дані. – Conversion Workflow documantation. – Режим доступу: https://docs.unity3d.com/Packages/com.unity.mathematics@1.0/manual/index.html.

15. Ian Griffiths , Programming C# 8.0 / O'Reilly Media, Inc. USA. – K., 2020. – 800 c.

16. Unity3d documantation [Електронний ресурс] : [Інтернет-портал]. - Електронні дані. – Entity debugging documantation. – Режим доступу: https://docs.unity3d.com/Packages/com.unity.entities@0.1/manual/ecs_debugging.html

17. Unity3d documantation [Електронний ресурс] : [Інтернет-портал]. - Електронні дані. – Entity. – Режим доступу: https://docs.unity3d.com/Packages/com.unity.entities@0.1/manual/ecs_entities.html

18. Unity3d documantation [Електронний ресурс] : [Інтернет-портал]. - Електронні дані. – Components. – Режим доступу: https://docs.unity3d.com/Packages/com.unity.entities@0.1/manual/ecs_components.html

19. Unity3d documantation [Електронний ресурс] : [Інтернет-портал]. - Електронні дані. – Systems. – Режим доступу: https://docs.unity3d.com/Packages/com.unity.entities@0.1/manual/ecs_systems.html

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інженерії програмного забезпечення

МАГІСТЕРСЬКА РОБОТА «ЗАСТОСУВАННЯ ІНФОРМАЦІЙНО-ОРІЄНТОВАНИХ ТЕХНОЛОГІЙ UNITY DOTS ДЛЯ ПІДВИЩЕННЯ ПРОДУКТИВНОСТІ МОБІЛЬНИХ ДОДАТКІВ»

Виконав: студент групи ПДМ-61,
Нагорний Євген Олегович
Керівник: директор ННІІТ, професор,
доктор технічних наук
Бондарчук Андрій Петрович

Київ - 2021

МЕТА, ОБ'ЄКТА ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: вдосконалення архітектури мобільних додатків для підвищення продуктивності на багато поточних процесорах з використанням інформаційно-орієнтованих технологій Unity DOTS.

Об'єкт дослідження: процес розробки архітектури мобільних додатків з використанням інформаційно-орієнтованих технологій Unity DOTS.

Предмет дослідження: архітектурні патерни проектування та інформаційно-орієнтовані технології.

АКТУАЛЬНІСТЬ РОБОТИ

Актуальність пов’язана з процесом розробки мобільних додатків в середовищі розробки Unity3d. Для оптимізації додатку існують рішення, які вдосконалюють класичну архітектуру, тим самим підвищуючи продуктивність додатку.

Загальною проблемою таких рішень є те, що вони не використовують потенціал процесорів, які б могли покращити продуктивність у рази.

Головною особливістю у використанні технологій Unity DOTS є те, що вони дозволяють чітко організувати дані у пам’яті та розпаралелити процеси пов’язані з використання даних, для того щоб використати багато поточність процесора і тим самим підвищити продуктивність мобільного додатку.

АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

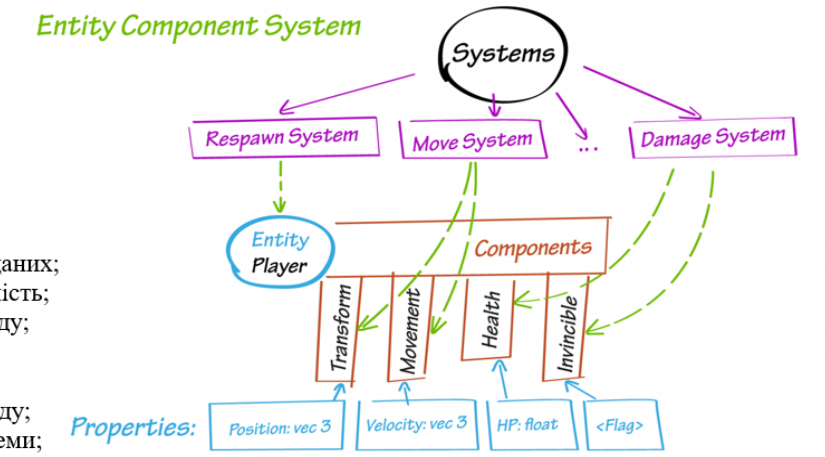
Для вирішення ряду проблем з класичною архітектурою компонентів в середовищі розробки Unity3d, зазвичай використовують архітектурний патерн проектування ECS. На сьогоднішній день існує кілька фреймворків які використовують цей архітектурний патерн проектування для розробки мобільних додатків.

	Мінуси
MonoBehaviour	Залежність коду
	Тестування
	Масштабування проектів
	Погана продуктивність
Entitas	Архітектура побудована на класах (Не велике збільшення продуктивності).
	Відсутність багатопоточності
Leopotam/ecs	Потребує багато написано коду власноруч
	Відсутність багатопоточності

МОДЕЛЬ АРХІТЕКТУРНОГО ПАТЕРНУ ПРОЕКТУВАННЯ ECS

Entity Component System (ECS) – це шаблон проектування, при якому відбувається поділ логіки на сутності, компоненти та системи. Сутності є контейнерами даних, які в свою чергу обробляються системами.

Архітектура ECS розрахована на максимальну продуктивність. Кожна сутність однакових архетипів лежать лінійно в пам'яті, під час кожної ітерації (додованні або видаленні) ми маємо швидкий доступ до даних.



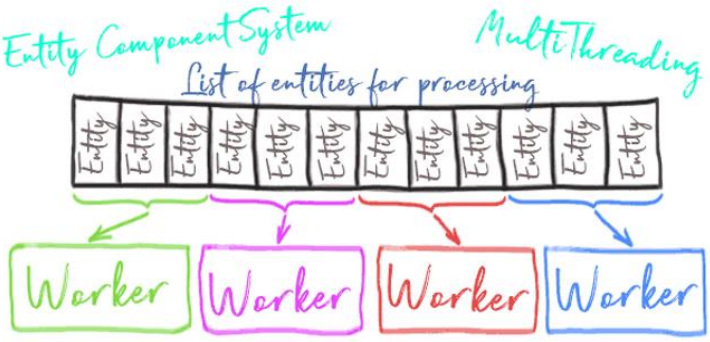
Переваги:

- швидкий доступ до даних;
- гнучкість і масштабність;
- низька залежність коду;
- швидкість розробки;
- продуктивність;
- перевикористання коду;
- легко тестувати системи;

ТЕХНОЛОГІЯ UNITY DOTS: C# JOB SYSTEM

C# Job System – дозволяє писати простий та безпечний багатопоточний код, який взаємодіє з Unity Engine для підвищення продуктивності.

Використовуючи архітектуру ECS та Job System ми можемо запускати системи в кілька потоків, використовуючи різні ядра центрального процесора.



ТЕХНОЛОГІЯ UNITY DOTS: BURST COMPILER

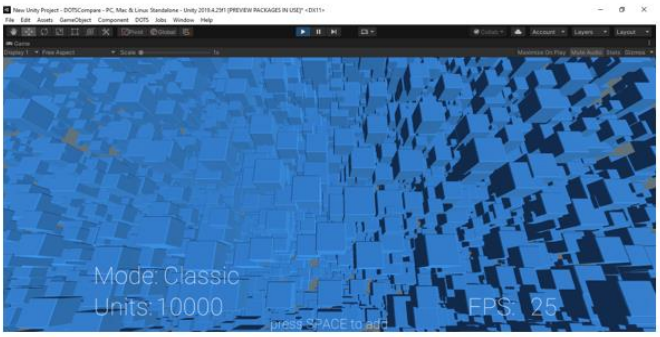
Burst Compiler – це технологія компіляції, яку можна використовувати для підвищення продуктивності проектів Unity, створених з використанням нового стеку технологій, орієнтованих на дані (DOTS) та системи завдань C# Job System.

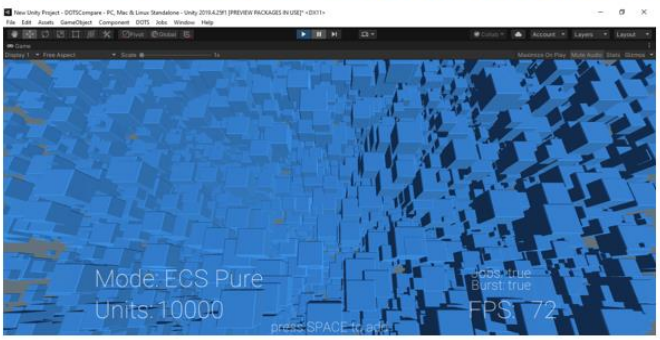
Компілятор Burst використовує завдання C# Job System для конвертації та створення оптимізованого машинного коду з урахуванням можливостей цільової платформи.

Особливості:

- компілятор конвертує .NET/IL (High Perfomance C#) в машиний код на основі LLVM;
- використовується для оптимізації написаного коду C# Job System;
- генерує найбільш оптимальний код під цільову платформу;
- оптимізована бібліотека математичних функцій (Math API);
- не потребує ніяких дій від програміста.

ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ДЛЯ ПРОВЕДЕННЯ СТРЕС-ТЕСТУВАННЯ





РЕЗУЛЬТАТИ ТЕСТУВАННЯ

Кількість об'єктів	5000	10 000	15 000
	Показники продуктивності		
Класична архітектура	Fps: **42** CPU: 14,2 мс.	Fps: **25** CPU: 26 мс.	Fps: **18** CPU: 38,1 мс.
Entity Component System (ECS)	Fps: **70** CPU: 9,5 мс.	Fps: **52** CPU: 14,9 мс.	Fps: **45** CPU: 18,4 мс.
ECS + Job System	Fps: **80** CPU: 6,6 мс.	Fps: **69** CPU: 7 мс.	Fps: **61** CPU: 8,8 мс.
ECS + Job System + Burst Compiler	Fps: **84** CPU: 5,3 мс.	Fps: **72** CPU: 6.8 мс.	Fps: **63** CPU: 8.3 мс.

ВИСНОВКИ

В ході магістерської роботи були проаналізовані та виявлені недоліки сучасних архітектурних фреймворків, які використовують архітектурний патерн проектування ECS. Для вдосконалення архітектурного патерну проектування був досліджений набір технологій Unity DOTS. Розроблено архітектурне рішення з використанням інформаційно-орієнтованих технологій Unity DOTS. Для проведення порівнянь продуктивності був створений проект в середовищі розробки Unity3d, який демонструє різницю між двома підходами проектування архітектури додатку.

Виходячи з результатів стрес-тестування ми можемо впевнено сказати, що високопродуктивний багатопоточний стек інформаційно-орієнтованих технологій Unity DOTS, дозволив використати багатоядерні процесори на повну потужність. Завдяки технологіям DOTS ми отримали збільшення продуктивності, при цьому не розробляючи складних алгоритмів для паралельних обчислень.

ПУБЛІКАЦІЇ ТА АПРОБАЦІЯ

Статті:

1. Нагорний Є.О. Застосування інформаційно-орієнтованих технологій Unity DOTS для підвищення продуктивності мобільних додатків / Зв'язок. №4 (146), 2021. С.35-42

Тези доповідей на конференціях:

1. Нагорний Є.О. Застосування інформаційно-орієнтованих технологій Unity DOTS для підвищення продуктивності мобільних додатків // XIII Науково-технічна конференція «Сучасні інфокомунікаційні технології», 10 грудня 2021 року, Державний університет телекомунікації, Київ, Україна.

ДЯКУЮ ЗА УВАГУ!