МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Державне некомерційне підприємство
«Державний університет» Київський авіаційний інститут»

Факультет комп'ютерних наук та технологій

Кафедра інженерії програмного забезпечення

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри

_____ Олена ГРІНЕНКО
«_____» _____ 2025 р.

КВАЛІФІКАЦІЙНА РОБОТА
(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ЗДОБУВАЧА ОСВІТНЬОГО СТУПЕНЯ «МАГІСТР»

Тема: Методика вдосконалення архітектури ігрового застосунку на рушії "Unreal Engine 5"

Виконавець: Хохлов Євгеній Олександрович

Керівник: к.ф.-м.н. Гололобов Дмитро Олександрович

Нормоконтролер: к.ф.-м.н. Гололобов Дмитро Олександрович

Київ 2025

Державне некомерційне підприємство
«Державний університет» Київський авіаційний інститут»

Факультет комп'ютерних наук та технологій
Кафедра інженерії програмного забезпечення
Спеціальність 121 «Інженерія програмного забезпечення»
Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Олена ГРІНЕНКО

«_____» _____ 2025 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи студента
Хохлова Євгенія Олександровича

1. Тема кваліфікаційної роботи: «Методика вдосконалення архітектури ігрового застосунку на рушії "Unreal Engine 5"»
затверджена наказом президента від ___17.11.2025 року No 2450/ст_____

2. Термін виконання проекту: з 29.09.2025 р. по 21.12.2025 р.

3. Вихідні дані до роботи: Необхідно дослідити особливості архітектури програмного забезпечення та ігрових рушіїв, проаналізувати архітектуру Unreal Engine 5, розробити методику вдосконалення архітектури ігрового застосунку та реалізувати програмний прототип із застосуванням Gameplay Ability System.

4. Зміст пояснювальної записки:
1. Дослідження архітектури програмного забезпечення та ігрових рушіїв.
2. Аналіз архітектури Unreal Engine 5 та можливостей її вдосконалення.
3. Розроблення методики побудови масштабованої архітектури ігрового застосунку.
4. Реалізація програмного прототипу з використанням Gameplay Ability System.
5. Експериментальне дослідження ефективності запропонованої методики.

5. Перелік обов'язкового графічного (ілюстративного) матеріалу:
1. Демонстрація роботи програми.
2. Демонстрація роботи модулів програми.

6. Календарний план-графік

№ пор.	Завдання	Термін виконання	Відмітка про виконання
1.	Розробка та затвердження графіка роботи	29.09-01.10.2025	виконано
2.	Ознайомлення з постановкою задачі, вивчення інформаційних джерел та складання плану роботи.	02.10-05.10.2025	виконано
2.	Підготовка 1 розділу та подання його керівнику	06.10-19.10.2025	виконано
3.	Підготовка 2 розділу та подання його керівнику	20.10-02.11.2025	виконано
4.	Підготовка 3 розділу та подання його керівнику	03.12-16.11.2025	виконано
5.	Загальне редагування пояснювальної записки, графічного матеріалу. Представлення роботи для перевірки на академічну доброчесність. Проходження нормоконтролю.	01.12-07.12.2025	виконано
6.	Отримання відгуку керівника. Підготовка презентації та тексту доповіді.	08.12-14.12.2025	виконано
7.	Попередній захист (представлення електронної версії пояснювальної записки, презентації, позитивного відгуку керівника).	08.12-14.12.2025	виконано
8.	Рецензування кваліфікаційної роботи	15.12-22.12.2025	виконано
9.	Здача секретарю ЕК пояснювальної записки: електронної версії кваліфікаційної роботи; презентації доповіді; відгуку керівника, рецензії; результату проходження перевірки на плагіат; довідки про успішність, декларації про академічну доброчесність.	15.12-22.12.2025	виконано
10.	Захист кваліфікаційної роботи перед екзаменаційною комісією	30.12.2025	

Дата видачі завдання 29.09.2025 р.

Керівник кваліфікаційної роботи:
к.ф.-м.н. Дмитро ГОЛОЛОБОВ

Завдання прийняв до виконання: Євгеній ХОХЛОВ

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи **«Методика вдосконалення архітектури ігрового застосунку на рушії Unreal Engine 5»**: 67 сторінок, 4 рисунка, 1 таблиця, 35 використаних джерел, 1 додаток.

Об'єкт дослідження – процеси проєктування архітектури ігрових застосунків у середовищі розробки багатокористувацьких тривимірних ігор.

Мета кваліфікаційної роботи – розроблення методики вдосконалення архітектури ігрового застосунку на рушії Unreal Engine 5, яка забезпечує масштабованість, модульність, стійкість до змін та можливість ефективного розширення ігрових механік без модифікації ядра системи.

Методи дослідження – аналіз і синтез архітектур програмного забезпечення, моделювання структури ігрових підсистем, застосування принципів інженерії ПЗ, експериментальне тестування у середовищі Unreal Engine 5, порівняльний аналіз архітектурних рішень, прототипування і розробка програмного забезпечення.

Результати роботи можуть бути використані при проєктуванні та розробці багатокористувацьких ігор, зокрема змагальних застосунків, де критично важливими є масштабованість, розширюваність і стабільність архітектури. Запропонована методика може застосовуватися у комерційних та інді-проєктах, що створюються на рушії Unreal Engine 5, а також у навчальних дисциплінах з програмної інженерії та розробки ігор.

Розробка та дослідження проводилися під управлінням ОС **Windows 11**. Створення і реалізація програмного забезпечення виконувалися у середовищі **Unreal Engine 5**, із використанням мови програмування **C++** та системи **Gameplay Ability System** для реалізації модульної ігрової логіки.

UNREAL ENGINE 5, GAMEPLAY ABILITY SYSTEM, АРХІТЕКТУРА ІГРОВОГО ЗАСТОСУНКУ, МАСШТАБОВАНІСТЬ, ПОДІЄВА МОДЕЛЬ, DATA-DRIVEN ДИЗАЙН, БАГАТОКОРИСТУВАЦЬКІ ІГРИ

ABSTRACT

Explanatory note to the qualification work **"Methodology for Improving the Architecture of a Game Application on the Unreal Engine 5 Framework"**: 67 pages, 4 figures, 1 table, 35 references, 1 appendices.

Object of research – the processes of designing the architecture of game applications in the context of developing multiplayer three-dimensional games.

Purpose of the qualification work – to develop a methodology for improving the architecture of a game application built on the Unreal Engine 5 framework, ensuring scalability, modularity, stability, and the ability to extend gameplay mechanics without modifying the core system.

Research methods include analysis and synthesis of software architectures, modeling of game subsystems, application of software engineering principles, experimental testing within Unreal Engine 5, comparative analysis of architectural solutions, prototyping, and software development.

The results of the work can be applied to the design and development of multiplayer and competitive game applications where architectural scalability, extendability, and stability are critical. The proposed methodology can be used in commercial and indie projects developed with Unreal Engine 5, as well as in educational courses related to software engineering and game development.

Development and experimentation were carried out under the **Windows 11** operating system. The implementation of the project was performed in **Unreal Engine 5**, using the **C++ programming language** and the **Gameplay Ability System** to create a modular and extensible gameplay logic architecture.

UNREAL ENGINE 5, GAMEPLAY ABILITY SYSTEM, GAME APPLICATION ARCHITECTURE, SCALABILITY, EVENT-DRIVEN MODEL, DATA-DRIVEN DESIGN, MULTIPLAYER GAMES

ЗМІСТ

ВСТУП

Актуальність теми. Сучасна індустрія розробки ігор характеризується високою динамікою розвитку та зростанням вимог до якості, продуктивності й масштабованості ігрових застосунків. Із переходом індустрії до складних тривимірних багатокористувацьких систем значного значення набувають архітектурні підходи, що забезпечують здатність застосунку до подальшого розширення, адаптації та підтримки. Традиційні монолітні або слабо структуровані рішення не відповідають сучасним вимогам, оскільки ускладнюють розвиток продукту та спричиняють зростання технічного боргу.

У цих умовах актуальність теми дослідження визначається потребою у створенні методики побудови масштабованої архітектури ігрового застосунку, яка дозволяє ефективно керувати складністю проєкту та забезпечувати можливість швидкого додавання нових механік. У роботі особливу увагу приділено рушію Unreal Engine 5, що є одним із найпотужніших інструментів сучасного ігробудування. Актуальність теми зумовлена тим, що, попри широке використання Unreal Engine, питання методичного підходу до проєктування масштабованої архітектури для багатокористувацьких ігор досі залишаються недостатньо опрацьованими, особливо в академічному середовищі.

Мета і завдання виконання кваліфікаційної роботи.

Мета кваліфікаційної роботи полягає в розробленні та обґрунтуванні методики вдосконалення архітектури ігрового застосунку на рушії Unreal Engine 5, що забезпечує його гнучкість, масштабованість та стійкість до подальшого розвитку.

Для досягнення поставленої мети необхідно вирішити такі завдання:

- дослідити теоретичні засади архітектури програмного забезпечення та особливості архітектури ігрових рушіїв;
- проаналізувати архітектуру Unreal Engine 5 і визначити можливості її вдосконалення в контексті ігрових застосунків;

- сформулювати методику побудови масштабованої та модульної архітектури з використанням Gameplay Ability System;
- розробити ігровий застосунок як демонстраційний приклад реалізації запропонованої методики;
- провести експериментальне дослідження масштабованості та стійкості архітектури під час додавання нових ігрових механік;
- оцінити ефективність запропонованої методики та визначити її обмеження.

Об'єктом дослідження є процес проєктування архітектури ігрових застосунків для багатокористувацьких тривимірних ігор.

Предметом дослідження є методика вдосконалення архітектури ігрового застосунку на рушії Unreal Engine 5, побудована на принципах модульності, подієвої моделі взаємодії та використання Gameplay Ability System.

Методи дослідження включають аналіз і синтез архітектурних підходів, моделювання структури застосунку, застосування принципів інженерії програмного забезпечення, експериментальне тестування механік у середовищі Unreal Engine 5, а також порівняльний аналіз отриманих результатів із відомими підходами.

Наукова новизна отриманих результатів полягає в удосконаленні архітектурного підходу до побудови масштабованих ігрових застосунків шляхом інтеграції Gameplay Ability System як базового механізму моделювання поведінки, а також у формулюванні методики проєктування розширюваних механік, яка забезпечує відкритість архітектури до еволюції без зміни її ядра. На відміну від традиційних підходів, запропонована методика використовує подієву модель та data-driven дизайн для зменшення зв'язаності компонентів та підвищення довготривалої підтримуваності системи.

Практичне значення роботи полягає в можливості застосування розробленої методики під час створення комерційних або інді-ігор, що потребують динамічного розширення ігрових механік. Отримані результати можуть бути використані як основа для побудови архітектури багатокористувацьких змагальних застосунків, що

реалізуються на Unreal Engine 5, а також для навчальних курсів із розробки ігор та інженерії програмного забезпечення.

Особистий внесок здобувача вищої освіти полягає у розробці архітектури ігрового застосунку, реалізації демонстраційного проєкту з використанням Gameplay Ability System, проведенні експериментального дослідження масштабованості та формулюванні методики вдосконалення архітектури.

Апробація отриманих результатів дослідження може бути здійснена під час захисту кваліфікаційної роботи

РОЗДІЛ 1
ТЕОРЕТИЧНІ ТА АРХІТЕКТУРНІ ОСНОВИ АРХІТЕКТУРИ ІГРОВИХ ЗАСТОСУНКІВ

1.1 Теоретичні основи архітектури ігрових застосунків

Архітектура програмного забезпечення відіграє ключову роль у процесі розробки будь-яких складних систем, і особливо — ігрових застосунків. На відміну від традиційних програмних продуктів, ігрові системи поєднують у собі візуальні, інтерактивні, фізичні та мережеві елементи, що робить їхню архітектуру значно більш вимогливою до продуктивності, гнучкості та масштабованості. Ігровий застосунок є прикладом комплексної системи реального часу, де затримки, неузгодженість даних або надмірна зв’язаність модулів безпосередньо впливають на якість користувацького досвіду.

Архітектура ігрового застосунку

З позицій програмного забезпечення архітектура ігрового застосунку визначає:

- структуру системи та взаємодію її компонентів;
- принципи обробки ігрової логіки;
- організацію даних, ресурсів і потоків управління;
- механізми оптимізації;
- обмеження та правила, згідно з якими розвивається проєкт.

Архітектура задає «каркас» гри, який повинен залишатися стабільним незалежно від того, як активно еволюціонує внутрішній геймплей. Це дозволяє тривалий час підтримувати проєкт, додавати контент, оптимізувати системи, не порушуючи загальної структури.

Функціональні вимоги до архітектури ігрового застосунку

Основні вимоги, що визначають якість архітектури ігрової системи:

1. Модульність

Поділ застосунку на окремі незалежні компоненти дозволяє:

- ізолювати зміни,
- спростити тестування,
- забезпечити повторне використання функціоналу.

2. Масштабованість

Архітектура повинна дозволяти:

- додавання нових механік,
- збільшення кількості гравців,
- розширення контенту без перепроєктування ядра.

3. Продуктивність та ефективність

Ігрові системи працюють у реальному часі, тому архітектура має:

- забезпечувати обробку логіки та рендеринг у фіксованому часовому циклі,
- мінімізувати затримки,
- оптимізувати доступ до пам’яті й ресурсів.

4. Гнучкість

Структура гри має легко адаптуватися до змін у геймплеї. Наприклад:

- додавання здібностей,
- нові типи об’єктів,
- нові правила взаємодії.

5. Підтримуваність

Здатність розробників розуміти, розширювати та модифікувати систему без ризику виникнення критичних помилок.

6. Мережева узгодженість

У багатокористувацьких проєктах архітектура повинна гарантувати:

- синхронність станів,

- коректність реплікації,
- справедливість ігрових процесів.

Базові структурні рівні ігрових застосунків

Типовий ігровий застосунок складається з кількох рівнів, кожен з яких має свою роль:

Рівень вводу

Обробка керування (клавіатура, миша, контролери, мобільні сенсори).

Логічний рівень

Містить правила гри, міжоб’єктну взаємодію, поведінку NPC.

Рівень фізики

Симуляція руху, зіткнень, сил, гравітації.

Рендеринговий рівень

Формування графічного кадру, робота з текстурами, геометрією, світлом.

Рівень даних і ресурсів

Завантаження моделей, анімацій, матеріалів, аудіо, кешування ресурсів.

Мережевий рівень

Передача станів між клієнтом і сервером, авторитетність, реплікація.

Ці рівні функціонують у рамках єдиного ігрового циклу (Game Loop), який забезпечує послідовність виконання всіх етапів та сталість частоти кадрів.

Архітектурні патерни в ігрових застосунках

У розробці ігор широко застосовуються патерни:

– Game Loop

Основний цикл обробки станів.

– Entity-Component-System (ECS)

Базова парадигма для створення гнучких ігрових структур.

– Observer / Event Bus

Забезпечує обмін подіями між системами.

– State Pattern

Реалізація станів об’єктів (наприклад, «оглушений», «прискорений»).

– Data-driven Architecture

Логіка задається даними, а не кодом — критично важливо для великих ігор.

Застосування таких патернів забезпечує високу адаптивність, повторне використання компонентів та масштабованість.

Архітектура ігрових застосунків як система реального часу

Ігрові системи працюють за жорсткими часовими обмеженнями (16–33 мс на кадр), тому архітектура повинна:

- мінімізувати кількість блокуючих операцій;
- підтримувати багатопоточність;
- оптимізувати використання CPU/GPU;
- ізолювати важкі обчислення у фонові задачі.

Структурно це реалізується через:

- task-graph,
- pipeline-системи,
- асинхронне завантаження ресурсів,
- кешування.

Теоретичні основи архітектури ігрових застосунків формують базу для проєктування, аналізу та подальшого вдосконалення ігрових систем. Архітектура визначає продуктивність, масштабованість та підтримуваність гри, а також рівень її адаптивності до змін геймплею. Саме тому вона є ключовим аспектом програмного забезпечення в контексті розробки сучасних інтерактивних застосунків.

1.2 Особливості архітектури ігрових рушіїв

Ігрові рушії виступають центральним елементом процесу розробки інтерактивних застосунків, забезпечуючи розробників готовою інфраструктурою для побудови складних систем у реальному часі. На відміну від традиційних програмних платформ, ігрові рушії повинні одночасно забезпечувати продуктивність, гнучкість, розширюваність і кросплатформеність, що пред'являє особливі вимоги до їхньої архітектури.

Особливості архітектури ігрових рушіїв формуються на перетині кількох галузей: комп'ютерної графіки, інженерії ПЗ, системного програмування, паралельних обчислень та розробки ігрової логіки. Нижче наведено основні структурні та функціональні характеристики, що визначають сучасні рушії.

Багаторівнева структура та розподіл відповідальностей

Сучасний ігровий рушій має багаторівневу архітектуру, у якій кожен шар виконує окремі функції та взаємодіє з іншими через стандартизовані API. Типова структура включає:

1. Низькорівневий системний рівень

Містить модулі для:

- управління пам'яттю,
- роботи з потоками,
- файлової системи,
- мережевих сокетів,
- абстракції апаратного забезпечення.

Цей рівень забезпечує стабільну роботу рушія на різних операційних системах і пристроях, уникаючи залежності від платформоспецифічних API.

2. Підсистеми середнього рівня

Реалізують основні можливості рушія:

- рендеринг і графічний пайплайн,

- фізичний рушій,
- анімаційні системи,
- аудіопроцесінг,
- AI-фреймворки,
- навігаційні системи,
- менеджери ресурсів.

Кожна підсистема розробляється ізольовано та має власні структури даних і механізми оптимізації.

3. Високорівневий ігровий рівень

Забезпечує:

- систему компонентів або ECS,
- об’єктно-орієнтовану модель ігрової логіки,
- scripting-системи (Blueprint, Lua, C#, Python),
- логіку поведінки персонажів.

Цей шар взаємодіє з дизайнерськими інструментами та безпосередньо підтримує розробку геймплею.

4. Рівень інструментів та редакторів

Включає:

- редактор рівнів,
- анімаційні редактори,
- редактор матеріалів,
- профайлери продуктивності,
- редактори поведінки (наприклад, Behavior Trees).

Він є частиною архітектури рушія, оскільки використовує ті самі API та підсистеми, що й runtime-середовище.

Модульність та плагінна архітектура

Однією з основних вимог до сучасного рушія є можливість розширення його функціональності без зміни ядра. Це досягається за рахунок:

- модульної структури,
- можливості підключення плагінів,
- чітких контрактів між підсистемами,
- стабільних інтерфейсів API.

Плагінна система дозволяє:

- інтегрувати нові системи рендерингу,
- додавати специфічні інструменти,
- замінювати або перевизначати поведінку базових модулів.

У рушії Unreal Engine, наприклад, будь-яка підсистема може бути винесена в окремий модуль, що дає змогу адаптувати рушій під конкретний проєкт.

Компонентна та ECS-модель побудови ігрових об’єктів

Сучасні рушії практично повсюдно застосовують:

- компонентно-орієнтовану архітектуру (Component-Based Architecture)
- або Entity-Component-System (ECS).

Ці підходи дозволяють:

- уникати складних ієрархій успадкування,
- комбінувати поведінку за рахунок композиції,
- підвищувати повторне використання логіки,
- зменшувати рівень зв’язаності,
- підсилювати масштабованість.

ECS-модель має особливо високу продуктивність завдяки оптимізованій роботі з пам’яттю та підтримці масивної паралелізації — що актуально для симуляції великої кількості об’єктів.

Паралельність, асинхронність і оптимізація

Ігрові рушії є системами реального часу, тому їх архітектура активно використовує:

- багатопоточність,
- task-graph моделі,
- SIMD-операції,
- GPU-паралелізацію,
- асинхронне завантаження ресурсів,
- стрімінг даних.

Паралельна обробка використовується для:

- фізичних розрахунків,
- систем частинок,
- навігаційних обчислень,
- анімаційних графів,
- підготовки рендер-пакетів.

Архітектура рушія повинна гарантувати, що всі ці процеси працюють послідовно та синхронізовано в рамках єдиного ігрового циклу.

Графічний пайплайн та рантайм-візуалізація

Графічна підсистема є однією з найскладніших частин рушія. Її архітектура включає:

- систему матеріалів,
- шейдерний пайплайн,
- управління рівнями деталізації (LOD),
- динамічне освітлення,
- глобальне освітлення (GI),
- системи тіней,
- постобробку,

- роботу з GPU та API (DirectX, Vulkan, Metal).

Сучасні рушії, такі як UE5, впроваджують інноваційні архітектурні рішення: Nanite, Lumen, Virtual Shadow Maps, що змінюють традиційні підходи до зберігання геометрії та освітлення.

Підсистема ресурсів (Asset Management)

Рушії працюють з великою кількістю ресурсів:

- моделі,
- анімації,
- текстури,
- звуки,
- скрипти,
- рівні.

Архітектура повинна забезпечувати:

- ефективне кешування,
- стрімінгове завантаження,
- відстеження залежностей,
- автоматичне вивільнення пам’яті (GC),
- компіляцію шейдерів і матеріалів у фоновому режимі.

Це особливо важливо у відкритих світах і високополігональних сценах.

Мережеві можливості та мультиплеєр

Мережева архітектура рушія повинна підтримувати:

- клієнт–серверну модель,
- синхронізацію станів через реплікацію,
- компресію пакетів,
- компенсацію затримок (lag compensation),

- прогнозування клієнта (client prediction),
- авторитетність сервера (server authority).

Це складна частина архітектури, яка визначає можливість використання рушія у багатокористувацьких іграх.

Scripting та візуальні системи

Сучасні рушії включають scripting-платформи:

- Blueprint (Unreal Engine),
- Lua (CryEngine),
- C# (Unity),
- Python API.

Скрипти дозволяють змінювати поведінку гри без перекомпіляції та надають розробникам можливість швидкого прототипування. Архітектура рушія повинна забезпечувати комунікацію між C++ ядром та scripting-шаром.

Особливості архітектури ігрових рушіїв визначають їхню здатність підтримувати складні інтерактивні системи, забезпечувати високу продуктивність та адаптуватися до потреб проекту. Багаторівнева структура, модульність, компонентний підхід, потужні графічні та мережеві підсистеми формують основу для створення сучасних ігрових застосунків. Ефективне використання цих можливостей напряму впливає на якість кінцевого продукту і визначає архітектурну гнучкість та масштабованість гри.

1.3 Аналіз архітектури рушія Unreal Engine 5

Unreal Engine 5 (UE5) є одним із найсучасніших ігрових рушіїв, архітектура якого поєднує високий рівень продуктивності, модульність, гнучкість і розширюваність. Рушій розроблений компанією Epic Games та позиціонується як універсальна платформа для створення проєктів різної складності: від мобільних ігор до високобюджетних AAA-титулів, VR-систем, симуляторів та навчальних застосунків.

У цьому підрозділі здійснюється аналіз архітектури UE5 як прикладу сучасного ігрового рушія, що відповідає вимогам інженерії програмного забезпечення та дозволяє ефективно реалізовувати масштабовані ігрові системи.

Архітектурна філософія Unreal Engine 5

UE5 розробляється на основі кількох ключових принципів:

- об’єктно-орієнтована архітектура з підтримкою компонентної моделі,
- модульність і плагінність усіх підсистем рушія,
- розширюваність на всіх рівнях (від ядра до редактора),
- data-driven підхід до керування поведінкою об’єктів,
- висока продуктивність завдяки новим GPU та CPU технологіям,
- кросплатформеність для широкого спектру пристроїв,
- інтеграція інструментів створення контенту в архітектуру рушія.

Ці принципи формують основу для побудови складних ігрових застосунків, здатних до тривалого розвитку.

Модульна та плагінна структура

UE5 реалізовано як набір модулів, кожен з яких відповідає за окрему підсистему:

- рендеринг,
- фізика,
- звук,
- анімація,
- мережа,
- AI,
- редакторські інструменти,
- управління ресурсами,
- ігровий фреймворк.

Кожен модуль представлений окремою збіркою (C++ module), що дозволяє:

- відключати непотрібні підсистеми,
- створювати кастомні модулі,
- розширювати редактор,
- інтегрувати зовнішні плагіни.

Архітектурний підхід UE5 дозволяє розподіляти функціональні блоки між командами розробників і забезпечує чіткий розподіл відповідальностей.

Об’єктно-орієнтована модель і система акторів

Основною одиницею ігрової логіки в UE5 є Actor — об’єкт, що існує в ігровому світі. Його функціональність формується за допомогою:

- компонентів (Scene Components, Collision Components, Movement Components),
- Blueprints (візуальна логіка),
- C++ класів.

UE5 використовує гібридну систему OCM + компонентну модель, у якій:

- сам Actor є контейнером,
- компоненти описують функціональні аспекти,
- логіка може бути розподілена між ними або винесена у системні модулі.

Це забезпечує повну гнучкість при створенні об’єктної структури гри.

Gameplay Framework — архітектурний рівень для ігрової логіки

UE5 містить розвинений ігровий фреймворк, який визначає архітектуру поведінки ігрових систем:

- GameMode — правила гри,
- GameState — загальний стан сесії,
- PlayerController — логіка вводу/керування,
- PlayerState — стан гравця,

- Pawn/Character — об’єкти керування,
- HUD/UI системи — відображення стану.

Такий фреймворк забезпечує структурованість і передбачуваність поведінки ігрової логіки, що критично для масштабованості.

Рендеринг і графічний пайплайн

Однією з ключових особливостей UE5 є впровадження Nanite та Lumen — нових систем рендерингу:

Nanite

- віртуалізована геометрія,
- автоматична оптимізація рівнів деталізації (LOD),
- можливість використання мільйонів полігонів у реальному часі,
- оптимізований доступ до пам’яті та GPU.

Lumen

- динамічне глобальне освітлення,
- реалістичні відображення,
- можливість змінювати геометрію сцени під час гри,
- зниження потреби у статичних lightmap.

Ці технології впливають на архітектуру ігрових застосунків — зокрема, спрощують пайплайн і дозволяють створювати динамічний контент без значної ручної оптимізації.

Підсистема фізики: Chaos Physics

Chaos — це нативний фізичний рушій UE5, що замінив PhysX. Архітектурні особливості:

- модульна структура,
- підтримка деструкції, тканин, рідин,
- паралельне виконання обчислень,

- підтримка складних симуляцій для великих сцен.

Chaos інтегрується з gameplay-framework і компонентами руху, що дозволяє будувати реалістичні системи взаємодії в грі.

Анімаційна система та Control Rig

UE5 надає засоби процедурної анімації:

- Control Rig,
- Full Body IK,
- Motion Warping,
- анімаційні графи.

Архітектура анімації повністю модульна та дозволяє поєднувати ключові кадри, фізику та процедурні обчислення.

AI Subsystem

Штучний інтелект у UE5 реалізовано через кілька підсистем:

- Behavior Trees,
- Environment Query System (EQS),
- NavMesh і NavigationSystem,
- AIController.

Це дозволяє створювати складну поведінку NPC без дублювання логіки.

Мережева архітектура та реплікація

UE5 використовує клієнт–серверну архітектуру з чітким пріоритетом авторитетності сервера.

Основні механізми:

- реплікація властивостей,
- RPC-виклики,
- мережеві оптимізації (серіалізація, delta-компресія),
- корекція прогнозування клієнта.

Архітектура побудована так, що будь-яка ігрова механіка може бути створена з урахуванням мультиплеєра з мінімальними додатковими змінами.

Scripting та візуальна логіка (Blueprints)

Blueprint — це повноцінна scripting-платформа, інтегрована у архітектуру рушія.

Особливості:

- реалізація логіки без компіляції C++ коду,
- подієва модель,
- інкапсуляція поведінки у вузлах,
- можливість створювати власні функції та компоненти,
- підтримка нативної інтеграції з C++.

Особливість UE5 — програмування на Blueprint є частиною архітектури, а не надбудовою.

Gameplay Ability System як окрема архітектурна підсистема

Особливо важливою для магістерської роботи є GAS, яка:

- інкапсулює логіку здібностей,
- реалізує data-driven підхід,
- забезпечує масштабування механік,
- підтримує multiplayer і реплікацію,
- дозволяє відокремити логіку від об’єктів.

GAS повністю інтегрується в архітектуру UE5 та виступає прикладом модульної та розширюваної архітектурної системи.

Аналіз архітектури Unreal Engine 5 демонструє, що це один із найрозвиненіших рушіїв сучасності, який поєднує модульну будову, компонентний підхід, data-driven архітектуру та інноваційні графічні й фізичні технології. UE5 забезпечує високий рівень гнучкості та масштабованості, що дозволяє ефективно розробляти ігрові застосунки різного рівня складності. Структура рушія, його модульність та

інструменти створюють сприятливе середовище для реалізації запропонованої у даній роботі методики вдосконалення архітектури ігрових систем.

Висновок

У першому розділі було проведено комплексний аналіз теоретичних засад побудови архітектури ігрових застосунків, особливостей архітектури сучасних ігрових рушіїв та структурних принципів, закладених в основу Unreal Engine 5. Здійснене дослідження дозволило сформувати системне уявлення про архітектуру ігор як галузь інженерії програмного забезпечення та визначити ключові вимоги, які повинні виконуватися для створення масштабованих, гнучких і продуктивних ігрових систем.

У підрозділі 1.1 розглянуто фундаментальні принципи архітектури ігрових застосунків, такі як модульність, масштабованість, гнучкість, продуктивність, підтримуваність та мережева узгодженість. Було проаналізовано типові структурні рівні ігрової системи — від вводу користувача та логіки взаємодії до фізики, рендерингу та мережевих механізмів. Також описано ключові архітектурні патерни, що застосовуються у розробці ігор, включно з Game Loop, ECS, Observer та data-driven підходом. Це дозволило визначити загальні закономірності та інженерні підходи до побудови архітектури ігрових проєктів.

У підрозділі 1.2 було досліджено особливості архітектури ігрових рушіїв як універсальних платформ розробки. Особливу увагу приділено багаторівневій структурі рушія, модульності, компонентному підходу, системам управління ресурсами, підтримці паралельних обчислень, графічному та фізичному пайплайнам, мережевим можливостям і scripting-середовищам. Аналіз показав, що архітектура рушія має забезпечувати високу продуктивність та розширюваність, одночасно залишаючись узгодженою з дизайнерськими інструментами, які теж є частиною загальної архітектури.

У підрозділі 1.3 проведено детальний аналіз архітектури Unreal Engine 5 як одного з найсучасніших і найпотужніших рушіїв. Розглянуто його модульну

структуру, компонентно-орієнтовану модель, Gameplay Framework, інноваційні графічні технології Nanite і Lumen, фізичну систему Chaos, засоби створення анімації, підсистеми штучного інтелекту та мережеві механізми реплікації. Окремо підкреслено значення Gameplay Ability System як архітектурного ядра для побудови масштабованих ігрових механік. Аналіз UE5 показав, що цей рушій забезпечує всі необхідні технічні передумови для реалізації застосунків високого рівня складності.

Підсумовуючи, перший розділ сформував теоретичний і методологічний фундамент для подальшого дослідження. На його основі у наступних розділах буде розроблено та обґрунтовано методику вдосконалення архітектури ігрового застосунку, орієнтовану на підвищення модульності, масштабованості та підтримуваності за допомогою архітектурних можливостей Unreal Engine 5 та Gameplay Ability System.

РОЗДІЛ 2

МЕТОДИКА ВДОСКОНАЛЕННЯ АРХІТЕКТУРИ ІГРОВОГО ЗАСТОСУНКУ НА РУШІЇ UNREAL ENGINE 5

2.1 Вимоги до масштабованої архітектури ігрового застосунку

Розробка сучасних ігрових застосунків характеризується високим рівнем складності, що зумовлено зростаючими вимогами до функціональності, графіки, інтерактивності та підтримки багатокористувацьких режимів. У таких умовах ключового значення набуває масштабованість архітектури ігрового застосунку, яка визначає здатність системи до розширення, адаптації та довгострокової підтримки без суттєвого ускладнення структури чи погіршення стабільності.

Під масштабованістю архітектури ігрового застосунку розуміється можливість збільшення кількості ігрових механік, контенту та систем взаємодії без необхідності повного перепроєктування вже реалізованих підсистем. Масштабована архітектура повинна забезпечувати гнучке додавання нових можливостей, зменшувати вплив змін на існуючий код та підтримувати цілісність ігрової логіки.

Показники масштабованості

Одним із ключових показників масштабованості є архітектурна гнучкість, тобто здатність системи реагувати на зміну вимог із мінімальними структурними модифікаціями. У контексті ігрових проєктів це проявляється у можливості додавати нові типи персонажів, здібностей, предметів або зон взаємодії без дублювання коду або жорсткої прив’язки до конкретних реалізацій.

Важливим показником також є стійкість до зростання складності, коли збільшення кількості ігрових механік не призводить до експоненційного ускладнення логіки. Досягнення цього можливе за умови правильного розподілу відповідальностей між підсистемами та чіткого дотримання архітектурних принципів.

Окремо слід виділити масштабованість команди розробки, адже архітектура повинна бути зрозумілою не лише для одного розробника, а й для колективної роботи. Чітка структура підсистем, стандартизовані інтерфейси та передбачувані точки розширення значно зменшують ризик помилок при паралельній розробці.

Розширюваність та підтримуваність

Розширюваність тісно пов’язана з підтримуваністю архітектури, оскільки будь-яке масштабування призводить до необхідності супроводження вже існуючого коду. Архітектура ігрового застосунку повинна забезпечувати можливість розширення функціоналу без внесення змін у вже стабільні частини системи. Це особливо важливо для ігрових проєктів, які розвиваються протягом тривалого часу або передбачають післярелізну підтримку.

Підтримуваність архітектури визначається читабельністю коду, логічною структурованістю підсистем та наявністю чітких меж відповідальності. Відсутність такої структури призводить до появи так званого спагеті-коду, що унеможливлює ефективне масштабування та значно ускладнює внесення змін.

Для забезпечення високої підтримуваності необхідно відокремлювати ігрову логіку від засобів її реалізації, мінімізуючи прямі залежності між окремими модулями. Такий підхід дає змогу змінювати або доопрацьовувати окремі компоненти без порушення роботи всієї системи.

Мінімізація міжмодульних залежностей

Однією з основних вимог до масштабованої архітектури є зменшення жорстких міжмодульних залежностей. У випадку, коли підсистеми тісно пов’язані між собою, будь-яка зміна в одному модулі може вимагати корекції в інших, що істотно підвищує ризик помилок і ускладнює розвиток застосунку.

Для ігрових застосунків це особливо критично, оскільки механіки часто взаємодіють між собою: персонажі отримують ефекти, змінюються характеристики, активуються здібності та відбувається реакція на події в ігровому світі. Масштабована архітектура повинна передбачати слабке зв’язування між такими

механіками, використовуючи абстрактні інтерфейси, подієві механізми та узагальнені структури даних.

Мінімізація залежностей також сприяє повторному використанню компонентів ігрової логіки, що дозволяє створювати універсальні рішення для різних сценаріїв ігрової взаємодії без дублювання реалізацій.

У підрозділі визначено основні вимоги до масштабованої архітектури ігрового застосунку, серед яких ключовими є гнучкість, розширюваність, підтримуваність та мінімізація міжмодульних залежностей. Дотримання цих вимог дозволяє створити архітектуру, здатну ефективно адаптуватися до зростання складності ігрового проєкту та спрощує подальший розвиток і супровід системи. Отримані положення слугують теоретичною основою для формування модульної архітектури та вибору інструментів її реалізації, що буде розглянуто в наступних підрозділах.

2.2 Принципи побудови модульної архітектури

Модульна архітектура — це підхід до побудови системи, за якого її функціональність розбивається на автономні (або слабозалежні) модулі з чітко визначеними інтерфейсами. В контексті ігрових застосунків, і особливо при роботі на рушії Unreal Engine 5, модульність дозволяє розподіляти роботу між спеціалізованими командами, спрощує тестування, пришвидшує розширення проєкту і знижує ризики регресій при модернізації систем. Нижче наведені ключові принципи та практики побудови модульної архітектури, адаптовані під особливості ігрової розробки.

2.2.1. Розподіл відповідальностей між підсистемами

Суть принципу. Кожен модуль або підсистема повинні мати чітко визначену сферу відповідальності — набір функцій, які він виконує, і обмеження щодо того, що не належить до його компетенції. Це дозволяє зменшити перехресні залежності і полегшує підтримку.

Практична реалізація:

- Визначити набір основних підсистем (наприклад: Input, Movement, Abilities, Effects, Inventory, UI, Networking, Persistence).
- Для кожної підсистеми сформулювати контракт: які події вона приймає, які події вона емінтує, які дані зберігає.
- Забезпечити "чисті" точки входу — фасади або API, через які інші модулі взаємодіють із підсистемою, без доступу до внутрішнього стану.
- Встановити межі авторитетності в мультиплеєрі: які підсистеми (або їх методи) «виконуються» лише на сервері, а які можуть бути локально предиктовані.

Приклад: Підсистема Abilities керує тільки декларацією і активацією здібностей, але не відповідає за візуальні ефекти — останні логічно делегуються системі VFX через подію або інтерфейс.

2.2.2. Інкапсуляція ігрової логіки

Суть принципу. Логіка повинна бути прихована за абстракціями; інші модулі працюють із контрактами, а не з внутрішніми деталями реалізації.

Практичні підходи:

- Використовувати компоненти (Component pattern) для інкапсуляції поведінки: AbilityComponent, InventoryComponent, HealthComponent.
- Мінімізувати публічні поля; використовувати геттери/сеттери або методи, що виконують перевірки інваріантів.
- Впроваджувати строгі API для взаємодії між модулями (фасади, інтерфейси).
- Розділяти стан (дані) і поведінку (логіку), де це має сенс (data-driven).

Приклад: Компонент здоров'я (HealthComponent) надає методи ApplyDamage, ApplyHeal, GetCurrentHealth; інші модулі викликають лише ці методи, і не читають/змінюють внутрішні поля напряму.

2.2.3. Подієва модель взаємодії (Event-driven architecture)

Суть принципу. Замість прямих викликів між модулями система будує комунікацію через події — модулі емінтують події, інші модулі підписуються на них.

Переваги:

- Зниження зв’язаності: емінтер не знає про слухачів.
- Легкість розширення: новий функціонал підписується на існуючі події.
- Краща масштабованість і можливість логування/моніторингу подій.

Реалізація в UE5:

- Використовувати Event Dispatchers/Delegates у Blueprints та C++.
- Впроваджувати централізовану EventBus/MessageBroker для крос-модульних повідомлень.
- Для мережі — визначити, які події слід реплікати/синхронізувати.

Приклад: Подія OnAbilityActivated(AbilityID, Caster, Target) емінтується AbilityComponent; UI, VFX та статистика гри підписуються і реагують відповідно.

Увага: надмірне використання глобальних подій може призвести до складності в відстежуванні потоку виконання; рекомендовано документувати кожну подію і її контракт.

2.2.4. Data-driven підхід

Суть принципу. Конфігураційні дані (параметри здібностей, налаштування зон, властивості предметів тощо) зберігаються у зовнішніх ресурсах (Data Assets, JSON, CSV), а код читає й інтерпретує ці дані. Логіка повинна бути максимально універсальною і керуватись даними.

Переваги:

- Зменшення необхідності змін у коді при балансуванні/додаванні контенту.
- Можливість підключення дизайнерів/тестувальників без програмування.
- Полегшення експериментування та A/B тестів.

Реалізація:

- Використовувати UE5 Data Assets / Data Tables для зберігання конфігурацій.
- Визначити схеми валідації даних.

- Підтримувати редакторські інструменти для зручного редагування цих даних.

Приклад: Замість хардкодингу значень дальності дебаф-зони, вказати їх у ZoneConfig Data Asset.

2.2.5. Інтерфейси, контракти і dependency inversion

Суть принципу. Інтерфейси — формальні контракти, що описують набір методів для взаємодії. Dependency inversion (інверсія залежностей) означає, що модулі звертаються до абстракцій, а не до конкретних реалізацій.

Рекомендації:

- Створювати інтерфейси на рівні гри: IAbilityUser, IEffectTarget, IZone.
- Використовувати фабрики/сервіси для створення об’єктів (залежності інжектяться через конструктор/ініціалізацію).
- При можливості застосовувати Service Locator або прості DI-механізми на рівні гри (UE5 не має повноцінного DI-контейнера, але патерни можна імплементувати вручну).

Приклад: AbilityManager отримує посилання на IEffectApplier, а не на конкретний клас EffectApplier_Simple.

2.2.6. Версифікація даних і сумісність

Суть принципу. При еволюції гри формати даних (savegames, data assets) можуть змінюватися; архітектура повинна передбачати механізм обробки старих версій.

Практики:

- Форматувати data assets з версіями та механізмами міграції.
- Писати backward compatibility handlers для savegames.
- Впроваджувати функції фіч-флагів, щоб нові можливості можна було вмикати/вимикати без деплою коду.

Приклад: ZoneConfig має поле Version; при завантаженні виконується міграція конфігурацій до останнього формату.

2.2.7. Плагінна архітектура та розширюваність

Суть принципу. Розробляти систему так, щоб нові модулі могли підключатися як плагіни — без зміни ядра.

Практики:

- Використовувати механізм плагінів UE (Plugins) для утримання окремого функціоналу.
- Визначати чіткі entry points (API) для плагінів.
- Документувати lifecycle плагіну (initialize, tick, shutdown).

Переваги:

- Розробка нових механік як окремих модулів.
- Можливість тестувати плагіни самостійно.

2.2.8. Тестування та автоматизація

Суть принципу. Модульна архітектура повинна бути легко тестованою. Тестування — ключовий елемент підтримуваності.

Рекомендації:

- Розробляти юніт-тести для бізнес-логіки (наприклад, для Ability/Effect систем).
- Писати інтеграційні тести для взаємодії модулів (через симуляцію подій).
- Автоматизувати CI/CD, щоб кожен коміт виконував базову батарею тестів.
- Використовувати інструменти для профілювання та регресійного тестування продуктивності.

Приклад: Тест на застосування effect: створити mock actor, застосувати Gameplay Effect та перевірити зміни атрибутів.

2.2.9. Моніторинг, логування та інструменти налагодження

Суть принципу. При зростанні розміру системи важливо мати інструменти для спостереження за її станом.

Практики:

- Впровадити централізоване логування подій (з рівнями: Info/Warning/Error).

- Додавати телеметрію для ключових подій (активація здібності, накладення ефекту, помилки реплікації).
- Використовувати in-editor debugging utilities (visualizers для зон, перегляд стеку ефектів).

Переваги: Швидше знаходження проблем у продакшн-сценаріях, полегшення балансування.

2.2.10. Продуктивність та оптимізації на архітектурному рівні

Суть принципу. Архітектурні рішення повинні враховувати вартість виконання: CPU, пам’ять, мережа. Модульна архітектура не повинна призводити до зайвих витрат (наприклад, надмірних крос-модульних викликів).

Рекомендації:

- Мінімізувати алокації у критичних шляхах (Tick, Physics callbacks).
- Використовувати пулінг для часто створюваних об’єктів (projectiles, temporary effects).
- Застосовувати батчинг подій, коли можливо, для зменшення кількості реплікованих повідомлень.
- Встановлювати QoS-політики для мережевих повідомлень: важливі події — reliable, другорядні — unreliable.

Приклад: Для зон, що тикають кожен кадр, використовувати централізований tick manager з групуванням оновлень замість індивідуального Tick в кожній зоні.

2.2.11. Життєвий цикл об’єктів і управління станами

Суть принципу. Керування створенням, ініціалізацією, активацією, деактивацією та знищенням об’єктів повинно бути стандартизоване.

Практики:

- Визначити мандатні методи: Initialize, Activate, Deactivate, Shutdown.
- Для мережі: розділити поняття Spawn (створення) і Possess (надання контролю).

- Забезпечити контроль над станами (state machines) для об’єктів зі складними lifecycle (наприклад, charged abilities, cooldowns).

2.2.12. Документація та стандарт коду

Суть принципу. Добре структурований проєкт потребує стандартів і документації.

Практики:

- Впровадити coding style guide та architectural guidelines.
- Документувати API для підсистем.
- Підтримувати приклади використання (snippets) та шаблони для типових задач (створення нової Ability, реалізація нової Zone).

У підрозділі висвітлено ключові принципи побудови модульної архітектури, які забезпечують масштабованість, розширюваність і підтримуваність ігрового застосунку. Основні тези:

- Чіткий розподіл відповідальностей зменшує зв’язаність і спрощує розвиток системи.
- Інкапсуляція та використання інтерфейсів роблять компоненти замінними і тестованими.
- Подієва архітектура та data-driven підхід значно полегшують розширення та балансування механік.
- Версифікація, плагінна архітектура, тестування та документування — невід’ємні елементи життєздатної архітектури.
- Архітектурні рішення повинні враховувати продуктивність і мережеві особливості (особливо у мультиплеєрі).

2.3 Інтеграція Gameplay Ability System у структуру застосунку

Gameplay Ability System (GAS) є однією з ключових підсистем Unreal Engine 5, яка забезпечує модульну, масштабовану та data-driven архітектуру для реалізації складних механік взаємодії, здібностей, ефектів та станів ігрових об’єктів. У контексті розробки ігрових застосунків високої складності GAS відіграє роль архітектурного ядра, що дозволяє відокремити логіку ігрових механік від конкретних об’єктів і реалізувати їх як незалежні функціональні модулі.

Архітектурний підхід GAS ґрунтується на чіткому розподілі даних, логіки та механізмів взаємодії. Система відокремлює визначення здібностей і ефектів від їхнього застосування, забезпечуючи гнучкість і можливість повторного використання.

Архітектурні компоненти GAS

Основними елементами GAS, які беруть участь у побудові ігрової логіки, є:

1. AbilitySystemComponent (ASC)

Компонент, що додається до будь-якого ігрового актора, який може володіти здібностями.

ASC відповідає за:

- активацію здібностей,
- керування ефектами,
- обробку Gameplay Tags,
- реплікацію станів у багатокористувацькому середовищі.

2. Gameplay Ability

Окремий модуль поведінки, що описує одну конкретну здібність. Здібність:

- визначає умови активації,
- описує послідовність дій,
- задає взаємодію з іншими системами (рух, фізика, пошкодження),
- повністю відокремлена від актора.

3. Gameplay Effect

Структура даних, яка задає параметри впливу:

- бафи та дебафи,
- модифікацію параметрів,
- тривалі ефекти,
- миттєві зміни стану.

У реалізації гри GameplayEffect використовується, наприклад, для:

- сповільнення гравця,
- переміни швидкості,
- тимчасових перешкод,
- штрафів за вхід у певні області карти.

4. Gameplay Tags

Система тегів, яка дозволяє:

- формувати категорії здібностей,
- визначати обмеження на активацію,
- впорядковувати ефекти,
- контролювати взаємодію модулів.

Це критично для масштабованості та уникнення жорсткої зв’язаності між здібностями.

GAS як архітектурне ядро проєкту

Головною причиною інтеграції GAS у даний ігровий застосунок є її здатність забезпечити:

Відокремлення ігрової логіки від реалізації

У моделі GAS конкретний актор не зберігає логіку здібності. Він лише “запускає” модуль здібності через AbilitySystemComponent. Це дозволило:

- реалізувати систему випадкових здібностей, що генеруються під час збирання об’єктів;
- уникнути дублювання логіки між гравцями;
- easily plug-and-play — додавання нової здібності без зміни коду персонажа.

Повну підтримку мультиплеєра

Оскільки змагання — це багатокористувацька система, GAS надає:

- авторитетність сервера,
- реплікацію станів,
- коректну синхронізацію ефектів між клієнтами,
- RPC-механізми, які гарантують справедливість гри.

Data-driven дизайн

У GAS усе описується у вигляді даних (DataAssets, GameplayEffect, AbilityTags), що дозволяє:

- змінювати логіку гри без перекомпіляції коду,
- легко налаштовувати параметри балансу,
- швидко додавати новий контент.

Інтеграція GAS у структуру гри

1. Здібності беруться із зібраних об’єктів, а не належать персонажу заздалегідь

Ігровий світ містить спеціальні предмети (Ability Pickups). Коли гравець входить у тригер об’єкта:

1. запускається логіка взаємодії;
2. випадково вибирається здібність із доступного пулу;
3. через AbilitySystemComponent ця здібність додається у “слот” гравця.

Таке проєктне рішення:

- мінімізує зв’язаність між Pickup і Ability,

- дозволяє легко додавати нові здібності,
- не потребує змін у логіці персонажа.

2. Кожна здібність абсолютно самостійна

Ability — повністю ізольований модуль. Вона не залежить:

- від конкретної кнопки,
- від Pickup,
- від конкретного персонажа,
- від типу рівня.

Це чудовий приклад модульної архітектури у геймдеві.

3. Використання GameplayEffects для впливів

- бафів,
- дебафів,
- модифікації швидкості,
- накладання станів.

Наприклад, зона на карті, що зменшує швидкість, — це:

- базовий клас “AreaVolume”,
- який при вході активує GameplayEffect на гравця.

Це показує, як GAS дозволяє реалізовувати архітектурно чисті механіки без дублювання коду.

4. Обмеження на дві активні здібності

Завдяки AbilitySystemComponent було легко реалізувати обмеження кількості активних здібностей. Архітектура GAS дозволяє:

- додавати здібності у слоти,
- відслідковувати їх активацію,

- очищати пасивні ефекти.

Це рішення повністю відповідає принципам інженерії ПЗ — чіткий контракт між системою і даними.

Переваги GAS у контексті запропонованої методики

У магістерській роботі ключовою темою є методика вдосконалення архітектури. GAS дозволяє реалізувати такі вдосконалення:

- мінімальна зв’язаність модулів,
- можливість додавання нового функціоналу без рефакторингу,
- універсальні архітектурні компоненти,
- централізований контроль станів,
- зручність тестування,
- адаптивність до різних типів ігор.

Інтеграція Gameplay Ability System у структуру ігрового застосунку забезпечує потужний фундамент для побудови гнучкої, модульної та масштабованої архітектури. GAS повністю відповідає принципам інженерії програмного забезпечення та дозволяє відокремити логіку здібностей від конкретних реалізацій, що робить її ключовим елементом методики вдосконалення архітектури, запропонованої у даній роботі.

2.4 Методика проєктування розширюваних ігрових механік

Проєктування архітектури ігрових механік є ключовим елементом у створенні сучасних інтерактивних застосунків. На відміну від статичних систем, ігрові механіки постійно змінюються та доповнюються в процесі розробки. Тому методика побудови таких механік повинна забезпечувати високу гнучкість, можливість подальшого розширення та мінімальну залежність між архітектурними компонентами. У контексті Unreal Engine 5 та Gameplay Ability System (GAS) особливо важливою стає здатність механік еволюціонувати без зміни базових класів, що відповідає принципам модульності та відкритості/закритості (OCP).

Архітектурні принципи проєктування розширюваних механік

Методика передбачає використання низки принципів, що дозволяють побудувати зручну, гнучку та масштабовану архітектуру:

1. Абстракція базових елементів ігрових механік

Будь-яка механіка ігрової системи повинна мати базову абстракцію — клас, інтерфейс або компонент, який визначає її загальну структуру та модель взаємодії. Це дозволяє:

- уникати дублювання коду,
- створювати нові механіки шляхом наслідування або композиції,
- забезпечувати єдиний контракт між різними частинами системи.

У контексті GAS така абстракція природно реалізується через:

- Gameplay Ability як базовий модуль поведінки,
- Gameplay Effect як універсальний спосіб впливу,
- Gameplay Tags як засіб визначення категорій та умов.

2. Використання data-driven підходу

У сучасній інженерії ПЗ механіки повинні конфігуруватися не кодом, а даними. Це дозволяє:

- змінювати поведінку без перекомпіляції системи,
- налаштовувати баланс гри,
- відокремлювати логіку від представлення.

Unreal Engine надає для цього:

- DataAsset-и здібностей,
- структури GameplayEffect,
- таблиці параметрів,
- теги як універсальні маркери станів.

Data-driven підхід забезпечує можливість швидкого масштабування проєкту без зміни архітектури.

Проєктування розширюваної системи областей впливу

Одним із типових елементів ігрової логіки є «області впливу» — сегменти ігрового простору, що модифікують стан персонажа під час входу, перебування або виходу з них.

Методика передбачає створення:

- базового класу області,
- який містить єдині точки входу: OnEnter, OnStay, OnExit,
- а конкретні області визначають лише дані — який саме GameplayEffect потрібно застосувати.

Такий підхід дозволяє:

- легко додавати нові області,
- змінювати їхню дію через GameplayEffect,
- уникати дублювання логіки,
- повністю відокремити реалізацію від поведінкових даних.

Це повністю відповідає принципу відкритості/закритості: нові області додаються без зміни кодової бази попередніх.

Розширюваність здібностей через GAS

Методика створення здібностей базується на принципах:

- ізоляції логіки всередині Ability-класів,
- конфігурації параметрів через GameplayEffect,
- контрольованої взаємодії через Gameplay Tags.

Переваги такого підходу:

- додавання нової здібності не змінює поведінку інших,
- архітектура не потребує централізованого керування здібностями,

- AbilitySystemComponent автоматично обробляє стан, cooldown, активацію та реплікацію.

У проєкті буде застосовано підхід, коли:

- гравець може мати обмежену кількість здібностей,
- здібності підбираються динамічно,
- їхній ефект визначається незалежним GameplayEffect або набором ефектів.

Це демонструє високу масштабованість системи при мінімальних вимогах до рефакторингу.

Подієва модель взаємодії

У розширюваних механіках важливо, щоб:

- модулі взаємодіяли через події,
- а не прямі виклики методів.

GAS частково забезпечує це через Gameplay Tags та Event Dispatchers. Подієва модель дозволяє:

- будувати складні системи без циклічних залежностей,
- динамічно підключати нові модулі,
- адаптувати поведінку до різних сценаріїв (наприклад, мультиплеєр).

Уникнення міжмодульних залежностей

Одна з ключових ідей методики — мінімальний прямий зв’язок між компонентами. Замість того щоб:

- здібність знала про Pickup,
- або область знала про конкретний клас персонажа,

використовуються абстрактні контракти:

- AbilitySystemComponent,
- GameplayEffect,
- події входу/виходу з тригеру,

- Gameplay Tags.

Це дозволяє розширювати систему у будь-якому напрямку без порушення архітектури.

Методика проєктування розширюваних ігрових механік на основі Unreal Engine 5 та Gameplay Ability System забезпечує гнучку та стійку до змін архітектуру, орієнтовану на довготривалий розвиток проєкту.

Використання абстракцій, data-driven підходу, системи тегів, базових компонентів і подієвої моделі взаємодії дозволяє створювати ігрові механіки, які можна розширювати без модифікації ядра системи.

Запропонована методика є універсальною та легко адаптується до різних типів ігор, забезпечуючи ефективне масштабування без зростання технічного боргу.

Висновок

У другому розділі було розроблено та обґрунтовано методику вдосконалення архітектури ігрового застосунку на рушії Unreal Engine 5. Проведений аналіз дозволив визначити ключові вимоги до масштабованої архітектури, серед яких — модульність, розширюваність, низька зв’язаність компонентів та підтримуваність у контексті динамічного розвитку ігрового застосунку.

Було встановлено, що побудова архітектури на основі чіткого розподілу відповідальностей між підсистемами, інкапсуляції логіки та використання подієвої моделі взаємодії забезпечує створення гнучкого програмного середовища. Застосування Gameplay Ability System як ядра обробки ігрової логіки й ефектів дозволило стандартизувати механізми взаємодії, підвищити рівень абстракції та забезпечити відокремлення даних від виконання поведінки.

Окрему увагу приділено методиці проєктування розширюваних ігрових механік, яка передбачає використання базових абстракцій, data-driven підходу, системи тегів та модульної організації компонентів. Завдяки цьому стає можливим додавання нових здібностей, зон впливу та поведінкових механізмів без змін у

структурі ядра, що відповідає принципам відкритості до розширення та закритості до модифікацій.

Узагальнюючи, розділ 2 сформував теоретичну та методичну основу, яка забезпечує побудову масштабованої та стійкої до змін архітектури. Ця методика стала фундаментом для реалізації ігрового застосунку та подальшої експериментальної перевірки, поданої у третьому розділі.

РОЗДІЛ 3

РЕАЛІЗАЦІЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ЗАПРОПОНОВАНОЇ МЕТОДИКИ

3.1 Опис архітектури ігрового застосунку

Архітектура розробленого ігрового застосунку побудована з урахуванням вимог до масштабованості, модульності та можливості подальшого розвитку, визначених у попередніх розділах. Структура гри сформована на основі клієнт–серверної моделі Unreal Engine 5 та використовує Gameplay Ability System як основу для реалізації гнучких механік взаємодії, здібностей та ефектів.

Застосунок є багатокористувацькою грою жанру «перегони з перешкодами», у якій основна мета гравця — досягти фінішу або набрати найбільшу кількість балів на основі проходження чекпоінтів та використання здібностей. Реалізація ігрової логіки спирається на структуровану архітектуру, що розділяє ігрові підсистеми між собою та мінімізує зв’язаність компонентів.

Склад ігрових підсистем

Архітектура застосунку складається з таких основних підсистем:

1. Підсистема ігрового світу (World Subsystem)

Відповідає за створення і керування рівнем, розташуванням перешкод, чекпоінтів та областей впливу.
Структура рівня побудована таким чином, щоб елементи ігрового світу були ізольованими один від одного і могли розширюватися незалежно.

2. Підсистема персонажа (Character Subsystem)

Містить:

- актор гравця (Character),

- компонент руху,
- AbilitySystemComponent,
- набір компонентів колізій та візуалізації.

Гравець взаємодіє з середовищем, Pickup-об’єктами та зонами впливу через чітко визначені тригери та інтерфейси.

3. Підсистема здібностей (Ability Subsystem)

Побудована на Gameplay Ability System і відповідає за:

- отримання здібностей через спеціальні об’єкти на мапі,
- виконання здібностей,
- застосування ефектів,
- накладання станів (бафи, дебафи),
- синхронізацію здібностей у мережевому середовищі.

Ця підсистема є ключовим прикладом модульності, оскільки здібності існують незалежно від персонажа чи Pickup-об’єктів.

4. Підсистема збору предметів (Pickup Subsystem)

Спеціальні об’єкти на карті дозволяють гравцеві отримати випадкову здібність. Архітектура побудована так, що:

- Pickup не знає про конкретні здібності,
- він лише генерує подію “GiveAbility”,
- AbilitySystemComponent вибирає здібність із пулу даних.

Таке рішення мінімізує залежності між системами.

5. Підсистема зон впливу (Area Effect Subsystem)

Використовує базовий клас «AreaVolume» і реалізує:

- ефекти на вході в область,
- ефекти під час перебування в ній,

- скасування ефектів після виходу.

Логіка впливу реалізована через GameplayEffect, що забезпечує повну узгодженість з підсистемою здібностей.

6. Підсистема змагального ігрового процесу (Race Logic Subsystem)

Включає:

- систему чекпоінтів,
- реєстрацію прогресу гравця,
- підрахунок балів,
- логіку визначення переможця.

Ця підсистема ізольована від здібностей і не залежить від підсистеми впливів, що робить її гнучкою та розширюваною.

Клієнт–серверна модель

Архітектура гри використовує стандартну модель Unreal Engine:

Сервер:

- зберігає авторитетний стан гри,
- обробляє пересування гравців,
- контролює активацію здібностей,
- перевіряє умови досягнення чекпоінтів,
- виконує логіку змагального процесу.

Сервер запобігає шахрайству та забезпечує узгодженість між усіма клієнтами.

Клієнти:

- відправляють інформацію про ввід гравця,
- виконують локальні передбачення руху (client-side prediction),
- отримують оновлення стану від сервера.

Клієнти не приймають критичних рішень щодо ігрової логіки, що відповідає принципам безпечного мультиплеєра.

Ролі сервера і клієнтів у контексті Gameplay Ability System

GAS має власну мережеву модель:

- Сервер контролює активацію здібностей та застосування ефектів.
- Клієнт відповідає за візуалізацію та попередню обробку вводу.
- Реплікація GameplayEffect, AbilityTags та станів відбувається автоматично.

У проєкті це дозволило:

- синхронізувати ефекти уповільнення, прискорення та перешкод;
- забезпечити коректну роботу здібностей, що впливають на інших гравців;
- гарантувати справедливість змагального процесу.

Архітектура ігрового застосунку побудована на основі чітко структурованих підсистем та відповідає принципам, описаним у другому розділі. Використання модульної структури, клієнт–серверної моделі та Gameplay Ability System забезпечує масштабованість, гнучкість і розширюваність гри. Запропоноване архітектурне рішення створює надійну основу для подальшого вдосконалення механік, додавання нових можливостей і проведення експериментальних досліджень, описаних у наступних підрозділах.

3.2 Реалізація ігрових механік

Реалізовані ігрові механіки у проєкті ґрунтуються на принципах модульності, повторного використання компонентів та мінімальної зв’язаності підсистем. Кожна з механік — система перегонів, система підрахунку балів, система здібностей та збирання об’єктів — побудована таким чином, щоб її можна було незалежно розширювати, модифікувати та інтегрувати з іншими елементами гри. Архітектурна основа, закладена у попередніх розділах, забезпечує узгодженість роботи систем на сервері й клієнті, а також підтримку масштабованості.

Реалізація системи перегонів і чекпоінтів

Система перегонів є основою геймплею. Ігровий рівень складається з послідовності чекпоінтів, розміщених у просторі таким чином, щоб визначати шлях до фінішу. Гравець повинен пройти всі чекпоінти у правильному порядку; відхилення або пропуск пропорційно впливає на результат.

Архітектурні особливості реалізації:

1. Чекпоінт як незалежний актор
   Кожен чекпоінт реалізовано як автономний актор, що містить тригерну зону, обробник подій входу та індекс порядку.
   Чекпоінт не містить логіки підрахунку очок — лише генерує подію.
2. Серверна обробка прогресу
   Перевірка правильності проходження виконується виключно на сервері, що унеможливлює маніпуляції та забезпечує справедливість.
3. Зберігання прогресу в PlayerState
   Кожен гравець має власний об’єкт PlayerState, у якому зберігається:
   - номер останнього пройденого чекпоінта,
   - кількість отриманих балів,
   - загальний прогрес.
4. Подієва модель
   Чекпоінт надсилає сигнал → GameMode обробляє → PlayerState оновлюється → HUD відображає.

   Це дозволяє:

- легко додавати нові типи чекпоінтів (ускладнені, таймерні, командні),
- змінювати логіку балів без зміни архітектури рівня,
- стандартизувати механіку у рамках змагальної системи.

Реалізація системи підрахунку балів

Система нарахування балів є ключовою для визначення переможця, оскільки перемога можлива не лише за допомогою фінішу, а й за рахунок кількості отриманих очок.

Підхід до реалізації:

1. Кожен чекпоінт має фіксовану або конфігуровану кількість балів.
2. Перший гравець, який проходить чекпоінт, отримує бонус, що реалізується через механізм “First Arriver Marking” на сервері.
3. Уся логіка підрахунку балів повністю ізольована в GameMode і PlayerState, що дозволяє:
    - легко змінювати алгоритм нарахування,
    - додавати нові типи очок (за час, за взаємодію зі здібностями),
    - не порушувати інші механіки.
4. Балансування реалізовано через data-driven підхід — значення балів зберігається в структурах даних (DataAssets або таблиці параметрів).

Реалізація системи збору та використання здібностей

Ця система є ключовою з точки зору архітектурного вдосконалення. Вона демонструє, як Gameplay Ability System дозволяє:

- мінімізувати зв’язаність,
- стандартизувати ігрову логіку,
- створювати модульний ігровий функціонал.

Система складається з трьох незалежних компонентів:

1. Pickup-об’єкти (збір предметів)

Pickup реалізовано як простий тригер:

- він виявляє потрапляння гравця в область,
- генерує подію “Запит здібності”,
- сам НЕ знає, яку здібність буде отримано.

Це важлива архітектурна особливість — предмет не містить логіки здібностей.

Pickup:

- може бути легко скопійований,
- може мати свій власний час відновлення (respawn),
- працює виключно на рівні подій.

2. Випадкове надання здібностей

У грі застосовано механізм випадкового вибору здібності з пулу.

Пул зберігається у вигляді:

- таблиці даних,
- або масиву GameplayAbility класів.

Це дозволяє:

- додавати нові здібності без зміни логіки Pickup,
- конфігурувати різні рівні складності,
- контролювати рідкість здібностей у майбутньому.

3. Використання здібностей через Gameplay Ability System

Кожна здібність:

- є незалежним модулем,
- активується через AbilitySystemComponent,
- містить власний набір GameplayEffects,
- має Gameplay Tags для контролю станів,
- може взаємодіяти з іншими гравцями.

У грі здібності можуть:

- ускладнювати рух суперника
- покращувати характеристики гравця,
- створювати тимчасові перешкоди.

Також реалізоване обмеження до двох здібностей, яке підтримується ASC і не потребує складного коду.

Синхронізація здібностей у мережевому середовищі

Оскільки гра є багатокористувацькою, важливу роль відіграє:

- реплікація ефектів,
- синхронізація змін стану,
- коректна обробка взаємодій між клієнтом і сервером.

GAS надає:

- автоматичну реплікацію GameplayEffects,
- повну серверну авторитетність,
- коректне оновлення станів персонажа на клієнтах.

Це дозволило уникнути розсинхронізації навіть у випадках швидкої зміни ефектів чи одночасного застосування кількох здібностей.

Реалізація ігрових механік у проєкті заснована на модульній архітектурі, що була визначена у другому розділі. Системи перегонів, підрахунку балів та здібностей демонструють застосування принципів розширюваності та інверсії залежностей. Взаємодія між підсистемами організована за допомогою подієвої моделі та Gameplay Ability System, що забезпечує гнучкість, масштабованість та легкість у подальшому розвитку проєкту.

3.3 Реалізація масштабування на прикладах

Однією з ключових цілей розробленої методики є забезпечення можливості масштабування архітектури ігрового застосунку без необхідності модифікації базових компонентів системи. Для перевірки ефективності підходу було проведено експериментальне розширення функціональності гри шляхом додавання нових здібностей, нових областей впливу та нових механік взаємодії, причому ядро системи

залишалося незмінним. У цьому підрозділі наведено результати такої експериментальної перевірки.

Додавання нових здібностей

Використання Gameplay Ability System дозволило реалізувати механіку, у якій кожна здібність існує як окремий модуль, що не залежить від решти системи. Це означає, що нові здібності можуть бути додані без змін у класах гравця, Pickup-об’єктів або існуючих Ability-класів.

Приклад масштабування: додавання нової здібності

У рамках експерименту було додано кілька різних за принципом дії здібностей, зокрема:

- здібність, що уповільнює іншого гравця;
- здібність, що створює тимчасову перешкоду на рівні;
- здібність, що дає гравцеві короткочасне прискорення.

Процес додавання нової здібності складався лише з таких кроків:

1. Створення нового класу Ability (наслідування від базового GameplayAbility).
2. Налаштування відповідного GameplayEffect.
3. Внесення цього Ability у пул доступних здібностей Pickup-об’єкта.

Жодних змін у:

- класі персонажа,
- AbilitySystemComponent,
- логіці взаємодії з Pickup,
- системі підрахунку очок,
- мережевій структурі гри

не потрібно було вносити.

Таким чином, система здібностей підтвердила свою модульність і гнучкість.

Додавання нових областей впливу

Другим прикладом масштабування стало створення нових типів зон впливу. Оскільки у грі реалізовано базовий клас «AreaVolume», який визначає:

- події входу,
- події перебування,
- події виходу,

а також надає універсальний інтерфейс для застосування GameplayEffect, додавання нової зони впливу стало завданням, що не потребує змін у кодовій базі.

Приклад масштабування: нова зона з періодичним дебафом

Було створено область, яка:

- кожні кілька секунд накладала на гравця дебаф,
- зменшувала швидкість,
- і могла конфігуруватися через GameplayEffect.

Для реалізації цього механізму потрібно виконати лише:

1. Створити новий Blueprint або C++ клас, що наслідує AreaVolume.
2. Призначити GameplayEffect, який має застосовуватись.
3. Налаштувати інтервал ефекту у властивостях об’єкта.

Важливо, що:

- логіка входу/виходу вже є у базовому класі;
- ASC самостійно обробляє застосування та зняття ефектів;
- інші механіки гри не зазнали жодних змін.

Таким чином, система областей підтвердила свою здатність до масштабування без рефакторингу.

Масштабування без модифікації ядра системи

Експеримент показав, що завдяки правильно обраній архітектурі:

- ядро гри (Character, GameMode, ASC) не потребує змін,

- підсистеми розширюються незалежно,
- усі нові механіки інтегруються через чіткі контракти.

Це стало можливим завдяки:

- розділенню відповідальності між підсистемами,
- data-driven підходу (усі налаштування — у даних, не у коді),
- використанню Gameplay Tags для визначення поведінки,
- мінімізації міжмодульних залежностей,
- подієвій моделі взаємодії.

Підтвердження принципу “Open-Closed”

Усі наведені розширення виконано за принципом:

система відкрита для розширення, але закрита для модифікації.

Жоден базовий клас не був переписаний. Жодна існуюча механіка не була порушена. Уся логіка інтегрувалася через існуючі точки розширення.

Проведені експериментальні розширення продемонстрували ефективність запропонованої методики проєктування. Завдяки використанню Gameplay Ability System, абстракцій для зон впливу та модульної структури гри стало можливим додавати нові елементи геймплею без необхідності редагувати ядро застосунку. Це підтвердило як масштабованість архітектури, так і її готовність до подальшого розвитку у майбутніх проєктах.

3.4 Аналіз результатів експерименту

Експериментальне дослідження, проведене в рамках реалізації ігрового застосунку, дозволило оцінити ефективність запропонованої методики вдосконалення архітектури на рушії Unreal Engine 5. Розширення функціональності, додавання нових здібностей, зон впливу та змагальних механік показало, що архітектура, побудована

на принципах модульності, подієвій моделі та Gameplay Ability System, є стійкою до змін та готовою до масштабування.

У цьому підрозділі наведено підсумкову оцінку складності розширення, порівняння з традиційними підходами та визначено ключові обмеження, які було виявлено під час експерименту.

Оцінка складності розширення системи

Результати показали, що запропонована архітектура забезпечує значне зниження складності розширення у порівнянні з монолітною або слабо структурованою архітектурою.

Важливі характеристики, підтверджені експериментально:

1. Незалежність модулів
   Нові здібності, зони впливу та правила підрахунку балів можна додавати без зміни існуючих класів персонажа, логіки руху чи серверного контролю.
2. Зменшення обсягу коду при розширенні
   У більшості випадків додавання нових механік вимагало:
   - створення нового Ability-класу,
   - створення GameplayEffect,
   - налаштування DataAsset.

   Це значно спрощує масштабування та скорочує час розробки.
3. Підвищена стабільність системи
   Оскільки ядро застосунку не модифікується, ризик виникнення помилок, пов’язаних з регресією, значно зменшується.
4. Висока адаптивність у мультиплеєрі
   GAS забезпечує коректну реплікацію ефектів та станів без ручного дописування логіки, зменшуючи кількість помилок при взаємодії з кількома клієнтами.

   Порівняльний аналіз архітектурних рішень

Порівняння запропонованої архітектури з традиційними підходами у розробці ігор виявило низку переваг (табл. 1.1).

Таблиця 1.1

Порівняння архітектур

Критерій	Монолітна архітектура	Запропонована архітектура
Залежність між компонентами	Висока	Мінімальна
Вартість розширення	Зростає експоненційно	Лінійна
Стійкість до змін	Низька	Висока
Масштабованість	Обмежена	Висока

У монолітних системах модифікація механік часто вимагає переписування пов’язаних компонентів. У запропонованій архітектурі завдяки GAS та подієвій моделі такого не виникає.

2. Порівняння з класичним компонентним підходом без GAS

Без використання Gameplay Ability System механіки зазвичай реалізуються через:

- ручні зміни станів,
- прямі виклики методів персонажа,
- логіку, розподілену між кількома класами,
- дублювання коду.

Це ускладнює підтримку і масштабування.

У архітектурі:

- стан зберігається централізовано в ASC,
- ефекти уніфіковані через GameplayEffect,
- здібності ізольовані й легко розширюються,
- вся логіка інтеграції стандартизована.

Таким чином, GAS забезпечує значно більшу структурованість та гнучкість.

Обмеження запропонованої методики

Попри численні переваги, дослідження виявило низку обмежень та потенційних складнощів, характерних для архітектури, побудованої на GAS та подієвих моделях.

1. Складність початкового впровадження

Базове налаштування GAS вимагає:

- створення компонентів,
- системи атрибутів,
- структури ефектів,
- відповідної ієрархії класів.

Для недосвідчених розробників це може бути складним порогом входу.

2. Залежність від коректної роботи тегів

Gameplay Tags виступають критичним елементом системи. Невірно налаштований тег може:

- блокувати здібність,
- викликати конфлікт ефектів,
- спричинити непередбачувану поведінку.

Це вимагає чіткої стандартизації тегів у командних проєктах.

3. Потенційний ріст складності при великій кількості здібностей

Хоча система дозволяє масштабування, при десятках або сотнях здібностей:

- виникає потреба у класифікації,
- групуванні ефектів,
- автоматизації балансу.

Це не є недоліком архітектури, але потребує додаткових інструментів.

4. Продуктивність у складних мережевих сценаріях

GAS є ефективною, але її інтенсивне використання у швидких змагальних іграх може створювати навантаження на реплікацію. Це потребує оптимізації при збільшенні кількості гравців.

Аналіз результатів експериментального розширення доводить, що запропонована методика вдосконалення архітектури ігрового застосунку забезпечує високу масштабованість, стійкість до змін та гнучкість у розробці нових механік. Завдяки використанню Gameplay Ability System, подієвої моделі та чіткої структуризації підсистем стало можливим додавати нову функціональність без суттєвих змін у кодовій базі.

Попри певні обмеження, загальна ефективність підходу підтверджена практичними результатами, що дозволяє рекомендувати запропоновану архітектуру для застосування у багатокористувацьких іграх, орієнтованих на динамічні та розширювані механіки.

Висновок

У третьому розділі було проведено практичну реалізацію запропонованої методики вдосконалення архітектури ігрового застосунку на рушії Unreal Engine 5, а також здійснено експериментальну перевірку її ефективності. Реалізована гра виступила демонстраційним прикладом застосування принципів модульності, розширюваності та інверсії залежностей, викладених у теоретичній частині.

У підпункті 3.1 було сформовано архітектурну модель ігрового застосунку, що включає структуру підсистем, клієнт–серверну організацію, розподіл ролей між сервером і клієнтами, а також інтеграцію Gameplay Ability System у загальну архітектуру гри. Було показано, що така структура забезпечує чітке розмежування обов’язків, узгодженість механік у мережевому середовищі та можливість ізольованого розвитку окремих підсистем.

У підпункті 3.2 наведено реалізацію основних ігрових механік: системи перегонів і чекпоінтів, механізму підрахунку балів, системи здібностей, а також механізму збору Pickup-об’єктів. Застосовані рішення підтвердили, що Gameplay Ability System та подієва модель взаємодії дозволяють створити гнучкі й універсальні механіки, які легко адаптуються під нові вимоги, не порушуючи існуючої структури.

У підпункті 3.3 було здійснено масштабування проєкту шляхом додавання нових здібностей і областей впливу. Експеримент показав, що запропонована архітектура дає змогу розширювати функціональність системи без внесення змін у її ядро. Нові здібності, зони впливу та поведінкові механізми інтегрувалися виключно через базові абстракції, GameplayEffect та GameplayTags, що підтверджує відповідність архітектури принципу відкритості до розширення та закритості до модифікації.

У підпункті 3.4 здійснено аналітичну оцінку результатів експерименту. Було визначено, що запропонована методика дозволяє значно зменшити складність розширення, підвищити стабільність застосунку та забезпечити краще керування залежностями між компонентами. Одночасно виявлено низку обмежень, пов’язаних із початковою складністю налаштування GAS, потребою стандартизації тегів та необхідністю оптимізації при великій кількості гравців або здібностей.

У сукупності результати третього розділу показують, що розроблена методика не лише є теоретично обґрунтованою, але й демонструє високу ефективність на практиці. Реалізований ігровий застосунок на Unreal Engine 5 підтвердив здатність архітектури до масштабування, стійкість до еволюції та відповідність вимогам сучасної інженерії програмного забезпечення.

ВИСНОВКИ

У магістерській кваліфікаційній роботі було розроблено та обґрунтовано методику вдосконалення архітектури ігрового застосунку на рушії Unreal Engine 5, орієнтовану на забезпечення масштабованості, модульності та стійкості до подальшого розвитку. Поставлена мета була повністю досягнута за рахунок комплексного дослідження теоретичних, методичних та практичних аспектів побудови ігрової архітектури, а також реалізації програмного прототипу, що підтвердив ефективність запропонованих рішень.

У першому розділі було досліджено теоретичні основи архітектури програмного забезпечення та особливості побудови ігрових рушіїв. Визначено ключові принципи, які впливають на якість архітектури: модульність, гнучкість, масштабованість, інкапсуляція та низька зв’язаність компонентів. Проведений аналіз сучасних ігрових рушіїв та, зокрема, архітектури Unreal Engine 5 показав, що UE5 створює потужні можливості для побудови комплексних ігрових систем, проте вимагає застосування системного підходу до організації ігрової логіки.

У другому розділі сформовано методику вдосконалення архітектури ігрового застосунку шляхом впровадження принципів модульного проєктування, подієвої моделі взаємодії та використання Gameplay Ability System як архітектурного ядра. Було визначено вимоги до масштабованої архітектури, розроблено підхід до розподілу відповідальностей між підсистемами, описано інтеграцію GAS у структуру застосунку та сформовано правила побудови розширюваних ігрових механік на основі абстракцій, GameplayEffect та GameplayTags. Запропонована методика орієнтована на відкритість архітектури до розширення і її закритість до модифікацій, що відповідає сучасним вимогам інженерії ПЗ.

У третьому розділі проведено реалізацію архітектури ігрового застосунку-прототипу та здійснено експериментальне дослідження ефективності розробленої методики. Реалізовано систему перегонів, систему чекпоінтів, механізм підрахунку

балів, компоненти збору здібностей і модульну систему ігрових умінь на базі GAS. Дослідження показало, що додавання нових здібностей, зон впливу та поведінкових механік не потребує зміни базових класів, а здійснюється за рахунок створення нових абстракцій і конфігураційних об’єктів. Це підтвердило низьку зв’язаність компонентів та високу гнучкість архітектури.

Аналіз результатів продемонстрував, що використання Gameplay Ability System, подієвої моделі та data-driven підходів забезпечує істотне зниження складності масштабування, спрощує підтримку коду та зменшує ризики появи помилок при розширенні функціональності. Разом із тим було визначено низку обмежень, пов’язаних з початковою складністю налаштування GAS, необхідністю стандартизації Gameplay Tags та можливими навантаженнями у складних мережевих сценаріях.

Узагальнюючи отримані результати, можна стверджувати, що запропонована методика вдосконалення архітектури ігрового застосунку є ефективною та практично придатною для розробки багатокористувацьких ігрових систем на Unreal Engine 5. Вона забезпечує можливість швидкої інтеграції нових механік, покращує керованість проєктом, підвищує якість архітектурних рішень та сприяє зниженню технічного боргу в процесі розвитку застосунку.

Отримані результати можуть бути використані у подальших дослідженнях архітектури ігрових систем, у проєктуванні комерційних та навчальних ігор, а також у розробленні методичних матеріалів для навчальних дисциплін з програмної інженерії та розробки програмних систем.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Bass, L., Clements, P., Kazman, R. Software Architecture in Practice. 4th ed. Addison-Wesley, 2021. 624 p.
2. Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994. 395 p.
3. Buschmann, F., Henney, K., Schmidt, D. Pattern-Oriented Software Architecture. Vol. 1–5. Wiley, 2007.
4. Fowler, M. Patterns of Enterprise Application Architecture. Addison-Wesley, 2002. 533 p.
5. Unreal Engine Documentation – Official Site. Epic Games, 2024. URL: https://docs.unrealengine.com (дата звернення: 15.02.2025).
6. Unreal Engine 5 GAS Documentation – Gameplay Ability System Overview. Epic Games, 2024. URL: https://docs.unrealengine.com/GameplayAbilitySystem (дата звернення: 15.02.2025).
7. The Unreal Engine 5 Source Code – GitHub Repository. Epic Games, 2024. URL: https://github.com/EpicGames (дата звернення: 15.02.2025).
8. Lewis, J. Blueprints Visual Scripting for Unreal Engine. Packt Publishing, 2022. 350 p.
9. Midgley, J. Unreal Engine 5 Game Development with C++. Packt Publishing, 2023. 580 p.
10. Veilleux, S. Game Programming Patterns. GPP Publishing, 2020. URL: https://gameprogrammingpatterns.com (дата звернення: 14.02.2025).
11. Nystrom, R. Game Architecture Patterns. O'Reilly Media, 2021. 412 p.
12. Gregory, J. Game Engine Architecture. 3rd ed. A K Peters/CRC Press, 2018. 1045 p.
13. Millington, I., Funge, J. Artificial Intelligence for Games. 3rd ed. CRC Press, 2016. 980 p.
14. Buckland, M. Programming Game AI by Example. Jones & Bartlett Learning, 2020. 739 p.
15. Unreal Engine Networking Overview. Epic Games, 2024. URL: https://docs.unrealengine.com/en-US/InteractiveExperiences/Networking/index.html (дата звернення: 14.02.2025).
16. Stroustrup, B. The C++ Programming Language. Addison-Wesley, 2014. 1368 p.
17. Sutter, H. Exceptional C++: 47 Engineering Puzzles. Addison-Wesley, 2000. 302 p.
18. Official Chaos Physics Documentation. Epic Games, 2024. URL: https://docs.unrealengine.com/Chaos (дата звернення: 15.02.2025).
19. Unity Documentation – Architecture and Components. Unity Technologies, 2024. URL: https://docs.unity3d.com (дата звернення: 13.02.2025).
20. Godot Engine Architecture Overview. Godot Foundation, 2024. URL: https://docs.godotengine.org (дата звернення: 12.02.2025).

21. Khronos Group. Vulkan API Specification. 2024. URL: https://www.khronos.org/vulkan (дата звернення: 13.02.2025).
22. ISO/IEC/IEEE 42010:2011. Systems and Software Engineering — Architecture Description. International Standard, 2011.
23. Sommerville, I. Software Engineering. 10th ed. Pearson, 2016. 820 p.
24. Pressman, R., Maxim, B. Software Engineering: A Practitioner’s Approach. 9th ed. McGraw-Hill, 2020. 976 p.
25. Fairhurst, M. Building Games with Unreal Engine 5. CRC Press, 2023. 460 p.
26. Unreal Engine Gameplay Tags Documentation. Epic Games, 2024. URL: https://docs.unrealengine.com/GameplayTags (дата звернення: 15.02.2025).
27. Wendel, S. Game Systems Design: Architecture, Mechanics, and Patterns. CRC Press, 2022. 320 p.
28. Paterson, J. Architecting Games: Proven Techniques for Game Developers. Packt Publishing, 2021. 350 p.
29. UE5 Multiplayer Best Practices. Epic Games, 2024. URL: https://docs.unrealengine.com/Multiplayer (дата звернення: 15.02.2025).
30. C++ Standards Committee — ISO/IEC 14882:2020. International Standard for C++.
31. Epic Games. Unreal Engine Coding Standard and Guidelines. 2024. URL: https://docs.unrealengine.com/CodingStandard (дата звернення: 12.02.2025).
32. Jansen, J. Data-Driven Gameplay Architecture. GDC Lecture, 2021. URL: https://gdcvault.com (дата звернення: 10.02.2025).
33. Unreal Engine UObjects and Actors Lifecycle Guide. Epic Games, 2024. URL: https://docs.unrealengine.com/UObject (дата звернення: 14.02.2025).
34. Module and Plugin System in Unreal Engine. Epic Games, 2024. URL: https://docs.unrealengine.com/Modules (дата звернення: 14.02.2025).
35. W3C. Event-Driven Programming Models. World Wide Web Consortium, 2019.

ДОДАТКИ

Додаток А

Демонстрація виконання програми

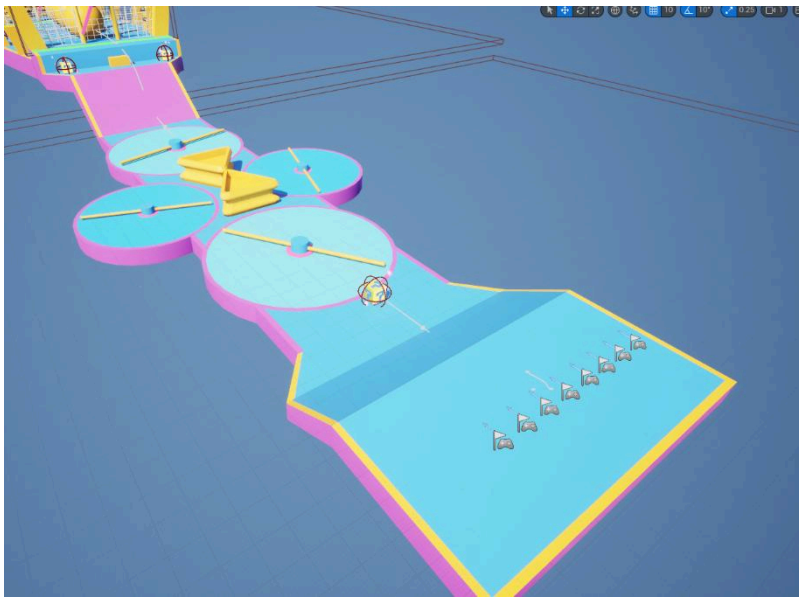

Рис.1. Ігрова мапа

Рис.2. Геймплей гри

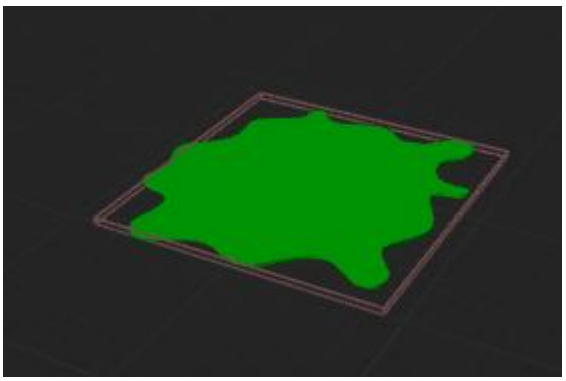

Рис.3. Вигляд однієї із здібностей

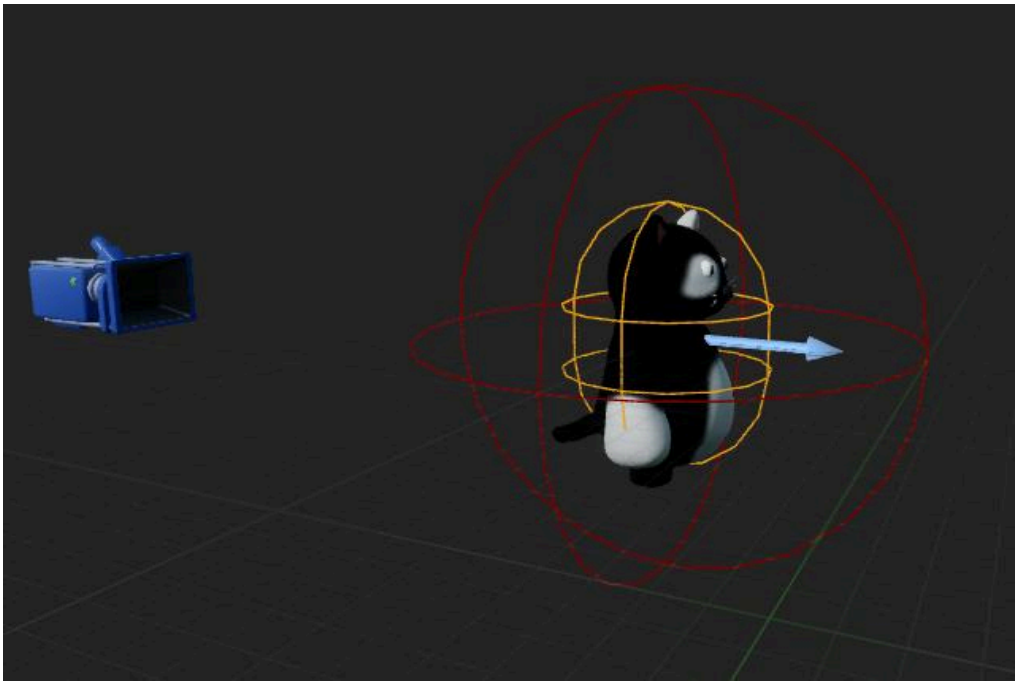

Рис.4. Вигляд класу ігрового персонажу