МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ БІЗНЕС-КОЛЕДЖ

кафедра комп’ютерної інженерії та інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА

на тему

Розробка та реалізація ігрового додатку на базі архітектури, орієнтованої на дані

Виконав: студент групи 1П-20
Спеціальності
121 – «Інженерія програмного забезпечення»

Євгеній ТЕРТИЧНИЙ

Керівник:
Станіслав МАРЧЕНКО

Черкаси 2024

ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ БІЗНЕС-КОЛЕДЖ

Кафедра комп'ютерної інженерії та інформаційних технологій
(повна назва випускової кафедри)

Спеціальність 121 "Інженерія програмного забезпечення"
(шифр і назва спеціальності)

Освітня програма Інженерія програмного забезпечення
(назва освітньої програми)

ЗАТВЕРДЖУЮ

Завідувач кафедри
комп'ютерної інженерії та
інформаційних технологій
(назва кафедри)
_____ Хотунов В.І.
(підпис) (ПІБ)
«_____»_____2023 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Тертичному Євгенію Сергійовичу
(прізвище, ім'я, по батькові студента)

1. Тема кваліфікаційної роботи Розробка та реалізація ігрового додатку на базі архітектури, орієнтованої на дані
Керівник роботи Марченко Станіслав Віталійович, спеціаліст І категорії
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом закладу вищої освіти від " 13 " жовтня 2023 року № 65У.
2. Строк подання студентом кваліфікаційної роботи 03.06.2024
3. Вихідні дані до кваліфікаційної роботи: мови програмування C++ та C#, технології EnTT та Unity Dots (стек орієнтований на дані).
4. Зміст випускної роботи (перелік питань, які потрібно розробити): огляд предметної області (аспекти розроблення ігрових додатків, архітектурні питання ігрової розробки, інструментальні засоби для ігрової розробки), архітектурні підходи до ігрової розробки (концептуальний опис демонстраційних прототипів, застосування традиційних шаблонів проектування, реалізація чистої архітектури), порівняння ігрових архітектур (виокремлення критеріїв для порівняння, тестування та архітектурний аналіз).
5. Дата видачі завдання 15.09.2023р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Терміни виконання етапів	Примітка про виконання
1	Вступ	20.10.2023	
2	Розділ 1. Огляд предметної області	22.12.2023	
3	Розділ 2. Архітектурні підходи до ігрової розробки	15.03.2024	
4	Розділ 3. Порівняння ігрових архітектур	15.05.2024	
5	Висновки	17.05.2024	
6	Оформлення випускної роботи (чистовий варіант)	27.05.2024	
7	Здача випускної роботи на кафедру для рецензування (за 14 днів до захисту)	31.05.2024	
8	Перевірка випускної роботи на наявність ознак плагіату (за 10 днів до захисту)	03.06.2024	
9	Подання випускної роботи на затвердження завідувачу кафедри (за 7 днів до захисту)	06.06.2024	

Студент _____ Тертичний Є.С
(підпис) (прізвище та ініціали)

Керівник роботи _____ Марченко С.В.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Ця дипломна робота присвячена дослідженню архітектурних підходів та розробці ігрового додатку на основі архітектури, орієнтованої на дані. Основними підходами, що розглядаються, є об’єктно-орієнтоване програмування (ООП) та проєктування, орієнтоване на дані (DoD). Крім того, значну увагу приділено реалізації архітектури Entity-Component-System (ECS), яка є ключовим аспектом сучасної розробки ігрових додатків.

Перший розділ роботи містить огляд предметної області, де висвітлюються аспекти розробки ігрових додатків. Вивчаються основні етапи розробки ігор, починаючи від концептуалізації ідей до кінцевого тестування та випуску. Також розглядаються відеоігри як м’які симуляції реального часу, що дозволяють створювати інтерактивні світи з високим рівнем деталізації. Окремо детально описані інструментальні засоби, які використовуються на різних етапах розробки, включаючи графічні рушії, середовища розробки та інші програмні засоби.

Другий розділ присвячений архітектурним підходам до розробки ігрових додатків. ООП розглядається як класичний підхід, що дозволяє структурувати код у вигляді взаємодіючих об’єктів. Проєктування, орієнтоване на дані (DoD), надає можливість оптимізувати роботу з великими обсягами даних, що є критичним для ігрових додатків. У цьому контексті представлено реалізацію ECS архітектури, яка поєднує гнучкість об’єктно-орієнтованого підходу та ефективність роботи з даними.

Третій розділ зосереджений на порівнянні архітектурних підходів. Визначено критерії відбору архітектур для ігрових додатків, що включають продуктивність, масштабованість, легкість підтримки та розширюваність. Виконано порівняльний аналіз ООП та DoD, виявлено їхні сильні та слабкі сторони. Особлива увага приділена аналізу архітектури ECS, яка поєднує переваги обох підходів, забезпечуючи високу продуктивність та гнучкість у розробці ігрових додатків.

ABSTRACT

This thesis is dedicated to the study of architectural approaches and the development of a game application based on a data-oriented architecture. The main approaches considered are object-oriented programming (OOP) and data-oriented design (DoD). Additionally, significant attention is given to the implementation of the Entity-Component-System (ECS) architecture, which is a key aspect of modern game development.

The first chapter provides an overview of the subject area, highlighting the aspects of game application development. The main stages of game development are studied, from the conceptualization of ideas to final testing and release. Video games are also considered as soft real-time simulations, allowing the creation of interactive worlds with a high level of detail. The chapter separately details the tools used at different stages of development, including graphics engines, development environments, and other software tools.

The second chapter is dedicated to architectural approaches to game application development. OOP is considered a classical approach that allows for structuring code in the form of interacting objects. Data-oriented design (DoD) provides the ability to optimize working with large volumes of data, which is critical for game applications. In this context, the implementation of the ECS architecture is presented, which combines the flexibility of the object-oriented approach with the efficiency of data handling.

The third chapter focuses on comparing architectural approaches. Criteria for selecting architectures for game applications are defined, including performance, scalability, ease of maintenance, and extensibility. A comparative analysis of OOP and DoD is performed, revealing their strengths and weaknesses. Special attention is given to the analysis of the ECS architecture, which combines the advantages of both approaches, ensuring high performance and flexibility in game application development.

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

ООП – об’єктно-орієнтоване програмування

DoD – проєктування, орієнтоване на дані

ECS – Entity-Component-System

CPU – центральний процесорний блок

GPU – графічний процесорний блок

RAM – оперативна пам’ять

FPS – кількість кадрів в секунду

IDE – інтегроване середовище розробки

API – інтерфейс прикладного програмування

UI – користувацький інтерфейс

UX – досвід користувача

AI – штучний інтелект

SDK – набір засобів розробки програмного забезпечення

MVC – Model-View-Controller

LFS – Large File Storage

ЗМІСТ

ВСТУП

Актуальність обраної теми. Зростання популярності ігрової індустрії призводить до необхідності підвищення якості та продуктивності ігрових додатків. Архітектура програмного забезпечення визначає основні принципи організації системи, що дає змогу досягти кращої модульності, масштабованості та зручності в розробці й підтримці. Дослідження цієї області важливе для пошуку оптимальних рішень в ході розроблення ігрових додатків та забезпечення їх конкурентоспроможності на ринку.

Ігрові архітектури впливають на кожний аспект створення гри, від її геймплею та візуального оформлення до оптимізації для різних платформ. Розуміння та використання сучасних архітектур допомагає розробникам побудувати ефективніші, швидші та більш гнучкі ігрові системи.

Тема ігрових архітектур особливо актуальна в контексті стрімкого зростання комплексності ігрових світів та геймплею. Розвиток технологій дозволяє створювати динамічні та іммерсивні ігрові досвіди, які вимагають від розробників високої ефективності та гнучкості в архітектурному плануванні.

Дослідження в галузі ігрових архітектур важливе також через зростання інтересу до розробки кросплатформних ігор. Відкриття нових ринків та платформ спонукає розробників до пошуку оптимальних рішень, які б забезпечили якість та ефективність гри на різних пристроях. Отже, аналіз та впровадження сучасних ігрових архітектур є критично важливим для досягнення успіху в динамічній та конкурентній галузі відеоігор.

Об’єкт дослідження. Об’єктом дослідження є ігрові архітектури, зокрема дата-орієнтовані системи компонентів (ECS), які використовуються у сучасних ігрових розробках.

Предмет дослідження. Предметом дослідження виступають тактики, моделі та технології, які застосовуються для розробки і впровадження ігрових архітектур.

Мета дослідження. Мета проєкту полягає в аналізі сучасного стану досліджень архітектур ігрових додатків та формуванні практичних навичок щодо їх програмної реалізації.

Завдання дослідження. У межах кваліфікаційної роботи передбачається вирішення таких завдань:

1) виконати літературний огляд сучасних джерел стосовно програмної архітектури ігрових додатків, виокремити популярні архітектурні шаблони та шаблони проєктування для розроблення ігор;
2) розглянути архітектурні підходи до розроблення ігрових додатків, виокремити основні критерії порівняння, особливості, переваги та недоліки;
3) програмно реалізувати демонстраційні прототипи, які підкреслюють ключові архітектурні аспекти при розробці ігор.

РОЗДІЛ 1
ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Аспекти розроблення ігрових додатків

У 2023 році дохід ігрової індустрії сягнув 390 млрд дол. У 2017 році цей показник становив 156 млрд дол. Щорічно він зростає в середньому на 30 млрд дол. З ростом популярності ігор вони стають все масштабніше та складніше у реалізації. Над багатьма проєктами працюють по декілька років із командою в декілька сотень людей. Наприклад, над S.T.A.L.K.E.R. 2: Heart of Chernobyl працюють понад 500 осіб [1]. Сучасний процес розробки гри з точки зору програмування фокусує увагу на підтримці постійно розширюваної кодової бази гри та внутрішніх інструментів. Саме тому разом з ростом ігрової індустрії, розвивалися й підходи до розробки ігор. Якщо спочатку було достатньо простого процедурного програмування, то зараз використовується найрізноманітніші архітектурні підходи. Тому в сучасному світі ігрової розробки, де швидкість та гнучкість є ключовими факторами, вибір відповідної ігрової архітектура має стратегічну важливість. Аналіз вимог до ігрових архітектур є важливим етапом проектування, що передбачає ретельний розгляд вимог для забезпечення успішного розвитку гри [1].

Процес розроблення ігор поєднує в собі творчість, технічність та величезний обсяг роботи. Він вимагає від команди розробників постійної уваги до подробиць, творчого мислення та величезної енергії. Цей процес розгортається від ідеї на початкових стадіях до втілення концепції в реальну гру, яка запам’ятається мільйонам гравців по всьому світу.

Спочатку команда розробників працює над ідеєю гри та формулює концепцію, яка буде лежати в основі проекту. Вони розробляють сценарій, механіку та загальну структуру гри, враховуючи побажання аудиторії та актуальні тенденції у світі геймінгу. Після цього настає етап передвиробничої підготовки, де розробники вже розробляють візуальну частину гри, роблять прототипи персонажів та пейзажів, а також створюють документацію проекту.

На наступному етапі – виробничому – розробники приступають до створення активів гри та програмування механіки. Це найбільш інтенсивний період у розробці, коли команда працює безпосередньо над створенням самої гри, засвоюючи велику кількість ресурсів та часу [15].

Останнім етапом є постпродакшн, коли гра проходить тестування, виправлення помилок та оптимізацію перед запуском. Це дуже важливий період, оскільки саме на цьому етапі вирішується якість гри та її прийняття гравцями. Тут кожна деталь має значення, і команда працює наполегливо, щоб забезпечити успішний реліз гри [7].

Ідея може бути сприйнята як фундамент у будь-якому успішному циклі розробки гри. Вона повинна бути унікальною та захопливою, з урахуванням того, що продукт повинен бути корисним та цікавим для певної аудиторії. Міцна концепція може вирішити відмінність між історією, яка загубиться на переповненому ринку, та історією, яка стане хітом. Важливо витратити час на мозкування і вдосконалення ідей перед переходом до етапів розробки [11].

1.2. Основні етапи розробки гри

Розроблення ігрових додатків є багатоетапним процесом. До основних фаз можна віднести:

1) планування;
2) передвиробничий етап, що передбачає формування гри;
3) прототипування;
4) робота над грою (виробництво);
5) тестування (вдосконалення);
6) підготовка до запуску;
7) запуск;
8) післярелізний період (Live-Ops).

Ця послідовність відображена на рис. 1.1 з уточненнями стосовно кожного етапу. Протягом фази планування команда розробників визначає концепцію гри, встановлює місію та візію, визначає цільову аудиторію, встановлює обсяг

проекту та встановлює бюджет і графік. Правильне планування має вирішальне значення, щоб забезпечити чітке розуміння командою цілей гри і працювати ефективно для створення успішного продукту.

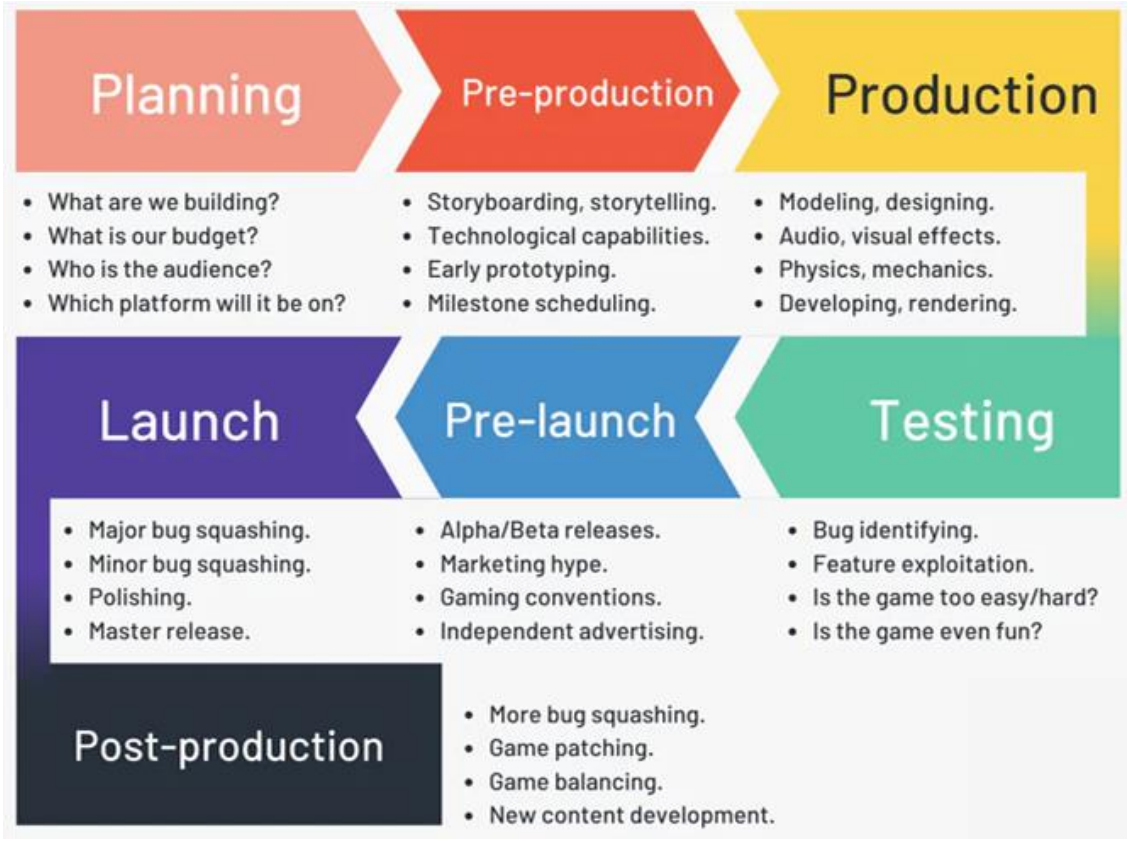

Рисунок 1.1 – Послідовність етапів розроблення гри

Під час передвиробничого процесу команда починає розробляти дизайн гри, розробляючи сюжет, проектуючи персонажів та мапуючи механіку гри. Щоб отримати уявлення про те, як буде виглядати гра, команда створює концепт-арт, сценарії та ескізи. Цей етап встановлює тон та напрямок гри і закладає основу для виробництва.

Прототипування дає змогу розробникам експериментувати з різними ідеями гри, механіками та елементами дизайну перед завершенням продукту. Основна мета – перевірити можливість гри та виявити будь-які потенційні проблеми під час розробки. Створюючи прототип, розробники можуть удосконалити свої ідеї та забезпечити, що кінцевий продукт відповідає їхнім вимогам.

Виробництво (production) – фактична розробка гри. Вона включає створення активів гри, програмування механіки гри, проектування рівнів та створення звукових ефектів і музики. Команда також регулярно тестує гру, щоб виявити та виправити будь-які проблеми. Виробничий етап зазвичай є найтривалішим у процесі розробки гри, і важливо мати присвячену команду, щоб забезпечити успішне завершення гри.

Фаза тестування допомагає виявити та виправити помилки або проблеми в грі. На цьому етапі команда ретельно тестує гру, шукаючи будь-які проблеми з ігроладом, глітчи або збої, і працює над процесом, що називається замороженням функцій. Команда також повинна збирати відгуки від тестувальників та вносити зміни відповідно.

На етапі перед запуском виробляються маркетингові матеріали, розробляється план запуску, налаштовуються канали розповсюдження та проводиться бета-тестування. Це створює шум та охоплення для успішного запуску гри. Розробники також можуть представити свою гру на конвенціях або використовувати незалежну рекламу, таку як огляди на YouTube або спонсорований контент, щоб залучити аудиторію власного хобі та створити базу шанувальників.

Під час етапу запуску команда зосереджується на просуванні гри, управлінні зворотним зв’язком від гравців та вирішенні будь-яких проблем, що виникають, включаючи виправлення як важливих, так і менших помилок. Цей етап також є можливістю збирати дані про продуктивність гри, включаючи залучення гравців, утримання та монетизацію. Після запуску команда може продовжувати випускати патчі та оновлення. Майстер-реліз позначає завершення процесу розробки гри та її остаточну версію.

Після випуску передбачається постійна підтримка та обслуговування гри після її запуску. Це включає в себе відслідковування відгуків гравців, вирішення багів та проблем, випуск патчів та оновлень, а також управління спільнотою гравців. Етап після випуску є критичним для підтримки залучення гравців та забезпечення тривалості гри. Цей етап також може включати розробку нового

контенту, функцій та розширень, щоб тримати гру свіжою та привабливою для гравців.

Ретельно підібрана команда допомагає зберігати цикл розробки гри на правильному шляху. Участь людей у розробці гри може значно відрізнятися в залежності від розміру проєкту, його складності та обсягу роботи. Зазвичай невеликі команди можуть включати від 5 до 20 осіб, тоді як великі проєкти можуть займати сотні або навіть тисячі співробітників.

1.3 Відеоігри як м'які симуляції реального часу

Більшість відеоігор у дво- та тривимірному форматі можна розглядати як приклади того, що в комп'ютерних науках відомо як м'які інтерактивні агент-орієнтовані комп'ютерні симуляції у реальному часі. У більшості грального ігор певна частина реального або уявного світу математично моделюється з метою управління нею за допомогою комп'ютера [14].

Агент-орієнтована симуляція – це симуляція, в якій взаємодіють кілька відомих об'єктів, які називають «агентами». Це добре узгоджується з описом більшості тривимірних комп'ютерних ігор, де агентами є транспортні засоби, персонажі, вогняні кулі, енергетичні краплі тощо. Враховуючи агент-орієнтовану природу більшості ігор, не дивно, що більшість ігор в наш час реалізовані на об'єктно-орієнтованій або, принаймні, на розріджено-об'єктній мові програмування [10].

Всі інтерактивні відеоігри є часовими симуляціями, тобто віртуальна модель ігрового світу є динамічною – стан світу гри змінюється з часом, коли відбуваються події та розвивається сюжет гри. Відеоігра також повинна реагувати на непередбачувані вхідні дані від гравця(ів). Нарешті, більшість відеоігор представляють свої сюжети та реагують на введення гравця в реальному часі, що робить їх інтерактивними часовими симуляціями реального часу. Один помітний виняток – це категорія покрокових ігор, таких як комп'ютеризований шахи або стратегічні ігри не в реальному часі. Але навіть ці

типи ігор зазвичай надають користувачу якусь форму графічного інтерфейсу в реальному часі.

У серці кожної системи реального часу лежить концепція дедлайну. Очевидним прикладом у відеоіграх є вимога, щоб екран оновлювався принаймні 24 рази на секунду, щоб створити ілюзію руху (більшість ігор відтворюють екран зі швидкістю 30 або 60 кадрів на секунду, оскільки ці значення кратні частоті оновлення монітора NTSC). Звичайно, є багато інших видів дедлайнів у відеоіграх. Наприклад, для стабільності симуляції фізики може бути потрібно оновлювати екран 120 разів на секунду. Системі штучного інтелекту персонажа може потрібно «мислити» принаймні один раз на секунду, щоб уникнути враження нерозумності. Бібліотека аудіо може знадобитися для виклику принаймні один раз за 1/60 секунди, щоб заповнювати аудіо буфери та запобігати слуховим збоям [10].

«М’яка» система реального часу – це система, в якій пропущені дедлайни не катастрофічні. Таким чином, всі відеоігри є системами м’якого реального часу: якщо частота кадрів падає, то людина-гравець, як правило, не постраждає. Порівняйте це з жорсткою системою реального часу, де пропущений дедлайн може означати важкі травми або навіть смерть людини-оператора. Авіаційна система вертольота або система керування в ядерній електростанції – приклади жорстких систем реального часу.

Числові симуляції зазвичай реалізуються за допомогою повторного виконання розрахунків, щоб визначити стан системи на кожному дискретному кроці часу [17]. Ігри працюють так само. Основний «цикл» гри повторюється, і під час кожної ітерації циклу різні ігрові системи, такі як штучний інтелект, ігрова логіка, фізичні симуляції тощо, мають можливість обчислювати або оновити свій стан для наступного дискретного кроку часу. Результати потім відрисовуються, відображаючи графіку, видаючи звук, і, можливо, створюючи інші виходи, такі як зворотний зв’язок вібрацією на геймпаді.

1.4. Інструментальні засоби для ігрової розробки

Ігрова розробка – це процес, що потребує ретельної підготовки та використання різноманітних інструментів для досягнення успішних результатів. Інструменти включають у себе програмне забезпечення для моделювання, редактори графіки та анімації, мови програмування, системи управління версіями та багато іншого. Розуміння та вміння використовувати ці інструменти є ключовими для успішної ігрової розробки, тому їх вивчення є важливою складовою будь-якого професійного геймдеву. У наступних розділах ми докладніше розглянемо кожен з цих інструментів та їхнє значення в процесі розробки ігор.

Гра передбачає багато складників з кута зору розробки [10]:

1) мова програмування та інструменти розробки. Розробка гри вимагає використання певної мови програмування (такої як C++, C#, Python тощо) та інструментів розробки, таких як інтегровані середовища розробки (IDE), компілятори, відладники тощо;

2) менеджер ресурсів: відповідає за завантаження, зберігання, кешування та управління різними типами ресурсів, такими як текстури, моделі, звукові файли, шрифти та інші;

3) двигун керування об’єктами (Object Management Engine) відповідає за створення, управління та взаємодію об’єктів у грі. Управління життєвим циклом об’єктів у грі включає в себе додавання, видалення та оновлення об’єктів залежно від потреб гри;

4) графічний рушій (Graphics Engine) – це компонент, який відповідає за відображення графіки в грі. Він включає в себе різноманітні технології, такі як рендерінг, освітлення, тіні, текстури та інші ефекти. Багато ігрових двигунів мають вбудований графічний двигун;

5) рушій фізики (Physics Engine) – це компонент, який відповідає за моделювання фізичних законів та взаємодію об’єктів у грі. Він може включати в себе обчислення зіткнень, сили тяжіння, імпульсів, динаміки руху та інших аспектів фізики;

6) аудіорушій (Audio Engine) керує звуковими ефектами та музикою в грі. Він включає в себе зчитування та відтворення аудіофайлів, налаштування об’єму, звукові ефекти та інші аспекти аудіо;

7) інтерфейс користувача (User Interface, UI) відповідає за відображення та взаємодію з користувачем в грі. Він може включати в себе елементи, такі як меню, кнопки, поля введення, індикатори та інші інтерфейсні елементи;

8) мережевий рушій (Networking Engine). Якщо гра підтримує мережевий режим, цей компонент відповідає за забезпечення комунікації між різними екземплярами гри через мережу Інтернет або локальну мережу.

Ігрові рушії в тій чи інший мірі покривають ці пункти. Вони є ключовими інструментами у розробці відеоігор, і вони можуть мати різну функціональність залежно від їхнього призначення та специфіки проєкту. Наприклад, Unity – потужний ігровий рушій, який підтримує мову програмування C# для написання скриптів, що дає змогу створювати складну логіку гри. Unity також має вбудовані конвеєри відрисовки (render pipeline – вбудований, URP та HDRP), які дозволяють реалізувати різні візуальні ефекти та оптимізувати якість графіки гри. Крім того, Unity має вбудовані рішення для фізичного моделювання, аудіо та розробки інтерфейсу, від старого IMGUI до сучасного UI Toolkit. Існують і більш компакті ігрові рушії, типу MonoGame, який надає лише базову функціональність для розробки ігор. Він використовується для створення ігор на базі платформи XNA та забезпечує базовий конвекєр відрисовки та менеджер ресурсів. Однак, на відміну від Unity, він не має вбудованих інструментів для фізики, аудіо або розробки інтерфейсу. Розробники, які використовують MonoGame, зазвичай повинні використовувати сторонні бібліотеки або розробляти власні рішення для цих компонентів [2, 3].

Однак для більш точного вирішення певних завдань у розробці ігор часто використовуються спеціалізовані бібліотеки та інструменти. Наприклад, для управління аудіо можуть використовуватися такі бібліотеки, як FMOD або Wwise, для рендерингу – Ogre, для фізичного моделювання – Box2D, Bullet або

PhysX, для створення інтерфейсу — NoesisGUI або ж Ultralight, а для управління об’єктами – EnTT. Ці бібліотеки надають розробникам різні інструменти та можливості для створення якісних ігрових проектів.

Також надзвичайно важлива система контролю версій, коли програмне забезпечення розробляється командою з кількох інженерів. Вона:

– забезпечує центральний репозиторій, з якого інженери можуть ділитися вихідним кодом;

– зберігає історію змін, внесених у кожний вихідний файл;

– надає механізми для тегування і відновлення конкретних версій базового коду;

– дозволяє відгалуження версій коду від основної лінії розробки. Ця можливість часто використовується для створення демонстрацій або випуску патчів для старих версій програмного забезпечення [19].

Система контролю версій може бути корисна навіть у проєкті з одним інженером. Найпоширеніші системи контролю версій:

– Subversion. Відкрита система контролю версій, спрямована на заміну і покращення CVS. Оскільки вона є відкритою, а отже, безкоштовною, вона є доречною для індивідуальних проєктів, студентських проєктів та малих студій.

– Git. Це відкрита система контролю версій, яка використовувалася для багатьох проектів з відкритим первинниим кодом, включаючи ядро Linux. У моделі розробки git програміст вносить зміни до файлів і фіксує зміни до гілки. Потім програміст може швидко і легко об’єднати свої зміни з будь-якою іншою гілкою коду, оскільки git вміє повертати послідовність відмінностей та застосовувати їх на нову базову ревізію – процес, який git називає перебазуванням. Як результат, отримується система контролю версій, яка є дуже ефективною та швидкою при роботі з кількома гілками коду [20].

Git Large File Storage (LFS) – це розширення Git, призначене для ефективного керування великими файлами, такими як графіка, аудіо або відео, що зазвичай використовуються в розробці відеоігор. Зазвичай зберігання таких

великих файлів безпосередньо в репозиторії Git може призвести до збільшення розміру репозиторію, що ускладнює роботу з ним і збільшує час синхронізації.

Git LFS пропонує спосіб замінити великі файли у репозиторії на посилання на зовнішнє сховище (наприклад, сервер LFS), що дає змогу зберігати файли окремо від основного репозиторію. Коли розробники клонують або стягують оновлення з цього репозиторію, Git LFS автоматично завантажує великі файли з сервера LFS, щоб репозиторій залишався легким і компактним [21].

У ході розроблення ігор Git LFS особливо корисний через великі обсяги графічних, аудіо та відео файлів, які зазвичай використовуються в розробці. Він допомагає зберігати ці файли сховищі поза Git репозиторієм, що полегшує спільну роботу над проєктом між розробниками, зменшує час синхронізації та збільшує продуктивність.

Використання Git LFS у геймдеві може бути особливо важливим для команд, які працюють над великими проєктами з великою кількістю графічних і аудіо активів, таких як тривалість відеороликів, текстури, звукові ефекти тощо.

Ігри завжди мають високі вимоги до продуктивності в реальному часі, тому програмісти ігрових рушіїв завжди шукають способи прискорення свого коду. Існує правило Парето (відоме як «80-20»), яке стверджує, що 80% ефектів можуть виникнути через лише 20% причин. У комп’ютерній науці використовується варіант цього правила, відомий як «90-10», де 90% часу виконання програми припадає на лише 10% коду. Для виявлення, яка частина коду споживає більше часу, використовують профайлери. Вони вимірюють час виконання кожної функції та допомагають спрямувати оптимізації саме туди, де це потрібно. Деякі профайлери також показують, скільки разів кожна функція викликається, що є важливою інформацією для визначення кращих напрямів оптимізації [4].

Профайлери можуть бути статистичними або інструментальними. Статистичні профайлери вимірюють час виконання програми без значного сповільнення її роботи. Вони працюють за допомогою вибіркового огляду

регістра програмного лічильника процесора та зауважують, яка функція в даний момент виконується [10].

Інструментальні профайлери надають більш точну та комплексну інформацію за рахунок сповільнення програми. Вони працюють шляхом попередньої обробки вашого виконавчого файлу та вставлення спеціального вступного та заключного коду в кожну функцію. Вступний та заключний код викликає бібліотеку профайлінгу, яка, в свою чергу, перевіряє стек викликів програми та записує різні подробиці, включаючи те, яка батьківська функція викликала функцію, що розглядається, та скільки разів цей батьківський компонент викликав дочірній. Цей тип профайлера може навіть бути налаштований на моніторинг кожного рядка коду в вашій первинній програмі, що дає йому змогу повідомляти, скільки часу займає виконання кожного рядка.

У великих проектах які розробляються баготьма розробниками критично важливим стає написання усього коду в одному стилі. У цій задачі нам допомагають різні лінтери та форматери. Наприклад у лістингу 1.1 подано конфігурацію для clang-format, який використовується у нашому проекті.

Лістинг 1.1 – конфігурація clang-format у проекті

```
Language: Cpp
BasedOnStyle: LLVM

IndentWidth: 2
ContinuationIndentWidth: 2
AlignAfterOpenBracket: false
UseTab: Never

AlignOperands: true
BreakBeforeBraces: Attach

AccessModifierOffset: -2
IndentAccessModifiers: false

AllowShortFunctionsOnASingleLine: None
AllowShortIfStatementsOnASingleLine: Never
AllowShortBlocksOnASingleLine: Never

PointerAlignment: Left

BinPackArguments: false
ColumnLimit: 0
```

Отже, в розділі було проведено аналіз предметної області розробки ігор, зокрема розглянуто різні архітектурні підходи та їхні особливості. Вибір архітектурного підходу визначає не лише ефективність розробки, але й продуктивність кінцевого продукту. Важливо організувати дані так, щоб забезпечити ефективну обробку великої кількості інформації. Це включає в себе кеш-локальність та оптимізацію використання ресурсів процесора і пам’яті. Використання унікальних ідентифікаторів для об’єктів дозволяє легко взаємодіяти з ними та виконувати динамічні операції. Важливу роль у розробці ігор відіграють інструментальні засоби, такі як інтегровані середовища розробки (IDE), набори засобів розробки (SDK), та інші програмні інструменти, що підтримують весь життєвий цикл розробки гри.

РОЗДІЛ 2
АРХІТЕКТУРНІ ПІДХОДИ ДО ІГРОВОЇ РОЗРОБКИ

2.1. Парадигми розробки ігрових додатків

Вибір конкретної архітектури залежатиме від відповіді на цілу низку питань та потреб розробників:

– як можна швидко впроваджувати нову функціональність і наскільки зручно підтримувати код в обраній архітектурі;

– як архітектура дає змогу ефективно розподіляти обов’язки серед розробників у команді;

– оцінки того, наскільки легко архітектура може адаптуватися до змін та розширюватися в разі потреби;

– масштабування за обсягом гравців або об’єктів: аналіз можливостей архітектури для безперервної та стабільної роботи гри при зростанні активності гравців чи об’єктів;

– розгляду специфічних потреб гри та визначення, яка архітектура краще відповідає цим вимогам;

– перевірки того, наскільки обрана архітектура легко інтегрується з використовуваними розробниками інструментами та мовами програмування;

– аналізу можливостей архітектури з точки зору оптимізації використання пам’яті та обчислювальних ресурсів.

– оцінки того, як архітектура впливає на продуктивність гри в реальному часі та рівня затримок;

– використання загальновизнаних патернів для забезпечення архітектурної якості та зручності розробки;

– визначення оптимального балансу між гнучкістю розробки та ефективністю гри.

Об’єктно-орієнтоване програмування (ООП) вже багато років є стандартним підходом до розробки переважної більшості додатків. Воно відображає світ через моделі, які містять стан (атрибути або поля) та

поведінку (методи або функції) об’єктів. Це доречно для моделювання реальних сутностей та сприяє легкості розуміння, тестування та підтримки коду.

В ООП кожен об’єкт володіє своїм власним станом і поведінкою, інкапсульованими в межах класу. Це дає змогу створювати ієрархії класів за допомогою наслідування, що сприяє повторному використанню коду і поліморфізму [18].

Проте традиційне ООП може бути менш ефективним у випадках, коли потрібна висока продуктивність або паралельне виконання. Зазвичай ООП надає більше абстрагування та модульності, але це може вплинути на продуктивність програми.

Для ігрових додатків однією з альтернатив часто стає проєктування, спрямоване на дані (DOD). Воно зосереджується на оптимізації обробки даних для досягнення кращої продуктивності та відділяє дані від логіки, намагаючись максимізувати ефективність кешування та паралельної обробки. У контексті розробки ігор часто застосовують архітектуру Entity-Component-System (ECS), в якій дані організовані в вигляді компонентів, які обробляються системами. Це сприяє покращенню кешування, оскільки дані зберігаються послідовно в пам’яті, і спрощує паралельне виконання, оскільки системи можуть обробляти дані незалежно [16].

Проте ECS може бути менш інтуїтивним для розуміння та вимагає більше уваги до організації даних. Цей підхід часто використовується у вимогливих до продуктивності додатках, таких як ігрові рушії або програми для обробки великих обсягів даних.

2.2. Реалізація архітектури Entity-Component-System

При реалізації Entity-Component-System постає питання, де зберігати компоненти. Невдалий варіант: кожна сутність зберігає свої компоненти самостійно, як показано на рис. 2.1:

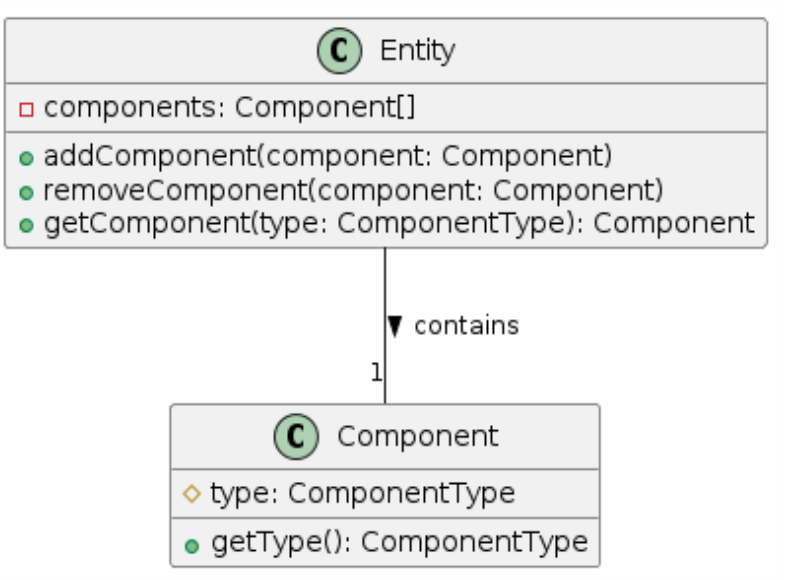

Рисунок 2.1 – Поганий варіант реалізації ECS

Такий підхід до ECS має деякі недоліки, особливо щодо ефективності використання кешу процесора [5]:

– послідовний доступ до даних. У цьому підході дані про сутності та їх компоненти зберігаються разом, тому при обробці сутностей ітерація по компонентам відбувається відносно сутностей. Це може призвести до розриву кешу процесора, оскільки доступ до даних буде не послідовним, що зменшує ефективність роботи процесора;

– неоптимальне використання кеш-пам’яті. Якщо компоненти розташовані в пам’яті далеко один від одного, це може призвести до великої кількості промахів кешу, коли процесор намагається отримати доступ до даних. Це відбувається через те, що розташування даних в пам’яті не враховується в оптимізації кешу;

– наслідування та велика кількість класів. При збільшенні числа типів сутностей та компонентів може збільшитися і кількість класів в системі. Це призводить до більшої складності управління кодом та збільшення часу розробки [12].

Підхід з використанням пулів компонентів є більш ефективним способом організації даних у ECS-системі. Замість того, щоб кожна сутність містила масив

компонентів, у цьому підході кожен тип компонентів зберігається в окремому пулі даних. Перевагами такого підходу є:

– покращена локальність даних. Компоненти одного типу зберігаються разом в пам’яті, що сприяє покращенню локальності даних. Це дає змогу зменшити кількість кеш-промахів та полегшує роботу з кешем процесора;

– контроль пам’яті. Пули компонентів можуть ефективно керувати пам’яттю, дозволяючи динамічно виділяти та звільняти пам’ять для компонентів. Це дає змогу уникнути зайвого використання пам’яті та оптимізувати його;

– простота і швидкість доступу. Доступ до компонентів здійснюється через індексацію в пулі даних, що забезпечує швидкий та простий доступ до необхідних даних;

– менша кількість класів. За рахунок використання пулів компонентів можна уникнути створення великої кількості класів для кожного типу сутностей та компонентів. Це зменшує складність системи [5].

На рисунку 2.2 наведено, як пули компонентів інтегруються у цю архітектуру

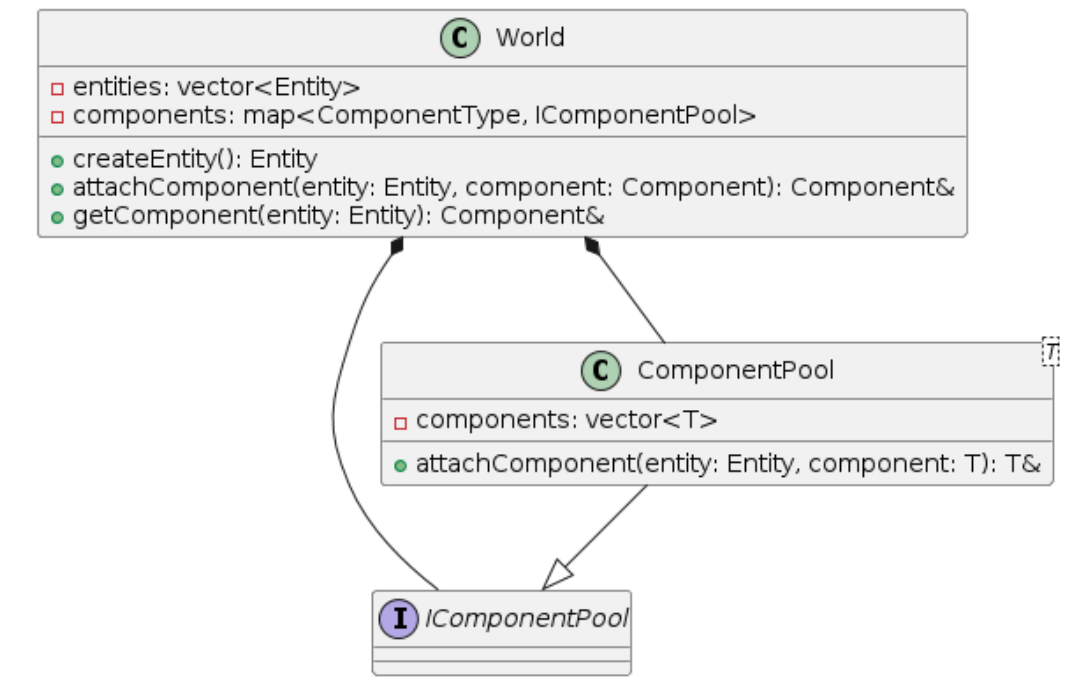

Рисунок 2.2 – Пули компонентів та світ

Код для пулу компонентів, реалізованого мовою C++, показано в лістингу 2.1.

Лістинг 2.1 – Код імплементації пулу компонентів мовою C++

```
template <typename Component>
class ComponentPool final : public IComponentPool {
public:
  Component& attachComponent(const Entity& entity, Component& component) noexcept {
    size_t index = m_components.size();
    m_components.push_back(component);

    m_entityToIndex.insert(std::make_pair(entity, index));
    m_indexToEntity.insert(std::make_pair(index, entity));

    return component;
  }

  void destroyEntity(Entity entity) override {
    if (m_entityToIndex.count(entity) == 0)
      return;

    size_t index = m_entityToIndex[entity];
    m_components.erase(m_components.begin() + index);
    m_entityToIndex.erase(m_entityToIndex.find(entity));
    m_indexToEntity.erase(m_indexToEntity.find(index));
  }

  bool hasEntity(Entity entity) override {
    return m_entityToIndex.count(entity);
  }

  Component& getByEntity(Entity entity) {
    if (!m_entityToIndex.count(entity))
      throw std::invalid_argument("ComponentPool doesn't have an entity");

    return m_components[m_entityToIndex[entity]];
  }

private:
  std::vector<Component> m_components;
  std::map<Entity, size_t> m_entityToIndex;
  std::map<size_t, Entity> m_indexToEntity;
};
```

Системи можуть отримати доступ до сутностей/компонентів через клас World (ігровий світ). Зазвичай системи працюють з певним набором компонентів, наприклад система Movement працювала б з компонентами Position та Velocity. Для того, щоб це було можливо, нам потрібен деякий механізм фільтрації сутностей за їх компонентами. Один з підходів — це механізм запитів до світу [6].

В ECS сутності складаються з компонентів, які визначають їх властивості або характеристики. Механізм запитів дозволяє вибирати сутності, які відповідають певним критеріям.

У ECS механізм запитів зазвичай реалізується через системи запитів або фільтри. Ці системи або фільтри аналізують сутності та їх компоненти та відбирають ті, які відповідають заданим умовам. Наприклад, якщо ви маєте ECS з компонентами «Позиція», «Швидкість» та «Тип», ви можете створити запит, щоб вибрати всі сутності з компонентом «Тип», який дорівнює «Ворог», та використати їх для обробки відповідно до логіки вашої гри або програми. Рис. 2.3 демонструє як системи взаємодіють зі світом

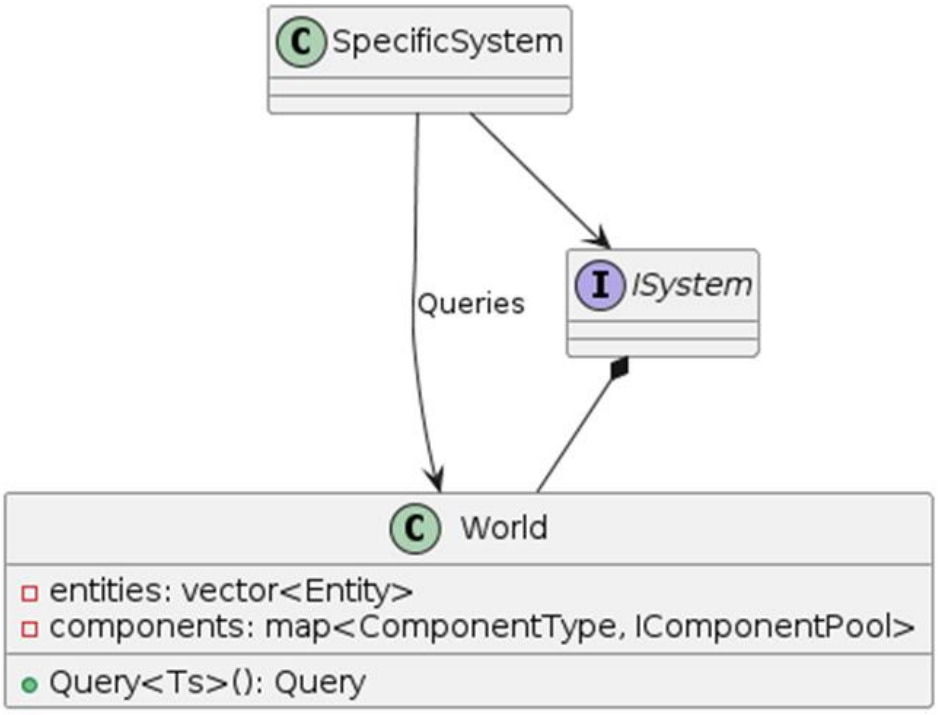

Рисунок 2.3 – Взаємодія системи зі світом

Лістинг 2.2 демонструє можливу реалізацію класу, що описуватиме ігровий світ в межах ECS-архітектури.

Лістинг 2.2 – Клас світу на C++:

```
class World {
public:
  Entity createEntity() {
    Entity newEntity;

    if (m_availableIds.empty()) {
      newEntity = m_lastEntity++;
    } else {
      newEntity = m_availableIds.back();
      m_availableIds.pop_back();
    }

    m_entities.push_back(newEntity);

    return newEntity;
  }

  template <typename Component>
  Component& attachComponent(Entity entity, Component component) {
    const char* type = typeid(Component).name();

    if (!m_components.count(type))
      m_components.insert(std::make_pair(type,
std::make_shared<ComponentPool<Component>>()));

    return
std::static_pointer_cast<ComponentPool<Component>>(m_components[type])->attachComponent(
entity, component);
  }

  template <typename Component>
  bool hasComponent(Entity entity) {
    const char* type = typeid(Component).name();

    if (!m_components.count(type))
      return false;

    return m_components[type]->hasEntity(entity);
  }

  template <typename Component>
  Component& getComponent(Entity entity) {
    const char* type = typeid(Component).name();

    if (!m_components.count(type))
      throw std::invalid_argument("ComponentPool doesn’t have an entity");
```

```
    return
std::static_pointer_cast<ComponentPool<Component>>(m_components[type])->getByEntity(entity
);
  }

  template <typename Func>
  void forEach(Func function) {
    for (const auto& entity : m_entities)
      function(entity);
  }

  template <typename... Components, typename Func>
  void forEach(Func function) {
    for (const auto& entity : m_entities)
      if ((m_components[typeid(Components).name()]->hasEntity(entity) && ...))
        function(entity,
std::static_pointer_cast<ComponentPool<Components>>(m_components[typeid(Components).nam
e()])->getByEntity(entity)...);
  }

  template <typename... Components, typename Func>
  void forEachWith(Func function) {
    for (const auto& entity : m_entities)
      if ((m_components[typeid(Components).name()]->hasEntity(entity) && ...))
        function(std::static_pointer_cast<ComponentPool<Components>>(m_components[typeid(Co
mponents).name()])->getByEntity(entity)...);
  }

  void addSystem(std::unique_ptr<ISystem> system) {
    m_systems.push_back(std::move(system));
  }

  void update() {
    for (auto& system : m_systems)
      system->update(*this);
  }

private:
  std::vector<Entity> m_entities;
  std::vector<Entity> m_availableIds;
  Entity m_lastEntity;

  std::map<const char*, std::shared_ptr<IComponentPool>> m_components;
  std::vector<std::unique_ptr<ISystem>> m_systems;
};
```

Приклад того, як може виглядати архітектура ігрової системи, наведено на рис. 2.4.

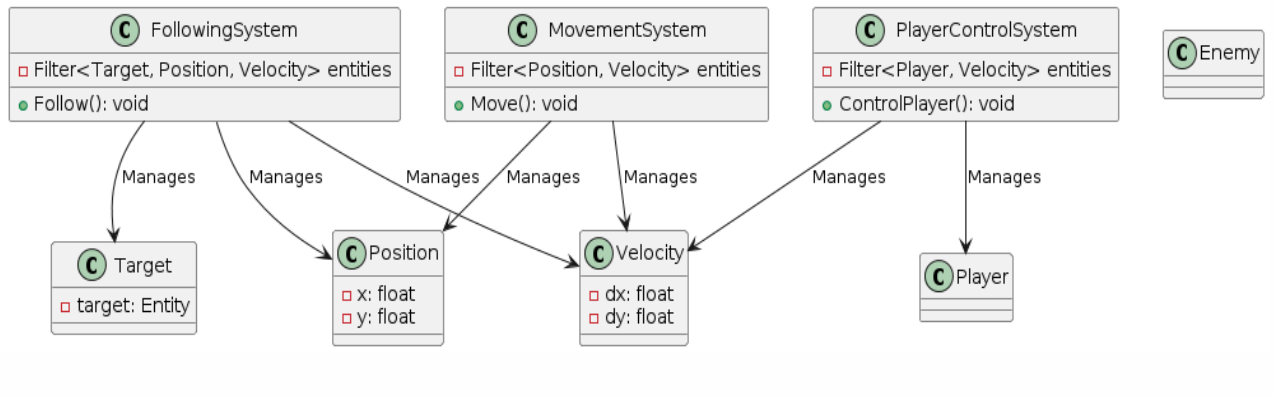

Рисунок 2.4 – Діаграма класів, яка відображає системи і компоненти в ігровому додатку

Як можна побачити з цієї діаграми, не всі компоненти повинні нести у собі якусь інформацію. Іноді використовується також спеціальні компоненти-«маркери», які несуть інформацію своєю присутністю або відсутністю. Наприклад, як показано на діаграмі, такі компоненти можна використовувати для ідентифікації сутностей. Такі маркери називаються «Entity Tags».

Використання таких маркерів (Unity DOTs) може виглядати так, як показано в лістингу 2.3 [6].

Лістинг 2.3 – Використання компонентів-маркерів

```
foreach (var moveable in SystemAPI.Query<RefRW<MoveableComponent>>()
                                  .WithAll<PlayerComponent>()) {
    moveable.ValueRW.Velocity = input;
}
```

Тут ми виконуємо запит на компоненти MoveableComponent і за допомогою WithAll() вказуємо, що нам потрібні тільки ті, сутності яких мають компонент PlayerComponent [9].

2.3. Завантаження ресурсів до гри та Asset Pipeline

Asset Pipeline призначений для оптимізації та вдосконалення процесу керування та завантаження ресурсів у вашому проєкті. Цей модуль забезпечує ефективну обробку та оптимізацію графічних, звукових та інших типів ресурсів.

Під час розробки ігор необроблені ресурси, такі як графіка, аудіо та інші мультимедійні елементи, часто спочатку не оптимізовані для ефективного використання. Ці ресурси можуть надходити у форматах із високою роздільною здатністю, що може вплинути на продуктивність гри, збільшити час завантаження та займати непотрібний простір для зберігання.

Модуль Asset Pipeline вирішує цю проблему, слугуючи життєво важливим проміжним кроком у процесі розробки. Він бере необроблені, неоптимізовані ресурси та перетворює їх у формат, придатний для бездоганної інтеграції та оптимальної продуктивності в грі. Загальний хід виконання операцій продемонстровано на рис. 2.5.

За допомогою Asset Pipeline грі не потрібно знати як завантажувати десятки різних форматів зображень, так як Asset Pipeline переведе усі зображення у єдиний формат. За рахунок цього, також стає можливим зберегання усього проекту в одному репозиторії: тепер дизайнери можуть імпортувати у гру ресурси у таких форматах як .psd і це не призвиде не до яких складностей.

Рисунок 2.5 – Хід роботи Asset Pipeline

Отже, в нашому випадку Asset Pipeline було розроблено на C++. Він складається з трьох модулів:

– Asset Bundler – це повноцінна консольна програма, яка дозволяє компілювати ресурси;

– Asset Manager – це бібліотека, за допомогою якої стає можливим загружати ресурси до програми;

– Pipes – це набір класів для обробки певних типів ресурсів.

Від Asset Pipeline вимагається мати можливість гнучко керувати ресурсами, при цьому так, щоб вони займали якомога менше пам’яті, й завантажувались до додатку якомога швидше. При цьому іноді виникають специфічні випадки, коли, наприклад, треба зробити певну попередню обробку ресурсів. Деякі ресурси повинні генеруватися з інших, наприклад шейдери чи певні скрипти можуть бути скомпільовані. Розроблений конвеєр активів дає змогу все це налаштовувати через файли конфігурацій.

Наприклад, у лістингу 2.3 показано конфігураційний файл, який буде запускати побудову UI, а потім збирати результати в бандл.

Лістинг 2.3 – конфігураційний файл для Asset Pipeline

```
preprocessing:
  execute: "pnpm i && pnpm build"

exports:
  - path: "(dist).*"
    pipe: default
```

Приклад організації ресурсів можна побачити на рисунку 2.6

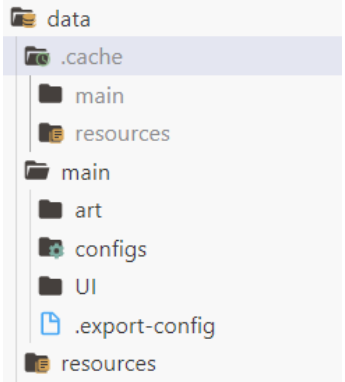

Рисунок 2.6 – Початкові ресурси для двох бандлів

Приклад запуску asset-bundler з IDE після збірки показано на рисунку 2.7.

```
PS C:\Users\NeutrinoZh\dev\cpp\dreich\build\wind\core\asset-pipeline\asset-bundler\Debug> ."C:/Users/N
eutrinoZh/dev/cpp/dreich/build/wind/core/asset-pipeline/asset-bundler/Debug/wind-asset-bundler.exe"
Asset-Bundler Tool. Version: 0.3a.
Is part of the Wind project.
Distributed under the terms of the MIT License.

Usage:
  wind-asset-bundler [OPTION...]

  -h, --help       Print usage
  -b, --build arg  Build directory
```

Рисунок 2.7 – запуск asset-bundler у терміналі

Для того, щоб автоматизувати процес збірки ресурсів, було розроблено CMake-функцію (лістинг 2.4), яка викликає asset-bundler кожного разу, коли запускається збірка проєкту.

Лістинг 2.4 – CMake функція для збірки проекту і всіх залежних бандлів ресурсів

```
function(add_wind_game TARGET_NAME BUNDLE_LIST)
  add_executable(${TARGET_NAME} ${ARGN})

  set(BUNDLES_TARGETS "")
  foreach(BUNDLE ${BUNDLE_LIST})
    set(BUDNLE_NAME ${TARGET_NAME}_${BUNDLE}_bundle)

    add_custom_target(
      ${BUDNLE_NAME} ALL
      COMMAND wind-asset-bundler -b ${BUNDLE}
      WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}/data/
      COMMENT "Build bundle: ${BUNDLE}"
    )

    list(APPEND BUNDLES_TARGETS ${BUDNLE_NAME})
  endforeach()

  add_custom_target(${TARGET_NAME}_bundles ALL DEPENDS ${BUNDLES_TARGETS})

  add_custom_command(TARGET ${TARGET_NAME}_bundles POST_BUILD
    COMMAND ${CMAKE_COMMAND} -E copy_directory
"${CMAKE_CURRENT_SOURCE_DIR}/res"
"$<TARGET_FILE_DIR:${TARGET_NAME}>/res"
  )

  target_link_libraries(${TARGET_NAME} PRIVATE wind-core)
  add_dependencies(${TARGET_NAME} ${TARGET_NAME}_bundles)
endfunction()
```

Також asset-bundler був впроваджений у наш CI/CD процес, де він окремо збирається і потім використовується для збірки усіх ресурсів у кожному підпроєкті, що продемонстровано на рис. 2.8.

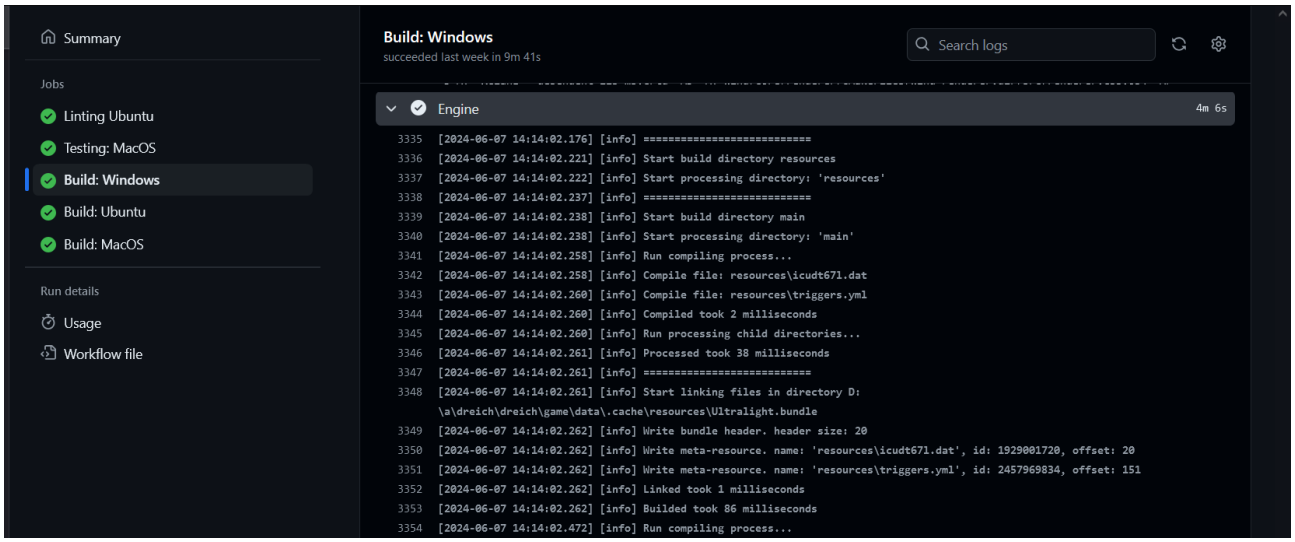

Рисунок 2.8 – Логи asset-bundler з CI на GitHub Actions

У результаті роботи asset-bundler ми отримуємо упаковані й оптимізовані ресурси. Кожен бандл має свій Table of Content (таблицю контента) у якій записано id ресурса, яке генерується як хеш імені, та зміщення у байтах до початку ресурса відносно початку файлу.

2.4. Реалізація скінченного автомата в ECS

Використання скінченного автомата (StateMachine) в архітектурі ECS дає змогу ефективно керувати складними поведінками сутностей шляхом розділення їх на окремі стани. Це допомагає підтримувати код чистим і модульним, забезпечуючи легке додавання нових станів і поведінок [13]. Реалізація StateMachine у ECS трохи відрізняється від класичної, що відображено на рис. 2.9. Тобто треба реалізувати його у межах систем та компонентів.

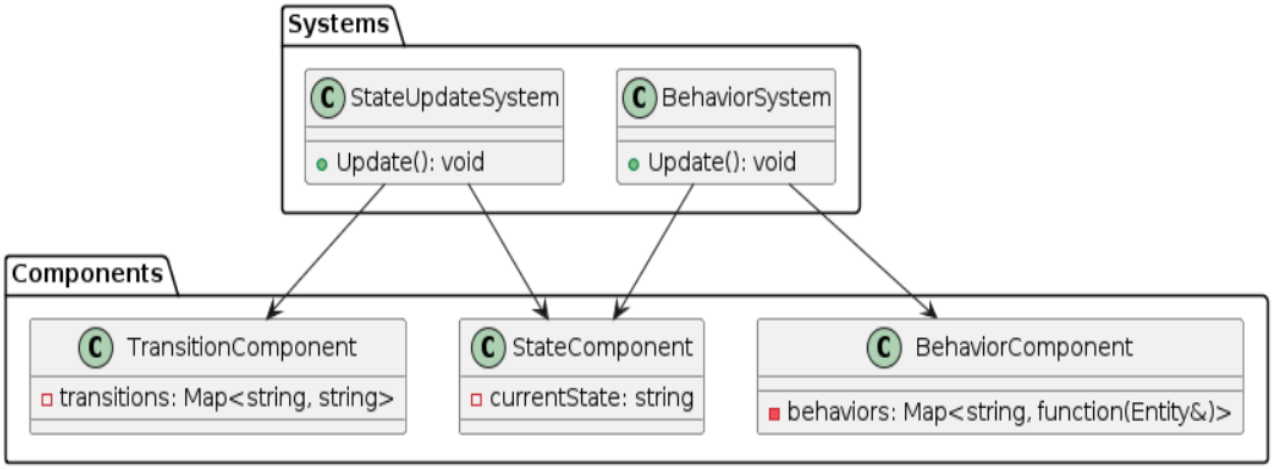

Рисунок 2.9 – Реалізація скінченного автомата в ECS

Використання подібного може виглядати так, як показано в лістингу 2.5.

Лістинг 2.5 – Використання скінченного автомата для управління станами

```
Entity enemy;

enemy.AddComponent<StateComponent>({"Idle"});
enemy.AddComponent<TransitionComponent>({
   { "Idle", "Patrol" },
   { "Patrol", "Chase" },
   { "Chase", "Attack" }
});
enemy.AddComponent<BehaviorComponent>({
   { "Idle", [](Entity& entity) { /* логіка бездіяльності */ } },
   { "Patrol", [](Entity& entity) { /* логіка патрулювання */ } },
   { "Chase", [](Entity& entity) { /* логіка переслідування */ } },
   { "Attack", [](Entity& entity) { /* логіка атаки */ } }
});
```

2.5. Реалізація шаблону Command у ECS

Шаблон Команда (Command) є потужним інструментом для реалізації дій, які можуть бути збережені, скасовані, або повторені. Він особливо корисний в ігровій розробці для обробки введення користувача або інших подій, таких як атаки, переміщення або використання предметів. В архітектурі ECS шаблон Команда можна використовувати для розмежування логіки дій від сутностей та систем, що полегшує управління складними взаємодіями. Діаграма класів на рис. 2.10 демонструє реалізацію шаблону «Команда» у ECS архітектурі.

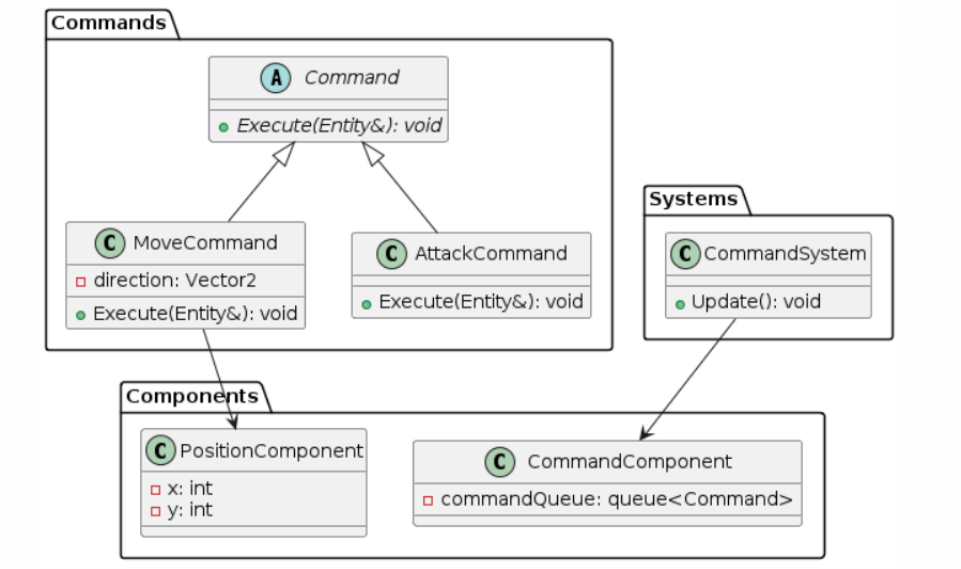

Рисунок 2.10 – Шаблон Команда в архітектурі ECS

Приклад використання шаблону Команда в коді наведено в лістингу 2.6.

Лістинг 2.6 – Використання шаблону Команда в ECS-архітектурі

```
Entity player;
player.AddComponent<CommandComponent>();

// Додавання команди переміщення
Vector2 direction = {1, 0}; // Наприклад, переміщення вправо
auto moveCommand = std::make_shared<MoveCommand>(direction);
player.GetComponent<CommandComponent>().commandQueue.push(moveCommand);

// Додавання команди атаки
auto attackCommand = std::make_shared<AttackCommand>();
player.GetComponent<CommandComponent>().commandQueue.push(attackCommand);
```

Отже, при розгляді ECS важливо враховувати його переваги та недоліки у порівнянні з MVC, враховуючи розмір проєкту, його складність та потреби в гнучкості та швидкодії. ECS має перевагу у масштабованості та гнучкості, особливо для геймдеву, де існує потреба у динамічному додаванні та видаленні об’єктів. Також ECS полегшує внесення змін та додавання нової функціональності без серйозних впливів на решту системи і дає змогу легко пристосовувати гру до різних сценаріїв та вимог. У результаті була реалізована ECS-бібліотека, яка надає можливість створювати ефективні дата-орієнтовані застосунки, а також Asset-pipeline для ефективного керування ресурсами.

РОЗДІЛ 3
ПОРІВНЯННЯ ІГРОВИХ АРХІТЕКТУР

Архітектура відіграє ключову роль у розробці ігор, оскільки вона визначає основну структуру та організацію гри. Аналіз вимог до ігрових архітектур є критичним етапом у визначенні оптимального рішення для ефективної розробки та масштабованості. Забезпечення відповідності обраної архітектури конкретним вимогам проекту дозволить побудувати стабільну, ефективну та масштабовану ігрову систему, здатну відповідати потребам сучасного геймдеву [5].

Порівняльний аналіз обох підходів допоможе зрозуміти їх відмінності та застосування:

– ООП частіше використовується для загальних застосувань та розробки програмного забезпечення, в той час як DOD частіше використовується для вимогливих до продуктивності застосувань, таких як ігрові двигуни або наукові обчислення;

– ООП може бути більш інтуїтивним для розуміння та використання, особливо для початківців, тоді як DOD може вимагати більше досвіду та розуміння для ефективного використання;

– DOD часто забезпечує кращу продуктивність за рахунок оптимізації обробки даних, але це може бути менш важливим для загальних застосувань, де основний акцент робиться на легкості використання та розуміння.

3.1. Аналіз архітектурного шаблону MVC для реалізації ігор

В контексті об’єктно-орієнтованого програмування традиційними архітектурними шаблонами є MVC (Model-View-Controller) та DI (Dependency Injection). Ці архітектурні підходи часто використовуються для розділення логіки програми та полегшення тестування та обслуговування коду.

Модель може представляти ґрунтовні дані гри. Вона може включати головні об’єкти гри, такі як гравець, вороги, об’єкти середовища тощо.

Представлення може бути реалізоване як класи, відповідальні за відображення даних користувачеві та обробку введення. Вони не повинні містити логіки гри, а лише відображати дані.

Контролери можуть слугувати посередниками між представленням та моделлю, оброблюючи введення від користувача та оновлюючи модель відповідно. Контролери можуть бути відповідальні за взаємодію з різними системами, такими як введення, звук чи фізика.

Разом з цим, впровадження залежностей (DI) є механізмом, який може допомагати вводити залежності між різними компонентами гри, забезпечуючи слабку зв’язаність і полегшуючи тестування та обслуговування коду [8]. При використанні MVC разом з DI важливо враховувати, що це лише шаблон, і реалізація може змінюватися залежно від конкретних вимог та потреб проекту. Головна ідея – розділити логіку гри, представлення та обробку введення для полегшення тестування та збереження кодової бази добре структурованою. Рисунок 3.1 демонструє загальний принцип взаємодії компонентів архітектури в такому випадку.

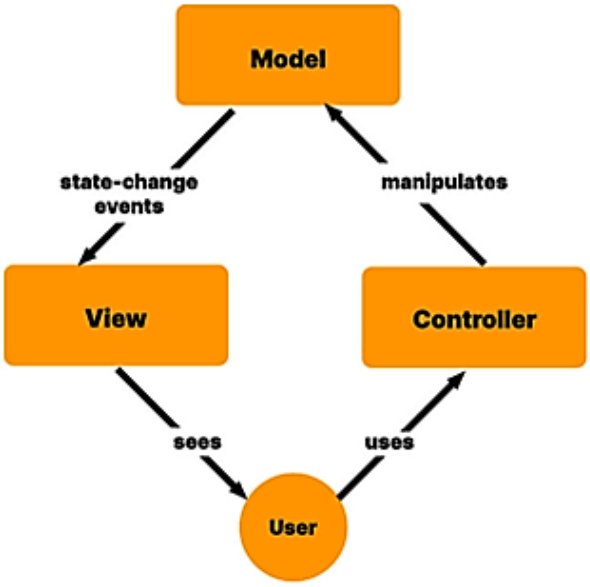

Рисунок 3.1 – Візуалізація взаємодії в архітектурному шаблоні MVC

3.2. Аналіз архітектурного підходу DoD / ECS

Архітектурний підхід ECS відповідає ряду вимог, які дозволяють розробникам більш ефективно та гнучко управляти компонентами та поведінкою об’єктів у грі [5, 6]. Розподіл відповідальностей при такому способі побудови архітектури додатку продемонстровано на рис. 3.2

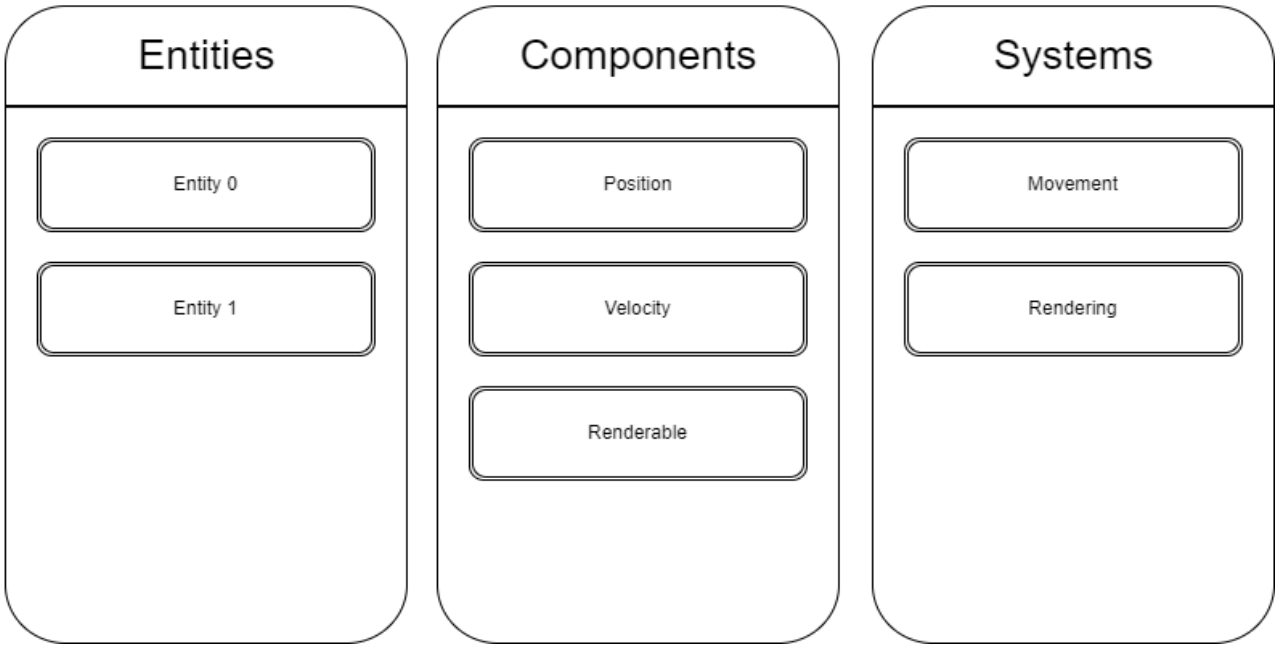

Рисунок 3.2 – Приклад розподілу відповідальностей за ECS

Обираючи між MVC та ECS важливо враховувати розмір проєкту, його складність та потреби в гнучкості та швидкодії. Кожен підхід має свої переваги та недоліки, і вибір залежить від конкретних вимог та характеристик проекту [7].

3.3. Метрики коду

Існують деякі метрики коду, за якими ми можемо оцінити рішення вже зараз:

– глибина спадкування: оскільки ECS більше функціонально-орієнтований, то глибина спадкування тут мінімальна, а у деяких реалізація його може не бути зовсім;

– зв’язаність класів: ECS має низьку зв’язаність класів – підсистеми мають деякі залежності від компонентів, але ні ті, ні інші не взаємодіють між

собою. За рахунок малої зв’язаності коду можливе легке впровадження розпаралелювання, як показано в лістингу 3.1.

Лістинг 3.1 – Приклад коду для обробки систем в паралельних потоках

```
#include <thread>
#include <vector>

// Припустимо, що у нас є клас System і його нащадки
class System {
public:
    virtual void Update() = 0;
};

// Кожна система оновлюється в окремому потоці
void RunSystem(System* system) {
    system->Update();
}

int main() {
    std::vector<System*> systems = { /* ініціалізація систем */ };

    std::vector<std::thread> threads;
    for (auto& system : systems) {
        threads.emplace_back(RunSystem, system);
    }

    for (auto& thread : threads) {
        thread.join();
    }

    return 0;
}
```

– кількість рядків коду та масштабованість: при додаванні нових механік до додатку на ECS нерідко може сильно зростати кількість рядків коду через те, що ви не зможете створити клас «Гравець» який буде відповідати за усю логіку гравця. У ECS вам потрібно буде створювати нову систему на кожну поведінку. Тобто, у вас вже може бути InputSystem, MoveSystem, DeadSystem, AttackSystem тощо. Тож у кількості рядків коду ECS буде програвати, але насправді це все сприяє масштабованості, оскільки так виявляється розділення відповідальностей, за рахунок чого ці класи буде легше підтримувати, і легше перевикористовувати;

– продуктивність: ECS легко піддається розпаралелюванню за рахунок того, що кожна підсистема є незалежною. Враховуючи це, а також те, що така архітектура в цілому краще використовує ресурси комп'ютера, ECS буде більш продуктивним рішенням у більшості випадків. У лістингу 3.2 показано код бенчмарку порівняння запуску реалізацій архітектурних підходів.

Лістинг 3.2 – Порівняння продуктивності реалізацій на базі ООП та ECS

```
#include <iostream>
#include <vector>
#include <chrono>

// Традиційний ООП
class GameObject {
public:
    int x, y;
    void Update() {
        x++;
        y++;
    }
};

void TraditionalOOPBenchmark() {
    std::vector<GameObject> objects(100000000);
    auto start = std::chrono::high_resolution_clock::now();

    for (auto& obj : objects) {
        obj.Update();
    }

    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> duration = end - start;
    std::cout << "Traditional OOP duration: " << duration.count() << " seconds\n";
}

// ECS
struct Position {
    int x, y;
};

struct Velocity {
    int dx, dy;
};

class ECS {
public:
    std::vector<Position> positions;
    std::vector<Velocity> velocities;
```

```
    void Update() {
        for (size_t i = 0; i < positions.size(); ++i) {
            positions[i].x += velocities[i].dx;
            positions[i].y += velocities[i].dy;
        }
    }
};

void ECSBenchmark() {
    ECS ecs;
    ecs.positions.resize(100000000, {0, 0});
    ecs.velocities.resize(100000000, {1, 1});
    auto start = std::chrono::high_resolution_clock::now();

    ecs.Update();

    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> duration = end - start;
    std::cout << "ECS duration: " << duration.count() << " seconds\n";
}

int main() {
    TraditionalOOPBenchmark();
    ECSBenchmark();
    return 0;
}
```

Результати запуску часового замірювання: реалізація на основі традиційного об’єктно-орієнтованого програмування виконувалась 1,7516с, а реалізація на основі ECS – 1.30224с. За рахунок кращого використання кешу процесора, ECS показує кращі результати.

3.4. Приклади ігрових додатків з різними архітектурами

Переважно архітектуру ігрового додатку визначає ігровий рушій, на якому ця гра пишеться. У той же час, деякі з популярних ігрових рушіїв підтримують декілька архітектур одночасно. Наприклад, Unity хоча й спочатку був суто ООП-рушієм, в останні роки активно розвиває свій дата-орієнтований стек, у тому числі Entity-Component-System фреймворк. Нещодавно було випущено демонстраційний проєкт Unity Megacity Metro, в якому показано переваги дата-орієнтованої архітектури (рис. 3.3). На прикладі цього проєкту можна побачити,

як використання ECS дає змогу обробляти величезні кількості об’єктів та анімацій у реальному часі.

MegaCity Metro містить приблизно 4,5 мільйона об’єктів, включаючи понад 100 000 окремих будівель. Завдяки ECS, система Unity здатна управляти цими об’єктами одночасно, забезпечуючи високу продуктивність. Важливість ECS у цьому контексті стає очевидною в світлі ефективності розподілу обробки даних між ядрами процесора.

Рисунок 3.3 – Демонстраційна гра Unity MegaCity Metro

Проєкт також включає близько 5 000 динамічних NPC (неігрових персонажів), що пересуваються по місту. ECS-архітектура дає змогу кожному NPC мати свої унікальні поведінкові патерни, зберігаючи високу продуктивність завдяки паралельній обробці та оптимізованому використанню пам’яті.

На сьогоднішній день подібні масштабні проєкти, які мають високі вимоги до продуктивності, в основному розробляються саме на дата-орієнтованому стеці. Проте багато менш вимогливих додатків все ще пишеться на традиційному об’єктно-орієнтованому підході, оскільки це може економити час розробників.

Наприклад, Terraria (рис. 3.4) реалізована засобами XNA Framework з використанням класичного ООП.

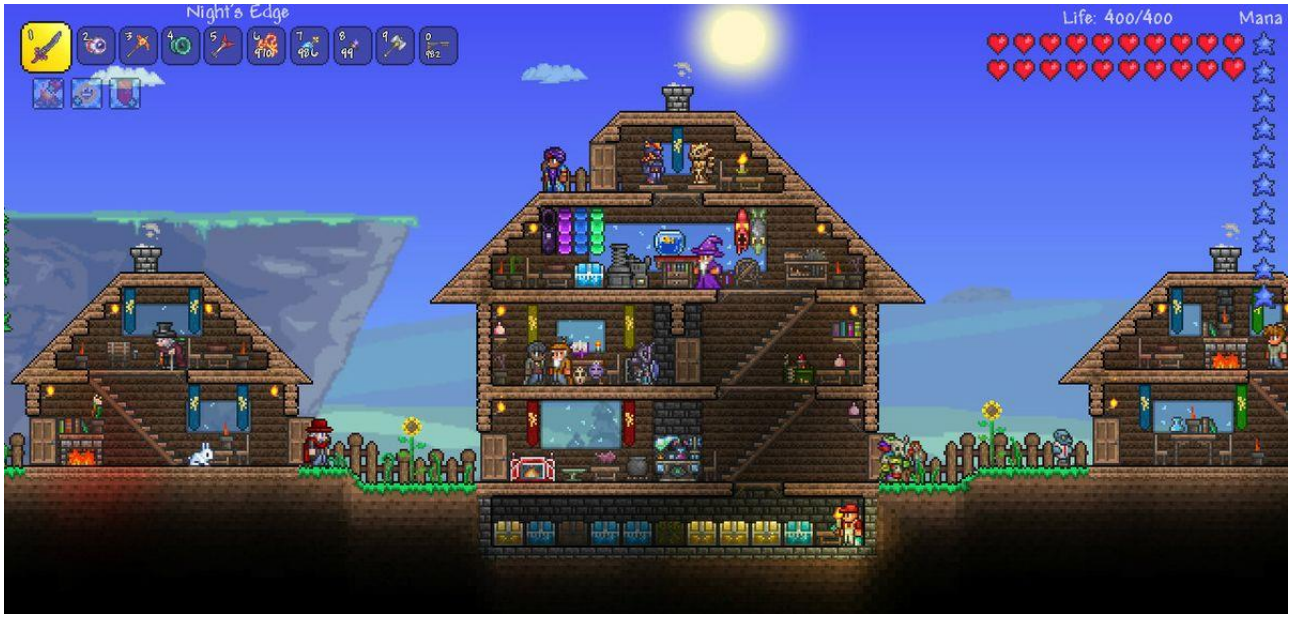

Рисунок 3.4 – Скріншот з гри, розробленої на основі об’єктно-орієнтованого підходу

Розглянувши основні архітектурні підходи, що лежать в основі розроблення ігрових додатків, можна зауважити, що, як і для інших типів додатків, присутні специфічні підходи залежно від потреб користувачів та можливостей розробників. У той же час, це накладає на команди розробки додаткові вимоги щодо рівня досвіду та навичок спеціалістів. Звідси, проведений вище порівняльний аналіз є доречним в якості дослідження сучасного стану та перспектив ігрових рушіїв та інших інструментів розробки.

ВИСНОВКИ

У ході виконання дипломної роботи було досліджено та проаналізовано архітектури, які можна використовувати для розробки ігор, у часності сучасний дата-орієнтований підхід з архітектурним паттерном Entity-Component-System. Розроблено бібліотеку Entity-Component-System (ECS), яка дає змогу створювати ефективні дата-орієнтовані застосунки. Ця бібліотека оптимізує управління компонентами та сутностями, забезпечуючи високу продуктивність і гнучкість. Також було створено Asset-pipeline для ефективного керування ресурсами. Це забезпечує зручне завантаження, зберігання та використання ресурсів (таких як текстури, моделі, звуки тощо) у процесі розробки та виконання ігрового додатку.

Було розглянуто застосування традиційних шаблонів проєктування в ігровій розробці, і, зокрема, в поєднанні з Entity-Component-System. Використано сучасні інструменти розробки, включаючи інтегровані середовища розробки, системи управління версіями та інші програмні інструменти, необхідні для ефективного створення ігор.

За результатами виконання кваліфікаційної роботи були розроблені демонстраційні прототипи для проведення архітектурного аналізу. Зроблено висновки щодо необхідності інвестування в навчання для ефективного використання ECS та надано рекомендації щодо використання інструментів для автоматизації управління пам’яттю та оптимізації розробки.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Українська студія GSC Game World уточнила, скільки людей працює над S.T.A.L.K.E.R. 2. *Новини України - останні новини України сьогодні - УНІАН*. URL: https://www.unian.ua/games/ukrajinska-studiya-gsc-game-world-utochnila-skilki-lyudey-pracyuye-nad-s-t-a-l-k-e-r-2-igri-11684839.html (дата звернення: 25.05.2024).

2. Atobiloye O. Game Development Patterns and Architectures in JavaScript by Olayinka Atobiloye – GitNation. *Open source technology conferences: JavaScript, React, Vue, NodeJS – GitNation*. URL: https://portal.gitnation.org/contents/game-development-patterns-and-architectures-in-javascript (дата звернення: 25.05.2024).

3. Cervantes H., Kazman R. Designing Software Architectures: A Practical Approach. Pearson Education, Limited.

4. Contributors to Wikimedia projects. Pareto principle - Wikipedia. *Wikipedia, the free encyclopedia*. URL: http://en.wikipedia.org/wiki/Pareto_principle (дата звернення: 25.05.2024).

5. ECS for Unity. *Unity*. URL: https://unity.com/ecs (дата звернення: 25.05.2024).

6. Entities overview | Entities | 1.2.1. *Unity - Manual: Unity User Manual 2022.3 (LTS)*. URL: https://docs.unity3d.com/Packages/com.unity.entities@1.2/manual/ (дата звернення: 25.05.2024).

7. Game Development Lifecycle Models | Studytonight. *Studytonight - Best place to Learn Coding Online*. URL: https://www.studytonight.com/3d-game-engineering-with-unity/game-development-models (дата звернення: 25.05.2024).

8. GitHub - modesttree/Zenject: Dependency Injection Framework for Unity3D. *GitHub*. URL: https://github.com/modesttree/Zenject#aot-support (дата звернення: 25.05.2024).

9. GitHub - Unity-Technologies/EntityComponentSystemSamples. GitHub. URL: https://github.com/Unity-Technologies/EntityComponentSystemSamples/tree/master (дата звернення: 25.05.2024).

10. Gregory J. Game Engine Architecture. A K Peters/CRC Press, 2009. URL: https://doi.org/10.1201/b10681 (дата звернення: 25.05.2024).

11. Guide on Stages of Game Development: From Concept To Release. *iLogos Game Studios* -. URL: https://ilogos.biz/stages-of-game-development-your-guide-on-game-development-process/ (дата звернення: 25.05.2024).

12. Kalbaska Y. Clean Architecture in Game Development. *Medium*. URL: https://betterprogramming.pub/clean-architecture-in-game-development-e57542a96e5e (дата звернення: 25.05.2024).

13. Table of Contents · Game Programming Patterns. *Game Programming Patterns*. URL: https://gameprogrammingpatterns.com/contents.html (дата звернення: 25.05.2024).

14. Wen J. Data Driven System Engineering: Automotive ECU Development. DDSE Consulting, LLC, 2022.

15. What is Agile? | Atlassian. *Atlassian*. URL: https://www.atlassian.com/agile (дата звернення: 25.05.2024).

16. What’s an Entity System? - Entity Systems Wiki. *What’s an Entity System? - Entity Systems Wiki*. URL: http://entity-systems.wikidot.com/ (дата звернення: 25.05.2024).

17. Учасники проектів Вікімедіа. Фізичний рушій – Вікіпедія. *Вікіпедія*. URL: https://uk.wikipedia.org/wiki/Фізичний_рушій (дата звернення: 25.05.2024).

18. Rollings A. Game architecture and design. verIndianapolis, Ind : New Riders, 2004. 926 p.

19. What is version control | Atlassian Git Tutorial. *Atlassian*. URL: https://www.atlassian.com/git/tutorials/what-is-version-control (дата звернення: 25.05.2024).

20. GitLab. What is version control? The most-comprehensive AI-powered DevSecOps platform | GitLab. URL: https://about.gitlab.com/topics/version-control/ (дата звернення: 25.05.2024).

21. Git Large File Storage. URL: https://git-lfs.com/ (дата звернення: 25.05.2024).