Харківський національний університет радіоелектроніки
Факультет Комп'ютерних наук
Кафедра Програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти - другий (магістерський)

Дослідження програмних архітектур реалізації комп'ютерних ігор в жанрі "Collectible Card Game»

Виконав: студент 2 курсу, групи ІПЗм-20-3
Таушанов М. П.
(прізвище, ініціали)

спеціальності 121- Інженерія програмного забезпечення
(код і повна назва спеціальності)

Освітньо-наукової програми
(тип програми)

Інженерія програмного забезпечення
(повна назва освітньої програми)

Керівник _____ доц. Мазурова О. О.
(посада, прізвище, ініціали)

Допускається до захисту

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

2022 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук

Кафедра Програмної інженерії

Рівень вищої освіти - другий (магістерський)

Спеціальність 121-Інженерія програмного забезпечення
(код і повна назва)

Тип програми освітньо-наукова програма

Освітня програма Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 20 ___ р.

ЗАВДАННЯ

НА АТЕСТАЦІЙНУ РОБОТУ

студента Таушанова Микити Павловича
(прізвище, ім'я, по-батькові)

1. Тема роботи: Дослідження програмних архітектур реалізації комп'ютерних ігор в жанрі "Collectible Card Game"

затверджена наказом університету від "_24_" _03_2022 р № ___412 Ст_
2. Термін подання студентом роботи до екзаменаційної комісії
15 травня 2022 р.
3. Вихідні дані до роботи *електронні ресурси за обраною тематикою, загальні вимоги до архітектури системи, методичні вказівки з дипломування*
4. Перелік питань, що потрібно опрацювати в роботі вступ, аналітичний огляд, аналіз існуючих підходів, опис програмної реалізації, опис експериментальних досліджень, висновки, перелік посилань.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Позначка про виконання
1	Аналіз проблемної області	17.01.22 – 22.01.22	виконано
4	Розробка постановки задачі	22.01.22 – 29.01.22	виконано
5	Аналіз існуючих архітектур	30.01.22 – 17.02.22	виконано
	Проектування та розробка ПЗ	17.02.22 – 15.03.22	виконано
	Планування експериментального дослідження	15.03.22 – 27.03.22	виконано
	Проведення експериментів	28.03.22 – 05.04.22	виконано
	Аналіз результатів експерименту	06.04.22 – 16.04.22	виконано
9	Створення висновків та рекомендацій на основі результатів експерименту	17.04.22 – 25.04.2022	виконано
13	Підготовка пояснювальної записки	01.04.22 – 15.05.2022	виконано
	Підготовка презентації та доповіді		
	Нормоконтроль		
	Рецензування		
	Занесення диплома в електронний архів		
	Попередній захист		
14	Допуск до захисту у зав. кафедри		

Дата видачі завдання _17_._01_.2022 р.

Студент _____(Таушанов М. П.)

Керівник роботи _____(доц. Мазурова О. О.)

РЕФЕРАТ/ABSTRACT

Кваліфікаційна робота магістра містить: 94 стор., 30 рис., 6 таблиць, 28 джерел, 6 додатків.

АРХІТЕКТУРА, ДОДАТОК, ДОСЛІДЖЕННЯ, ККІ, C#, ECS, HMVC, UNITY

Обєкт дослідження - програмні архітектури реалізації комп'ютерних ігор.

Метою роботи є підвищення ефективності реалізації комп'ютерних ігор в жанрі "Collectible Card Game" за рахунок правильно подібраних програмних архітектур.

Методи розробки архітектур базуються на C#, Unity з використанням Rider.

Результатом роботи є проведенене дослідження та описані висновки та рекомендації щодо обраних архітектур.

Explanatory note contains: 94 p., 30 fig., 6 tables, 28 sources, 6 applications..

APP, ARCHITECTURE, CCG, C#, ECS, HMVC, RESEARCH, UNITY

The object of the study are software architectures of computer games implementation.

The purpose of this work is to increase the efficiency of the implementation of computer games in the "Collectible Card Game" genre based on properly selected software architectures.

Architecture development methods are based on C #, Unity using Rider.

The result of the work is the conducted research and described the conclusions and recommendations for selected architectures.

Я, Таушанов Микита Павлович, студент гр. ІПЗм-20-3, здобувач вищої освіти на другому (магістерському) рівні кафедри «Програмна інженерія», заявляю: моя кваліфікаційна робота на тему «Дослідження програмних архітектур реалізації комп'ютерних ігор в жанрі "Collectible Card Game», що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElAr KhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

ЗМІСТ

Вступ 8

1 Аналіз проблемної області та постановка задачі 10

1.1 Аналіз проблемної області розробки комп'ютернох ігор в жанрі CCD 10

1.2 Постановка задачі 14

2 Опис прийнятих проектних рішень 16

2.1 Аналіз та моделювання предметної області 16

2.2 Аналіз існуючих підходів до проектування архітектур 22

2.3 Вибір підходів до проектування архітектур для подальшого дослідження 31

2.4 Проектування обраних архітектур ПЗ 34

3 Опис програмної реалізації 40

3.1 Вибір засобів програмної реалізації 40

3.2 Опис клієнтської розробки 40

3.3 Опис обраних фреймворків та бібліотек 45

3.4 Опис реалізації обраних архітектур 50

4 Опис експеріментальних досліджень 58

4.1 Планування експерименту 58

4.2 Результати проведення експерименту 60

4.3 Висновки та рекомендації з експериментального дослідження 67

Висновки 69

Перелік джерел посилання 70

Додаток А - Перелік джерел посилання за науковими напрямами керівника та науковців кафедри програмної інженерії 73

Додаток Б - Результати перевірки роботи на академічну доброчесність 74

Додаток В - Слайди презентації 75

Додаток Г - Тези доповіді 87

Додаток Д - Код сервісу для роботи з Addressables 89

Додаток Е - Код ECS-системи 93

ВСТУП

Сучасний всесвіт неможливо уявити без комп’ютерних ігор, індустрія яких зайняла величезну частину ринку розваг та дозвілля. Вони дають людству неповторну можливість перенестися в світ фантазій. Серед усього різноманіття усіляких жанрів відеоігор насамперед окреме місце займають стратегічні ігри. Саме в таких іграх перед гравцем виникають доволі складні завдання, тому що він повинен своєчасно реагувати на неочікуванні ігрові події, продумувати тактику своїх дій наперед задля того, щоб досягти успіху в грі. Однією з найпопулярніших категорій таких стратегій є стратегії у жанрі «Колекційних карточних ігор».

Крім того, останні роки серед усіх платформ для комп'ютерніх ігор саме мобільні платформи отримують найбільшу популярність – багато популярних компаній орієнтуються переважно на мобільні пристрої, що робить розробку таких застосунків актуальною та необхідною задачею та задає тренд на певні архітектурні підходи в проєктуванні ігор. Так як розробка ігор для мобільних девайсів є відносно новою галузією, то ще немає єдиного підходу до розробки таких застосунків. Це призводить до великої кількості поганих архітектурних рішень, які працюють неоптимізовано, втрачаючі аудиторію користувачів зі старими пристроями, а також ускладнюють розробку, тестування та супровід програмного забезпечення.

Метою роботи є проведення аналізу програмних архітектур, що використовуються для реалізації комп'ютерних ігор, аналізу та моделювання предметної області комп'ютерніх ігор в обраному жанрі, проектування та реалізація програмного рішення для проведення дослідження обраних архітектур; планування та проведення експериментального дослідження розроблених архітектур.

Під час виконання магістерської кваліфікаційної роботи було проведено аналіз проблемної області штучного інтелекту в ігрових додатках на основі публікацій світових та вітчизняних фахівців в проблемній області (див. додаток А).

В результаті роботи було проаналізовано найбільш популярні підходи до проектування ігрових архітектур в обраному жанрі. На основі аналізу архітектур,

було сформовано вимоги до архітектури, що повина бути найкращою для реалізації комп'ютерних ігор в жанрі "Collectible Card Game". Та обрано дві найбільш привабливі архітектури для дослідження та проведення порівняння. Після проведенного дослідення було сформовано висновки та рекомендації по користуванню обраних архітектур.

Робота пройшла успішну перевірку на академічну доброчесність (див. додаток Б).

На основі плану проведення дослідження та його результатів було розроблено презентацію (див. додаток В) та подано тези доповіді на Дванадцяту міжнародну науково-технічну конференцію (див. додаток Г). Частина програмного коду наведена в додатках Д та Е.

1 АНАЛІЗ ПРОБЛЕМНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз проблемної області розробки комп'ютерних ігор в жанрі CCD

У сучасному світі, комп'ютерні ігри стали не тільки розвагою, але і носієм культури. Виникнувши в далекій давнині, ще на ранніх етапах розвитку родовому суспільстві, гра розвивалася і вбирала в себе різноманітні явища дійсності [1]. Зараз же комп'ютерні ігри дають людям нову унікальну можливість перенестися в світ фантазій. Раніше не було способу глибоко занурюватися в нереальний світ і мати там свободу дій. Саме інтерактивність ігор є їх перевагою над кіно, театром і книгами.

Комп'ютерні ігри – один з найбільш значущих і швидко зростаючих сегментів глобальної масової культури. Вражають темпи розширення ігрового співтовариства, так само як і перспективи розвитку індустрії, що займається розробкою, просуванням і продажем ігор.

Аналітична компанія SuperData Research опублікувала статистичний звіт індустрії відеоігор за 2019 рік [2]. Загальний дохід сфери інтерактивних розваг і відеоігор склав 120 млрд доларів – це рекордний показник за всю історію індустрії. Майже 30 млрд доларів принесли продукти для ПК-гейминга, консолі – 15 млрд. Найвищий результат зафіксований в сфері мобільного гейминга – вона дала індустрії 65 млрд доларів прибутку. Хоч і мобільні пристрої (смартфони, планшети) за технічними характеристиками набагато слабкіше комп'ютерів, але це не заважає їх популярності серед користувачів.

Через це виникає питання технічної якості розробляємого додатку на мобільних платформах. Спростити розробку кросплатформної гри можна завдяки ігровому рушію. Ігровий рушій – програмний рушій, центральна програмна частина будь-якої відеогри, яка відповідає за всю її технічну сторону, дозволяє полегшити розробку гри шляхом уніфікації та систематизації її внутрішньої структури. Важливим значенням рушія є можливість створення багатоплатформових ігор [3]. Основну функціональність гри зазвичай забезпечує її

рушій, до якого входить рушій рендерингу, фізичний рушій, звук, система скриптів, анімація, ігровий штучний інтелект, мережевий код, керування пам’яттю, багатопоточність і граф сцени. Часто на процесі розробки можна заощадити шляхом повторного використання одного рушія гри для створення декількох різних ігор.

Тому ігровій додаток у жанрі "Collectible Card Game", архітектурні підході до якого будуть досліджуватіся, буде являти собою кросплатформний мобільний додаток, оскільки це дає можливість користуватися ним у будь-який зручний для користувача час, в будь-якому місці та на будя-якій операційній системі (ОС). В даний час найбільш популярними мобільними ОС є Android від компанії Google та iOS від Apple. Підтримка цих платформ може буди досягнута за допомогою використання ігрового рушія Unity. Лідируючу позицію вже не перший рік займає операційна система Android, і завдяки своїй поширеності вона була обрана головною платформою для проведення дослідження.

При створенні багатокористувацької гри дуже важливим чинником є вибір мережевої архітектури додатку [4]. Під мережевий архітектурою мається на увазі засіб підключення і спілкування клієнтів (комп'ютерів гравців) між собою в багатокористувацьких іграх [5]. Як правило, для ігор використовується два способи підключення це: Peer-To-Peer і Client-Server [6].

Peer-to-peer — варіант архітектури системи, в основі якої стоїть мережа рівноправних вузлів. Комп'ютерні мережі типу peer-to-peer (або P2P) засновані на принципі рівноправності учасників і характеризуються тим, що їх елементи можуть зв'язуватися між собою, на відміну від традиційної архітектури, коли лише окрема категорія учасників, яка називається серверами, може надавати певні сервіси іншим [7]. Але головним недоліком такої архітектури є «читерство» [6], тобто діяльність, яка змінює поведінку гравця, шляхом застосування спеціально створених або модифікованих програм або обладнання в онлайн-іграх, щоб дати одному гравцю перевагу над іншими [8].

Тому останнім часом велику популярність здобула клієнт-сервісна архітектура [9]. Архітектура клієнт-сервер передбачає взаємодію та обмін даними між такими основними компонентами:

– набір серверів, які надають інформацію або інші послуги програмам, які звертаються до них;
– набір клієнтів, які використовують сервіси, що надаються серверами;
– мережа, яка забезпечує взаємодію між клієнтами та серверами.

Також окрім платформи та моделі коммунікації між сервером та клієнтом, важливим чинником є жанр гри. За останні роки в сфері комп'ютерних ігор набувають популярності колекційні карткові ігри.

Колекційні карткові ігри – це досить складні стратегічні ігри. Колекційними картковими іграми (ККІ або CCG) вважаються тільки ті ігри, в яких гравці повинні збирати зі своїх карт колоди і грати ними. Практично у всіх CCG гравцеві доступна не вся колода карт відразу, а тільки її частину, яка приходить випадковим чином. Основний вид бою в більшості CCG – це дуель між двома гравцями, але в комп'ютерних іграх в якості суперника часто виступає ігровий персонаж (ШІ).

Також дуже важливим етапом при розробці грі є створення її гейм-дизайну, тобто вигадування правил, стратегії та ігрового балансу для створюємої гри [11]. Як правило, цим займається гейм-дизайнер. Геймдизайнер – це людина, яка придумує концепцію гри, її основні риси. Він проєктує базову ігрову механіку, ігровий процес, образи персонажів, коригує їх впродовж розробки. Тому дуже важливо мати таку архітектуру гри, щоб вона була доволі легко розширювальною не тільки для розробників, а й для людей, які не вміють програмувати та взаємодіють з розробниками завдяки ігровому рушію та спеціальніх ігрових інструментів.

Цінність будь-якої системи, тим паче ігрової системи, полягає у даних, які вона має та які вона використовує у своїх обчисленнях. Тому дуже важливим етапом створення програмних систем є вибір бази даних.

Існують різні підходи до організації даних [12]: ієрархічний, реляційний, об’єктно-орієнтований, NoSQL-підхід та інші. Всі вони мають свої недоліки та

переваги, однак для побудови великих програмних комплексів використовують реляційні бази даних. Адже вони мають чітке математичне обґрунтування до збереження даних (виконуються закони математичної логіки). Проте існують ситуації, в яких небажано використовувати реляційний підхід, адже він може знизити швидкість виконання запитів. В таких ситуаціях слід звернути увагу на NoSQL системи, які мають здатність швидко реагувати на зміну даних. Слід звернути увагу на те, що NoSQL підхід є зручним та ефективним для зберігання простих даних, а реляційний – для складних.

Окрім всього цього, маємо також і хардварні проблеми сучасних протсроїв. Сучасний телефон може працювати так само швидко, як і сучасний комп'ютер (але не дуже довго через перегрівання). Також треба розуміти, що збільшення продуктивності це не тільки використання всіх ядер, але й оптимізація одноядерної продуктивності.

Наприклад, однією з сучасних проблем в геймдеві та звічайних мобільних застосунків є те, що продуктивність процесерів йде вгору, а швидкість доступу до пам'яті збільшується не так сильно, що можна бачини на рисунку 1.1:

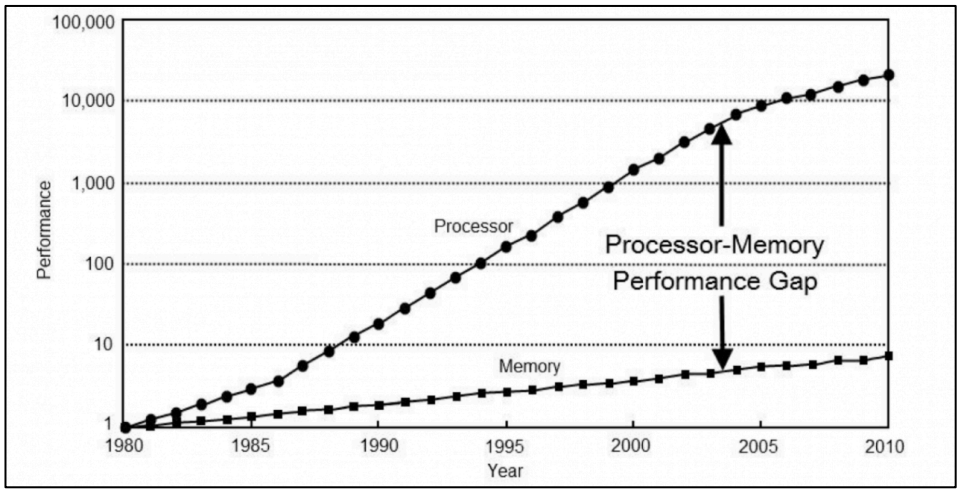

Рисунок 1.1 – Перформанс Gap

Видно, що зараз доступ до пам'яті є надзвичайно повільним. Виробники процесорів роблять дуже багато для того, щоб нівелювати цю різницю - додають кеші, CPU займаються спекулятивним обчисленням, намагаючись передбачити,

який код виконуватиметься далі. То гарна архітектура повіна скористатися всіма перевагами сучасних процесорів.

Виходячи з усього вищеназваного, стає дуже непростою задача дослідження клієнтських програмних архітектур реалізації комп'ютерних ігор в жанрі "Collectible Card Game", бо така архітектура повинна бути зручною при співпраці з гейм-дізайнером, повинна бути універсальною та стандартізованою, мінімізуючи витрати на супровід та розширення програмного коду. Окрім того, вона не повинна залежати від того, яку саме модель взаємодії з сервером було обрано (P2P чи Client-Server), на якому ігровому рушії було зроблено гру (Unity, Unreal Engine, Godot, etc), на якій платформі гра була запущена (Android чи iOS), як саме організована база даних (SQL чи NoSQL) та чи є в грі штучній інтелект. Окрім цих обмежень, існують ще обмеження апаратного плану слабких девайсів: кількість оперативної пам’яті, графічна API, що підтрімується, кількість ядер процесору, архітектура процесору, підтрімка SIMD-команд процесором та інші оптимізаційни пітання.

1.2 Постановка задачі

Метою є дослідження програмних архітектур реалізації комп'ютерних ігор в жанрі «Колекційних карточних ігор» задля того, щоб проаналізуваті можливі рішення існуючіх недоліків та реалізоваті таку архітектуру, що дозволить розробляти мобільні ігрі більш стандартизовано, враховуючі всі обіеження, та з ціллю покращення якості та можливості тестування мобільних ігрових застосунків.

Актуальність дослідження зумовлено популярністю ігрової індустрії, популярністю розробки саме мобільних ігр під операційнімі системамі iOS та Android, ігрового рушія Unity, жанру ігор ККІ, а також відсутністю чітко сформованого підходу до проектування клієнтської частини архітектури таких застосунків.

Ще одним фактором, який обумовлює актуальність дослідження – поступовий перехід Unity від стандартної компонентної архітектурі до DOTS-стеку

розробки, що відбувається останнім часом, та впливає на всю ігрову індустрію [13]. Саме такий новий підхід є трендом останніх років усієї ігрової індустрії, що робить дослідження цього підходу корисним для покращення якості коду та його супроводжуваності.

У якості завдання було обрано дослідження програмних архітектур реалізації комп'ютерних ігор в жанрі "Collectible Card Game".

Через це об’єктом дослідження є сучасні архітектурні підходи, які можуть бути використані для розробки мобільних ігор та які відповідають усім зазначеним вище обмеженням.

Головне питання, на яке необхідно отримати відповідь у результаті дослідження та після проведення експерименту – “Яку саме клієнтську архітектуру краще використовувати при реалізації комп'ютерних ігор в жанрі “Collectible Card Game”?.

Для досягнення поставленої мети необхідно вирішити наступні задачі:

– провести аналіз програмних архітектур, що використовуються для реалізації комп'ютерних ігор;

– провести аналіз та моделювання предметної області комп'ютерної гри в жанрі «Collectible Card Game»;

– спроектувати та реалізувати програмні рішення для проведення дослідження обраних та розроблених архітектур;

– спланувати та провести експериментальне дослідження розроблених архітектур, розробити відповідні рекомендації.

2 ОПИС ПРИЙНЯТИХ ПРОЕКТНИХ РІШЕНЬ

2.1 Аналіз та моделювання предметної області

Проведемо аналіз передметної області, а саме аналіз механіки та принципів гри у жанрі CCD.

Основна механіка гри жанру CCD – це стратегічний бій між двома гравцями колекційними картками, що є в них у колоді. Це може бути бій або між людьми через Інтернет, або між людиною та штучним інтелектом.

Битва відбувається на ігровому полі розбитому на дві частини. Нижня частина – це територія гравця, верхня – зона супротивника. Знизу гравець бачить карти, які він має на руках, та які він може використовувати у бою. Карти супротивника він не бачить, але бачить іх кількість, бо можна бачити їх зворотню сторону. Карти які гравець використовує у бою, він кладе у нижню центральну частину поля, а його супротивник – відповідно у верхню. Справа від поля боя розташована колода, з якої гравець випадково достає картку з заздалегідь сформованої колоди.

Колода – це простий набір в кількість 20 карток. Кожна картка має назву, опис, картинку, бали здоров’я, категорію, бали атаки та ціну у ресурсах, які необхідно вистачити на неї під час дуелі.

Після початком боя між гравцями випадковим чином визначається право першого ходу, і кожен з гравців витягує, в залежності від черговості, по 3 або 4 карти зі своєї колоди.

Мета гравця – перемогти свого супротивника, завдавши йому достатню кількість шкоди, щоб його бали здоров'я стали нижче нуля. Для цього він повинен обирати картки, які потрібно запустити на поле.

Картки поділяються на дві категорії: істоти та магія. У істот є бали здоров’я, бали атаки та ціна у ресурсах. Тобто бали атаки позначають шкоду, яку вони можуть завдати істотам супротивника або самому супротивнику. Бали здоров’я –

кількість шкоди, яку вони можуть «витримати» перед тим як покинути поле боя. Карти магії не мають балів здоров'я та використовуються одноразово.

Картки під час боя купуються за ресурси. Гравець заробляє ресурси кожен раз, коли він отримує хід в бою. Максимальна кількість ресурсів – 10.

Після покупки картки істот та розміщенні її на полі, вона має змогу атаковати тількі у наступному раунді, картки ж магії використовуються одразу після придбання.

Важливо відмітити, що атаковати самого гравця можна тільки в тому випадку, якщого всі його істоти були знищені.

Перейдемо до моделювання предметної області з метою проектування БД компютерної гри.

Головною метою концептуального моделювання є огляд предметної області та створення концептуальної моделі. Концептуальна (змістовна) модель – це абстрактна модель, що визначає структуру модельованої системи, властивості її елементів і причинно-наслідкові зв'язки, властиві системі і суттєві для досягнення мети моделювання

Тобто концептуальна модель включає опис об'єктів і їх взаємозв'язків, що представляють інтерес у розглянутій предметній області і виявляються в результаті аналізу даних.

Для заданої предметної області головними поняттями є користувач, колода та гральна колекційна картка (далі просто картка). Щоб визначити, чи є ще якісь допоміжні сутності, необхідно провести аналіз заданих понять та виявити їх характеристики.

Користувач може мати декілька колод карт, тобто колода – це простий набір в кількість 20 карток. Кожна картка має назву, опис, картинку, бали здоров’я, категорію, бали атаки та ціну у ресурсах, які необхідно вистачити на неї під час дуелі. З цього випливають ще дві сутності предметної області – категорія та дуель. Категорія поділяє на звичайні картки та картки зі спеціальними ефектами, які можна використовувати під час дуелі. У той час як дуель – це протистояння користувача та ШІ.

Отже основні сутності предметної галузі – «Користувач», «Колода» та «Картка». Крім того, було вирішено ввести допоміжні сутності «Категорія» та «Дуель». Враховуючи виявлені зв’язки, буда побудована схема взаємозв’язку об’єктів, що зображена на рисунку 2.1 у вигляді загальної діаграми класів на мові UML:

Рисунок 2.1 – Загальна діаграма класів

Таким чином, вибір об’єктів та сутностей, встановлення взаємозв’язків та сценаріїв показує цілісну картину розроблюваної програмної системи.

Як вже було виявлено раніше при концептуальному моделюванні, для обраної ігрової системи основними поняттями, які повинні зберігатися в базі даних є «Користувач», «Колода» та «Картка». Крім того, було вирішено ввести допоміжні сутності «Категорія» та «Дуель».

Користувач системи матиме змогу реєструватися у системі, редагувати свої дані, розвиватись переглядати свої карткові колоди, бойову статистику та історію матчей. Також користувач має змогу переглядати загальний рейтинг користувачів, та може приймати участь в бою, змагаясь при цьому з іншим користувачем.

Бій відбувається між двома користувачами, він має дату та час початку та дату та час кінця. Бій може біти завершеним та незавершеним. Завершений бій містить також інформацію про перемігшого гравця.

Також бій (дуель або матч) має інформацію про колоди гравців під час бою, що являє собою інформацію про картки, які використовував гравець, щоб грати зі своїм опонентом.

Кожна картка в свою чергу належить до якоїсь категоріх та має ще купу інших характеристик, які будуть приведені пізнише.

Також слід зауважити, що природа гри є ріалтаймовою, тобто дані повинні синхронізуваттся для всіх клієнтів у режимі реального часу та залишаються доступними, коли гра переходить у режим офлайн. Крім того, є обмеження самої технології. Це також накладає певні обмеження на стуктуру бази даних.

Firebase Realtime Database - це хмарна база даних. Дані зберігаються як JSON і синхронізуються в режимі реального часу з кожним підключеним клієнтом. Коли створюються крос-платформні програми з пакетами SDK для iOS, Android та JavaScript, всі клієнти діляться одним екземпляром бази даних Realtime і автоматично отримують оновлення з найновішими даними [14]. В парі з нею використовується файлова база даних Firebase Cloud Storage [15].

Realtime Database є NoSQL базою даних та підкорюється вимогам програми, бувши максимально оптимізована під найбільш часті запити. Для таких баз даних, на відміну від зберігання вийнятково зовнішніх ключів, звичний підхід зберігати зовнішні значення заразом з даними моделі, тобто їм властива денормалізація даних.

Отже, у ході аналізу та концептуального моделювання були виявлені основні сутності системи та їх атрибути, що мають стати базовими при моделюванні та розробці системи.

Сутність «Користувач» має наступні атрибути:

– унікальний ідентифікатор;

– логін;

– пароль;

– остання дата візиту;

– масив з колодами;

– масив з дуелями;

– кількість дуелей (матчей);

– відношення перемог до поразок.

Сутність «Дуель» повинна мати такі атрибути:

– унікальний ідентифікатор;

– ідентифікатор користувача, що грає проти поточного гравця;

– дату та час початку;

– дату та час закінчення;

– колода поточного гравця;

– колода опонента;

– статус;

– результат матча;

Сутність «Колода» має такі атрибути:

– унікальний ідентифікатор;

– назва;

– масив ідентифікаторів карток.

Сутність «Картка» містить такі атрибути:

– унікальний ідентифікатор;

– назва;

– опис;

– бали атаки (шкоди, що завдається);

– кількість одиниць здоров’я;

– ціна;

– категорія;

– ідентифікатор зображення, що зберігається в Cloud Storage.

NoSQL, на відміну від SQL, який має ER та діаграми класів, не має обмежень для діаграм в моделювання даних. Очевидна причина - відсутність жорстких правил відносин NoSQL.

ER-модель — модель даних, яка дозволяє описувати концептуальні схеми за допомогою узагальнених конструкцій блоків. ER-модель — це мета-модель даних, тобто засіб опису моделей даних. Існує ряд моделей для представлення знань, але одним з найзручніших інструментів уніфікованого представлення даних, незалежного від програмного забезпечення, що його реалізує, є модель «сутність-зв'язок». Важливим є той факт, що з моделі «сутність-зв'язок» можуть бути породжені всі існуючі моделі даних (ієрархічна, мережева, реляційна, об'єктна), тому вона є найзагальнішою [16].

Після опису необхідних для системи сутностей, було створено ER-діаграму бази даних, що зображена на рисунку 2.2:

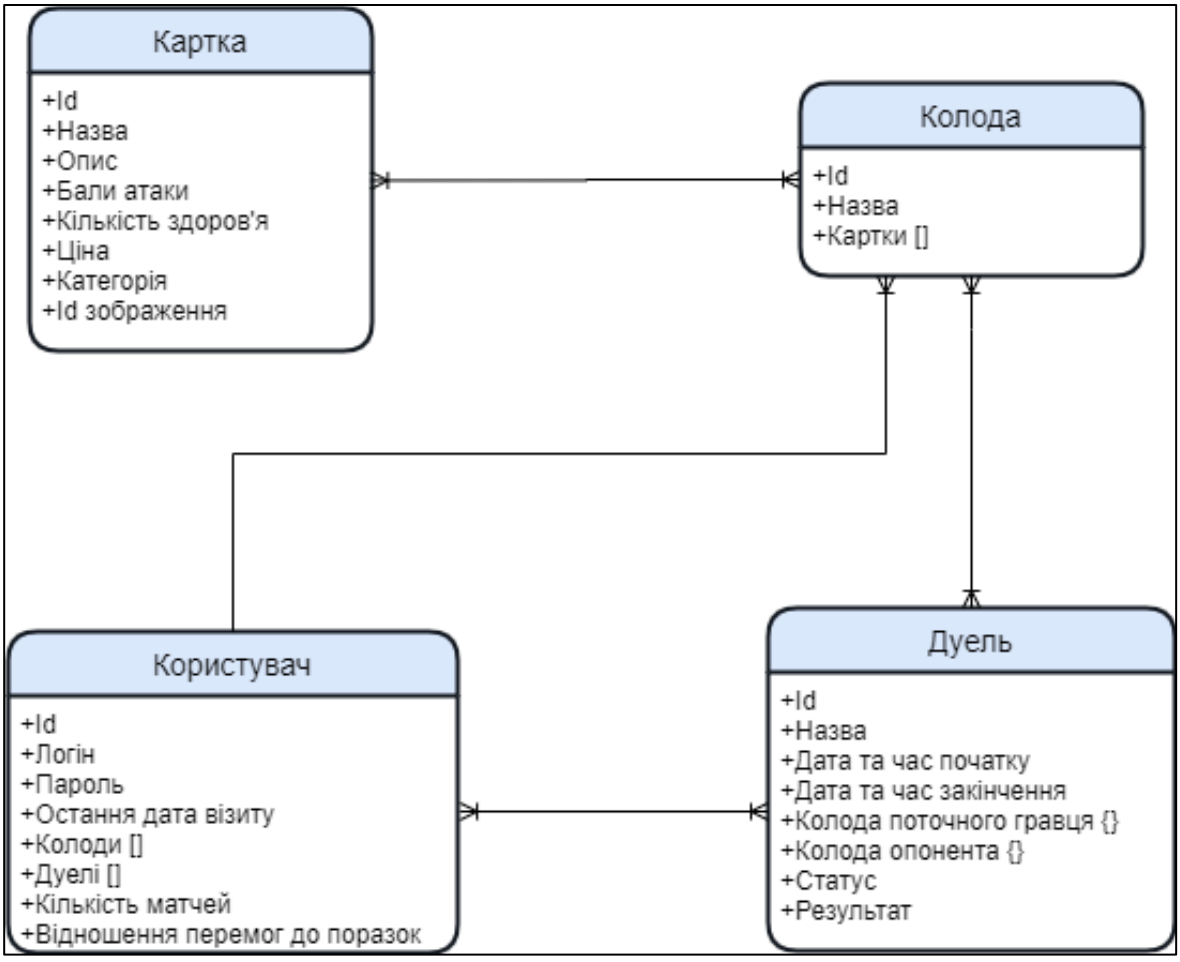

Рисунок 2.2 – ER-діаграма бази даних

На діаграмі окрім сутностей та їх атрибутів також позначені зв’язки, які присутні між цими сутностями.

3.2 Аналіз існуючих підходів до проектування архітектур

Підходи до проектування архітектури ігрових мобільних застосунків умовно можна поділити на два типи: підхід з окремою архітектурою до геймплея та мета-гейму з інтерфейсом користувача, та підхід з однаковою архітектурою для усіх частин гри. Таке розділення обумовлено специфікою роботи та вимог до різних частин гри.

Ігровий процес (геймплей, core) — термін, яким називають особливості взаємодії людини із відеогрою. Ці особливості створюються за допомогою правил, завдань та способів їх вирішення, які пропонує гра. Тобто це “серце” гри, яке закладє правила взаємодії гравця з додатком, ці правила можуть бути дуже спеціфічними та значно відрізнятися від того, як працює інша частина гри. Як правило, геймплей – це логічна частіна гри, яка схована від гравця за графічним інтерфейсом, але відркита для геймдизайнера, тому вона повинна бути прорахованою та зручною для розширення.

Іншою частиною гри є інтерфейс користувача, який дає змогу потрапити у геймплей, та відкріває взаємодію з так званим мета-геймом. Метаігра (мета) - в комп'ютерних іграх являє собою поняття, що описує активності поза грою, але що впливають на її ігровий процес. Немає єдиного визначення мета-ігри. Під метагрою можна розуміти запровадження розробниками додаткових елементів (досягнень, доповнень та інших), які є частиною центрального ігрового процесу. Ця частина гри більш схожа на класичні мобільні застосунки, тому для неї є популярним підхід з сімейством MV*-архітектур.

На перший погляд, перший підхід може здатися важким та непотрібним, але якщо застосунок має складну бізнес-логіку або велику кількість компонентів – підхід з розділенням архітектур полегшує розширення системи, реалізацію та тестування функціоналу.

Найбільш популярними у розробці мобільних застосунків є наступні архітектурні підходи:

- MVC (Model-View-Controller);
- MVP (Model-View-Presenter);
- MVVM (Model-View-ViewModel);
- HMVC (Hierarchical Model–View–Controller);
- компонентній підхід (компонентно-орієнтовна);
- ECS (Entity-Component-System).

Шаблон проектування MVC розбиває програмне забезпечення на три основні категорії: моделі, представлення та контролери [17].

Моделі (Model) зберігають значення. У прикладі, пов’язаному з грою, модель гравця зберігає його загальні та поточні очки здоров’я, швидкість, силу атаки тощо. Моделі не виконують жодних обчислень чи рішень, а також не взаємодіють з іншими блоками чи компонентами. Вони просто обробляють дані.

Подання (View) — це інтерфейс тріо MVC. Він обробляє вхідні дані та події та відповідає за графічне представлення або об’єкти в моїй концепції. Представлення також є єдиною частиною блоку MVC, яка взаємодіє з іншими сценаріями за межами самого блоку.

Нарешті, контролер (Controller) є мозком пристрою. Він обробляє інформацію, робить з нею розрахунки та приймає рішення і відправляє її в модель або повертає до представлення.

Зв'язок між трьома частинами MVC також суворо регламентується. Представлення повинно мати доступ до обох, щоб надсилати вхідні дані контролеру та перевіряти збережені значення в моделі та, наприклад, відображати їх. І модель, і контролер не мають посилань на представлення. Таким чином, конструкція залишається гнучкою та модульною.

Принцип роботи якої зображено на рисунку 2.3:

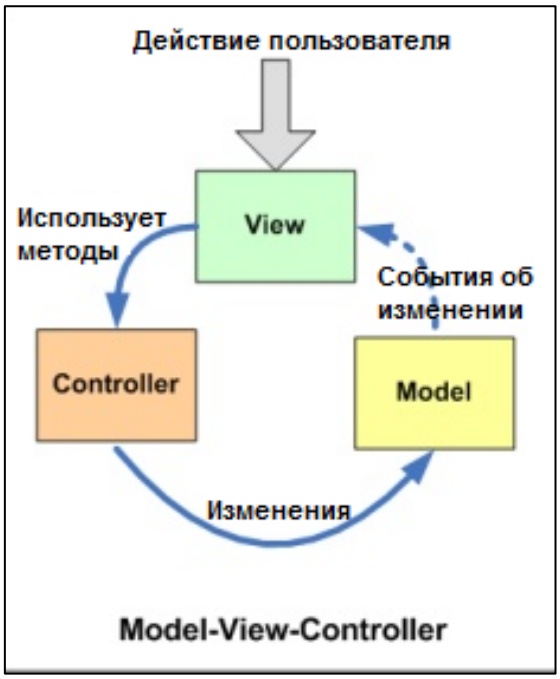

Рисунок 2.3 - Архітектура MVC

Основна ідея цього патерну в тому, що і контролер і уявлення залежать від моделі, але модель не залежить від цих двох компонентів.

Для більш чіткого розподілу обов'язків між компонентами було адаптовано інший архітектурний підхід – MVP, принцип роботи якої зображено на рисунку 2.4:

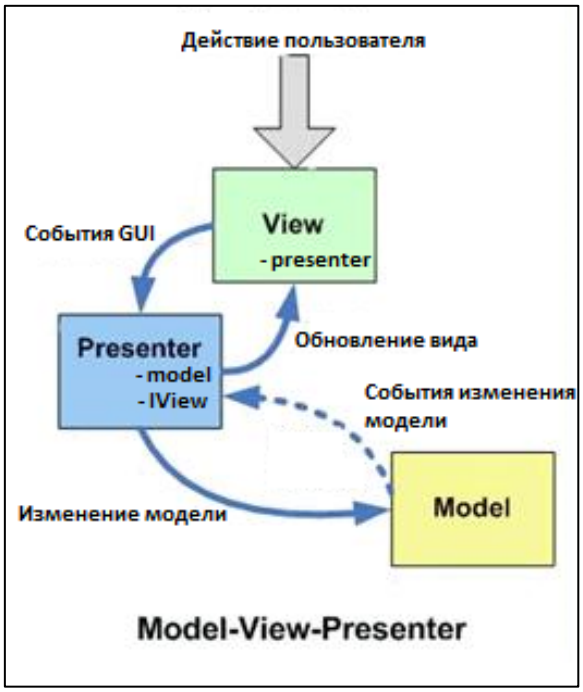

Рисунок 2.4- Архітектура MVP

В свій час MVP підхід дозволяє створювати абстракцію уявлення. Для цього необхідно виділити інтерфейс уявлення з певним набором властивостей та методів. Презентер (Presenter), у свою чергу, отримує посилання на реалізацію інтерфейсу, підписується на події уявлення та за запитом змінює модель.

Кожне уявлення має реалізовувати відповідний інтерфейс. Інтерфейс представлення визначає набір функцій та подій, необхідних для взаємодії з користувачем. Презентер повинен мати посилання реалізацію відповідного інтерфейсу, яку зазвичай передають у конструкторі. Логіка уявлення повинна мати посилання на екземпляр презентера. Всі події уявлення передаються для обробки презентатором і практично ніколи не обробляються логікою уявлення (в т.ч. створення інших уявлень).

Слід також зауважіті, що у MVP архітектурі шари Model та View ізольовані один від одного, на відміну від MVC, що призводить до значно більшої чіткості розподілу відповідальності між сутностями.

MVVM підхід дозволяє пов'язувати елементи уявлення з властивостями та подіями View-моделі [18]. Можна стверджувати, що кожен шар цього патерну не знає існування іншого шару. На відміну від MVC/MVP, MVVM забезпечує вищий рівень абстрагування UI від логіки та даних, з якими UI працює (рисунок 2.5).

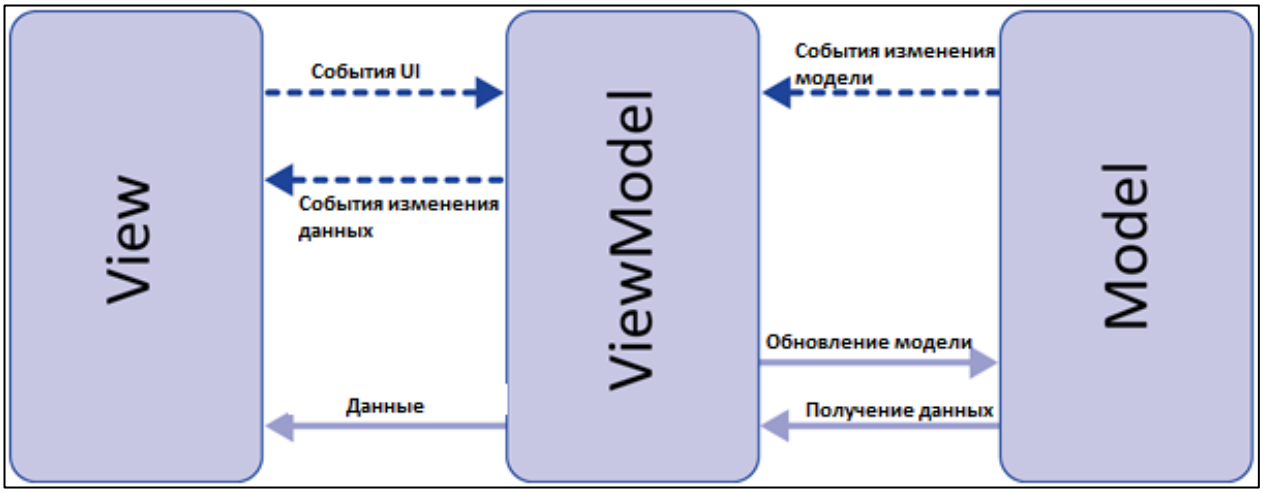

Рисунок 2.5- Архітектура MVVM

З використанням цього патерну, уявлення не реалізує відповідний інтерфейс (IView). Подання повинно мати посилання джерело даних (DataContex), яким у разі є View-модель. Елементи уявлення пов'язані (Bind) з відповідними властивостями та подіями View-моделі. У свою чергу View-модель реалізує спеціальний інтерфейс, який використовується для автоматичного оновлення елементів представлення.

Реалізація MVVM і MVP-патернів, на перший погляд, виглядає досить схожою. Однак, для MVVM зв'язування подання з View-моделлю здійснюється автоматично, а для MVP необхідно програмувати. MVC, мабуть, має більше можливостей з управління поданням.

HMVC (Hierarchical model–view–controller) — Ієрархічні Модель-Вид-Контролер, одне з розширень архітектурного паттерна MVC, що дозволяє вирішити деякі проблеми масштабованості програм, що мають класичну MVC-архітектуру.

Згідно з парадигмою HMVC [19], кожна окрема MVC тріада використовується як шар в ієрархічній структурі. При цьому, кожна тріада в цій ієрархії незалежна від інших, і може звернутися до контролера іншої тріади. Принцип роботи якої зображено на рисунку 2.6.

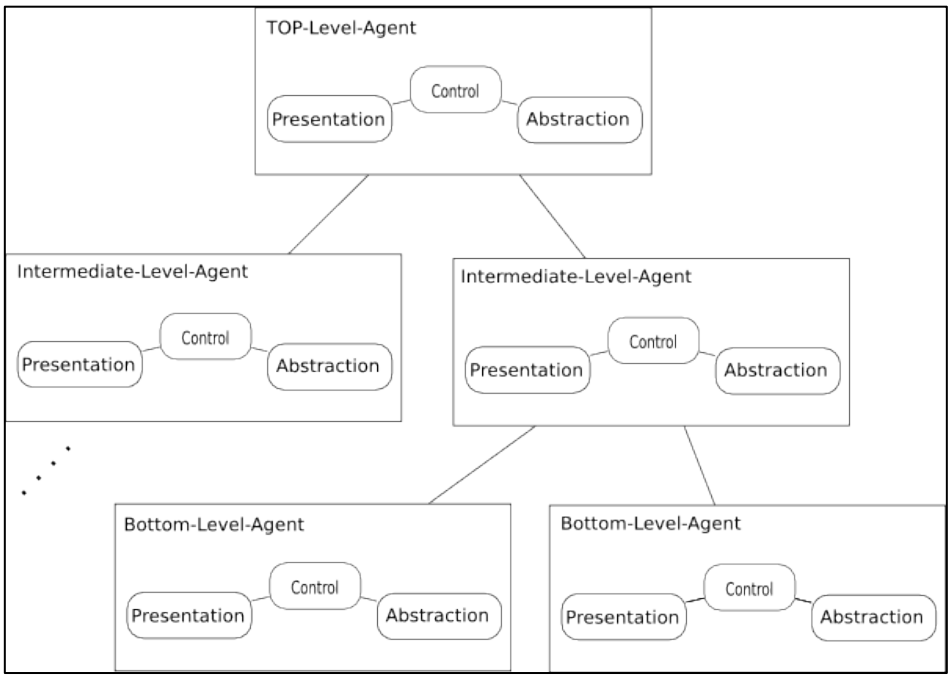

Рисунок 2.6- Архітектура HMVC

Такий підхід істотно полегшує і прискорює розробку великих програм, полегшує їх подальшу підтримку та масштабування, сприяє повторному використанню коду.

Спільною ідеєю MVP, MVVM, HMVC архітектурних підходів є одна особливість - виділення окремого презентаційного шару (presentation layer) та шару відображення (view layer), що робить модель незалежною від конкретного відображення. Таке розподілення відповідає принципу Single Responsibility, а також підвищує можливість тестування коду, а також зменшує кількість помилок, які можна допустити у великих проектах. Так у MVP за презентаційний шар відповідає сутність Presenter, а у MVVM – ViewModel.

Щодо компонентного підходу, то Unity робить процес розробки гри досить простим, даючи ряд зрозумілих кроків, щоб створити будь-який бажаний сценарій гри. Unity не прив'язаний до якогось певного ігрового жанру, це «чисте полотно», на якому можна створити будь-яку бажану гру. Основною концепцією Unity є використання Game Object (GO), це дає можливість розбити свою гру на легко керовані об'єкти, які, в свою чергу, складаються з безлічі компонентів (Components). Це э так звною компонентно-оріентованною архітектурою. Створюючи окремі ігрові об'єкти і розширюючи їх функціональність за допомогою додавання різних компонентів, стає можливо нескінченно розширювати і ускладнювати гру.

Компоненти (components) мають різне призначення – вони можуть впливати на поведінку, зовнішній вигляд і багато інших функцій об'єктів в грі. Прикріплюючи компонент до об'єкта можна відразу ж скористатися можливостями, що надаються цим компонентом. Unity надає безліч компонентів різного призначення.

Для створення власних компонентів потрібно створювати власні скрипти. Unity надає можливість використовувати для написання скриптів мову програмування C#. Скрипти – це файли, в яких описується логіка поведінки об'єктів на сцені. MonoBehaviour – базовий клас, з якого походить кожен сценарій

Unity. За таким підходом дані та логіка об'єднані в компоненті MonoBehaviour, а GameObject (якщо порівнювати з сутністю в ECS) має додаткові дані - ім'я, місце в ієрархії та ін.

Але в даний час стає дуже популярним інший підхід до побудування ігрових додатків. Це так званий паттерн ECS. Основна ціль ECS — розподіл різних проблем і завдань між сутностями (Entities), компонентами (Components) і системами (Systems). Це три основні поняття цього шаблону. Сутність це контейнер для компонентів. Компонент — це невеликий об'єкт, який не володіє ніякою логікою. Зазвичай компонент містить одне чи два поля. В ідеалі це об'єкти з простою структурою даних. Кожен тип компонента можна прикріпити до сутності, щоб дати їй щось на зразок властивості. Системи містять в собі логіку, наприклад, переміщення чи запуск анімації. Системи працюють с сутностями, які мають потрібний для неї набір компоненті. Їх взаємодію можна побачити на рисунку 2.7.

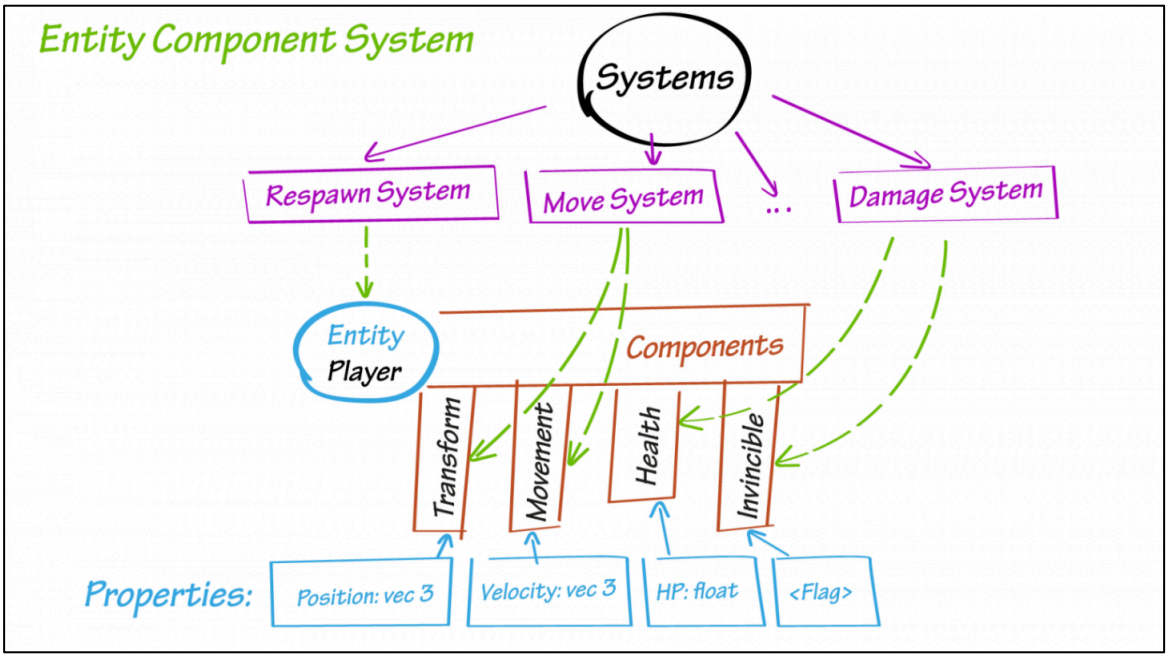

Рисунок 2.7 - Архітектура ECS

Цей підхід на перший погляд схожий на компонентний, але основна відмінність між компонентною системою Unity та ECS – логіка в ECS обов'язково має бути відокремлена від даних. Це дозволяє дуже гнучко змінювати логіку

(навіть видаляти/додавати її), не ламаючи дані. А крім цього, ще й утілізується механізм кешу процесору через механізм data locality [20].

Кожна з зазначеніх архітектурних реалізацій є похідною від певного підходу та наслідує усі переваги та недоліки від нього. Але окрім цього кожна архітектура має власні риси, які слід відзначити.

Архітектура MVC була історічно першою та заклала фундаментальні концепції для своїх нащадків, але в свій час вона не мала чіткого розподілу на зони відповідальності, а також найменш відповідала принципам чистої архітектури, які були створені “Дядюшкою Бобом”. Через те, що недоліки цього старого підходу вже давно відомі програмістській спільноті, а інші архітектури обрали головні плюси MVC та позбулися недоліків оригінальної архітектури, тому немає сенсу розглядати цей підхід взагалі.

Як вже було зазначено раніше, реалізація MVVM і MVP-патернів, на перший погляд, виглядає досить схожою. Однак, для MVVM зв'язування подання з View-моделлю здійснюється автоматично, а для MVP необхідно програмувати. Тому MVP використовується в ситуації, коли неможливе зв'язування даних (не можна використовувати Binding), в той час MVVM використовується у ситуації, коли можливе зв'язування даних без необхідності введення спеціальних інтерфейсів представлення (тобто відсутня необхідність реалізовувати IView).

Але шаблон HMVC розкладає рівень клієнта на ієрархію батьківських і дочірніх шарів MVC. На кожному рівні може бути одна з реалізацій MVC-паттерну (тобто MVC, MVP або MVVM), що є дуже масштабованим та гнучкім підходом та використовує окрмий потрібний шаблон залежачі від контексту задачі (рис. 2.8).

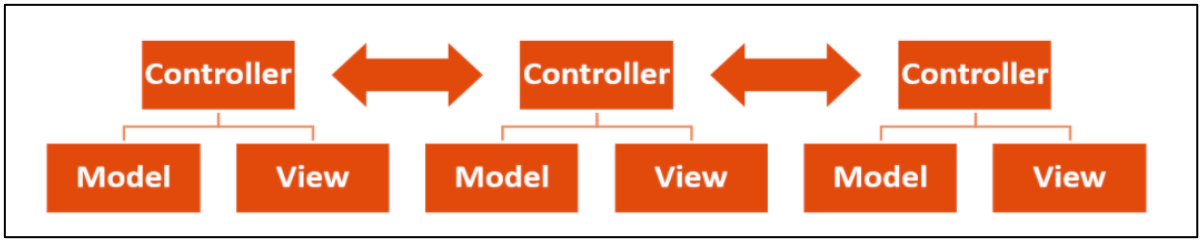

Рисунок 2.8- Архітектура HMVC

Підхід створює досить складний рівень клієнта.

Деякі з ключових переваг використання HMVC підкреслюють переваги ООП, а саме:

– зменшує залежності між різними частинами програми;

– заохочує повторне використання коду, компонентів і модулів;

– збільшує розширюваність, одночасно полегшуючи ремонтопридатність.

Таким чином, така ієрархічна організація коду є дуже привабливаю для великих ігор, бо вона дуже гнучка, та її легко розширювати.

Щодо порівняння компонентно-орієнтованої архітектури з архітектурою ECS, то використовуючи компонентний підхід, рано чи пізно можна наталкнутися на те, що ієрархія об’єктів на сцені має велікій оверхед, а якісь сутності не обов’язково повинні бути GameObject’ами. А якщо в грі велике число компонентів і апдейтів у них (а якщо це велика гра, то це трапиться обов’язково), то все стає сильно повільним.

І найважливіше в контексті процесорів це те, що всі компоненти при такому підході та всі дані розкидані в пам'яті, що ламає використання кешу процесора. Схематічно така модель зображена на рисунку 2.9.

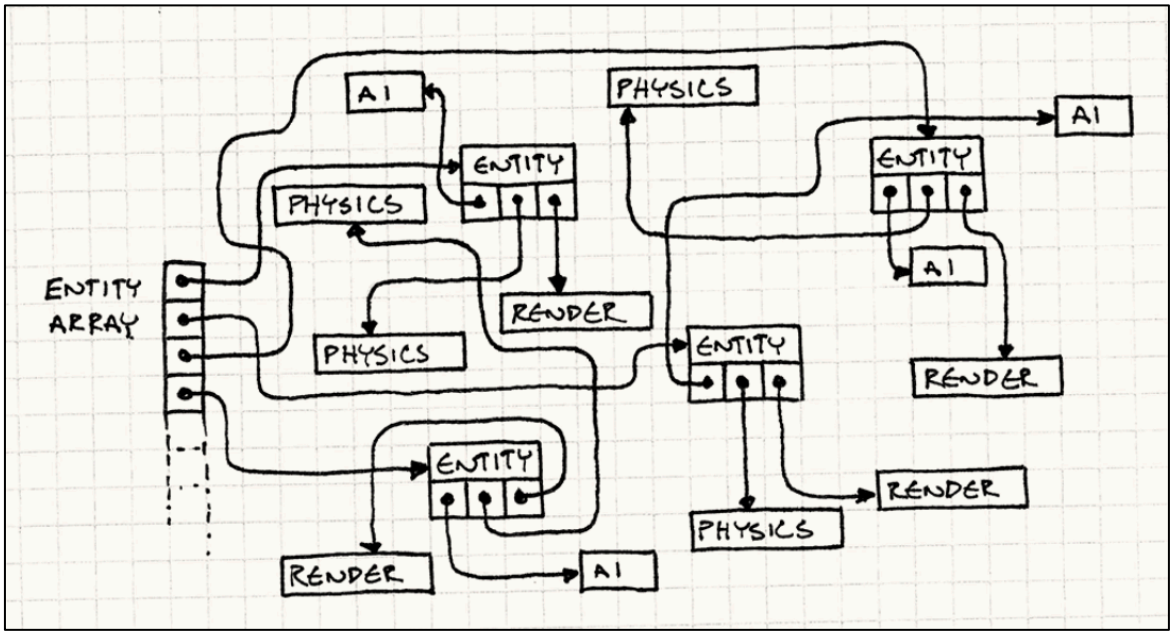

Рисунок 2.9 - Пам’ять при використанні компонентного підходу

В той час як при використанні ECS, пам’ять виглядятиме таким чином, як зображено на рисунку 2.10:

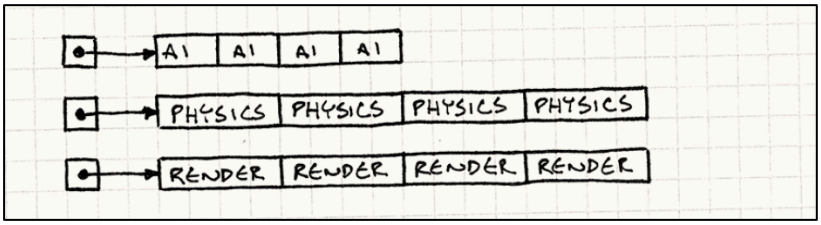

Рисунок 2.10 - Пам’ять при використанні ECS-підходу

Це дає значну перевагу для використання сучасних процесорів. Та крім цього такій підхід є простішим у використанні, незалежить від конкретно фреймворку та інструментів, є дуже зручним для масштабування ігри та роботи геймдізайнера.

2.3 Вибір підходів до проектування архітектур для подальшого дослідження

Як вже було відзначено, існує два основних типи проектування архітектур для застосунків с користувацьким інтерфейсом:

- підхід з окремою архітектурою до геймплея та мета-гейму;
- підхід з однаковою архітектурою для усіх частин гри.

Кожен з цих підходів має як свої переваги, так і недоліки. Серед переваг першого типу підходу можна відмітити наступні:

- масштабованість;
- можливість інтеграції будь-якої бізнес-логіки;
- незалежність від жанру гри;
- простота тестування;
- незалежність гейм-плея та мета-гейму;
- простота розширюваності.

Але серед недоліків першого підходу можна виділити наступні:

– велика кількість коду для реалізації базової архітектури;
– необхідність у створенні або використанні бібліотек.

Що стосується другого типу підходу, то можна виділити наступні переваги:

– невелика (порівняно з першим підходом) кількість коду;
– незалежність від жанру гри;
– швидкість та простота розробки.

Але цей підхід також має власні недоліки:

– сильна залежність від бізнес-логіки та жанру;
– залежність гейм-плея та мета-гейму;
– незалежність від жанру гри;
– складність тестування;
– складність розширюваності;
– складність масштабованості.

Тому слід відмітити, що як першій так і другій підхід мають свої сценарії використання. Другій підхід є більш доцільним для використання у невеликих застосунках, де критичною рисою є швидкість розробки (наприклад, ігри в жанрі hypercasual чи clickers), в той час як першій підхід більш доцільно використувати у великих застосунках, насичених бізнес-логікою (наприклад, ігри в жанрі Action, RPG, CCG та ін.).

Треба врахувати всі перелічені обмеження щодо вибору архітектури, а саме - архітектура повинна бути зручною при співпраці з гейм-дізайнером, повинна бути універсальною та стандартізованою, мінімізуючи витрати на супровід та розширення програмного коду. Окрім того, вона не повинна залежати від того, яку саме модель взаємодії з сервером було обрано, на якому ігровому рушії було зроблено гру, на якій платформі гра була запущена, як саме організована база даних та чи є в грі штучній інтелект. Окрім цих обмежень, існують ще обмеження апаратного плану слабких девайсів: кількість оперативної пам’яті, графічна API, що підтрімується, кількість ядер процесору, архітектура процесору, підтрімка SIMD-команд процесором та інші оптимізаційни пітання.

З усіх вищеописаних критеріїв для вибору архітектури гри, треба взяти тільки ті, що найбільше стосуються розробки мобільних ігр в жанрі "Collectible Card Game", а саме:

- простота розширюваності ігрових фіч (Ext);
- удобство співпраці з гейм-дизайнером (Gd) ;
- стандартизація розробки (Std);
- агностична до серверу/рушія/бази даних (Agn);
- простота тестування (Test);
- підтримка слабких пристроїв (Sup);
- відокремелення мета-гейму від геймплею (Sep).

Для наглядності порівняння архитектур наведемо таблічний опис переваг та недоліків існуючих архітектур в таблиці 2.1:

Таблиця 2.1 – Порівняння архітектур

	Ext	Gd	Std	Agn	Test	Sup	Sep
MVC	-	-	+	+	+	-	-
MVVM	-	-	+	+	+	-	-
MVP	-	-	+	+	-	-	-
HMVC	-	-	+	+	+	-	+
Компонентно-орієнтована	+	+	+	-	+	-	-
ECS	+	+	+	-	+	+	-
HMVC + ECS	+	+	+	-	+	+	+
HMVC + компонентна	+	+	+	-	+	-	+

Як бачимо з порівняльної таблиці, з усіх описаних архітектур, більш всього до реалізації комп'ютерної гри в жанрі "Collectible Card Game", підходить перший

описаний підхід з окремою архітектурою до геймплея та мета-гейму з інтерфейсом користувача. Таку гібрідну архітектуру можна реалізувати двома способами: поєднавши HMVC та ECS, а також поєднавши HMVC з компонентно-орієнтовною архітектурою. То ж було обрано дві архітектури для порівняння.

Порівняння архітектур - це перед усім емпіричне дослідження. Для того, щоб порівнювати декілька архітектур, потрібно проводити експериментальні дослідження.

Експериментальне дослідження проводиться у вигляді набору тестів. Для цього створюється середовище, до якого розробляються тести. Мета експерименту - довести наукову теорію з визначеними залежними і незалежними змінними.

2.4 Проектування обраних архітектур ПЗ

Як вже було зазначено вище, для дослідження було обрано підхід з гібрідними архітектурами: поєднання HMVC та ECS, а також поєднання HMVC з компонентно-орієнтовною архітектурою. Спільною частиною цих двох архітектур є HMVC частина, бо саме вона буде являти собою головний скелет гри. Розглянємо її детальніше та спроектуємо її зв’язок з ECS в першому випалку та з компонентно-орієнтовною архітектурою в другому.

Найбільш поширінішим підходом до реалізації HMVC архітектури є паттерн Controllers Tree (Дерево конролерів). Він дозволяє будувати менш заплутаний код, що заохочує писати розділені та ізольовані функції. Окрім того, дерево контролерів — це архітектура, орієнтована на код, вона спрощує рефакторинг і підвищує гнучкість, а ще він є дружнім до DI (Dependency Injection) [21].

На рисунку 2.11 можна побачити як саме співвідносяться головні компоненти паттерна Controllers Tree:

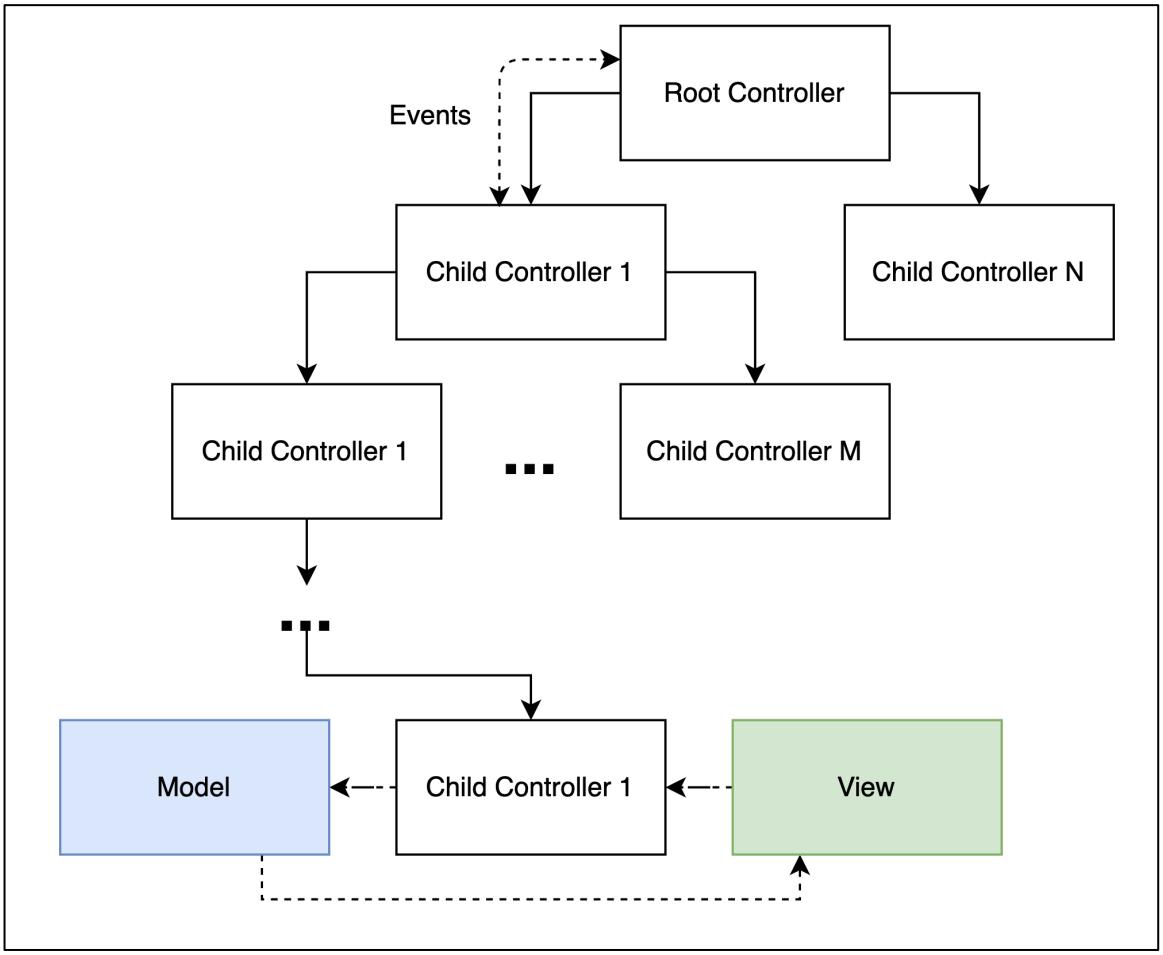

Рисунок 2.11 - Архітектура Controllers Tree

За своєю природою цей патерн рекурсівен, бо на кожному рівні можна створювати контролери-нащадки та створювати будь-який рівень вкладенності ієрархії.

На кожному рінві контролер може роботати з ресурсами гри, з мережею, з ігровими сервісамі, працювати з моделями даних та регулювати діяльність представлення.

Для того, щоб мати уніфікований вигляд для всіх контролерів, незважаючи на то, задля чого був створений цей конролер та за яке представлення він контролює, паттерн накладає певний перелік станів в яких може перебувати конролер.

Цей набір та послідовність станів зображено на рисунку 2.12:

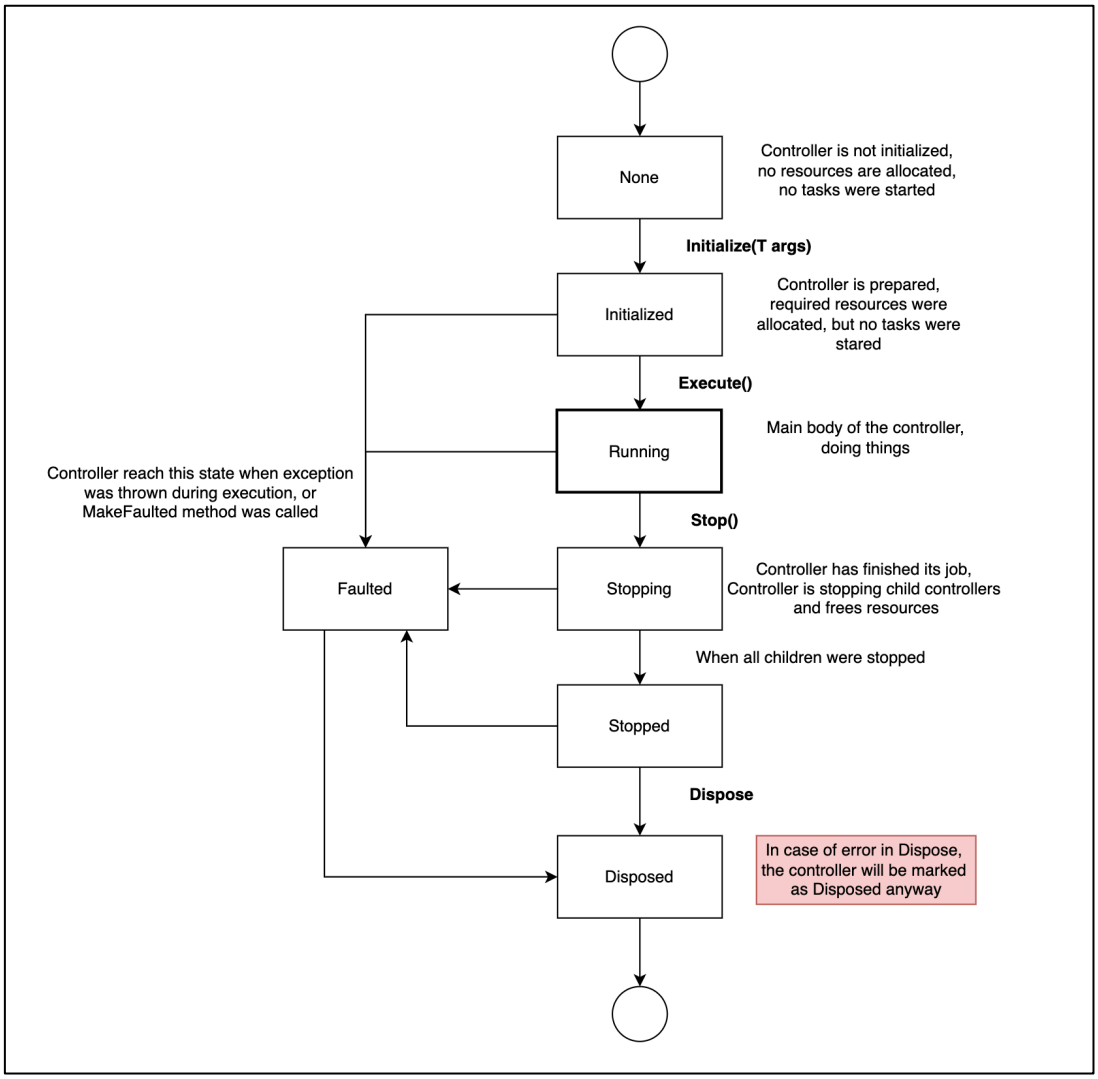

Рисунок 2.12 – Стани паттерну Controllers Tree

Як бачимо з рисунку, контролер може перебувати в одному з наступних станів:

– None – контролер створено, але він не ініціалізований, ресурси не виділені;

– Initialized – контролер ініціалізовано, усі ресурси підготовлені та готові до використання;

– Running – це основна частина контролера, тут треба розмістити всі головні завдання на виконання, наприклад відкриття спливаючого вікна, запуск дочірнього контролера або зв’язок із сервером;

– Stopping – контролер звільняє ресурси та зупиняє своїх дітей;

– Stopped – контролер завершив свою роботу та звільнив усі ресурси, але він все ще існує в дереві контролера;

– Disposed – контролер утилізується та видаляється з дерева контролерів.

– Faulted – під час виконання стався необроблений виняток, контролер зламаний.

Також цей паттерн накладає ряд обмежень на його використання. Це можна побачити на діаграмі послідовностей створення контролеру на рисунку 2.13:

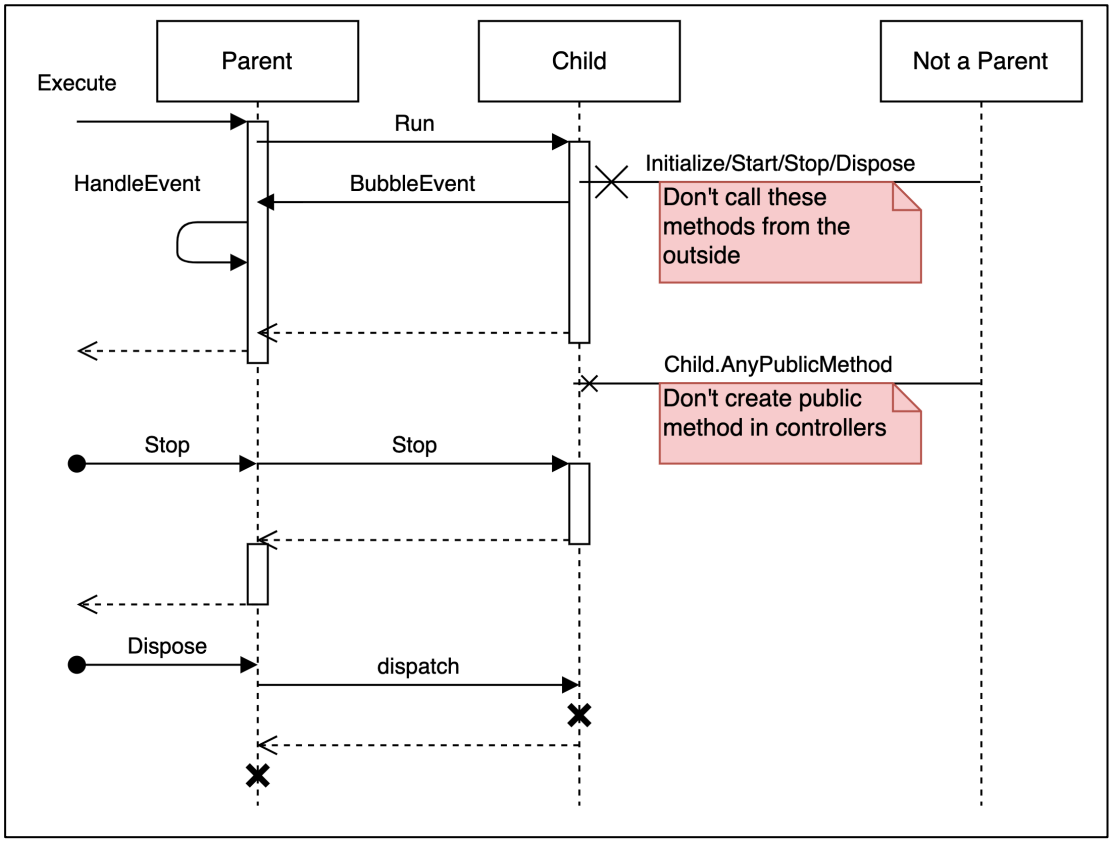

Рисунок 2.13 – Діаграма послідовностей створення контролеру

В ці обмеження входять такі пункти:

– усі контролери (крім RootController) мають бути створені всередині іншого контролера;

– неможна викликати методи контролера за межами дерева, контролер може викликати тільки власні методи або методи дітей;

– контролер займається зупинкою та утилізацією всіх дітей. Якщо під час зупинки або видалення дочірнього контролера виникли винятки, батьківський контролер має бути завершено правильно;

– контролер не повинен містити загальнодоступні методи;

– діти спілкуються з контролерами предків за допомогою повідомлень. Якщо безпосередній батько не може обробити повідомлення, повідомлення надсилатимуться далі, доки будь-який контролер не обробить повідомлення, або повідомлення не досягне RootController.

Але виникає питання, як саме можна поєднати Controllers Tree з ECS та з компонентною архітектурою. На високому рівні вирішення цієї проблеми як для архітектури з ECS, так і для архітектури з компонентнім підходом, є однаковим – буде створено окремий контролер, задачею якого стає управління життевим циклом та логікою гри. Таке співвідношення можна схематично зобразити на прікладі ECS архітектури на рисунку 2.14:

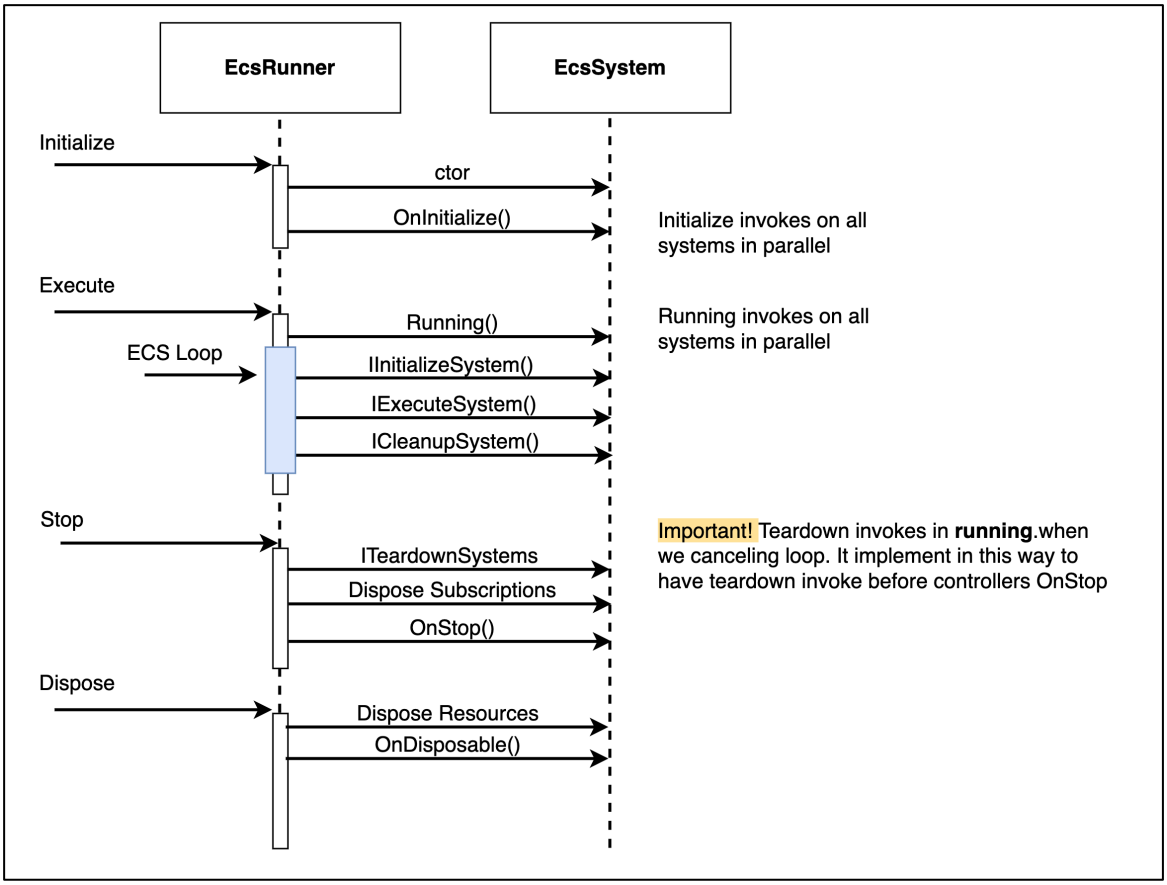

Рисунок 2.14 – Життевий цикл контролера для ECS архітектури

Контролером для ECS в даному випадку виступає EcsRunner, що фактично стає RootController для логіки усієї ECS-архітектури, яка схематично зображена як EcsSystem. Кожен стан контролеру накладається на життевий цикл ECS системи, що було названо як ECS Loop. У випадку з компонентною архітектурою принцип залишається той ж самий, тільки замість EcsSystem буде життевий цикл для компонентів.

Також слід зауважити, що сильною стороною цього паттерну є те, що ми можемо відкласти питання з’єднання Controllers Tree з ECS або компонентно-орієнтованною архітектурою до моменту конкретної реалізації, бо відповідь на це питання дуже залежіть від конкретних фреймворків, що будуть обрані для реалізації геймплея.

3 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

3.1 Вибір засобів програмної реалізації

Для розробки системи відбираються відповідні засоби, які будуть легкі у розробці, підтримувані, дадуть змогу у майбутньому легко масштабувати систему. Перед початком роботи необхідно обрати мови програмування, за допомогою яких будуть реалізовані частини системи, та систему управління базами даних, яка б задовольняла всі вимоги системи.

Для ігрового клієнту, що являє собою мобільний додаток, було обрано разробку за допомогою платформи Unity, до якої входить інтегроване середовище розробки та ігровий рушій Unity. Обрана технологія дозволяє розробляти ігрові додатки дуже швидко та порівняно легко. Ігровий рушій повністю пов'язаний із середовищем розробки. Це дозволяє випробовувати гру прямо в редакторі. При розробці на даній технології код пишеться на мові програмування C#.

3.2 Опис клієнтської розробки

Клієнтська частина ігрового додатку побудована за допомогою платформи Unity, до якої входить інтегроване середовище розробки та ігровий рушій Unity. Обрана технологія дозволяє розробляти ігрові додатки дуже швидко та порівняно легко, бо при розробці на даній технології код пишеться на мові програмування C#.

Unity робить процес розробки гри досить простим, даючи ряд зрозумілих кроків, щоб створити будь-який бажаний сценарій гри. Unity не прив'язаний до якогось певного ігрового жанру, це «чисте полотно», на якому можна створити будь-яку бажану гру. Основною концепцією Unity є використання Game Object (GO), це дає можливість розбити свою гру на легко керовані об'єкти, які, в свою

чергу, складаються з безлічі компонентів (Component). Це э так звною компонентно-оріентованною архітектурою.

Але в даний час стає дуже популярним інший підход до побудування ігрових додатків. Це так званий паттерн ECS. Основна ціль ECS — розподіл різних проблем і завдань між сутностями (Entities), компонентами (Components) і системами (Systems). Це три основні поняття цього шаблону. Сутність це контейнер для компонентів. Компонент — це невеликий об'єкт, який не володіє ніякою логікою. Зазвичай компонент містить одне чи два поля. В ідеалі це об'єкти з простою структурою даних. Кожен тип компонента можна прикріпити до сутності, щоб дати їй щось на зразок властивості. Системи містять в собі логіку, наприклад, переміщення чи запуск анімації. Системи працюють с сутностями, які мають потрібний для неї набір компонентів.

Створюючи окремі ігрові об'єкти і розширюючи їх функціональність за допомогою додавання різних компонентів, стає можливо нескінченно розширювати і ускладнювати гру. У компонентів, в свою чергу, є налаштування (змінні об'єкта), за допомогою яких здійснюється управління компонентами. Змінюючи ці змінні, можна отримати повний контроль над впливом цього компонента на об'єкт.

Крім того кожна гра складається не тільки з коду, а й з інших ресурсів. Ресурси (Assets) – будівельні / складові блоки всіх проектів Unity, що можна побачити на рисунку 3.1:

Ігровий рушій посилається на файли зображень, 3D-моделей, звуків і т.д., які будуть використовуватися, при створенні гри, в якості ресурсів (assets). Саме тому в будь-якій папці проекту Unity, є підкаталог, де зберігаються всі файли ресурсів, з ім'ям Assets.

Але ці об'єкти десь потрібно зберігати та розміщувати, щоб гравець бачив її під час гри. Для його використовуються сцени. У Unity сцена (scene) – це абстракція, про яку можна думати, як про ігровий рівень або область ігрового контенту (наприклад, меню).

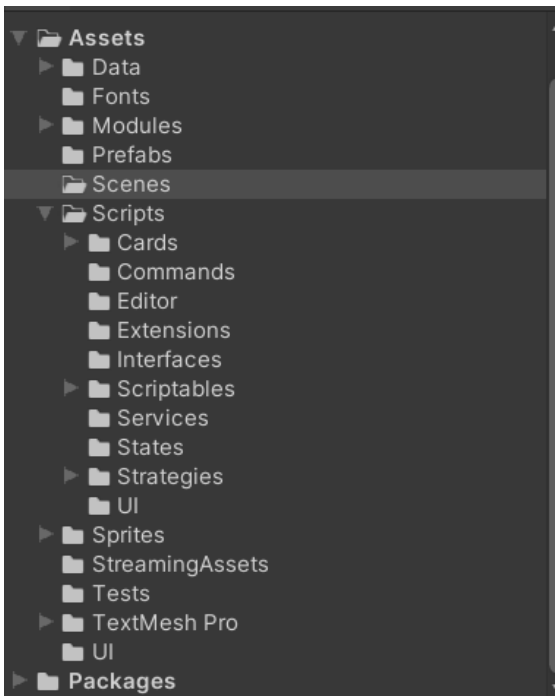

Рисунок 3.1 - Структура папки Assets

Список усіх сцен гри можна побачити на рисунку 3.2. Створюючи гру з безліччю сцен, з'являється можливість розподіляти час завантаження і тестувати різні частини гри окремо.

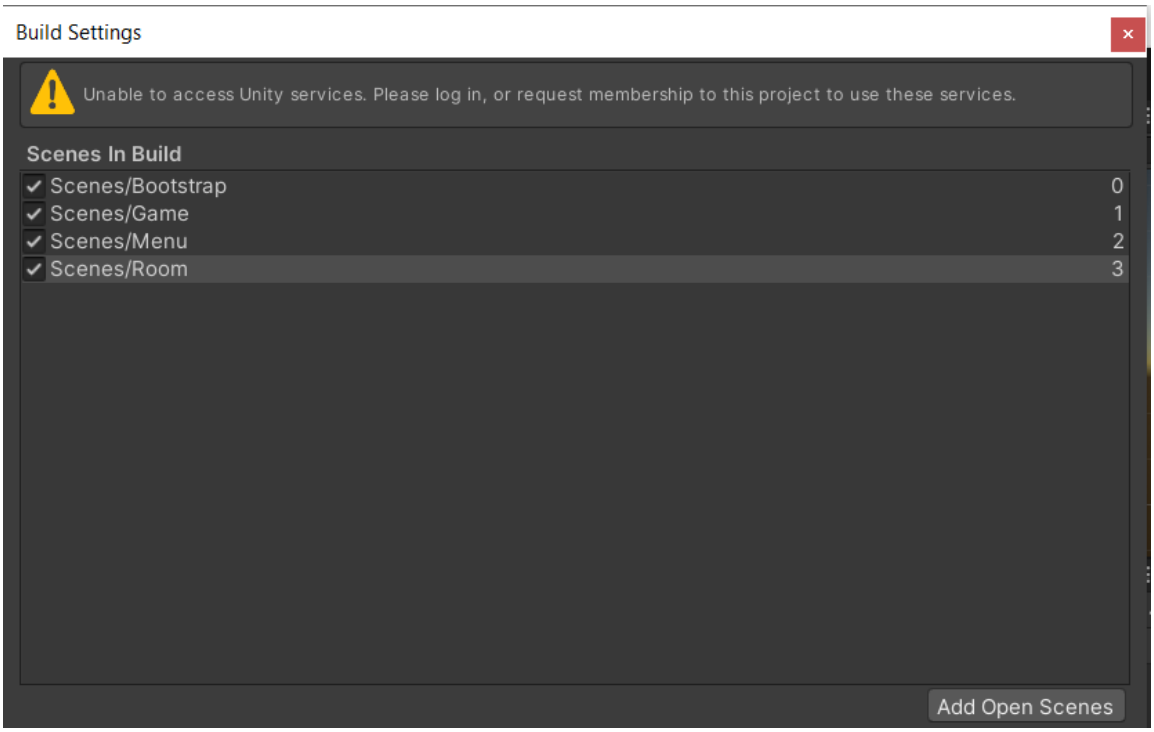

Рисунок 3.2 – Сцени у грі «Card Battle Arena»

Коли ресурс (asset) використовується в сцені гри, він ставати Ігровим Об'єктом (Game Object) – в термінології Unity. Всі об'єкти (GameObjects), спочатку, мають хоча б один компонент (component) – Transform (Перетворення), що можна побачити на рисунку 3.3.

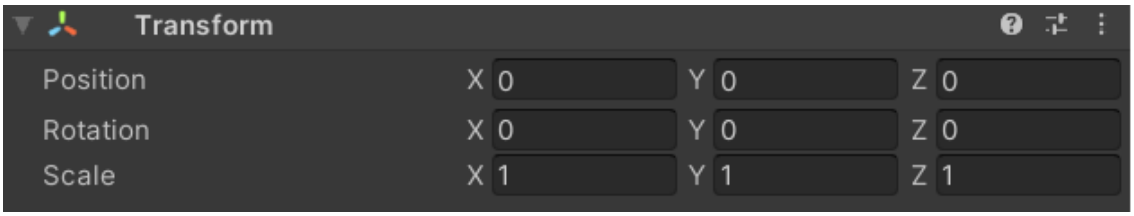

Рисунок 3.3 – Компонент Transform

Transform говорить движку Unity про становище (position), поворот (rotation) і масштаб (scale) об'єкта, описаних в X, Y, Z координатах (або у випадку з масштабом, розмірність). До того ж, до цього компоненту можна звернутися з скрипта, що б встановити його положення (position), поворот (rotation) або масштаб (scale). Додаючи різні компоненти до об'єкта, ми можемо значно розширити його функціонал, для забезпечення реалізації будь-якого бажаного сценарію гри.

Компоненти (components) мають різне призначення – вони можуть впливати на поведінку, зовнішній вигляд і багато інших функцій об'єктів в грі. Прикріплюючи компонент до об'єкта можна відразу ж скористатися можливостями, що надаються цим компонентом. Unity надає безліч компонентів різного призначення. Для забезпечення інтерактивності різних ігрових елементів, треба використовувати скрипти, які розглядаються Unity як компоненти.

Наприклад, у карточній грі «Card Battle Arena» головним ресурсом є картка, що можна побачити на рисунку 3.4

Створення такого вигляду ресурсу та всього інтерфейсу гри стало можливим завдяки існування вбудованих компонентів для побудови інтерфейсу користувача: Canvas, Image, Mask, Text, Outline, RectTransform та інших.

Але для створення власних компонентів потрібно створювати власні скрипти. Unity надає можливість використовувати для написання скриптів мову програмування C#. Скрипти – це файли, в яких описується логіка поведінки

об'єктів на сцені. MonoBehaviour – базовий клас, з якого походить кожен сценарій Unity.

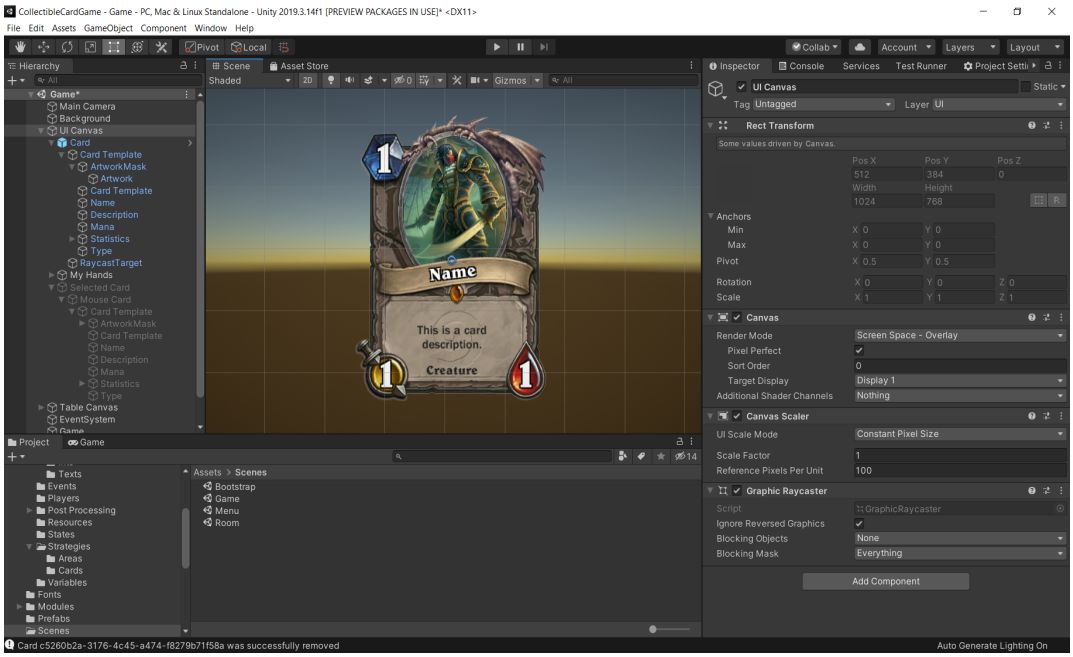

Рисунок 3.4 – Ресурс картки

Кожна модель картки Card з цього коду являє собою ScriptableObject. ScriptableObject – це контейнер даних, який можна використовувати для збереження великої кількості даних, незалежно від екземплярів класу. Один з головних випадків використання ScriptableObjects – зменшення використання пам'яті проекту, уникаючи копій значень. Як і MonoBehaviours, ScriptableObjects походять від базового об'єкту Unity, але, на відміну від MonoBehaviours, їх не можна приєднувати до GameObject. Натомість зберігати їх можливо лише як ресурси у своєму проекті.

Дані, які зберігаються з інструментів редактора до ScriptableObjects як ресурсу, записуються на диск і зберігаються між сесіями. Це дало змогу значно спростити створення карток та використовувати їх під час гри, це можна побачиті на рисунку 3.5.

Рисунок 3.5 – Автоматизація створення карток

Також було створено власне розширення редактора. Він дозволяє створювати нові картки у редакторі та автоматично писати дані про них у клауд-базу, що дає користувачам синхронізуватися з оновленими даними не виходячи з гри. В основі цього рішення лежить використання трьох класів, таких як, ScriptableObject (зберігання даних), CustomEditor (додатковий функсціонал інспектора для Scriptable Object) та UIElements. UIElements замінює поточну систему GUI-редактора Unity (IMGUI), що була введена в Unity 2019.1. Можна комбінувати HTML для Unity "UXML" та таблиці стилів для Unity "USS", щоб створити графічний інтерфейс Unity так, як ніби створюється веб-сторінку з HTML та CSS.

3.3 Опис обраних фреймворків та бібліотек

Окрім різноманітного базового функціоналу, що предаставляє Unity для розробки проектів, будь-яка гра потребує той чі інший додатковий функціонал, що може бути представленний завдяки набору фреймворків та бібліотек. Важливо

зазначити, що описані нижче технології будуть використовуватися в реалізаціях різних архітектур, що дає можливість спростити та стандартизувати деякий спільний функціанал, такий як, наприклад, асінхронні операції або работа з ресурсами проєкта.

Для спрощення роботи з ресурсами в Unity було використано нову систему під назвою Addressables (або Addressable Asset System) [22]. Усі ресурси карток було додано до цієї системи, як можна побачити на рисунку 3.6:

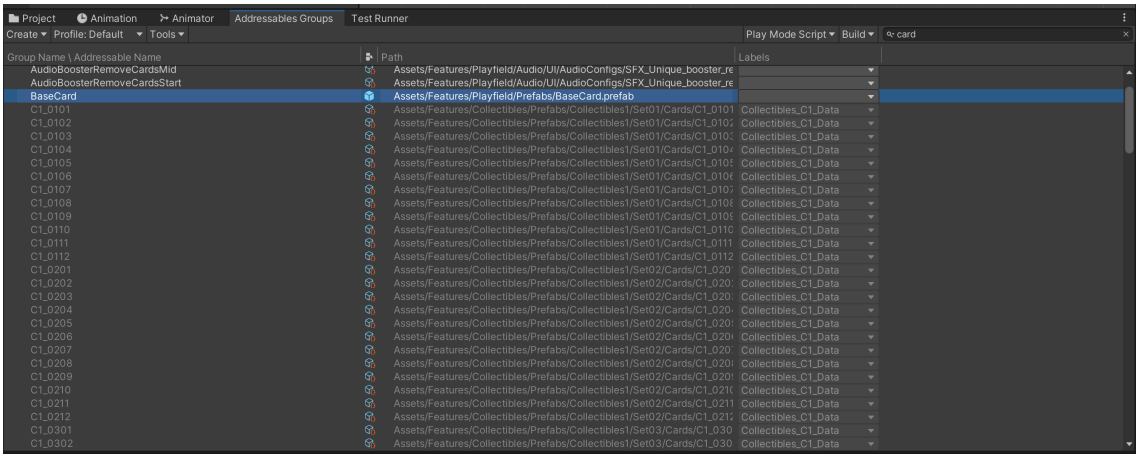

Рисунок 3.6 – Використання Addressables в проєкті

Традиційні засоби структурування ігрових асетів ускладнюють ефективне завантаження контенту. Система Addressables скорочує цикли ітерацій, дозволяючи приділяти більше часу розробці, кодуванню та тестуванню програми. Бо вона забезпечує простий спосіб завантаження активів за «адресою» (або ключом).

Основні переваги такої системи:

– управління залежностями: система повертає всі залежності запитуваного асету, щоб усі сітки, шейдери, анімації тощо завантажувалися, перш ніж повернути вміст асету;

– управління пам’яттю: система вивантажує асети, а також завантажує їх, автоматично підраховуючи посилання та надаючи надійний профайлер, який допомагає виявити потенційні проблеми з пам’яттю;

– упакування контентом: оскільки система відображає та розуміє складні ланцюжки залежностей, вона дозволяє ефективно пакувати пакети, навіть під час переміщення чи перейменування ресурсів. Можна легко підготувати асети як для локального, так і для віддаленого розгортання, щоб підтримувати завантажуваний вміст і зменшити розмір програми;

Ця система використовує асинхронне завантаження для підтримки завантаження з будь-якого місця з будь-якою колекцією залежностей. Але стандартні засоби для роботи з асинхронними операціями є слабким місцем для Unity. Серед вбудованих засобів роботи з асинхроними операціями є стандартна бібліотека для C# TPL, головним об'єктом для якої є Task, та вбудовані в Unity сопрограми, що називаються Coroutine. Бібліотека UniTask [23] є альтернативою цим підходам, вона використовує новий функціонал C# - структурні Task і кастомний AsyncMethodBuilder для досягнення zero allocation по пам'яті. Також UniTask не використовує ExecutionContext і SynchronizationContext, напротивагу звичайному Task, що дозволяє отримати високу продуктивність в Unity, так як виключає накладні витрати на переключення контекстів. Unity-спільнотою було проведено багато експеріментів [24] з порівнянням цих підходів, та саме UniTask виявився найкращім підходом на данний момент. Результати порівняння цих підходів можна побачити на рисунку 3.7:

Так як швидкодія гри є важливим критерієм проєкту, то саме тому було використовано дуже легку з точки зору пам'яті бібліотеку UniTask, яка до цього всього ще й має вбудованну підтрмку системи Addressables.

Також важливим елементом технічного стеку проєкту є DI-бібліотека Zeject, яка закладає базу для архітектур проєкта та спрощує розробку. Zeject — це легкий високопродуктивний фреймворк для ін'єкції залежностей (dependency injection), створена спеціально для Unity (однак її можна використовувати і за межами Unity). Ця бібліотека використовується для того, щоб перетворити програму на набір слабо пов'язаних частин із дуже сегментованими обов'язками. Zeject може потім склеювати частини разом у багатьох різних конфігураціях, щоб дозволити легко

писати, повторно використовувати, реорганізувати та тестувати код масштабованим і надзвичайно гнучким способом [25].

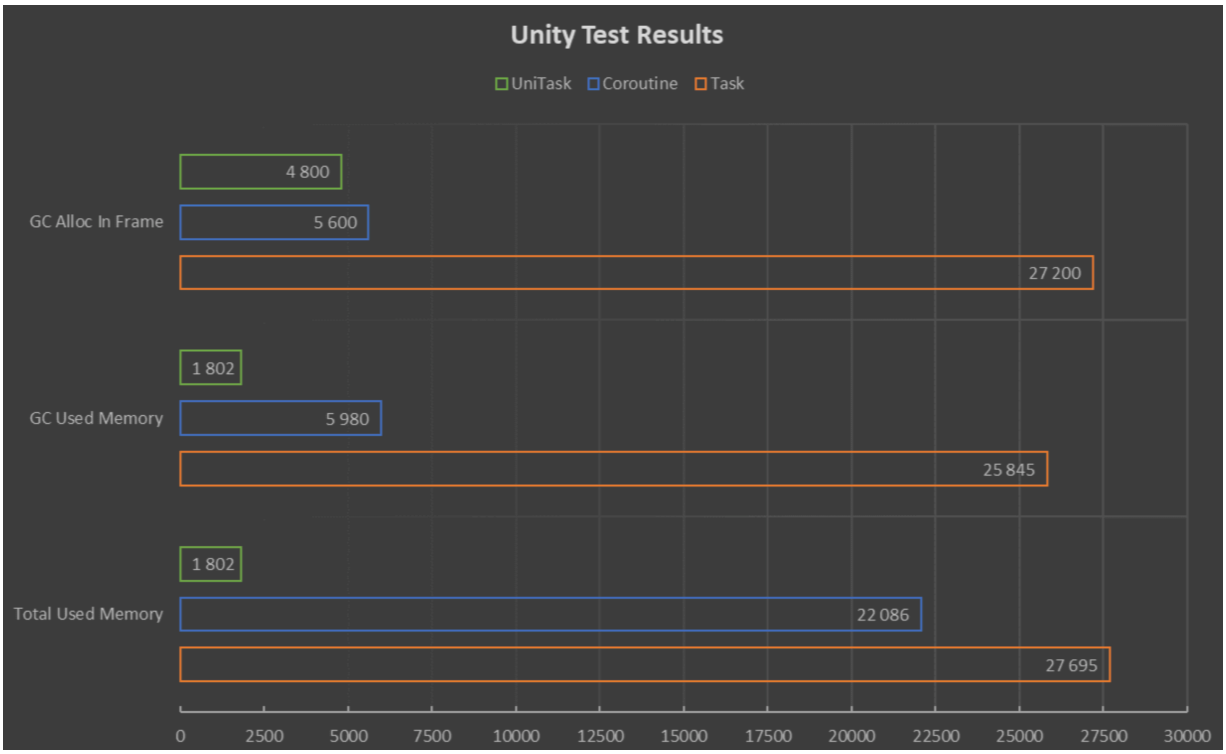

Рисунок 3.7 – Порівняння UniTask з вбудованим функціоналом

Та останнім фреймворком в технічному стекі проекту є Entitas. Це надшвидкий та легкий Entity-Component-System (ECS) фреймворк, спеціально розробленій для використання з движком Unity. Внутрішнє кешування та надзвичайно швидкий доступ до компонентів роблять його неперевершеним. Він також був ретельно розроблений для оптимальної роботи в середовищі зі збором сміття (Garbage collector) [26]. Як можна мобачити на рисунку 3.8, в середньому Entitas працює швидше, ніж вбудована компонентна система Unity:

Серед усіх існуючих фреймворків Entitas — саме популярне рішення, яке підтримується великою кількістю спільнот (тому, що був першим таким фреймворком). Крім того, що він досить швидкий, важливо що в ньому є інтеграція з Unity-редактором для візуалізації ECS-об’єктів, присутня кодогенерація для створення оберток із зручним API поверх користувацьких компонентів [27].

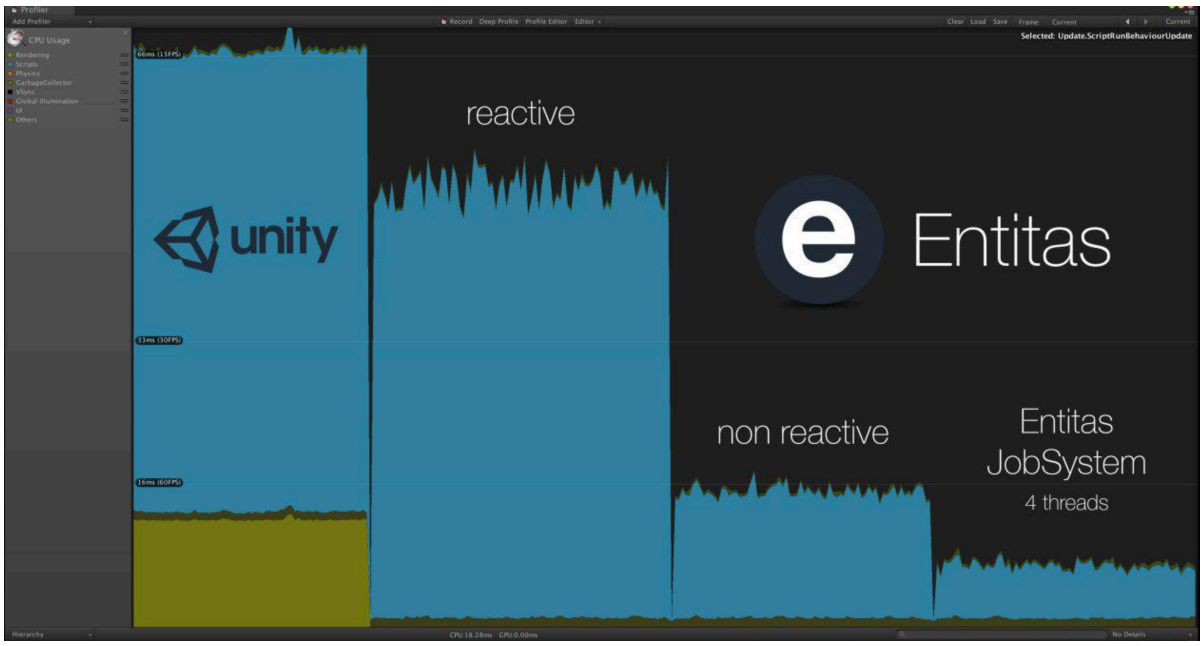

Рисунок 3.8 – Порівняння Entitas з вбудованим функціоналом

Для розуміння архітектурних рішень, важливо разглянути головні базові компоненти, з яких складається Entitas:

– Component - це в основному якимось чином залейблені дані. Компоненти можуть виконувати роль сховища даних (як тимчасове, так і для всього сеансу), як щось, що можна відфільтрувати (звужуючи область застосування певної системи), в якості тригеру/повідомлення (запуск систем реагування та подій), як ідентифікатор (що полегшує пошук однієї конкретної сутності);

– Entity - це набір компонентів;

– Group - завжди оновлюються і містять усі сутності, що відповідають вказаному збігу;

– Matcher - генеруються генератором коду і можуть бути комбіновані. Збіги зазвичай використовуються для отримання груп сутностей із контексту, що цікавить, тобто працюють як фільтр;

– Systems – існує 5 різних типів систем: InitializeSystem (виконується один раз перед початком гри), ExecuteSystem (виконує кожен кадр), CleanupSystem (виконує кожен кадр після завершення роботи інших систем), TearDownSystem (виконується один раз після закінчення гри), ReactiveSystem (виконується при зміні спостережуваної групи).

Як можна побачити, саме системи завдають життевий цикл усіх сутностей в ECS середовищі. Це є важливим чинником, бо накладє певні обмеження на архітектуру такої системи.

3.4 Опис реалізації обраних архітектур

Для обраних архітектур було розроблено власну реалізацію паттерну Controllers Tree як приклад підходу HMVC, використано фреймворк Entitas в якості ECS підходу, а для компонентно-орієнтованого підходу було взято вбудовану компонентну модель Unity.

Реалізація паттерну Controllers Tree була зроблена згідно з раніше описаним проєктуванням. Співвідношення між класами такої архітектурі можна побачити на діаграмі класів на рисунку 3.9:

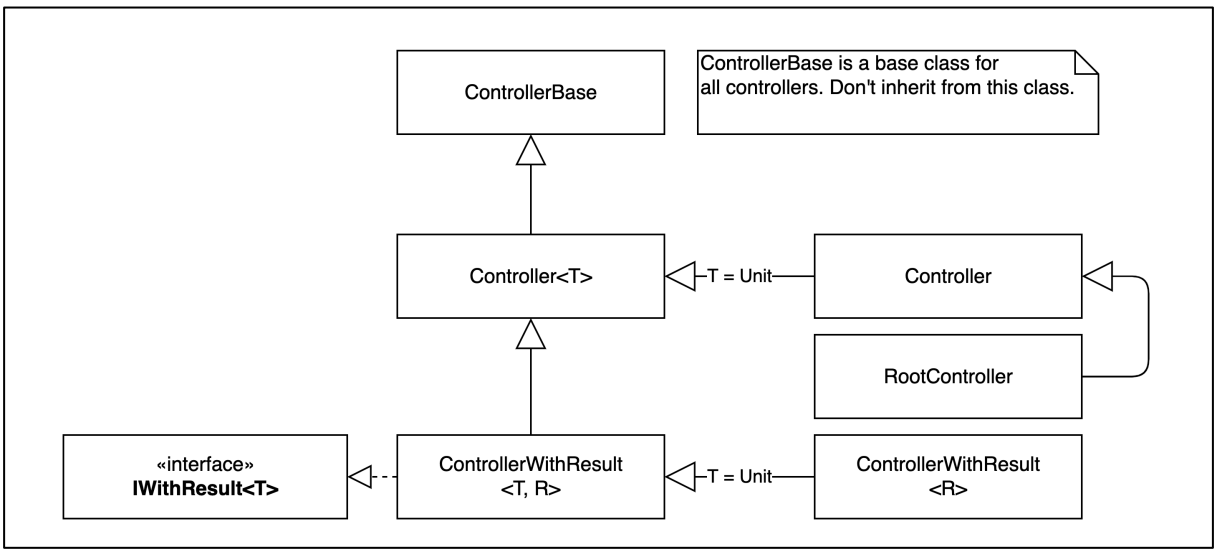

Рисунок 3.9 – Діаграма класів реалізації Controllers Tree

З діаграми бачимо головні класи реалізації:

– ControllerBase - клас містить всю базову логіку для контролерів, таку як обробка дітей, зупинка та видалення. Хоча в ньому відсутні деякі методи, такі як Initialize, тому використовувати цей контролер безпосередньо не рекомендується;

– Controller<TArgs> - контролер містить метод Initialize, який приймає об’єкт TArgs як параметри для ініціалізації;

– Controller - спадкоємець Controller<TArgs> з методом Initialize без параметрів;

– ControllerWithResult<TArgs, TResult> - представляє контролер, який повертає TResult після завершення виконання Start. Наприклад, це може бути корисно для діалогових вікон;

– ControllerWithResult<TResult> - те саме, що й Controller, але з результатом

Важливо розуміти життевий цикл конролеру, щоб знати на якому етапі роботи контролера виконується загрузка та вигрузка ресурсів гри, це можна побачити на рисунку 3.10:

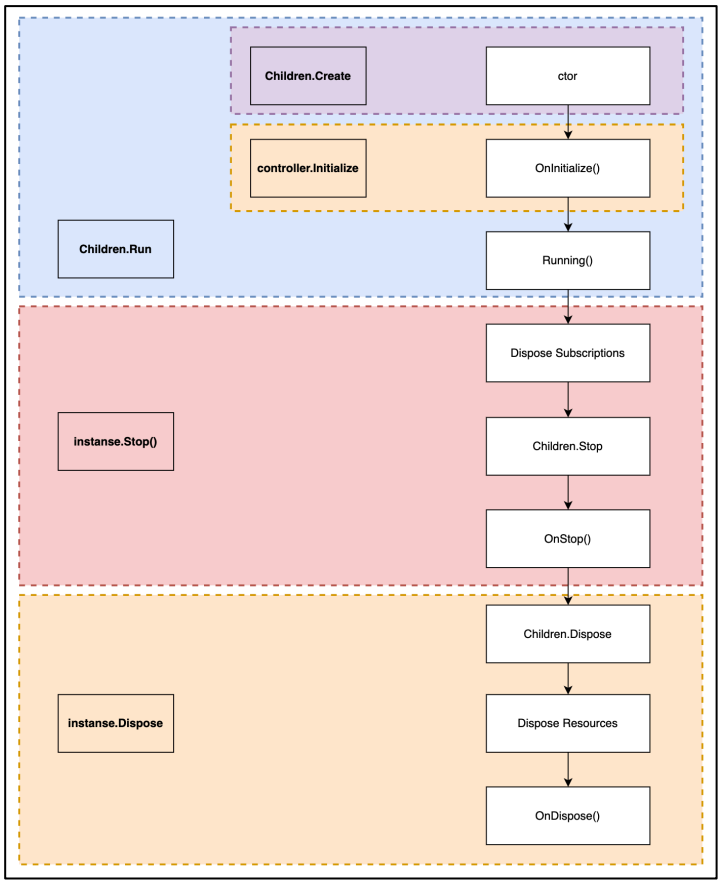

Рисунок 3.10 – Життевий цикл контролера

Жовтим кольором на рисунку відокремлено методи, коли саме краще всього здійснювати загрузку та вигрузку асетов: метод OnInitialize(), що викликається на початку роботи контролера, та метод OnDispose(), що відповідає за очистку. Це робить уніфікованим работу з ресурсами по всьому проєкту.

Для загрузки ассетов використається обгортка понад бібліотекою Addressables, що була описана раніше. Так як для проєкту непотрібен весь функціонал бібліотекі, то її API було зведено до мінімуму. За загрузку та вигрузку ресурсів відповідає нащадок інтерфейсу IAssetProvider, сервіс, що називається AddressablesAssetProvider. Наведемо код інтерфейсу IAssetProvider:

```
public interface IAssetProvider : IDisposable
{
      UniTask Initialize();
      UniTask<T> LoadAsync<T>(string key, CancellationToken token = default) where T : UnityEngine.Object;
      UniTask<IEnumerable<T>> LoadAsync<T>(IEnumerable<string> keys, CancellationToken token = default) where T : UnityEngine.Object;
      UniTask PreloadAssetsAsync(IProgress<DownloadStatus> progress, CancellationToken token = default, params string[] keys);
      UniTask<long> GetAssetsDownloadSize(CancellationToken token = default, params string[] keys);
}
```

Повну версію коду реалізації цього інтерфейса сервісом AddressablesAssetProvider наведено у додатку В. Як видно з інтерфейсу цього сервісу, робота з асетами є асінхронною. Саме тому для таких операцій використовється бібліотека UniTask. Також ця асинхронна природа операцій є причиною, чому методи життєвого циклу контролеру теж є асинхронними та повертають той ж самий тіп даніх – UniTask.

Також важливо згадати що для реїстрації контролерів було дещо розширено базовий функціонал DI-контейнеру Zenject та створено метод розширення:

```
public static void BindController<TContract>(this DiContainer container)
            where TContract : ControllerBase
        {
```

```
container.BindIFactory<TContract>().To<TContract>().AsSingle();
                container.Bind<IControllerFactory<TContract>>()
                    .To<ExtenjectControllerFactory<TContract>>()
                    .AsSingle();
            }
```

Це зв’язує реалізацію паттерну Controllers Tree з вбудованною функціональністю фреймворка Zenject, шо значно полегшує розширення та створення нового коду в проєкті.

Дуже важливою частиною всієї архітектури проекту є наявність стейт-машини та реалізація паттерну State. Але State в інтерпретації Controllers Tree – це нащадок ControllerWithResult, що повертає в якості результату об’єкт, що описує перехід з поточного стейту в наступний. Наведемо приклад такого стейту:

```
public class RootMapState : State
{
   private readonly IFactory<MapStateMachine> _mapStateMachine;
   private readonly IFactory<CoreMapState> _coreMapState;

   public RootMapState(IFactory<MapStateMachine> mapStateMachine,
IFactory<CoreMapState> coreMapState)
   {
      _mapStateMachine = mapStateMachine;
      _coreMapState = coreMapState;
   }

   protected override UniTask OnInitialize()
   {
      return UniTask.CompletedTask;
   }

   protected override async UniTask<StackedControllerInfo>
RunningForResult()
   {
      var subStateMachineResult = await
Children.RunForResult<MapStateMachine, StackedControllerInfo[],
StackedControllerInfo>(
          _mapStateMachine,
```

```
            new[] {

_coreMapState.FromExtenject().CreateState().WithoutPayload()
            });


        if                  (subStateMachineResult                  is
ControllerSuccess<StackedControllerInfo> controllerResult)
        {
            return controllerResult.Value;
        }

        return await base.RunningForResult();
    }

    protected override UniTask OnStop(bool isForced = false)
    {
        return UniTask.CompletedTask;
    }
}
```

Як бачимо, метод RunningForResult повертає об’єкт UniTask<StackedControllerInfo>, що і є указівкою для батьківського контролеру про майбутній перехід.

Щодо зв’язку Controllers Tree з ECS частиною проекту, то для їх поєднання кожна система повинна походити від EcsSystemController. Іншими словами – кожна система є контролером. Він має такі переваги, як використання системи асинхронної ініціалізації, однаковий конвеєр, який працює з ресурсами, приєднання ресурсів до контролера та додаткові методи життєвого циклу на додаток до інтерфейсів від Entitas. Таий контролер виконує всі системи в кожному кадрі. Крім того, можна примусово запустити будь-яку систему в будь-який момент. Наприклад, можна змоделювати основну гру без інтерфейсу користувача, просто відключивши системи, на які реагує візуально. Цей головний клас системи виглядає наступним чином:

```
public abstract class EcsSystemController : Controller, ISystem
{
```

```
        protected CustomCancellationToken DisposeCancellation =>
_disposeCancellation;
        private CustomCancellationToken _disposeCancellation;

        protected override UniTask Running(CancellationToken
cancellationToken)
        {
            _disposeCancellation = CustomCancellationTokenSource
                .CreateLinkedTokenSource(GetCancellationToken())
                .CustomCancellationToken;

            return UniTask.CompletedTask;
        }

        protected override UniTask OnStop(bool isForced)
        {
            return UniTask.CompletedTask;
        }

        protected override UniTask OnInitialize(CancellationToken
cancellationToken)
        {
            return UniTask.CompletedTask;
        }
    }
```

Тобто всі ECS-системи є конролерами та повинні бути нащадками цього класу. Приклад однієї з таких систем наведено в додатку Г.

Все це дає можливість поєднати дві частини проекту. Умовно таке поєднання можна відокрмити в домени що контролюються стейтами від Controllers Tree та домен з ECS-системою. Це можна побачити на рисунку 3.11:

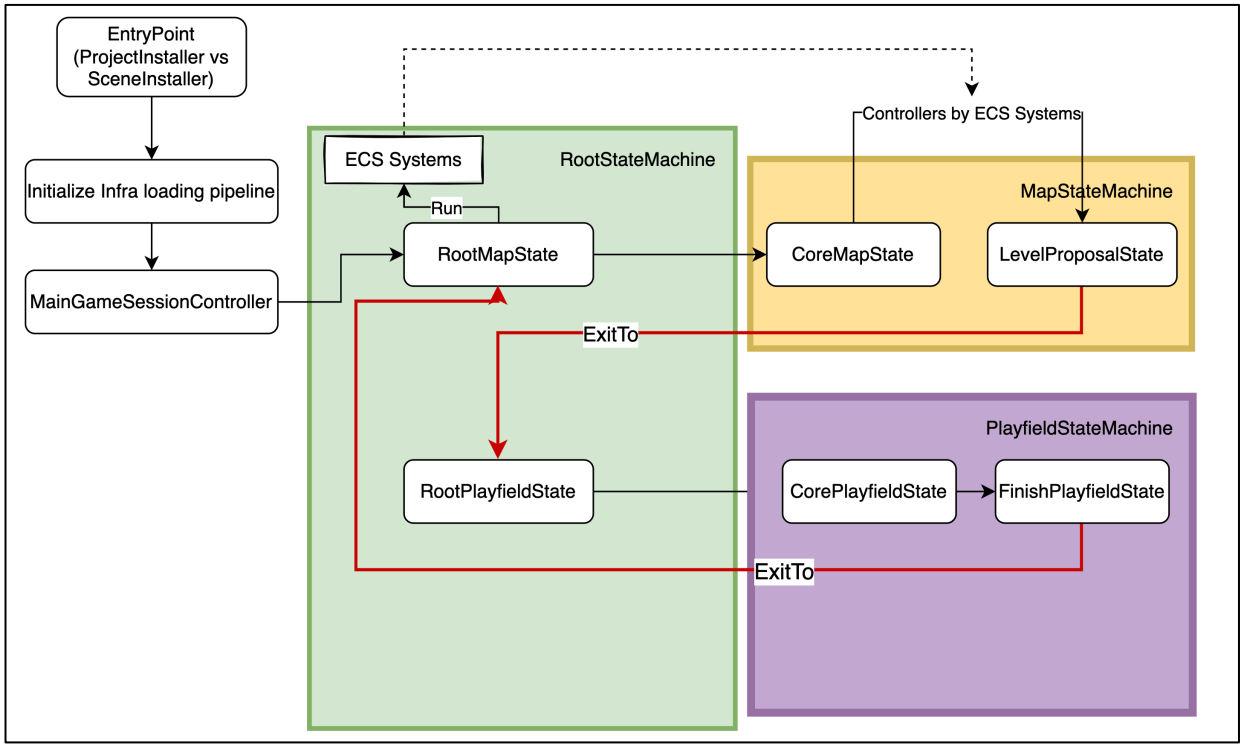

Рисунок 3.11 – Архітектура HMVC + ECS

Таке поєднання забезпечує можливість поєднання двух архітектур та вбудовує ECS-системи в світ контролерів від Controllers Tree.

Щодо поєднання Controllers Tree з компонентною системою, то тут все набагато простіше, бо компонентно-орієнтована система вбудована в Unity. Головне, про що треба пам’ятати – це життевий цикл компонентів [28], що можна побачити на рисунку 3.12:

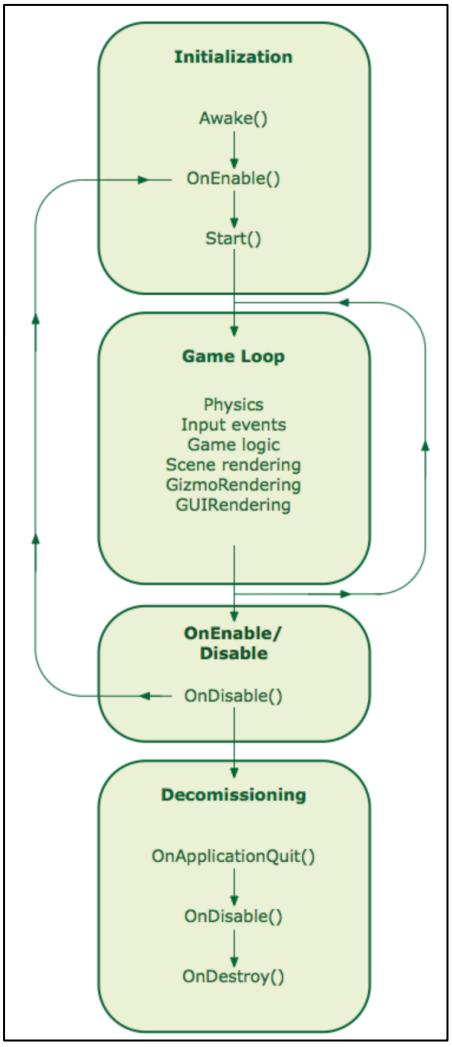

Рисунок 3.12 – Життевий цикл компонента

Кожен компонент поєднується з ресурсом, та починає працювати з моменту загрузки цього ресурсу до оперативної пам’яті. Тобто достаньо використовувати вищеописані контролери та сервіс-нащадок IAssetProvider для загрузки цих об’єктів і початку їх роботи.

4 ОПИС ЕКСПЕРІМЕНТАЛЬНІХ ДОСЛІДЖЕНЬ

4.1 Планування експеріменту

Головна мета експерименту – порівняння обраних разроблених архітектур. Визначимо набір головних метрик, які ми можемо порівнювати, щоб провести цей експеримент:

- швидкодія застосунку;
- здатність до тестування;
- здатність до розширення (додавання нового функціоналу);
- здатність до підтримки (простота внесення змін у вже існуючий код);
- кількість допущених помилок під час розробки функціоналу;
- час на інтеграцію архітектури у проект;
- час, необхідний на додавання нового функціоналу.

Враховуючи усі наведені метрики, придатність до використання певної архітектури (*Q)* можна визначити як значення функції:

$$Q = f(h_r, h_c, t, e, m, t_i, t_s),$$

де h_r - використання оперативної пам’яті;

h_c - використання CPU;

t - показник здатності до тестування;

e - показник розширюваності;

m - здатність до подальшої підтримки;

t_i - час інтеграції;

t_s - витрати часу на додавання подальшого функціоналу.

Найбільш чіткими та легко вимірюваними є показники швидкодії (завантаженість процесору, зайнятий обсяг оперативної пам’яті, тощо), які можна вимірювати за допомогою профайлеру.

Показники здатності до тестування, розширення, та підтримки – є більш розмитими. Здатність до розширення коду можна виміряти як час або кількість рядків коду чи кількість сутностей, необхідних для реалізації нового функціоналу у системі. Здатність до підтримки вимірюється аналогічно. Здатність до тестування архітектури можна визначити через два головних параметри - кількість рядків коду, які необхідні для тестування конкретного функціоналу, а також крихкість коду, що є показником, який відображає яка кількість написаних тестів для певного функціоналу зламається в результаті внесення змін.

Нажаль найбільш трудним показником є показник кількості допущених під час розробки помилок. Досягти адекватних показників цієї метрики неможливо, тому що у випадку реалізації функціоналу одним розробником, кількість помилок залежить від порядку - перша архітектура буде завжди матибільше помилок. Тому не будемо включати цю метріку в кінцевий експерімент.

Час, який потрібен на інтеграцію архітектури у проект, та час, необхідний для додавання нового функціоналу – метрики, які слід вимірювати відношенням показника однієї архітектури до показника аналога. Також зазначимо, що не можна розглядати значення цих двох параметрів як витрати часу - через те, що час інтеграції може змінити результати експерименту - деякі архітектури мають великий показний, який компенсується потім на великому проміжку часу.

Також введемо поняття величини C для відносного порівняння двох архітектур:

$$C_{AB} = (q_{ab} / q_m),$$

де q_{ab} - кількість метрик архітектури А, які є кращими за відповідні метрики В;

q_m - загальна кількість метрик для порівняння.

Маючи показники кожної архітектури, відповідь на питання «чи є доцільним використання архітектури А замість В» можна розглядати як параметр з наступними можливими інтервалами значень:

- $C_{AB} >= 0.75$ - Архітектура А є рекомендованою замість В майже в усіх випадках;
- $0.5 <= C_{AB} < 0.75$ - А є рекомендованою замість В у певних випадках;
- $C_{AB} < 0.5$ - А є рекомендованою замість В у виключних випадках.

Таким чином, ітогом експерименту є зібрані метрики відносно кожної з порівнюваних архітектур, які надають можливість зробити висновки відносно того, яку саме архітектуру краще використовувати.

4.2 Результати проведення експеріменту

Для проведення експерименту було вибрано єдине для всіх архітектур та стабільне середовище виконання - фізичний пристрій iPhone 13 Pro операційної системи iOS. Єдиний пристрій гарантує контрольований метод експерименту, бо для експеріменту повинні бути зібрані числові дані для всіх метрик обраних архітектур. Технічні характеристики обраного пристрою наведено в таблиці 4.1:

Таблиця 4.1 - Характеристики тестового пристрою

Device Name	iPhone 13 Pro
CPU	A15 Bionic
RAM	6GB
OS	iOS 15.4.1

Як було відзначено раніше, експеримент проводився за допомогою порівняння ідентичного реалізованого функціоналу, але з використанням двох різних архітектур: поєднання архітектури HMVC + ECS, та поєднання HMVC з компонентно-орієнтовною архітектурою. У результаті було зроблено виміри показників для кожного етапу розробки проекту, що дало можливість порівняти ці дві архітектури.

В проєкті існують певні use-cases (або сценарій використання), які використовувалися для заміру показників швидкодії застосунку. Під сценарієм використання мається на увазі проведення певних дій у грі, які не залежать від специфіки певної архітектури (наприклад, розрахунок штучного інтелекту або підготовка ігрового поля).

Для виміру швидкодії архітектур було використано стандартний інструмент профайлінгу - Unity Profiler. Зовнішний вигляд цього профайлеру зображено на рисунку 4.1:

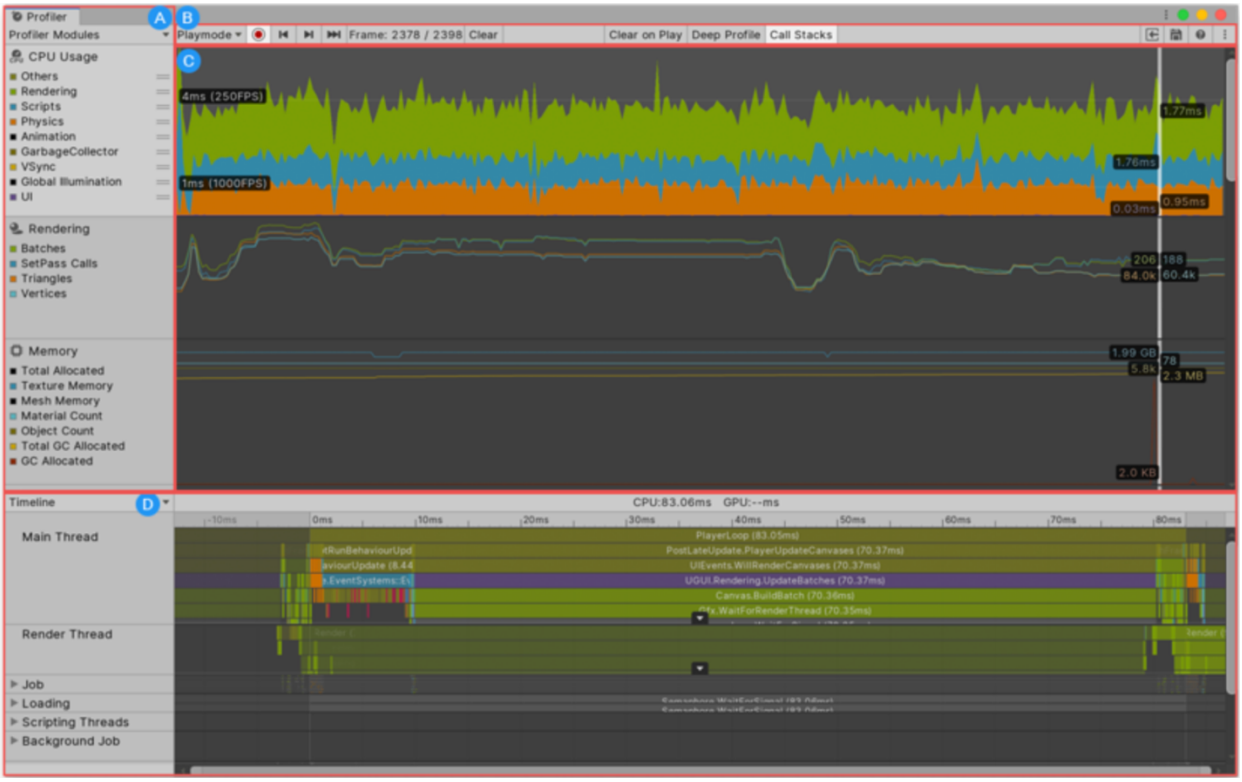

Рисунок 4.1 – Вигляд Unity Profiler

Це інструмент, який можна використовувати для отримання інформації про продуктивність програми. Дуже важливо, що його можна підключити до пристрою у мережі або пристрою, підключеного до комп'ютера, щоб перевірити, як гра працює на призначеній платформі. Ми будемо використовувати тільки розділи CPU Usage та Memory від цього профайлеру.

Перший сценарій використання, що було заміряно профайлером – це використання пам'яті при створенні та загрузки ігрового поля для карт. Слід відмітити, що сценарії виконувалися в однаковому середовищі та без запущених інших застосунків (для чистоти експеріменту). Результати заміру для першої архітектури наведено на рисунку 4.2:

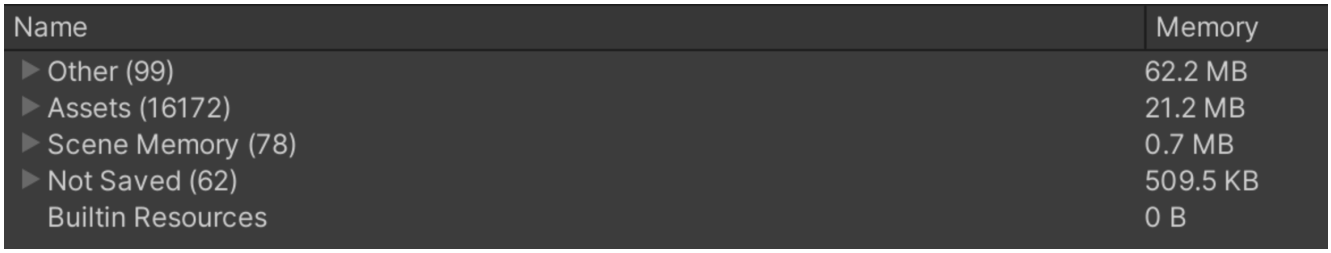

Рисунок 4.2 – Використання пам'яти для HMVC + ECS

Аналогічний замір було зроблено для другої архітектури, що наведено на рисунку 4.3:

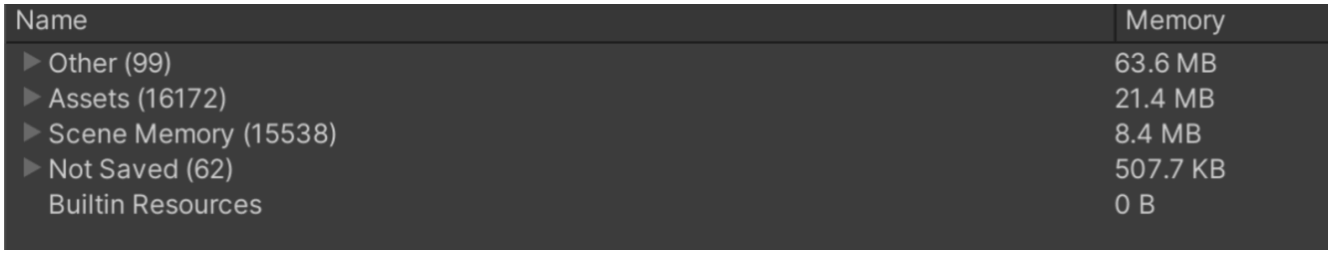

Рисунок 4.3 – Використання пам'яти для HMVC та компонентів

Як бачимо з рисунків, друга архітектура використовує більше пам'яті на ігровій сцені (scene memory), що негативно впливає на швидкодію гри.

Також для цього сценарію було зроблено замір навантаження на CPU для двох архітектур. Надалі всю інформацію по метріках буде надано в табличному вигляді.

Результати порівняння наведено у таблиці 4.2:

Таблиця 4.2 – Результати перошого профайлінгу

Метріки / Архітектура	HMVC + ECS	HMVC + компоненти
Навантаження на CPU	10.5%	12%
Використаня RAM	84.6 mb	93.9 mb

Як можна бачити з результатів першого профайлінгу, виміряні показники відрізняються. Перша архітектура виявляється більш оптимізованною.

Другий сценарій використання, що було заміряно профайлером – це очищнення пам’яті при виходу з ігрового поля для карт та перехід до головного меню. Результати порівняння наведено у таблиці 4.3:

Таблиця 4.3 – Результати другого профайлінгу

Метріки / Архітектура	HMVC + ECS	HMVC + компоненти
Навантаження на CPU	9.2%	11%
Використаня RAM	70 mb	78.7 mb

Як можна бачити з результатів другого профайлінгу, виміряні показники відрізняються приблизно в тому ж порядку, що й при першому замірі. Перша архітектура знов виявляється більш оптимізованною.

Настуні показники, а саме - здатність до тестування, розширювання та підтримки - вимірювалися після реалізації необхідного функціоналу. Незалежна змінна в даному випадку – це вимоги, які реалізуються за допомогою різних архітектурних підходів. Таким чином, логіка застосунку залишається незмінною, а на реалізацію впливають лише особливості архітектури. Дослідження цього функціоналу на предмет кількості строк коду/створених сутностей проводилося за допомогою порівняння двох версій коду у системі контроля версій – git.

Час на додавання нового функціоналу вимірювався у відносній формі – наприклад, час реалізації певного функціоналу, використовуючи одну архітектуру

приймається за X. Тоді час реалізації цього ж функціоналу за допомогою іншої архітектури можна було відносно представити як 1.5X або 0.5X. Враховуючи складність об’єктивної оцінки цього параметру, у результатах експерименту треба розглядати його з найменшим пріоритетом.

Першим функціоналом для порівняння виступає механізм доставання карти з колоди. Отримані результати порівняння наведено у таблиці 4.4:

Таблиця 4.4 – Порівняння функціоналу доставання карти з колоди

Метріки / Архітектура	HMVC + ECS	HMVC + компоненти
Кількість доданих рядків коду	190	356
Кількість створених сутностей	8	2
Кількість відповідальностей у класі	1	5
Кількість об’єктів, для тестування яких необхідні окремі залежності	0	2
Кількість коду для тестування	20	60
Відносний час для реалізації	1x	1.3x

Розберемо отримані результати. Кількість доданих рядків коду – показник розширюваності коду. Чем менше рядків коду додано – тим краще для архітектури. Треба зазначити, що при вимірі цього показника виключався автоматично

згенерований код від Entitas, так як цей фреймворк використовує дуже потужний кодогенератор, що полешує рутинні дії в разробці. Тому в даному випадку, архітектура HMVC + ECS має перевагу – кількість рядків коду, які необхідно додати, суттєво менша за іншу архітектуру.

Кількість створених сутностей – це також показник розширюваності. У першому випадку потрібно створити багато класів – Components, Groups, Systems, але частина з них генерується автоматично, в той час як інша архітектура потребує лише дві нові сутності. Але в цих двох сутностей набагато більше відповідальності, ніж у першій архітектурі, а це вже грає проти такої архітектури, бо це показник підтримки коду та розширюваності. Він говорить про те, наскільки створені сутності ізольовані та чи перетинаються їх відповідальності одна з одною (Single Responsibility Principle).

Кількість об’єктів, для тестування яких необхідні окремі залежності та кількість рядків, необхідних для тестування – це все показники придатності коду до тестування. В даному випадку, HMVC + ECS має перевагу – для тестування коду з використанням цієї архітектури не потрібно сторонніх залежностей. У випадку з компонентною архітектурою стає складнішим написання тестів.

Останній показник – відносний час, витрачений на реалізацію – це показник розширюваності коду. Порівняння є відносним, за базову величину береться та архітектура, час реалізації якої зайняв менше часу. Як вже було відзначено, цей показник не є надійним через те, що його дуже складно виміряти, та його не слід вважати основним, але в даному випадку перевага HMVC + ECS є досить значною – реалізація аналогічного функціоналу з використанням іншого підходу зайняла на 30% більше часу. Така швидкість також досягається завдяки кодогенерації який є присутнім для першого підходу.

Як можна бачити, з двох реалізованих частин функціоналу вже видно певну тенденцію порівняння двох архітектур, але для того, щоб зробити більш точні висновки порівняємо ще іншу частину гри.

Для настопного порівняння візьмемо більш складний функціонал – розрахунок поведінгкі штучного інелекту, який приймає рішення на основі

поточного положення карток на полі. Отримані результати порівняння наведено у таблиці 4.5:

Таблиця 4.5 – Порівняння функціоналу зі ШІ

Метріки / Архітектура	HMVC + ECS	HMVC + компоненти
Кількість доданих рядків коду	309	428
Кількість створених сутностей	4	1
Кількість відповідальностей у класі	1	6
Кількість об’єктів, для тестування яких необхідні окремі залежності	1	3
Кількість коду для тестування	53	87
Відносний час для реалізації	1.2x	1x

Як бачимо з результатів цього порівняння, головні тенденції у співвідношені метрік залішаються тиж самими. Єдине що змінилося – це відносний час для реалізації. В цей раз друга архітектура виявилася швидчішою в реалізації. Тому надалі будемо вважати, що за цим показником воні приблізно рівні.

Останній показник – це час на інтеграцію певної архітектури. Під час порівнянь архітектури існує певний час та обсяг коду, який необхідно написати для того, щоб обрана архітектура почала працювати. Цей час та обсяг існує ще до

того, як починається реалізація функціоналу. Порівняємо ці показники для двох обраних архітектур.

Спільною частиною обох архітектур є HMVC або ж реалізація конкретного паттерну Controllers Tree. Цей час для обоз архітектур можна вважати рівним.

Тоді для порівняння часу на інтеграцію залишаються інші частини обраних архітектур. Компоненти є вбудованим в Unity підходом. Тому за замовчуванням, нам не потрібно втрачати ніякого часу на інтеграцію цього підходу. В той час як для ECS потрібно заінтегрувати фреймворк Entitas, зробити розширення контролерів для вписування ECS до ієрархії конролерів та інше. Таким чином, час та витрачені зусилля на реалізацію такої архітектури суттєво збільшуються, та використання такої архітектури вже не є настільки простим, як здається.

Тому, у результаті порівняння часу на інтеграцію та обсягу коду можна вважати, що архітектура HMVC + копмонентно-орієнтована має перевагу над іншою за цим крітерієм.

4.3 Висновки та рекомендації з експериментального дослідження

За результатами проведенного дослідження обраних архітектур можна зробити висновок, що за такими показниками, як швидкодія, здатність до розширення, тестування та підтримки переважає архітектура на базі HMVC та ECS. Метріка кількості допущених помилок під час розробки функціоналу не бралась до експеріменту. За метрікою часу, що необхідно для додавання нового функціоналу, обидві архітектури показали приблизно однакові результати. А за метрікою часу, який ми втрачаємо на інтеграцію, перемогла архітектура на базі HMVC та компонентного підходу.

Таким чином $C_{AB} = (q_{ab} / q_m) = 4/6 = \sim 0.66$. Тобто архітектура на базі HMVC та ECS є рекомендованою замість архітектури на базі HMVC та компонентного підходу у певних випадках. А CCG жанр ігор якраз і є саме таким випадком.

Як можна бачити з показників швидкодії, перша архітектура працює швидшке в середньому на 15% та використовує на 10% менше оперативної пам’яті. Це є доволі великими значеннями для сучасних мобільних пристроїв, зокрема це стає дуже крітично для підтримки слабких девайсів.

Але такі переваги в швидкодії маюють і свої недоліки. Через те, що такий підхід до архітектури проектів на Unity покі що не є стандартизованим, ми втрачаємо більше часу на інтеграцію та не виграємо жодної хвилини на додавання нового функціоналу, якщо порівнювати зі стандартним підходом з компонентно-орієнтованою архітектурою. Хоча перша архітектура стає все більш і більш популярною та певні дослідження ведуться компаниєю Unity задля того, щоб зробити еволюційний стрибок в розробці ігор та зробити такий підхід другим стандартизованим на цьому рушії у майбутньому. Цей проект називаєтсь DOTS та поки що знаходиться на етапі розробки, але вже показує гарні результати.

Таким чином, можна зробити висновки, що проекти, у якіх рекомендовано архітектуру на базі HMVC та ECS – це середні та великі проекти, у яких важливе місце займає швидкодія да підтримка слабких девайсів, а також проекти, для яких за своє природою (наприклад, на це впливає жанр гри) важлива простота розширення та підртимки проекту, ніж час на реалізацію прототипу. Такі проекти характеризуються великим обсягом часу підтримки та модифікації – ці сроки досягають років, що робить показники розширюваності, підтримки та можливості тестування критичними.

В той час як для невеликих проектів та маленькіх прототипів навпаки є рекомендованою архітектура на базі HMVC та компонентного підходу через незначний час первинної реалізації та відстутности необхідність в навчанні розробників роботі з цією архітектурою, бо вана є частково стандартизованою в Unity. Для таких проектів важливо бути мобільними та вчасно реагувати на зміни ринку та трендів, це робить критичним час на реалізацію, в той час як метрики розширюваності, можливості підтримки та тестування часто відходять на другий план.

ВИСНОВКИ

В ході виконання магістерської кваліфікаційної роботи було проаналізовано найбільш популярні підходи до проектування ігрових архітектур в контексті реалізації комп'ютерних ігор в жанрі "Collectible Card Game", а саме два типи – підхід з окремою архітектурою до геймплея та мета-гейму з інтерфейсом користувача, та підхід з однаковою архітектурою для усіх частин гри. Для кожного з основних типів було проведено аналіз найбільш типових представників та проаналізовано кожного з цих представників. Крім цього, було зроблено висновки, що можна покращити у рамках проектування архітектур мобільних ігрових застосунків.

На основі аналізу архітектур, було сформовано вимоги до архітектури, що повина бути найкращою для реалізації комп'ютерних ігор в жанрі "Collectible Card Game". Та обрано дві найбільш привабливі архітектури для дослідження та проведення порівняння.

В результаті проектування та розробки було спроектовано та створено мобільний ігровий додаток, який реалізує ці два архітектурний підходи, для проведення дослідження.

Для аналізу результатів проектування та розробки було обрано ряд критеріїв та метрик, за якими проводилося дослідження та порівняння розроблених архітектур. У результаті порівняння було зроблено висновки про можливості використання обох архітектур, їх сильні та слабки сторони, а ткжож було розроблено рекомендації до того, яку архітектуру треба обрати для різних видів проектів. Результати роботи було опубліковано у науково-технічній конференції «Сучасні напрями розвитку інформаційно-комунікаційних технологій та засобів управління» (див. додаток Г).

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Игра как феномен культуры [Электронный ресурс] : учеб. пособие / М. А. Гузик. – 2-е изд., стер. – М. : Флинта : 2012. – 268 с.

2. Звіт від команії SuperData про стан ігрової індустрії на 2019 рік: Початок роботи. - URL: https://www.superdataresearch.com/blog/superdata-reports-games-and-interactive-media-earned-a-record-1201b-in-2019 (дата звернення: 20.03.2022)

3. Визначення терміну «Ігровий рушій» : Початок роботи. - URL: https://uk.wikipedia.org/wiki/%D0%86%D0%B3%D1%80%D0%BE%D0%B2%D0%B8%D0%B9_%D1%80%D1%83%D1%88%D1%96%D0%B9 (дата звернення: 20.03.2022)

4. Визначення терміну «Багатокористувацька гра» : Початок роботи. - URL: https://uk.wikipedia.org/wiki/%D0%91%D0%B0%D0%B3%D0%B0%D1%82%D0%BE%D0%BA%D0%BE%D1%80%D0%B8%D1%81%D1%82%D1%83%D0%B2%D0%B0%D1%86%D1%8C%D0%BA%D0%B0_%D0%B3%D1%80%D0%B0

5. Player.IO. Сетевые архитектуры : Початок роботи. - URL: http://www.ant-karlov.ru/PlayerIO-setevie-arhitekturi.html

6. Многопользовательские игры. Разработка сетевых приложений. — СПб.: Питер, 2017. — 368 с.: ил. — (Серия «Библиотека программиста»).

7. Визначення терміну «Peer-to-peer» : Початок роботи. - URL: https://uk.wikipedia.org/wiki/%D0%A7%D0%B8%D1%82_%D0%B2_%D0%BE%D0%BD%D0%BB%D0%B0%D0%B9%D0%BD_%D1%96%D0%B3%D1%80%D0%B0%D1%85

8. Визначення терміну «Читерство» : Початок роботи. - URL: https://uk.wikipedia.org/wiki/%D0%A7%D0%B8%D1%82_%D0%B2_%D0%BE%D0%BD%D0%BB%D0%B0%D0%B9%D0%BD_%D1%96%D0%B3%D1%80%D0%B0%D1%85 (дата звернення: 20.03.2022).

9. Визначення терміну «Клієнт-серверна архітектура» : Початок роботи. - URL:

https://uk.wikipedia.org/wiki/%D0%9A%D0%BB%D1%96%D1%94%D0%BD%D1%82-%D1%81%D0%B5%D1%80%D0%B2%D0%B5%D1%80%D0%BD%D0%B0_%D0%B0%D1%80%D1%85%D1%96%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0 (дата звернення: 20.03.2022).

10. Васильев, Ф. П. Линейное программирование / Ф.П. Васильев, А.Ю. Иваницкий. - М.: Факториал Пресс, 2016. - 352 с.;

11. Семь этапов создания игры: от концепта до релиза : Початок роботи. - URL: https://habr.com/ru/company/miip/blog/308286/

12. Д. Грофф, П. Вайнберг, Э. Оппель. SQL: полное руководство 3-е издание. – М.: Издательский дом «Вильямс», 2015 г. - 959 с.

13. Getting started with DOTS: Початок роботи. - URL: https://unity.com/dots

14. Firebase Realtime Database: Початок роботи. -. URL: https://firebase.google.com/docs/database (дата звернення: 20.03.2022)

15. Firebase Cloud Storage: Початок роботи. -. URL: https://firebase.google.com/docs/storage (дата звернення: 20.03.2022)

16. Модель «сутність — зв'язок»: Початок роботи. -. URL: https://uk.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D0%B5%D0%BB%D1%8C_%C2%AB%D1%81%D1%83%D1%82%D0%BD%D1%96%D1%81%D1%82%D1%8C_%E2%80%94_%D0%B7%D0%B2%27%D1%8F%D0%B7%D0%BE%D0%BA%C2%BB (дата звернення: 20.03.2022)

17. Визначення терміну «MVC» : Початок роботи. - URL: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

18. Визначення терміну «MVVM» : Початок роботи. - URL: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93viewmodel

19. Визначення терміну «HMVC» : Початок роботи. - URL: https://en.wikipedia.org/wiki/Hierarchical_model%E2%80%93view%E2%80%93controller

20. Locality of reference: Початок роботи. - URL: https://en.wikipedia.org/wiki/Locality_of_reference

21. Визначення терміну «Dependency Injection»: Початок роботи. - URL: https://en.wikipedia.org/wiki/Dependency_injection

22. Unity Addressable Asset system: Початок роботи. - URL: https://docs.unity3d.com/Packages/com.unity.addressables@1.10/manual/index.html

23. UniTask v2 — Zero Allocation async/await for Unity, with Asynchronous LINQ: Початок роботи. - URL: https://neuecc.medium.com/unitask-v2-zero-allocation-async-await-for-unity-with-asynchronous-linq-1aa9c96aa7dd

24. Unity: 8 причин отказаться от Coroutine в пользу Async: Початок роботи. - URL: https://habr.com/ru/post/652483/

25. Zenject: Початок роботи. - URL: https://github.com/modesttree/Zenject

26. What is Entitas?: Початок роботи. - URL: https://github.com/sschmid/Entitas-CSharp/wiki

27. Unity, ECS и все-все-все: Початок роботи. - URL: https://habr.com/ru/post/358108/

28. Order of execution for event functions: Початок роботи. - URL: https://docs.unity3d.com/Manual/ExecutionOrder.html