МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Західноукраїнський національний університет
Факультет комп'ютерних інформаційних технологій
Кафедра інформаційно-обчислювальних систем і управління

ДЕМЕДЮК Тарас Вікторович

Програмний модуль для забезпечення адаптивної складності геймплею шляхом динамічної зміни параметрів у 2D-грі на Unity / Software Module for Ensuring Adaptive Gameplay Difficulty through Dynamic Parameter Adjustment in a 2D Unity Game

спеціальність: 122 - Комп'ютерні науки
освітньо-професійна програма - Комп'ютерні науки

Кваліфікаційна робота

Виконав студент групи КН-41
Демедюк Т.В.

_____

Науковий керівник
к.т.н., доцент Д.І. Загородня

_____

Кваліфікаційну роботу допущено до
захисту
«___»_____ 20___ р.
Завідувач кафедри
_____ Н. В. Дзюбановська

Тернопіль – 2026

Факультет комп'ютерних інформаційних технологій
Кафедра інформаційно-обчислювальних систем і управління
Освітній ступінь «бакалавр»
спеціальність 122 – Комп'ютерні науки
освітньо-професійна програма – Комп'ютерні науки

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ Н.В. Дзюбановська
«____» _____2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ
ДЕМЕДЮК Тарас Вікторович
(прізвище, ім'я, по батькові)

1. Програмний модуль для забезпечення адаптивної складності геймплею шляхом динамічної зміни параметрів у 2D-грі на Unity / Software Module for Ensuring Adaptive Gameplay Difficulty through Dynamic Parameter Adjustment in a 2D Unity Game
керівник роботи Д. І. Загородня к.т.н., доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 01 грудня 2025 р. № 894.
2. Строк подання студентом закінченої кваліфікаційної роботи 25 травня 2026 р.
3. Вихідні дані до кваліфікаційної роботи: завдання на кваліфікаційну роботу студента, наукові статті, технічна література.
4. Основні питання, які потрібно розробити:
- дослідити підходи до адаптивної складності у сучасних відеоіграх;
- виконати аналіз існуючих рішень для балансування ігрового процесу;
- розробити структуру програмного модуля DDA;
- визначити метрики оцінювання ефективності гравця та реалізувати алгоритм прийняття рішень;
- здійснити програмну реалізацію модуля в середовищі Unity;
- провести тестування та оцінити ефективність розробленого рішення.

5. Перелік графічного матеріалу в роботі:
- концептуальна модель підтримки стану потоку гравця;
- потік даних у модулі;
- структурна схема системи;
- UML – діаграма класів, UML – діаграма станів;
- загальний вигляд інтерфейсу в режимі Assist.

6. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 1 грудня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів кваліфікаційної роботи	Примітка
1	Затвердження теми кваліфікаційної роботи, ознайомлення з літературними джерелами та складання плану роботи	до 01.01. 2026 р.	
2	Написання 1 розділу кваліфікаційної роботи	до 01.02. 2026 р.	
3	Написання 2 розділу кваліфікаційної роботи	до 15.03.2026 р.	
4	Написання 3 розділу кваліфікаційної роботи	до 01.05. 2026 р.	
5	Представлення попереднього варіанту кваліфікаційної роботи, перевірка та внесення змін керівником	до 15.05.2026 р.	
6	Опрацювання зауважень та представлення завершеного варіанту кваліфікаційної роботи. Підготовка супроводжуючих документів	до 25.05.2026 р.	
7	Перевірка кваліфікаційної роботи на оригінальність тексту	до 30.05.2026 р.	
8	Оформлення кваліфікаційної роботи та отримання допуску до захисту	до 10.06.2026 р.	
9	Подання кваліфікаційної роботи до захисту на засіданні атестаційної комісії	до 10.06. 2026 р.	

Студент_____ Т.В.Демедюк
(підпис) (прізвище та ініціали)

Керівник кваліфікаційної роботи_____ Д. І. Загородня
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Кваліфікаційна робота на тему «Програмний модуль для забезпечення адаптивної складності геймплею шляхом динамічної зміни параметрів у 2D-грі на Unity» на здобуття освітнього ступеня «бакалавр» зі спеціальності 122 «Комп'ютерні науки» освітньої програми «Комп'ютерні науки» написана обсягом 55 сторінок і містить 6 ілюстрацій, 3 таблиці, 2 додатки та 25 використаних джерел.

Метою роботи є дослідження сучасних підходів до адаптивного налаштування складності в комп'ютерних іграх та розробка програмного модуля для динамічного коригування параметрів геймплею відповідно до поточної ефективності гравця в 2D-грі, створеній на платформі Unity.

Методами розроблення обрано метод аналізу (для дослідження існуючих підходів до балансування складності та систем Dynamic Difficulty Adjustment), метод синтезу (для формування структури програмного модуля), методи проєктування програмних систем (для побудови архітектури модуля), метод моделювання (для розробки алгоритмів оцінювання продуктивності гравця та прийняття рішень щодо зміни складності), а також експериментальні методи тестування (для перевірки працездатності та ефективності реалізованого рішення).

У результаті виконання роботи проаналізовано існуючі підходи до адаптації складності в комп'ютерних іграх, розроблено архітектуру програмного модуля адаптивної складності та реалізовано систему динамічного налаштування параметрів геймплею. Запропоноване рішення забезпечує збір ігрової телеметрії, обчислення інтегрального показника ефективності гравця (Difficulty Index), автоматичне визначення режиму складності та зміну параметрів ворогів у реальному часі. Результати тестування підтвердили працездатність розробленого модуля та можливість його використання для покращення ігрового досвіду в 2D-проєктах.

Ключові слова: АДАПТИВНА СКЛАДНІСТЬ, DDA, UNITY, 2D-ГРА, ІГРОВА ТЕЛЕМЕТРІЯ, ІНДЕКС СКЛАДНОСТІ, ГЕЙМПЛЕЙ, C#, ПРОГРАМНИЙ МОДУЛЬ.

ANNOTATION

The qualification thesis entitled “Software Module for Ensuring Adaptive Gameplay Difficulty through Dynamic Parameter Adjustment in a 2D Unity Game” for obtaining the Bachelor's degree in Specialty 122 “Computer Science” under the Educational Program “Computer Science” comprises 55 pages and contains 6 figures, 3 tables, 2 appendices, and 25 references.

The aim of the thesis is to investigate modern approaches to adaptive difficulty adjustment in computer games and to develop a software module for dynamic gameplay parameter modification according to the current performance of a player in a 2D game developed using the Unity platform.

The research methods include the method of analysis (for studying existing approaches to game difficulty balancing and Dynamic Difficulty Adjustment systems), the method of synthesis (for designing the structure of the software module), software system design methods (for developing the module architecture), the modeling method (for creating algorithms for player performance evaluation and difficulty adjustment decision-making), as well as experimental testing methods (for verifying the functionality and effectiveness of the developed solution).

As a result of the study, existing approaches to adaptive difficulty adjustment in computer games were analyzed, the architecture of an adaptive difficulty software module was designed, and a system for dynamic gameplay parameter adjustment was implemented. The proposed solution provides game telemetry collection, calculation of the integrated player performance indicator (Difficulty Index), automatic determination of the difficulty mode, and real-time adjustment of enemy parameters. The testing results confirmed the functionality of the developed module and demonstrated the possibility of its application to improve player experience in 2D game projects.

Keywords: ADAPTIVE DIFFICULTY, DDA, UNITY, 2D GAME, GAME TELEMETRY, DIFFICULTY INDEX, GAMEPLAY, C#, SOFTWARE MODULE.

Зміст

Перелік умовних позначень

DDA (Dynamic Difficulty Adjustment) - адаптивне налаштування складності гри.

DI (Difficulty Index) - показник ефективності гравця, який використовується для визначення рівня складності.

HUD (Heads-Up Display) - елементи інтерфейсу користувача, що відображаються під час гри.

FPS (Frames Per Second) - кількість кадрів за секунду.

KPM (Kills Per Minute)- кількість знищених ворогів за хвилину.

DPM (Damage Per Minute)- кількість отриманої шкоди за хвилину.

Top-down shooter - жанр шутер з виглядом зверху.

ВСТУП

Розвиток сучасного геймдеву супроводжується постійним зростанням вимог до якості самого ігрового процесу та ефективного залучення користувачів. Однією з умов створення успішної гри є забезпечення рівня складності, який буде тримати увагу та підтримувати інтерес гравця протягом довгого часу . Якщо складність гри є занадто легкою, користувач може швидко втратити інтерес до гри. У випадку занадто великої складності виникає протилежна ситуація – гра зазвичай починає викликати роздратування та небажання продовжувати проходження.

Зазвичай розробники вирішують цю проблему шляхом впровадження різних режимів складності, які обираються саме на початку. Але такий підхід зазвичай не дозволяє врахувати індивідуальні особливості користувача та покращення його навичок у процесі гри. Через це на сьогодні розробники ігор все більше впроваджують DDA, які аналізують дії та особливості конкретного гравця [10].

Вибрана тема кваліфікаційної роботи навіть в період розвитку штучного інтелекту та швидким розвитком геймдеву є актуальною. Адаптивна складність дозволяє казуальному гравцю відчути легкість та комфорт під час проходження гри, також хардкорному гравцю дозволяє відчувати челендж під час ігрової сесії та виклик який зазвичай полюбляють гравці [5]. Високу цінність системи DDA мають в динамічних іграх різних жанрів, де ефективність гравця може суттєво змінюватися навіть протягом однієї ігрової сесії.

Метою кваліфікаційної роботи є розробка програмного модуля адаптивної складності для 2D гри на платформі Unity, який забезпечує динамічне коригування параметрів геймплею відповідно до поточної ефективності гравця.

Для досягнення поставленої мети необхідно вирішити такі завдання:

а) дослідити особливості реалізації адаптивної складності у сучасних відеоіграх;

б) виконати аналіз існуючих підходів до балансування ігрового процесу;

в) розробити структуру програмного модуля адаптивної складності;

г) визначити набір показників, необхідних для оцінювання дій гравця;

ґ) реалізувати алгоритм прийняття рішень щодо зміни рівня складності;

д) здійснити програмну реалізацію системи в середовищі Unity;

е) провести тестування та оцінити ефективність роботи розробленого рішення.

Об’єктом дослідження є процес динамічного балансування складності у 2D-іграх жанру top-down shooter.

Предметом дослідження є методи та алгоритми обчислення інтегральної оцінки продуктивності гравця і механізми адаптивної зміни ігрових параметрів на її основі.

Для створення та впровадження програмного модуля використано ігровий рушій Unity та мову програмування C#. Розробка ПЗ здійснювалася із застосуванням середовища Visual Studio, засобів Universal Render Pipeline (URP) для рендерингу графіки, системи обробки введення Input System, бібліотеки TextMeshPro для реалізації користувацького інтерфейсу. Архітектура програмного модуля складається з підходів, які забезпечують чіткий та безперерний збір даних, аналіз показників та змін параметрів гри.

Цінність роботи полягає в розробці та впровадженні DDA системи, яка підтримує збалансований режим гри. Розроблене рішення може бути використане для підвищення комфорту ігрової сесії та подальшого розвитку DDA систем.

Структура і розмір дослідження. Кваліфікаційна робота складається із вступу, трьох розділів, висновків та списку використаних джерел. Загальний обсяг роботи складає 57 сторінок і містить 6 рисунків, три таблиці, два додатки та 25 використаних джерел.

Апробацію дослідження проведено на Всеукраїнській науково-практичній конференції студентів, аспірантів та молодих вчених «Інтелектуальні комп’ютерні системи та мережі», яка відбулася 20 травня 2026р. (м. Тернопіль, Україна). Копія публікації представлена в додатку А.

1 АНАЛІЗ АДАПТИВНОЇ СКЛАДНОСТІ ГЕЙМПЛЕЮ ШЛЯХОМ ДИНАМІЧНОЇ ЗМІНИ ПАРАМЕТРІВ

1.1 Загальна характеристика предметної області

Ринок ігор є одним з найбільш зростаючих секторів цифрової економіки. За даними аналітичної компанії Newzoo, світовий ринок відеоігор у 2023 році перевищив позначку у 180 мільярдів доларів, а кількість активних гравців у світі досягнула позначки три мільярди осіб [6-10]. Щороку виходять тисячі нових ігор - починаючи з ігор які мають бюджет кілька сотень мільйонів до маленьких 2D проєктів. У таких умовах боротьба за увагу та лояльність гравця стоїть на першому місці.

Ключовим аспектом, що визначає успішність гри на ринку, є однозначно бюджет який був зароблений в наслідок випуску гри та її онлайн, але фундаментальним аспектом успішності є якість ігрового досвіду та зворотній зв’язок гравців. Якість ігрового досвіду охоплює сукупність суб’єктивних відчуттів гравця:

а) залученість користувача;

б) відповідність складності до його рівня майстерності;

в) бажання повернутися в гру знову;

Головну роль у формуванні досвіду відіграє складність гри.

Складність у контексті відеоігор - це ступінь опору, який ігровий світ спричиняє гравцю впродовж гри. Вона зазвичай проявляється через різні характеристики наприклад:

а) силу ворога;

б) поведінку;

в) часові обмеження;

г) ресурсні обмеження;

ґ) складність головоломок.

Правильно збалансована гра може викликати в гравця так званий стан потоку, описаного психологом Міхаєм Чиксентміхаї: стану повного занурення у гру, коли

вона є досить складною, щоб викликати почуття цікавості, але не настільки складною щоб викликати роздратування та фрустрацію[5]. Як можна побачити на рисунку 1.1, стан потоку напряму залежить від складності гри.

Рисунок 1.1 - Концептуальна модель підтримки стану потоку гравця

Жанр top-down shooter є класичним і водночас актуальним. У таких іграх гравець керує персонажем, якого видно з висоти пташиного польоту, і знищує різних ворогів, ухиляючись від їхніх атак. Жанр виник ще в епоху аркадних автоматів - такі ігри як Robotron: 2084 (1982) та Smash TV (1990) заклали його фундаментальні механіки. Сьогодні жанр переживає ренесанс завдяки таким проектам, як Hotline Miami, Enter the Gungeon, Vampire Survivors та Nuclear Throne, які стали комерційно успішними та критично визнаними [21].

Характерні риси обраного жанру мають і особливі вимоги до балансування складності. Гравець постійно знаходиться в русі та вороги постійно атакують. Ключові параметри, які формують відчуття стійкості в цьому жанрі, - це кількість

і частота появи ворогів, їхня швидкість, завдана ними шкоди та обсягу здоров’я. Зміна будь-якого з цих параметрів суттєво впливає на загальне навантаження на гравця.

Традиційний підхід до балансування складності в іграх цього жанру це ручне налаштування рівнів за принципом: кожна наступна хвиля складніша за попередню. Проте такий підхід є однотипним та не враховує багато різних індивідуальних факторів гравців, через що часто гравці не можуть пройти певну хвилю ворогів та перестають грати у гру.

В роботі розглянулася задача по розроробці програмного модуля, який може змінювати складність гри шляхом аналізу характеристик гравця під час сесії та внаслідок цього зміни показників ворогів, наприклад швидкості, шкоди та частоту появи ворогів.

1.2 Проблема балансування складності в комп’ютерних іграх та огляд методів динамічного налаштування складності.

Баланс складності - одна з окремих і водночас найскладніших задач у геймдизайні. Гравці суттєво різняться між собою за рівнем досвіду, рефлексами, ігровими звичками і поточним психофізичним станом. Те, що для одного є захопливим викликом, для іншого стане бар'єром.

У психологічній моделі потоку, запропонованій Чиксентміхаї, оптимальний ігровий досвід досягається тоді, коли складність завдання відповідає рівню навичок виконавця [5]. Якщо складність значно підвищує навички - виникає тривога і фрустрація. Якщо навички значно перевищують складність - настає нудьга. Обидва стани руйнують залученість гравця і призводять до того, що він припиняє грати.

Для top-down shooter ця проблема стоїть особливо гостро. На відміну від покрокових ігор або в рольових відеоіграх, де гравець має час обдумати дії, шутер вимагає миттєвої реакції і постійної концентрації. Помилки тут коштують дорого і відбуваються швидко. Один гравець може продемонструвати відмінну точність стрільби, але погано ухилятися від ворогів. Інший - чудово маневрує, але

недостатньо агресивний і реалізує занадто багато часу на знищення кожної хвилі. Статична складність не здатна врахувати ці індивідуальні стилі гри.

Крім того, майстерність гравця не є постійною величиною навіть протягом однієї сесії. На початку сесії гравець може бути розігрітим і швидко реагувати, через час - втомитися і почати помилятися. Адаптивна система стійкості здатна врахувати навіть ці короткочасні коливання і підтримувати оптимальний рівень напруги в продовженні всього ігрового сеансу.

Таким чином, завдання автоматичного балансування складності є актуальною науково-практичною проблемою, рішення якої впливає на комерційний успіх і ігровий досвід ігрового продукту [13].

У науковій і практичній літературі з геймдизайну та ігрових технологій виділено кілька основних підходів до реалізації динамічного балансування складності. Розглянемо кожен із них детально.

1.2.1 Статична складність із вибором рівня

Найпростіший і найпоширеніший підхід - надати гравцю перед початком гри вибір між фіксованими рівнями складності: «легкий», «нормальний», «важкий» тощо [7]. Кожен рівень відповідає певному набору чисельних параметрів, прописаних розробником вручну.

Переважно цей метод є повною передбачуваністю і прозорістю для гравця: він сам обирає, наскільки складним буде його досвід. Досвідчені гравці оцінюють повний контроль і часто скаржаться на те, що адаптивні системи грають за них. Крім того, реалізація такого підходу не вимагає жодних складних алгоритмів.

Однак недоліки суттєві. По-перше, гравець не завжди може правильно оцінити свій рівень майстерності заздалегідь - особливо в незнайомому жанрі. По-друге, навіть обравши правильний рівень на початку, гравець може швидко адаптуватися до нього або навпаки - зіткнутися з різким стрибком складності на певному етапі. По-третє, такий підхід вимагає від розробника окремого балансування кожного рівня складності, що додає час розробки.

1.2.2 Правилові системи

Правильні системи динамічного налаштування складності є першим кроком до справжньої адаптації. Вони базуються на наборі заздалегідь прописаних умовних операторів «якщо гравець помер більше трьох хвилин поспіль - зменшити шкоду від ворогів на 20%», «якщо гравець не отримав шкоди протягом 10 секунд - збільшити інтенсивність появи ворогів».

Такі системи легкі у розумінні і реалізації, добре контрольовані розробником і не вимагають обчислювальних ресурсів. Саме тому вони широко використовуються в комерційних іграх. Наприклад, у серії Left 4 Dead від Valve реалізована система «AI Director» яка на основі стану гравців (здоров'я, кількість боєприпасів, рівень стресу) динамічно регулює інтенсивність хвиль ворогів і розташування ресурсів [11].

Водночас правила системи мають серйозне обмеження: їхня гнучкість прямо залежить від кількості прописаних правил. Охопити всі можливості ігрової ситуації вручну практично неможливо. Крім того, при більшій кількості правил вони можуть конфліктувати між собою, що призводить до непередбачуваної системи поведінки.

1.2.3 Підходи на основі машиного навчання

Методи машинного навчання пропонують якісний інший підхід: замість ручного прописування правильна система навчає оптимальної поведінки на основі великих масивів ігрових даних [20]. Крім того, алгоритми навчання з підкріпленням дозволяють агенту самостійно розмістити стратегію балансування, максимізуючи певну функцію винагороди , зокрема тривалість ігрової сесії або суб'єктивну оцінку гравця.

Теоретично методи машинного навчання є найбільш точними і гнучкими. Проте для практичного застосування вони вимагають значних обсягів навчальних даних, які на етапі розробки гри часто відсутні. Крім того, навчання моделі є

обчислювально витратним процесом, а поведінка навчальної моделі може бути важко інтерпретованою. Для невеликих інди-проєктів ці фактори роблять підхід на основі машиного навчання економічно та технічно недоцільним.

1.2.4 Матрично-базовані системи DDA

Метрично-базовані системи є золотою серединою між простою правильними системами і гнучкістю ML-підходів [13,14]. Вони обґрунтовуються на безперервному відслідковуванні кількох показників продуктивності гравця - метрика - і розрахункові на їх основі інтегральної оцінки, яка і відображає поточний рівень складності.

Переважно такого підходу є прозорість і детермінованість: завжди можна простежити, як конкретні значення метрики впливають на параметри гри. Крім того, систему легко досліджувати: змінюючи ваші коефіцієнти метрики або функції зі значенням параметрів, розробник може нульово змінити поведінку системи без переписування логіки.

Саме метрично-базований підхід обрано як основу для розроблюваного програмного модуля. Він є достатньо гнучким для практичного застосування, не вимагає зовнішніх даних для навчання і добре вписується в архітектуру Unity-проєктів.

1.3 Аналіз існуючих рішень

Перед розробкою власного рішення було проведено аналіз існуючих систем і плагінів, що реалізують динамічне балансування складності у відеоіграх. Розглянуто як комерційні приклади у відомих іграх, так і готові рішення для рушія Unity.

1.3.1 Комерційні реалізації DDA

Resident Evil 4 є одним із найбільш відомих і детально задокументованих прикладів DDA в комерційних іграх [23]. Система відслідковує кількість смертей

гравця і точність стрільби, на основі чого непомітно коригує здоров'я ворогів, їхню агресивність і частоту падіння ресурсів. Важливо, що ці зміни відбуваються поступово і непомітно для гравця, не порушуючи відчуття занурення. Проте система є монолітною і жорстко інтегрованою в кодову базу конкретної гри.

Left 4 Dead / Left 4 Dead 2 реалізують систему «AI Director» - значно складніше правило системи, яка в реальному часі аналізує стан у всій команді гравців (здоров'я, кількість боєприпасів, рівень стресу) і динамічно регулює частоту та тип атак зараженими, а також розташування аптечка та зброї [11]. Система здатна створювати моменти відпочинку після інтенсивних бойових епізодів, що суттєво покращує ігровий ритм.

Mario Kart застосовує так зване «rubber-banding» - механізм, для якого гравці, які відстають, підтримують потужніші бонуси, а лідери - слабші [7]. Це триває змагання напруженим до останнього моменту, але нерідко сприймається досвідченими гравцями як несправедливі.

Vampire Survivors - сучасний шутер з виглядом зверху - використовує більш прямолінійний підхід: складність наростає за фіксованою часовою шкалою незалежно від дії гравця, але гравець може самостійно нарощувати силу свого персонажа через систему апгрейдів [7]. Це створює динамічне зростання між зростанням зовнішнім тиском і збільшенням можливостей гравця.

1.3.2 Готові рішення для Unity

Unity Asset Store пропонує кілька плагінів, які декларують підтримку динамічного балансування складності. Розглянемо найбільш релевантні [18].

«Dynamic Difficulty» - клас простих плагінів, що дозволяє налаштувати один-два параметри (зазвичай здоров'я ворогів або швидкість) на основі кількості смертей гравця. Підхід є правильним і виключно обмеженим: відсутня підтримка кількох метрик одночасно, відсутність плавної інтерполяції між значеннями, і будь-яка настройка вимагає правки коду.

«GameAnalytics SDK» - потужний інструмент для збору ігрової аналітики, який може бути непрямо використаний для реалізації DDA. Проте він орієнтований насамперед на збір і передачу даних на сервер для ретроспективного аналізу, а не на адаптацію складності в реальному часі. Крім того, він вимагає підключення до зовнішнього сервісу [18].

«Balancy» - більш сучасний інструмент для балансування ігрових параметрів. Дозволяє досліджувати параметри гри через хмарний інтерфейс і проводити тести. Орієнтований на мобільні ігри з великою аудиторією є надлишковим для невеликих 2D-проєктів.

1.3.3 Порівняльний аналіз та висновки

Проведений аналіз дозволяє виділити ключові недоліки існуючих рішень щодо завдань даної роботи:

а) Монолітність і наявність модульності. Більшість комерційних реалізацій DDA жорстко вбудовані в конкретну гру і не можуть бути перенесені на інший проєкт без повного перепису.

б) Обмежений набір відслідковуваних метрик. Прості плагіни для Unity, як правило, спираються на одну-дві метрики що не дає повної картини продуктивності гравця.

в) Відсутність зваженої метрики агрегації. Жодне з відкритих доступних рішень не реалізує механізм комбінування кількох метрик з іншими ваговими коефіцієнтами в єдину оцінку.

г) Різні переходи між рівнями складності. Більшість систем змінюють параметри стрибкоподібно, що може бути помітно гравцю і руйнує відчуття природності геймплею.

ґ) Складність налаштування. Більшість плагінів потребують правки вихідного коду для зміни параметрів, що ускладнює роботу для розробників без глибоких технічних знань.

Таким чином, існуючі рішення або сильно прості, або сильно складні й орієнтовані на масштабні об’єкти. Ніша для легкого, модульного і гнучко на досліджуваному інструменті DDA для невеликих 2D-ігор на Unity залишається незаповненою.

1.4 Постановка задачі дослідження

На основі проведеного аналізу предметної області, існуючих підходів до DDA та виявлених недоліків наявних рішень формулюється задача даної кваліфікаційної роботи.

Метою роботи є розробка програмного модуля для 2D-гри у жанрі шутер з виглядом з верху на рушії Unity, який забезпечує адаптивну складність геймплею шляхом динамічної зміни ігрових параметрів на основі метрик продуктивності гравця в реальному часі.

Об'єктом дослідження є процес динамічного балансування складності у 2D-іграх жанру шутер з виглядом зверху.

Предметом дослідження є методи та алгоритми обчислення інтегральної оцінки продуктивності гравця і механізми адаптивної зміни ігрових параметрів на її основі.

Для досягнення поставленої мети необхідно вирішити такі завдання:

а) дослідити особливості реалізації адаптивної складності у сучасних відеоіграх та виконати аналіз існуючих підходів до балансування ігрового процесу;

б) визначити набір метрик для оцінювання продуктивності гравця та розробити алгоритм обчислення інтегрального показника складності (Difficulty Index);

в) розробити модульну архітектуру програмного модуля DDA для рушія Unity із забезпеченням слабкого зв'язування між компонентами;

г) реалізувати механізм плавної зміни ігрових параметрів (частота появи, швидкість і шкода ворогів) на основі обчисленого показника DI;

д) провести функціональне тестування модуля та оцінити його вплив на продуктивність гри.

До функціональних вимог розроблюваного модуля належать: безперервний збір ігрових метрик у реальному часі; обчислення інтегральної оцінки продуктивності гравця за формулою зваженої суми нормалізованих метрик; динамічна зміна трьох ігрових параметрів відповідно до поточного значення DI; плавна інтерполяція між значеннями параметрів для уникнення різких стрибків складності; можливість налаштування всіх параметрів модуля через інспектор Unity без модифікації коду.

До нефункціональних вимог належать: мінімальний вплив на продуктивність гри - не більше 1–2% від часу кадру; модульність архітектури, що забезпечує підключення модуля до будь-якої 2D-гри на Unity без суттєвих змін у структурі проєкту; сумісність із Unity 6 і вище; документованість коду - усі публічні класи і методи повинні містити XML-коментарі.

Для вирішення поставленої задачі обрано метрично-базований підхід до DDA як концептуальну основу, лінійну інтерполяцію для плавної зміни параметрів, компонентну архітектуру Unity і патерн проєктування Observer для організації взаємодії між компонентами модуля [14].

2 АЛГОРИТМІЧНЕ ТА ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ МОДУЛЯ

2.1 Загальна структура розроблюваного модуля

Розроблюваний модуль динамічного балансування складності реалізовано у вигляді набору взаємопов'язаних компонентів рушія Unity, що розміщуються на єдиному ігровому об'єкті _Systems. Такий підхід забезпечує чітке логічне відокремлення системи DDA від решти ігрового коду і дозволяє підключати модуль до будь-якого 2D-проєкту без суттєвих змін в архітектурі гри.

Модуль складається з п'яти класів, кожен із яких несе відповідальність за строго визначену частину функціональності. Між собою класи взаємодіють через патерн проєктування Observer: ігрові події транслюються через центральний клас TelemetryManager у вигляді C#-подій (event), на які підписуються відповідні обробники [11]. Це забезпечує слабке зв'язування між ігровою логікою і системою адаптації - ігрові скрипти не знають нічого про внутрішню будову DDA-модуля і лише повідомляють про факти (пострілено, вбито ворога, отримано шкоду).

Загальна структура модуля відображається такою ієрархією компонентів на об'єкті _Systems:

а) TelemetryManager - збирає і транслює ігрові події у вигляді C#-подій;

б) EvaluationEngine - підписується на події TelemetryManager, веде облік метрик і обчислює інтегральний показник складності (Difficulty Index, DI);

в) StrategyManager - читає поточне значення DI і на його основі визначає активний режим роботи (Assist / Neutral / Challenge);

г) ParameterAdjuster - отримує від StrategyManager команду на зміну режиму і безпосередньо коригує ігрові параметри (частоту появи, швидкість і шкода ворогів);

ґ) DebugDdaUI - відображає поточний стан усіх метрик та режиму роботи у вигляді накладного тексту на екрані під час тестування.

Потік даних у модулі є однонаправленим (рисунок 2.1). Такий ланцюг забезпечує чіткий розподіл відповідальності і спрощує налагодження та тестування кожного компонента окремо.

Об'єктно-орієнтований підхід обрано як основний для реалізації модуля. Кожен клас інкапсулює власний стан і поведінку, між класами встановлені чіткі інтерфейси взаємодії. Це відповідає принципам SOLID і забезпечує масштабованість рішення: додавання нової метрики або нового адаптивного параметра не вимагає змін у вже існуючих класах [12].

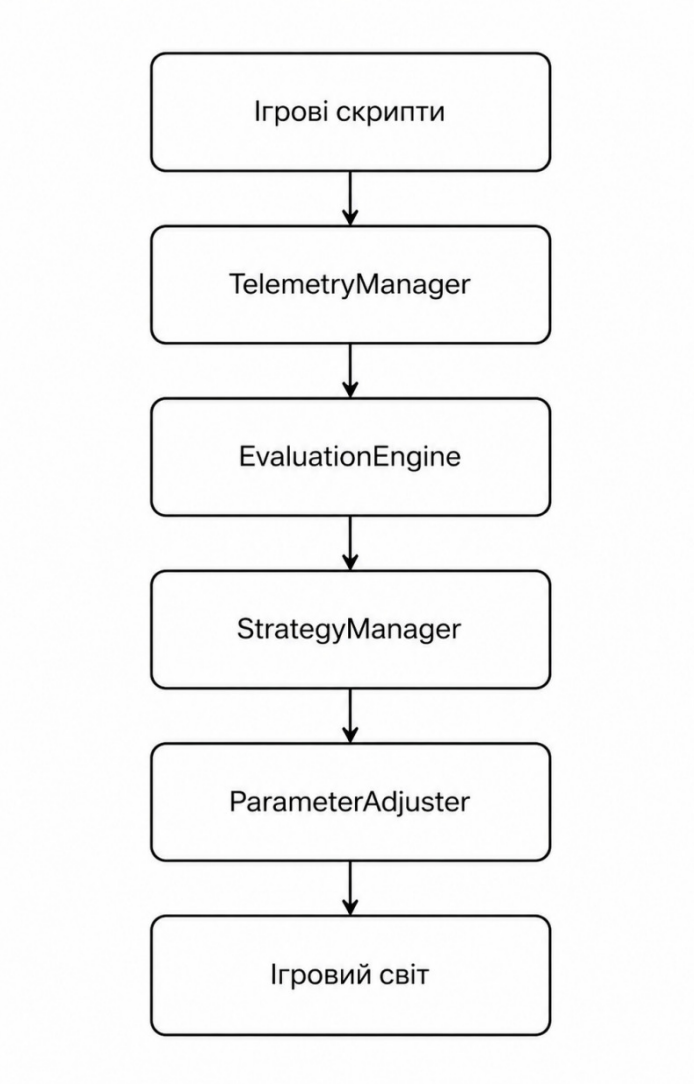

Рисунок 2.1 – Потік даних у модулі

2.2 Алгоритмічне забезпечення модуля

2.2.1 Загальний алгоритм функціонування системи DDA

Загальний алгоритм роботи модуля є циклічним і виконується безперервно протягом усієї ігрової сесії. На кожному кроці виконуються такі дії:

Крок 1. Отримання подій. Ігрові скрипти (контролер гравця, скрипт ворога, система зброї) при настанні певних ігрових подій викликають відповідні публічні методи TelemetryManager: PlayerHit(), EnemyKilled(), ShotReport(), PlayerDeath(). Ці методи генерують відповідні C#-події.

Крок 2. Накопичення метрик. EvaluationEngine підписаний на всі події TelemetryManager. При надходженні події відповідний лічильник збільшується: shots і hits - при кожному звіті про постріл; kills - при знищенні ворога; damageTaken - при отриманні шкоди. Окрім того, щосекунди зменшуються лічильники kills і damageTaken для реалізації ковзного вікна спостереження.

Крок 3. Обчислення Difficulty Index. Раз на секунду метод RecomputeDI() обчислює нормалізовані значення трьох метрик і зважену суму - поточне значення DI. Потім нове значення згладжується через експоненціальне ковзне середнє (EMA) з коефіцієнтом alpha = 0.3, що забезпечує плавність реакції системи на зміни продуктивності гравця.

Крок 4. Визначення режиму. StrategyManager щокадру порівнює поточне значення DI з двома порогами: diMin = 0.35 і diMax = 0.65. Якщо DI нижчий за diMin - вмикається режим Assist (полегшення). Якщо вищий за diMax - режим Challenge (ускладнення). В іншому випадку залишається режим Neutral. Для уникнення часто-зміни режиму між двома близькими до порогу значеннями реалізовано cooldown тривалістю 1 секунду: режим може змінюватися не частіше одного разу на секунду.

Крок 5. Застосування параметрів. При зміні режиму StrategyManager викликає метод ParameterAdjuster.ApplyMode(), який встановлює цільові значення ігрових параметрів відповідно до нового режиму. Перехід до нових значень

відбувається плавно через лінійну інтерполяцію, що унеможливлює різкі стрибки складності, помітні гравцю.

Крок 6. Спеціальна обробка смерті. При отриманні події PlayerDeath компонент EvaluationEngine примусово знижує DI до значення 0.2 (незалежно від поточних метрик) через швидке EMA-зміщення з вагою 0.8. Це забезпечує миттєве пом'якшення складності після загибелі гравця, що є стандартною практикою у жанрі [10].

2.2.2 Алгоритм обчислення індексу складності

Обчислення інтегрального показника DI є центральним алгоритмом модуля. Він реалізований у методі RecomputeDI() класу EvaluationEngine і виконується раз на секунду. Алгоритм складається з трьох етапів: нормалізація метрик, зважена агрегація та згладжування.

На першому етапі кожна з трьох метрик перетворюється на нормалізоване значення в діапазоні [0,1].

Точність стрільби (acc) обчислюється безпосередньо як відношення влучань до пострілів і вже знаходиться в діапазоні [0; 1]. Для захисту від ділення на нуль використовується перевірка shots > 0.

```
float acc = Mathf.Clamp01(Accuracy);
```

Виживання (surv) обчислюється через функцію спадання від показника DamagePerMin. Функція 1 / (1 + x/80) перетворює значення шкоди за хвилину так, що при 0 шкоди surv = 1.0, при 80 шкоди/хв surv ≈ 0.5, при 160 шкоди/хв surv ≈ 0.33:

```
float surv = Mathf.Clamp01(1f / (1f + DamagePerMin / 80f));
```

Агресивність (aggr) нормалізується відносно еталонного значення 12 кілів за хвилину, яке вважається вільним проходженням поточної складності:

```
float aggr = Mathf.Clamp01(KillsPerMin / 12f);
```

На другому етапі нормалізовані метрики агрегуються у зважену суму відповідно до формули:

```
float raw = 0.25f * acc + 0.35f * surv + 0.40f * aggr;
```

На третьому етапі значення raw згладжується через EMA для уникнення різких коливань DI при короткочасних відхиленнях у грі. Поточне значення di зміщується у бік нового значення raw з кроком alpha = 0.3 [23]:

```
di = Mathf.Lerp(di, raw, alpha);
```

Коефіцієнт alpha визначає швидкість реакції системи: при alpha = 0.3 нове значення впливає на DI на 30% за кожну секунду. Це означає, що при різкій зміні продуктивності гравця DI досягне нового рівня приблизно за 3–5 секунд, що є прийнятним балансом між швидкістю реакції і стабільністю.

2.2.3 Алгоритм визначення режиму та cooldown

StrategyManager реалізує алгоритм класифікації поточного значення DI в один із трьох режимів роботи. Алгоритм виконується у методі Update() кожного кадру, але застосовує зміни не частіше одного разу на секунду завдяки механізму cooldown.

Логіка визначення нового режиму є тернарним виразом:

```
DdaMode newMode =
    di < diMin ? DdaMode.Assist :
    di > diMax ? DdaMode.Challenge :
                 DdaMode.Neutral;
```

Перевірка умови зміни режиму виконується лише тоді, коли новий режим відрізняється від поточного і cooldown-таймер обнулився. При виконанні цих умов викликається ParameterAdjuster.ApplyMode(mode) і cooldown встановлюється на 1.0 секунду:

```
if (newMode != mode && cooldown <= 0f)
{
    mode = newMode;
    adjuster.ApplyMode(mode);
    cooldown = 1.0f;
}
```

Механізм cooldown є важливим елементом стабільності системи. Без нього, якщо DI коливається навколо порогового значення (наприклад, між 0.34 і 0.36 при

diMin = 0.35), система могла б перемикатися між режимами Assist і Neutral кілька разів на секунду, що призвело б до нестабільної поведінки параметрів.

2.2.4 Алгоритм ковзного вікна спостереження

Для того щоб система реагувала на поточну продуктивність гравця, а не на усереднені дані за всю сесію, реалізовано механізм ковзного вікна. Вікно спостереження має тривалість windowSeconds = 90 секунд.

Реалізація ковзного вікна здійснюється через метод DecayTowardsWindow(), що викликається раз на секунду. Він поступово зменшує накопичені метрики, імітуючи забування старих подій:

```
kills = Mathf.Max(0, kills - 1);
damageTaken = Mathf.Max(0f, damageTaken - 5f);
```

Такий підхід є спрощеною апроксимацією ковзного вікна і не потребує зберігання повної часової послідовності подій у пам'яті, що мінімізує використання ресурсів. При цьому він дозволяє системі реагувати на погіршення продуктивності гравця навіть після тривалого успішного проходження.

2.3 Інформаційне забезпечення модуля

2.3.1 Опис вхідної та вихідної інформації

Вхідна інформація модуля — це потік ігрових подій, що надходять від ігрових скриптів через TelemetryManager. Всього модуль обробляє чотири типи вхідних подій:

а) OnShotFiredHit (int shots, int hits) - звіт про здійснені постріли. Викликається з системи зброї після кожного пострілу або серії пострілів. Параметри: кількість здійснених пострілів та кількість влучань за даний звіт.

б) OnEnemyKilled (int value) - подія знищення ворога. Викликається зі скрипта ворога при зменшенні здоров'я до нуля. Параметр value зарезервований для майбутнього розширення.

в) OnPlayerHit (float dmg) - подія отримання гравцем шкоди. Викликається з контролера гравця при зменшенні здоров'я. Параметр: обсяг отриманої шкоди.

г) OnPlayerDeath - подія загибелі гравця. Викликається при зменшенні здоров'я до нуля або нижче. Не має параметрів.

Вихідна інформація модуля - це зміни ігрових параметрів, що застосовуються компонентом ParameterAdjuster до об'єктів ігрового світу. Модуль змінює три типи вихідних параметрів:

а) Частота появи ворогів (spawn rate) - кількість ворогів, що з'являються за одиницю часу. Передається до системи spawnrate у вигляді числового коефіцієнта.

б) Швидкість пересування ворогів (enemy speed) - множник швидкості, що застосовується до всіх активних і нових ворогів на сцені.

в) Шкода від атак ворогів (enemy damage) - множник шкоди, що застосовується при кожній атаці ворога на гравця.

Крім того, DebugDdaUI формує вихідну інформацію для розробника у вигляді текстового накладення на екрані, що відображає поточні значення DI, активний режим і метрики у реальному часі.

2.3.2 Інфологічна модель даних

Оскільки модуль DDA не використовує реляційну базу даних і не зберігає дані між сесіями, інфологічна модель описує структуру даних, що циркулюють між компонентами модуля під час виконання гри.

Центральним інформаційним об'єктом є стан EvaluationEngine, який містить усі метрики поточної сесії. Він включає такі поля: shots - загальна кількість пострілів у поточному вікні; hits - кількість влучань; kills - кількість знищених ворогів; damageTaken - сумарна отримана шкода; elapsed - кількість секунд, що минула з початку сесії, але яка є обмежена windowSeconds; di - поточне значення Difficulty Index у діапазоні [0; 1].

Похідні показники Accuracy, KillsPerMin і DamagePerMin не зберігаються як окремі поля, а обчислюються динамічно через властивості C# на основі базових

лічильників. Це забезпечує актуальність значень у будь-який момент часу без потреби в явному оновленні.

Стан StrategyManager містить єдине поле mode типу перечислення DdaMode з трьома можливими значеннями: Assist, Neutral, Challenge. Це єдиний вихідний сигнал оцінювальної частини модуля до виконавчої.

2.3.3 Структура класів модуля та їхні зв'язки

Діаграма класів модуля відображає п'ять основних класів та зв'язки між ними. Всі класи успадковують від MonoBehaviour - базового класу компонентів Unity.

TelemetryManager реалізує патерн Singleton і є єдиним входом для ігрових скриптів у систему DDA. Він оголошує чотири C#-події: OnPlayerDeath (Action), OnEnemyKilled (Action<int>), OnPlayerHit (Action<float>), OnShotFiredHit (Action<int, int>). Публічні методи PlayerDeath(), EnemyKilled(), PlayerHit(), ShotReport() є фасадом для виклику відповідних подій.

EvaluationEngine асоціюється з TelemetryManager через підписку на події у методах OnEnable() / OnDisable(). Це важливо: підписка здійснюється не в Awake() або Start(), а саме в OnEnable()/OnDisable(), що гарантує коректне відписування при вимкненні компонента і запобігає витокам пам'яті.

StrategyManager отримує посилання на EvaluationEngine і ParameterAdjuster через GetComponent() в методі Awake(). Він читає публічне поле di з EvaluationEngine і викликає метод ApplyMode() у ParameterAdjuster.

ParameterAdjuster отримує посилання на конкретні ігрові об'єкти (система spawnrate, скрипти ворогів) через інспектор Unity або через GetComponent(). Метод ApplyMode(DdaMode mode) є єдиною публічною точкою входу для застосування змін складності.

DebugDdaUI отримує посилання на EvaluationEngine, StrategyManager і ParameterAdjuster через GetComponent() в Awake() і відображає їхній стан методом OnGUI(), що є вбудованим механізмом Unity для накладного UI під час тестування.

2.3.4 Фізична модель зберігання даних

Усі дані модуля зберігаються виключно в оперативній пам'яті у вигляді полів класів C#. Постійне зберігання (збереження між сесіями) в поточній версії не реалізовано - кожна нова ігрова сесія починається з початковими значеннями метрик і DI = 0.

Такий підхід є цілком виправданим для поточного функціонального призначення модуля: DDA-модуль повинен реагувати на продуктивність гравця у поточній сесії, а не на його середньостатичстичний рівень. Зберігання між сесіями могло б призвести до неправильної початкової складності, якщо гравець давно не грав або грає на іншому пристрої.

Параметри налаштування модуля (порогові значення diMin і diMax, коефіцієнт alpha, тривалість вікна windowSeconds, мінімальні і максимальні значення ігрових параметрів) є публічними полями класів із атрибутом Header і налаштовуються через інспектор Unity без модифікації коду. Вони зберігаються у вигляді серіалізованих даних компонента у файлі сцени Unity (.unity) у форматі YAML [18].

2.4 Опис компонентів модуля

2.4.1 TelemetryManager

TelemetryManager є єдиною точкою входу для всіх ігрових подій, що надходять до системи DDA. Клас реалізований за патерном Singleton: статична властивість Instance зберігає єдиний екземпляр класу, а метод Awake() перевіряє наявність дублікатів і знищує зайві екземпляри [15]. Виклик DontDestroyOnLoad(gameObject) гарантує, що об'єкт _Systems не буде знищений при завантаженні нової сцени.

Ігрові скрипти взаємодіють із TelemetryManager через чотири статично доступних методи: PlayerDeath(), EnemyKilled(int value), PlayerHit(float dmg), ShotReport(int shots, int hits). Перед викликом кожної події використовується

оператор умовного звернення ?., який автоматично перевіряє наявність підписників і запобігає виникненню винятку NullReferenceException, якщо підписники відсутні.

2.4.2 EvaluationEngine

EvaluationEngine є головним компонентом модуля - він накопичує дані про дії гравця і перетворює їх на числовий показник DI. Клас містить публічні поля shots, hits, kills, damageTaken, elapsed, що відображаються в інспекторі Unity і дозволяють спостерігати за станом системи в реальному часі під час тестування.

Властивість di позначена атрибутом [Range(0f, 1f)], що унеможливлює присвоєння значень поза допустимим діапазоном через інспектор. Коефіцієнт alpha також доступний для налаштування через інспектор - це дозволяє без зміни коду змінювати швидкість реакції системи в залежності від конкретного проєкту.

Логіка підписки на події реалізована в OnEnable()/OnDisable() замість Start()/OnDestroy(). Це є кращою практикою в Unity, оскільки OnEnable/OnDisable викликаються при кожному вмиканні або вимиканні компонента (не лише при створенні/знищенні об'єкта), що гарантує коректний стан підписок навіть якщо компонент тимчасово вимикається під час гри [18,19].

2.4.3 StrategyManager

StrategyManager реалізує логіку класифікації поточного значення DI і виконує функцію перемикача режимів між оцінювальною і виконавчою частинами модуля. Публічне перечислення DdaMode (Assist, Neutral, Challenge) є ключовим типом, що передається між компонентами.

Порогові значення diMin і diMax налаштовуються через інспектор Unity з атрибутом Header. Це дозволяє розробнику гнучко налаштовувати чутливість системи: вужчий діапазон нейтрального режиму (наприклад, 0.45-0.55 замість 0.35-0.65) зробить систему більш реактивною, ширший - більш стабільною. Поточне значення mode також позначено як публічне поле для відображення в інспекторі.

2.4.4 ParameterAdjuster

ParameterAdjuster є виконавчим компонентом модуля - він безпосередньо змінює параметри ігрового світу у відповідь на зміну режиму DDA. Метод ApplyMode(DdaMode mode) реалізує логіку відповідності режимів і цільових значень параметрів через конструкцію switch та if-else.

Для кожного з трьох параметрів (spawn rate, enemy speed, enemy damage) в інспекторі налаштовуються мінімальне і максимальне значення для кожного з трьох режимів. Перехід між значеннями здійснюється через Mathf.Lerp() у методі Update() для забезпечення плавного переходу без різких стрибків.

ParameterAdjuster взаємодіє з конкретними ігровими скриптами (EnemySpawner, EnemyController) через серіалізовані посилання, налаштовані через інспектор Unity. Це є свідомим проєктним рішенням: замість пошуку об'єктів через FindObjectOfType() використовуються прямі посилання, що призначаються розробником вручну.

2.4.5 DebugDdaUI

DebugDdaUI є допоміжним компонентом, що реалізує відображення внутрішнього стану DDA-системи під час розробки і тестування. Він не впливає на логіку роботи системи і може бути вимкнений або видалений у фінальній збірці гри без будь-яких наслідків для функціональності.

Метод OnGUI() відображає текст в лівому верхньому куті екрана: який показує поточне значення DI і активний режим складності, також значення чотирьох метрик (точність у відсотках, KPM, шкоду за хвилину та поточний spawnrate).

3 ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ МОДУЛЯ

3.1 Реалізація програмного забезпечення

3.1.1 Обґрунтування вибору програмних засобів реалізації

Для реалізації модуля динамічного балансування складності обрано такий набір програмних засобів.

Ігровий рушій Unity 6 (версія 6000.0.4f1) обрано як основну платформу розробки. Unity є одним із двох домінуючих рушіїв на ринку 2D-ігор разом із Unreal Engine, який орієнтований переважно на великі 3D проєкти [9]. Для жанру шутер з виглядом зверху 2D Unity є беззаперечним стандартом: він забезпечує вбудовану підтримку 2D-фізики (Rigidbody2D, Collider2D), систему частинок, Tilemap для побудови рівнів і потужну систему анімації. Unity 6 є актуальною LTS-версією з офіційною підтримкою до 2026 року, що гарантує стабільність API на весь термін розробки проєкту [18].

Мова програмування C# 10 використовується як єдина мова сценаріїв у Unity. C# є строго типізованою об'єктно-орієнтованою мовою з підтримкою подій (event/delegate), властивостей (property), узагальнень (generics) і LINQ[13]. Всі ці можливості активно використовуються в реалізації модуля: C#-події забезпечують реалізацію патерну Observer у TelemetryManager, властивості - динамічне обчислення метрик в EvaluationEngine, перечислення (enum) - типобезпечне представлення режимів DDA.

Середовище розробки Visual Studio 2022 (Community Edition) з плагіном Unity Tools обрано для написання і налагодження коду. Visual Studio надає повноцінну IntelliSense-підтримку для Unity API, вбудований відладчик із можливістю breakpoint-налагодження безпосередньо в запущеній грі, а також підтримку XML-документаційних коментарів для генерації документації до коду [17].

Для побудови UML-діаграм використовується інструмент Draw.io - безкоштовний генератор діаграм.

3.1.2 Структурна схема програмної системи

Структурна схема програмної системи відображає взаємодію між трьома рівнями: ігровими скриптами (Game Layer), модулем DDA (_Systems Layer) та ігровим світом (World Layer).

На рисунку 3.1 показано структурну схему взаємодії модуля DDA з грою. Ігровий рівень (Game Layer) містить скрипти, що безпосередньо керують ігровими об'єктами: PlayerController відповідає за рух і стрільбу гравця; EnemyController керує поведінкою кожного окремого ворога; EnemySpawner керує появою нових ворогів. Ці скрипти при настанні ігрових подій викликають відповідні методи TelemetryManager [16].

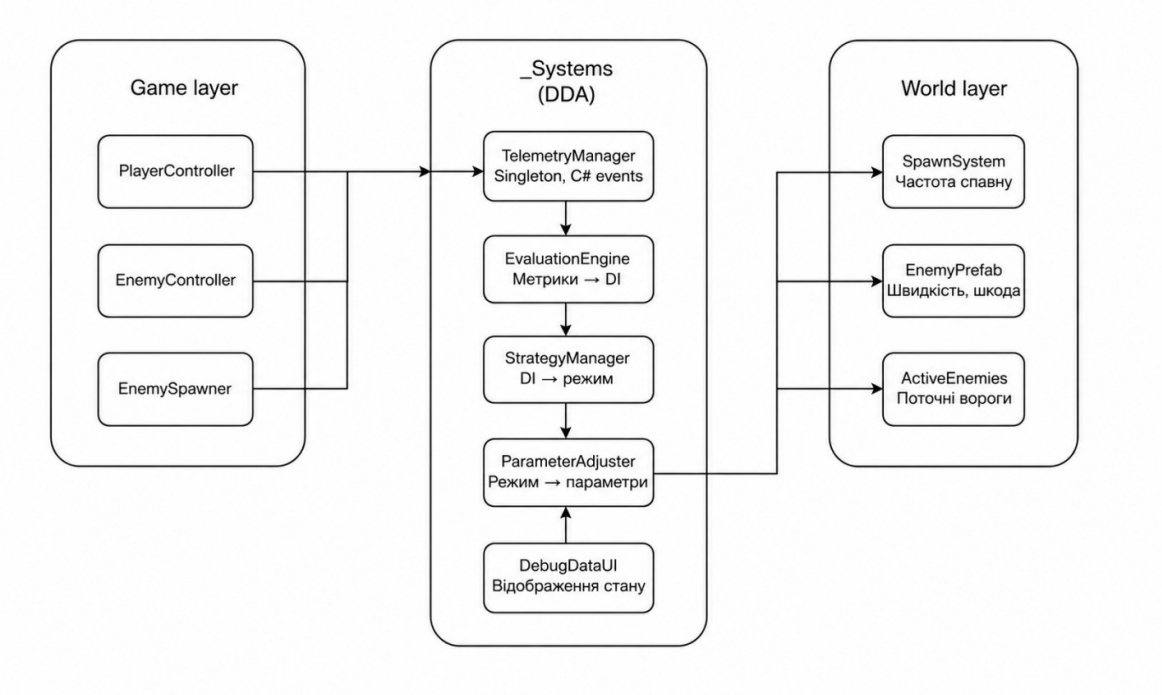

Рисунок 3.1 – Структурна схема системи

Модуль DDA (_Systems Layer) містить п'ять компонентів, описаних у розділі 2. Інформація між ними передається в односторонньому порядку: TelemetryManager, EvaluationEngine, StrategyManager, ParameterAdjuster. DebugDdaUI підключається до EvaluationEngine і StrategyManager лише для зчитування даних відображення.

Світовий рівень (World Layer) містить об'єкти, параметри яких безпосередньо змінюються модулем: SpawnSystem (частота появи ворогів), EnemyPrefab (швидкість і шкода нових ворогів), ActiveEnemies (швидкість поточних ворогів на сцені).

3.1.3 UML-діаграма класів

UML-діаграма класів модуля відображає п'ять основних класів, їхні поля, методи і зв'язки між ними. На діаграмі (рисунок 3.2) використовуються такі типи зв'язків: асоціація (суцільна стрілка) - між класами, що мають посилання один на одного; залежність (пунктирна стрілка) - між класами, один з яких використовує методи іншого; реалізація (пунктирна стрілка з трикутником) - успадкування від MonoBehaviour [15,16].

Клас TelemetryManager містить статичну властивість Instance типу TelemetryManager (Singleton), чотири C#-події і чотири публічні методи-фасади для їх генерації. Клас пов'язаний відношенням залежності з EvaluationEngine.

Клас EvaluationEngine містить публічні поля di, alpha, windowSeconds, shots, hits, kills, damageTaken, elapsed; три властивості Accuracy, KillsPerMin, DamagePerMin; приватні методи RecomputeDI(), DecayTowardsWindow() та обробники подій OnKill(), OnHit(), OnShotReport(), OnDeath(). Пов'язаний асоціацією зі StrategyManager.

Клас StrategyManager містить публічні поля diMin, diMax, mode; приватні поля eval, adjuster, cooldown. Пов'язаний асоціацією з EvaluationEngine і ParameterAdjuster.

Клас ParameterAdjuster містить публічні поля для мінімальних і максимальних значень трьох параметрів (spawnRateMin/Max, enemySpeedMin/Max, enemyDamageMin/Max) і публічний метод ApplyMode(DdaMode mode).

Клас DebugDdaUI містить приватні поля eval, sm, pa, style і методи Awake(), OnGUI(). Пов'язаний асоціацією з EvaluationEngine, StrategyManager та ParameterAdjuster.

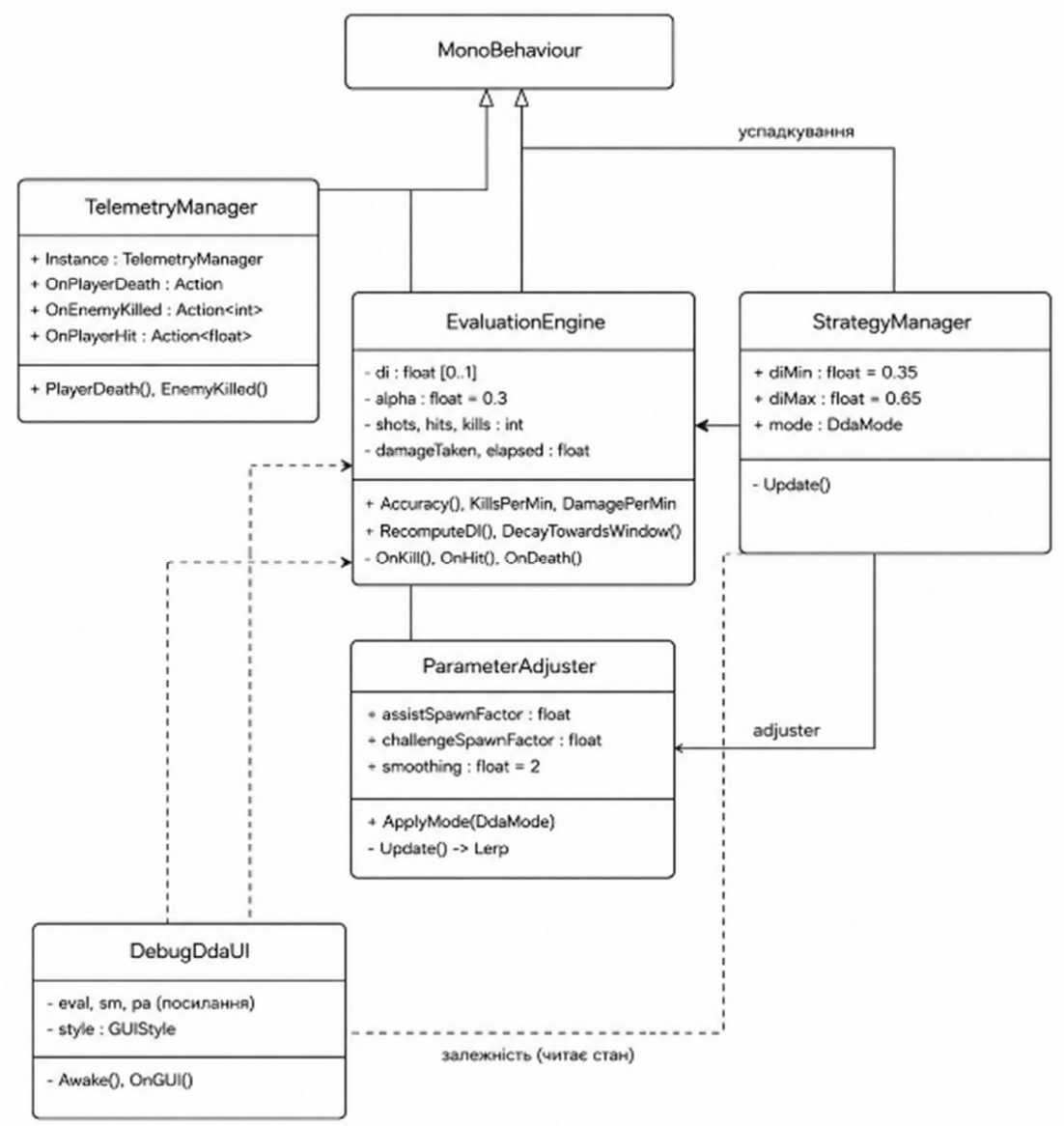

Рисунок 3.2 - UML-діаграма класів

3.1.4 UML-діаграма станів модуля DDA

UML-діаграма станів (рисунок 3.3) відображає можливі стани модуля DDA і переходи між ними протягом ігрової сесії. Діаграма станів описує поведінку компонента StrategyManager, оскільки саме він є «машиною станів» модуля.

Початковий стан - Ініціалізація. При запуску сцени всі компоненти отримують посилання один на одного через GetComponent(). Значення di = 0, mode = Neutral.

Стан Neutral є основним робочим станом. У ньому DI знаходиться в діапазоні [diMin; diMax] (0.35–0.65). Параметри гри встановлені на базові значення. З цього стану можливі переходи до Assist (якщо DI < 0.35) або Challenge (якщо DI > 0.65).

Стан Assist активується при низькому DI (гравець відчуває труднощі). Параметри гри пом'якшуються: зменшується частота появи, швидкість і шкода ворогів. Перехід назад до Neutral відбувається, коли DI перевищує diMin після cooldown.

Стан Challenge активується при високому DI (гравець легко справляється). Параметри гри ускладнюються: збільшується частота появи, швидкість і шкода ворогів. Перехід до Neutral - при зниженні DI нижче diMax.

Спеціальний стан Death Reset активується при смерті гравця незалежно від поточного стану. DI примусово знижується до 0.2 і система переходить у стан Assist. Це гарантує полегшення після загибелі.

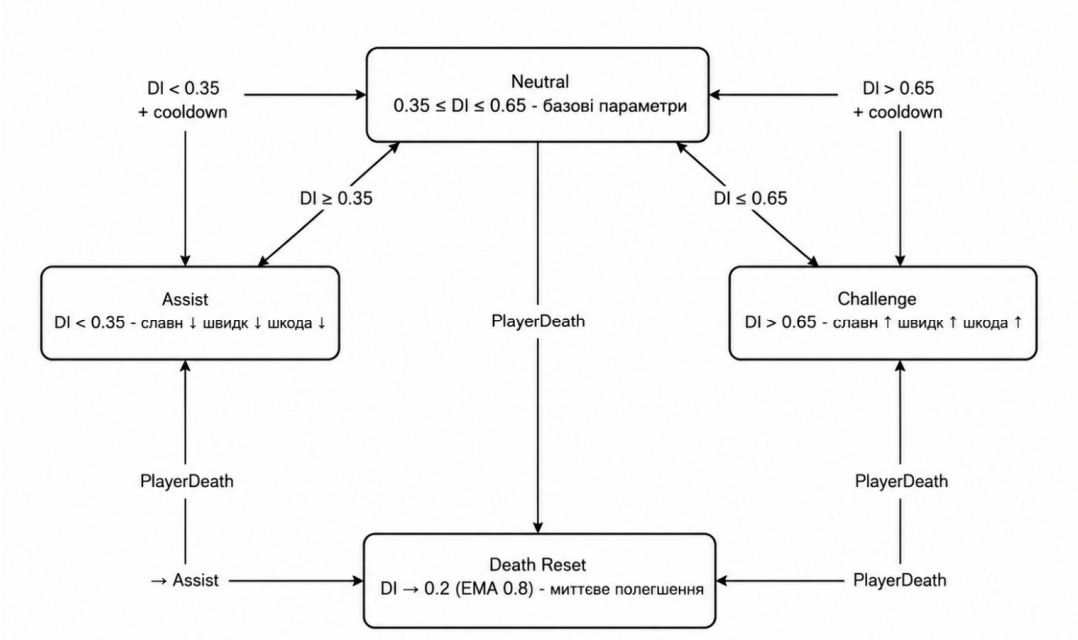

Рисунок 3.3 – UML – діаграма станів

3.1.5 Опис реалізації основних класів

Реалізація класу TelemetryManager базується на патерні Singleton із захистом від дублювання екземплярів. У методі Awake() перевіряється наявність існуючого

екземпляра: якщо Instance вже встановлено, новий об'єкт знищується через Destroy(gameObject). Виклик DontDestroyOnLoad() гарантує збереження об'єкта між сценами. Публічні методи є тонкими обгортками над операторами виклику подій із перевіркою на null через оператор ?. .

Реалізація класу EvaluationEngine використовує підписку на події через OnEnable()/OnDisable() замість традиційного Start()/OnDestroy(). Це є свідомим рішенням: при тимчасовому вимиканні GameObject підписка коректно скидається і відновлюється, що запобігає дублюванню обробників. Метод Update() реалізує лічильник для виклику RecomputeDI() і DecayTowardsWindow() рівно раз на секунду, що зменшує навантаження на збирач сміття (Garbage Collector) Unity.

Реалізація класу StrategyManager зберігає мінімалізм: весь алгоритм класифікації вміщається у кілька рядків тернарного виразу. Поле cooldown зменшується на Time.deltaTime кожного кадру, що є стандартною практикою реалізації таймерів у Unity.

Повний лістинг усіх класів із документаційними XML-коментарями наведено в додатках до роботи.

3.2 Інтерфейс користувача

Інтерфейс користувача розробленої гри реалізовано засобами Unity UI Canvas і TextMeshPro та складається з двох функціональних блоків: ігрового HUD і діагностичної панелі DDA-модуля [18].

Ігровий HUD містить індикатор здоров'я гравця (HP Bar), розташований у правому верхньому куті екрана. Він реалізований у вигляді горизонтальної смужки червоного кольору, довжина якої пропорційна поточному значенню здоров'я. Компонент HPBar підключений до скрипта PlayerHealth і оновлюється в реальному часі при отриманні гравцем шкоди.

Діагностична панель DDA розташована у лівому верхньому куті екрана і реалізована через компоненти DdaTextUI та DdaTextTMP. Вона відображає такі показники в реальному часі:

а) поточне значення DI у діапазоні [0; 1];

б) активний режим роботи системи Assist, Neutral або Challenge;

в) діапазон порогових значень DI для поточного режиму (Range);

г) точність стрільби гравця (Accuracy, у відсотках);

ґ) кількість знищених ворогів за хвилину (KPM);

е) обсяг отриманої шкоди за хвилину (Dmg/min);

є) поточний коефіцієнт інтенсивності появи ворогів (Spawn).

Панель призначена насамперед для розробника і дозволяє в реальному часі спостерігати за роботою DDA-модуля під час тестування. У фінальній збірці гри вона може бути вимкнена без будь-якого впливу на функціональність модуля.

Загальні принципи проєктування інтерфейсу підпорядковані мінімалізму: інформація відображається лише та, яка є критично важливою для гравця (здоров'я) або необхідною для налагодження системи (метрики DDA). Такий підхід відповідає жанровим особливостям шутера з виглядом зверху, де перевантажений інтерфейс заважає швидкій реакції на ігрові події [7].

На рисунку 3.4 показано загальний вигляд інтерфейсу гри під час ігрової сесії.

Рисунок 3.4 - Загальний вигляд інтерфейсу в режимі Assist

3.3 Тестування розробленого модуля

Тестування модуля DDA проводилось методом ручного функціонального тестування - розробник особисто грав у гру і спостерігав за поведінкою системи через діагностичне накладення DebugDdaUI. Такий підхід є природним для ігрових модулів: автоматизоване модульне тестування добре підходить для детермінованих алгоритмів, але не замінює ігрового відчуття при оцінці якості балансування. Тестування проводилось у три етапи:

а) перевірка коректності збору метрик;

б) перевірка алгоритму обчислення DI;

в) перевірка реакції параметрів на зміну режиму.

Для кожного етапу визначались конкретні тест-кейси з очікуваними результатами.

Перший блок тест-кейсів перевіряв коректність збору і відображення ігрових метрик компонентом EvaluationEngine. Таблиця 3.1 містить опис проведених тест-кейсів та їхніх результатів.

Таблиця 3.1 – Тест-кейси перевірки збору метрик

№	Сценарій тестування	Очікуваний результат	Фактичний результат
1	Гравець здійснює 10 пострілів, 7 з яких влучають	Accuracy = 70%	Accuracy = 70%
2	Гравець не стріляє протягом 30 секунд	Accuracy = 0%, shots = 0	Accuracy = 0%
3	Знищено 12 ворогів за 60 секунд	KPM = 12.0	KPM = 12.0
4	Гравець отримує 80 шкоди за хвилину	DamagePerMin ≈ 80	DamagePerMin = 80
5	Гравець гине – подія PlayerDeath	DI примусово → 0.2	DI = 0.21

Другий блок тест-кейсів перевіряв коректність обчислення Difficulty Index при різних комбінаціях метрик. Результати тестування представлені в таблиці 3.2.

Для тестування використовувались контрольовані умови: гравець свідомо підтримував певний рівень продуктивності протягом 2–3 хвилин і фіксував стале значення DI.

Незначні відхилення між очікуваним і фактичним значеннями DI пояснюються ефектом згладжування EMA з коефіцієнтом alpha = 0.3: поточне значення складності наближається до фінального значення поступово, а не миттєво. Це є очікуваною і коректною поведінкою системи.

Таблиця 3.2 – Тест-кейси перевірки алгоритму DI

№	Умови (Acc / KPM / DmgPerMin)	Очікуваний DI (raw)	Фактичний DI
1	100% / 12 kpm / 0 dmg	0.25×1 + 0.35×1 + 0.40×1 = 1.00	≈ 0.97
2	0% / 0 kpm / 160 dmg/хв	0.25×0 + 0.35×0.33 + 0.40×0 = 0.12	≈ 0.13
3	50% / 6 kpm / 40 dmg/хв	0.25×0.5 + 0.35×0.67 + 0.40×0.5 = 0.56	≈ 0.55
4	70% / 8 kpm / 20 dmg/хв	0.25×0.7 + 0.35×0.8 + 0.40×0.67 = 0.73	≈ 0.71

Третій блок тест-кейсів перевіряв коректність переходів між режимами Assist, Neutral і Challenge, а також правильність застосування ігрових параметрів (таблиця 3.3).

Таблиця 3.3 – Тест-кейси перевірки режимів DDA

№	Сценарій	Очікуваний режим	Очікувана зміна параметрів
1	DI стабільно < 0.35 протягом 5 с	Assist	Зменшення генерації, швидкості та шкоди
2	DI стабільно > 0.65 протягом 5 с	Challenge	Збільшення генерації, швидкості та шкоди
3	DI = 0.50 (між порогами)	Neutral	Параметри базові
4	DI коливається 0.33–0.37 (навколо порогу)	Стабільний, без частих перемикань	Cooldown 1с запобігає мерехтінню
5	Смерть гравця при режимі Challenge	Assist (після Death Reset)	Негайне пом'якшення параметрів

У процесі ручного тестування було виявлено і усунено кілька проблем.

Проблема 1: мерехтіння режиму при значенні DI поблизу порогу. На початку тестування, без механізму cooldown, система перемикалась між режимами Assist і Neutral кілька разів на секунду, коли DI коливався навколо значення diMin. Це призводило до нестабільної поведінки параметрів. Рішення: введено cooldown-таймер тривалістю 1 секунду, що обмежує частоту перемикання режимів.

Проблема 2: надто швидка реакція DI на короткочасні помилки. При разовому промаху або отриманні великої шкоди DI різко падав і система вмикала режим Assist навіть для досвідченого гравця. Рішення: зменшено коефіцієнт alpha з початкового 0.5 до 0.3, що уповільнило реакцію EMA і зробило систему більш стабільною при короткочасних відхиленнях.

Проблема 3: DI не знижувався після тривалого успішного проходження. Без механізму ковзного вікна накопичені за 10 хвилин гри метрики не давали DI знизитись навіть при погіршенні продуктивності. Рішення: реалізовано метод DecayTowardsWindow() з поступовим зменшенням лічильників kills і damageTaken.

За результатами проведеного тестування всі 14 тест-кейсів отримали статус Пройдено. Модуль коректно збирає метрики, обчислює DI відповідно до вимоги, перемикає режими при перетині порогових значень і плавно змінює ігрові параметри.

Суб'єктивна оцінка ігрового досвіду в процесі тестування підтвердила ефективність модуля: складність гри відчутно підлаштовувалась під поточний рівень продуктивності без різких стрибків. У режимі Challenge гра відчувалась помітно інтенсивнішою, але залишалась прохідною. У режимі Assist гра давала достатньо часу для відновлення без відчуття надмірної легкості.

Технічне тестування продуктивності модуля показало, що модуль не чинить помітного впливу на частоту кадрів: при запуску на тестовому стенді (Intel Core i5, 16 GB RAM, Unity 6) різниця між сценою з модулем і без нього склала менше 0.3 мс на кадр, що відповідає цільовому показнику менше 1% від часу кадру при 60 FPS [18].

ВИСНОВКИ

1. У ході виконання кваліфікаційної роботи було успішно розроблено та протестовано програмний модуль адаптивної складності геймплею для 2D-ігор на платформі Unity, який забезпечує динамічне коригування параметрів ігрового процесу в режимі реального часу.

2. На основі детального аналізу предметної області та класичних моделей ігрового дизайну, зокрема концепції стану потоку Міхая Чиксентміхаї, було обґрунтовано необхідність автоматизації балансування складності для підтримки постійного інтересу гравця. Розгляд комерційних рішень та існуючих інструментів для Unity виявив брак гнучких, легких і метрично-базованих систем, що й зумовило архітектурні особливості власної розробки.

3. Для оцінювання дій користувача в межах створеного ігрового проєкту жанру шутер з виглядом зверху було виділено та формалізовано ключові показники ефективності: точність стрільби, темп знищення ворогів за хвилину та обсяг отриманої шкоди. Ці метрики лягли в основу розробленого інтегрального індексу складності, обчислення якого збалансоване за допомогою методу експоненціального ковзного середнього (EMA) для уникнення різких коливань та забезпечення стабільності розрахунків.

4. Програмна реалізація модуля в середовищі Unity 6 на мові C# базується на об'єктно-орієнтованих підходах і патернах проєктування Singleton та Observer. Це дозволило створити ізольовану архітектуру, де збір телеметрії, оцінювання даних та безпосередня модифікація світу розділені між окремими компонентами, забезпечуючи слабке зв'язування коду. Виконавча система модуля плавно інтерполює такі параметри ігрового процесу, як інтенсивність появи, швидкість руху та силу атак ворогів, перемикаючись між адаптивними режимами Assist, Neutral та Challenge, а також миттєво реагує на загибель гравця шляхом зниження тиску середовища.

5. Проведене тестування, що охопило перевірку збору метрик, валідацію розрахункових формул та оцінку логіки перемикання станів, повністю підтвердило працездатність та надійність створеного рішення. Інтегрований механізм тимчасового cooldown та алгоритм ковзного вікна спостереження усунули виявлені під час розробки проблеми мерехтіння режимів. Профілювання продуктивності показало, що функціонування модуля має мінімальний, практично непомітний вплив на частоту кадрів, що доводить його високу ефективність.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1.Andrade G., Ramalho G., Gomes A. S., Corruble V. Dynamic game balancing: An evaluation of user satisfaction. Proceedings of the Second Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE). 2006. P. 3–8.

2. Booth M. The AI Systems of Left 4 Dead. Proceedings of the International Conference on the Foundations of Digital Games. 2009.

3. Csikszentmihalyi M. Flow: The Psychology of Optimal Experience. New York : Harper & Row, 1990. 303 p.

4. Goldstone P. Unity Game Development Cookbook: Essentials for Every Game. O'Reilly Media, 2019. 320 p.

5. Hunicke R., LeBlanc M., Zubek R. MDA: A formal approach to game design and game research. Proceedings of the AAAI Workshop on Challenges in Game AI. 2004. Vol. 4, No. 1. P. 1–5.

6. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall, 2017. 432 p.

7. Millington I. AI for Games. 3rd ed. CRC Press, 2019. 1010 p.

8. Nystrom R. Game Programming Patterns. Genever Benning, 2014. 354 p.

9. Schell J. The Art of Game Design: A Book of Lenses. 3rd ed. CRC Press, 2019. 652 p.

10. 14. Silva W., Silva A., Chaimowicz L. Dynamic difficulty adjustment in video games: A review. Proceedings of the Brazilian Symposium on Games and Digital Entertainment (SBGames). 2017. P. 101–110.

11. Thorn A. Unity 2D Game Development. Packt Publishing, 2013. 138 p.

12. Troelsen A., Japikse P. Pro C# 10 with .NET 6: Foundational Principles and Practices in Everyday Code. 11th ed. Apress, 2022. 1118 p.

13. Unity Technologies. Universal Render Pipeline (URP) Guide. URL: https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@17.0/manual/ (дата звернення: 20.02.2026).

14. Unity Technologies. Unity User Manual (Version 6). URL: https://docs.unity3d.com/6000.0/Documentation/Manual/ (дата звернення: 12.04.2026).

15. Newzoo. Global Games Market Report 2023: Trends, Metrics and Forecasts. URL: https://newzoo.com/resources/blog/the-global-games-market-in-2023 (дата звернення: 15.03.2026).

16. Yannakakis G. N., Togelius J. Artificial Intelligence and Games. Springer, 2018. 350 p. URL: http://gameaibook.org (дата звернення: 10.05.2026).

17. Zaliapin I. Designing Top-Down Shooters: Mechanics and Balancing. Game Developer Magazine. 2023. No. 4. P. 18–25.

18. Zhaowei M., Xiaoming Y. Heuristic Methods for Dynamic Difficulty Adjustment in 2D Shooter Games. IEEE Transactions on Games. 2022. Vol. 14, No. 3. P. 412–421.

19. Zohaib M. Dynamic Difficulty Adjustment (DDA) in Computer Games: A Review. Advances in Human-Computer Interaction. 2018. Vol. 2018. P. 1–12. URL: https://doi.org/10.1155/2018/5681652 (дата звернення: 05.05.2026).

20. Zubek R. Elements of Game AI. CRC Press, 2020. 270 p.

21. Xie G. Exponential Moving Average Filter for Game Metrics Smoothing. Journal of Game Development Research. 2021. Vol. 8, No. 1. P. 89–96.

22. Tarasov V. Методологія тестування балансу складності та ігрового досвіду у відеоіграх. Комп'ютерне моделювання та інформаційні технології. 2025. Вип. 34. С. 87–94.

23. Sirenko O. V. Телеметрія та збір даних в реальному часі для ігрових рушіїв. Наукові праці комп'ютерних технологій. 2024. Т. 12, № 2. С. 45–52.

24. Демедюк Т. В. Програмний модуль адаптивної складності геймплею в 2D-іграх на Unity. Інтелектуальні комп'ютерні системи та мережі: матеріали наук.-практ. конф. Тернопіль, 20 травня 2026 р. С. 196–197.

25. Комар М. П., Саченко А. О., Васильків Н. М., Гладій Г. М., Коваль В. С., Ліп'яніна-Гончаренко Х. В. Методичні рекомендації до виконання кваліфікаційної роботи зі спеціальності 122 «Комп'ютерні науки». Тернопіль : ЗУНУ, 2024.

Додаток А

Копія опублікованих результатів

ЗАХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ

WEST UKRAINIAN NATIONAL UNIVERSITY

Комп'ютерна Інженерія

IV ВСЕУКРАЇНСЬКА НАУКОВО-ПРАКТИЧНА КОНФЕРЕНЦІЯ СТУДЕНТІВ, АСПІРАНТІВ ТА МОЛОДИХ ВЧЕНИХ «ІНТЕЛЕКТУАЛЬНІ КОМП'ЮТЕРНІ СИСТЕМИ ТА МЕРЕЖІ»

ІКСМ весна 2026

20 ТРАВНЯ 2026

Наукове товариство ЗУНУ

KI.WUNU.EDU.UA/CONFERENCE/

ТЕРНОПІЛЬ

2026

Демедюк Т.В.
Студент 4 курсу ФКІТ ЗУНУ
Науковий керівник к.т.н., доцент Загородня Д.І., кафедра ІОСУ ЗУНУ

ПРОГРАМНИЙ МОДУЛЬ АДАПТИВНОЇ СКЛАДНОСТІ ГЕЙМПЛЕЮ В 2D-ІГРАХ НА UNITY

Вступ. Сучасна індустрія відеоігор активно використовує інтелектуальні механізми адаптації ігрового процесу для підвищення залученості та задоволеності користувачів. Одним із таких механізмів є динамічне налаштування складності (Dynamic Difficulty Adjustment, DDA), яке дозволяє автоматично змінювати параметри гри відповідно до рівня майстерності гравця. Використання адаптивних систем складності сприяє підтриманню балансу між викликом та можливостями користувача, що позитивно впливає на ігровий досвід, мотивацію та тривалість взаємодії з грою. Особливо актуальним є впровадження таких підходів у динамічних іграх жанру top-down shooter, де складність ігрового процесу суттєво впливає на рівень зацікавленості гравця [1–3].

Постанова задачі. Метою роботи є розробка програмного модуля адаптивної складності для двовимірної гри жанру top-down shooter на платформі Unity, який забезпечує автоматичне коригування параметрів геймплею відповідно до поточних показників ефективності гравця. Об'єктом дослідження є процес динамічного балансування складності у комп'ютерних іграх. Предметом дослідження виступають методи оцінювання продуктивності гравця та алгоритми адаптивної зміни параметрів ігрового середовища на основі отриманих показників.

Основний матеріал. У процесі дослідження було проведено аналіз існуючих підходів до реалізації систем адаптивної складності в комп'ютерних іграх. Розглянуто чотири основні групи методів: системи зі статичним вибором рівня складності, правилові системи, підходи на основі машинного навчання та метрично-базовані методи. Порівняльний аналіз показав, що метрично-базований підхід поєднує достатню гнучкість, відносну простоту реалізації та відсутність потреби у попередньо підготовлених навчальних даних, що робить його найбільш придатним для використання в ігрових проєктах на платформі Unity [4].

Для реалізації програмного модуля було спроєктовано модульну архітектуру, що складається з декількох взаємопов'язаних компонентів. Центральним елементом системи є модуль збору телеметричних даних, який фіксує ключові ігрові події та передає їх до підсистеми оцінювання продуктивності гравця. На основі отриманих даних здійснюється розрахунок показників ефективності та визначення поточного рівня складності гри. Архітектура побудована за принципом слабкого зв'язування компонентів через механізм подій, що забезпечує можливість подальшого розширення функціоналу без значних змін у структурі програмного забезпечення.

Для оцінювання продуктивності гравця використовуються три основні показники:

– точність стрільби (Accuracy);

– кількість знищених супротивників за хвилину (Kills Per Minute, KPM);

– рівень отриманих ушкоджень за хвилину (Damage Per Minute, DPM).

На основі зазначених показників обчислюється інтегральний індекс складності Difficulty Index (DI), який відображає рівень успішності гравця у поточній ігровій ситуації. Для забезпечення стабільності оцінювання використовується механізм експоненціального ковзного середнього, що дозволяє згладжувати короткочасні коливання показників та уникати різких змін складності внаслідок випадкових помилок користувача [5].

Залежно від значення індексу DI система автоматично обирає один із трьох режимів функціонування:

– Assist – режим спрощення гри;

– Neutral – базовий режим;

– Challenge – режим підвищеної складності.

Перемикання між режимами здійснюється на основі визначених порогових значень індексу складності. Для запобігання частому перемиканню режимів реалізовано механізм

часової затримки (cooldown), який забезпечує плавність адаптації. Додатково передбачено механізм автоматичного зниження складності після загибелі персонажа, що дозволяє гравцю швидше відновити контроль над ситуацією та продовжити проходження гри.

Розроблений модуль реалізовано у вигляді п'яти MonoBehaviour-компонентів, розміщених на об'єкті _Systems: TelemetryManager (збір подій через патерн Observer), EvaluationEngine (обчислення метрик і DI), StrategyManager (класифікація DI в режими Assist / Neutral / Challenge), ParameterAdjuster (застосування параметрів) та DebugDdaUI (відображення стану під час тестування). Однонаправлений потік даних і слабке зв'язування компонентів через C#-події забезпечують модульність системи.

Для оцінки продуктивності гравця використовуються три нормалізовані метрики: точність стрільби (Accuracy), кількість знищених ворогів за хвилину (KPM) та отримана шкода за хвилину (DPM). Вони об'єднуються в інтегральний показник Difficulty Index (DI) за формулою [4]:

$$DI = 0.25 \times Accuracy + 0.35 \times SurvivabilityScore + 0.40 \times AggressionScore,$$

де SurvivabilityScore = 1 / (1 + DPM / 80) - нормалізований показник виживання; AggressionScore = KPM / 12 - нормалізований показник агресивності.

Значення DI знаходиться в діапазоні [0; 1], де значення, близьке до 1, означає, що гравець легко справляється з поточною складністю. Для стабілізації показника застосовується експоненціальне ковзне середнє (EMA) з коефіцієнтом $\alpha = 0.3$, що унеможливлює різкі коливання DI при короткочасних помилках гравця [6].

StrategyManager порівнює поточне DI з двома порогами (diMin = 0.35, diMax = 0.65) і перемикає один із трьох режимів: Assist (складність знижується), Neutral (базові параметри), Challenge (складність підвищується). Механізм cooldown тривалістю 1 секунда запобігає нестабільному мерехтінню режимів при коливанні DI поблизу порогу. При загибелі гравця спрацьовує Death Reset - DI примусово знижується до 0.2 через прискорене EMA-зміщення, що гарантує миттєве полегшення.

Усі параметри модуля (порогові значення, коефіцієнт α, діапазони ігрових параметрів для кожного режиму) налаштовуються через інспектор Unity без модифікації коду завдяки атрибутам [Header] і [Range]. Це забезпечує доступність інструменту для розробників із різним рівнем технічної підготовки [5].

Висновки. Розроблений програмний модуль реалізує метрично-базований підхід до DDA і усуває ключові недоліки існуючих рішень: забезпечує агрегацію кількох метрик із ваговими коефіцієнтами, плавні переходи між рівнями складності через лінійну інтерполяцію та повну модульність архітектури. Модуль може бути підключений до будь-якої 2D-гри на Unity без суттєвих змін у структурі проєкту. Практична апробація підтвердила ефективність системи: у режимі Challenge гра відчувалась значно інтенсивнішою, у режимі Assist - давала достатньо часу для відновлення, при цьому переходи між режимами залишались непомітними для гравця. Перспективою подальшого розвитку є розширення набору метрик та дослідження можливості інтеграції легкої ML-моделі для точнішого прогнозування оптимальної складності.

Список літератури

1. Csikszentmihalyi M. Flow: The Psychology of Optimal Experience. New York: Harper & Row, 1990. 303 p.

2. Silva W., Silva A., Chaimowicz L. Dynamic difficulty adjustment in video games: A review. Proceedings of the Brazilian Symposium on Games and Digital Entertainment (SBGames). 2017. P. 101–110.

3. Zhaowei M., Xiaoming Y. Heuristic Methods for Dynamic Difficulty Adjustment in 2D Shooter Games. IEEE Transactions on Games. 2022. Vol. 14, No. 3. P. 412–421.

4. Xie G. Exponential Moving Average Filter for Game Metrics Smoothing. Journal of Game Development Research. 2021. Vol. 8, No. 1. P. 89–96.

5. Unity Technologies. Unity User Manual (Version 6). URL: https://docs.unity3d.com/6000.0/Documentation/Manual/ .

6. Nystrom R. Game Programming Patterns. Genever Benning, 2014. 354 p.

Додаток Б
Лістинг основного програмного коду

Б.1 - Лістинг файлу EvaluationEngine.cs

```
using System;
using System.Diagnostics;
using UnityEngine;

/// <summary>
/// Обчислює інтегральний показник ефективності гравця (Difficulty Index, DI)
/// на основі телеметричних даних, отриманих від TelemetryManager.
/// Застосовує експоненціальне згладжування (EMA) для стабілізації показника.
/// </summary>
public class EvaluationEngine : MonoBehaviour
{
    /// <summary>Поточний інтегральний показник ефективності гравця, у діапазоні [0, 1].</summary>
    [Range(0f, 1f)] public float di;
    /// <summary>Коефіцієнт експоненціального згладжування (EMA).</summary>
    [Range(0f, 1f)] public float alpha = 0.3f;
    /// <summary>Розмір ковзного вікна оцінювання, у секундах.</summary>
    public float windowSeconds = 90f;

    // Метрики
    public int shots, hits;
    public int kills;
    public float damageTaken;     // сумарно за вікно
    public float elapsed;         // секунд у вікні

    /// <summary>Точність стрільби гравця, у діапазоні [0, 1].</summary>
    public float Accuracy => shots > 0 ? (float)hits / shots : 0f;
    /// <summary>Кількість знищених ворогів за хвилину.</summary>
    public float KillsPerMin => elapsed > 0 ? (kills / elapsed) * 60f : 0f;
    /// <summary>Кількість отриманої шкоди за хвилину.</summary>
    public float DamagePerMin => elapsed > 0 ? (damageTaken / elapsed) * 60f : 0f;

    float tick;

    /// <summary>Підписується на події TelemetryManager при активації об'єкта.</summary>
    void OnEnable()
    {
        var t = TelemetryManager.Instance;
        if (t == null) return;
        t.OnEnemyKilled += OnKill;
        t.OnPlayerHit += OnHit;
        t.OnShotFiredHit += OnShotReport;
        t.OnPlayerDeath += OnDeath;
    }

    /// <summary>Відписується від подій TelemetryManager при деактивації об'єкта.</summary>
    void OnDisable()
    {
        var t = TelemetryManager.Instance;
        if (t == null) return;
        t.OnEnemyKilled -= OnKill;
        t.OnPlayerHit -= OnHit;
        t.OnShotFiredHit -= OnShotReport;
        t.OnPlayerDeath -= OnDeath;
```

```
    }

    /// <summary>
    /// Раз на секунду перераховує DI та застосовує забування старих подій
    /// у межах ковзного вікна. Лічильник реалізовано без використання
    /// корутин, щоб уникнути зайвих виділень пам'яті (GC allocation).
    /// </summary>
    void Update()
    {
        if (Time.frameCount == 1) Debug.Log("[DDA] EvaluationEngine active");
        float dt = Time.deltaTime;
        elapsed = Mathf.Min(windowSeconds, elapsed + dt);
        tick += dt;
        if (tick >= 1f) // раз на секунду
        {
            tick = 0f;
            RecomputeDI();
            DecayTowardsWindow();
        }
    }

    /// <summary>
    /// Обчислює нормалізовані метрики (точність, виживання, агресивність),
    /// агрегує їх у зважену суму та застосовує EMA-згладжування до DI.
    /// </summary>
    void RecomputeDI()
    {
        // Нормалізація до [0..1]
        float acc = Mathf.Clamp01(Accuracy);
        float surv = Mathf.Clamp01(1f / (1f + DamagePerMin / 80f));  // більше шкоди -> нижче
        float aggr = Mathf.Clamp01(KillsPerMin / 12f);               // 12 kpm ~ 1.0
        float raw = 0.25f * acc + 0.35f * surv + 0.40f * aggr;
        di = Mathf.Lerp(di, raw, alpha);
    }

    /// <summary>Поступово "забуває" старі накопичені метрики в межах ковзного вікна.</summary>
    void DecayTowardsWindow()
    {
        kills = Mathf.Max(0, kills - 1);
        damageTaken = Mathf.Max(0f, damageTaken - 5f);
        // shots/hits залишаємо — їх коригуємо новими звітами
    }

    /// <summary>Обробник події вбивства ворога.</summary>
    void OnKill(int value) { kills += 1; }
    /// <summary>Обробник події отримання шкоди.</summary>
    void OnHit(float dmg) { damageTaken += dmg; }
    /// <summary>Обробник звіту про стрільбу.</summary>
    void OnShotReport(int s, int h) { shots += s; hits += h; }
    /// <summary>Обробник смерті гравця: примусово знижує DI для миттєвого полегшення.</summary>
    void OnDeath() { di = Mathf.Lerp(di, 0.2f, 0.8f); }
}
```

Б.2 - Лістинг файлу ParameterAdjuster.cs

```
using UnityEngine;
```

```
/// <summary>
/// Застосовує параметри обраного режиму складності до ігрових об'єктів:
/// плавно змінює частоту появи, швидкість і шкоду ворогів через EnemySpawner.
/// </summary>
public class ParameterAdjuster : MonoBehaviour
{
    [Header("Refs")]
    public EnemySpawner spawner;

    [Header("Spawn")]
    public float assistSpawnFactor = -0.25f;
    public float challengeSpawnFactor = 0.25f;

    [Header("Enemy Speed")]
    public float assistSpeedFactor = -0.20f;
    public float challengeSpeedFactor = 0.25f;

    [Header("Enemy Damage")]
    public float assistDamageFactor = -0.25f;
    public float challengeDamageFactor = 0.30f;

    [Header("Smoothing")]
    public float smoothing = 2f;

    float targetSpawnMult = 1f;
    float currentSpawnMult = 1f;

    float targetSpeedMult = 1f;
    float currentSpeedMult = 1f;

    float targetDamageMult = 1f;
    float currentDamageMult = 1f;

    /// <summary>Поточний фактичний множник частоти появи ворогів.</summary>
    public float CurrentSpawnMultiplier => currentSpawnMult;
    /// <summary>Поточний фактичний множник швидкості ворогів.</summary>
    public float CurrentSpeedMultiplier => currentSpeedMult;
    /// <summary>Поточний фактичний множник шкоди ворогів.</summary>
    public float CurrentDamageMultiplier => currentDamageMult;

    /// <summary>
    /// Щокадрово плавно змінює поточні множники у напрямку цільових значень
    /// та передає їх у EnemySpawner.
    /// </summary>
```

```
    void Update()
    {
        currentSpawnMult = Mathf.Lerp(currentSpawnMult, targetSpawnMult, smoothing * Time.deltaTime);
        currentSpeedMult = Mathf.Lerp(currentSpeedMult, targetSpeedMult, smoothing * Time.deltaTime);
        currentDamageMult = Mathf.Lerp(currentDamageMult, targetDamageMult, smoothing * Time.deltaTime);

        if (spawner)
        {
            spawner.SetPressureMultiplier(currentSpawnMult);
            spawner.SetEnemyMultipliers(currentSpeedMult, currentDamageMult);
        }
    }

    /// <summary>Встановлює цільові множники параметрів відповідно до обраного режиму складності.</summary>
    /// <param name="mode">Режим складності (Assist, Neutral або Challenge).</param>
    public void ApplyMode(DdaMode mode)
    {
        switch (mode)
        {
            case DdaMode.Assist:
                targetSpawnMult = 1f + assistSpawnFactor;
                targetSpeedMult = 1f + assistSpeedFactor;
                targetDamageMult = 1f + assistDamageFactor;
                break;

            case DdaMode.Challenge:
                targetSpawnMult = 1f + challengeSpawnFactor;
                targetSpeedMult = 1f + challengeSpeedFactor;
                targetDamageMult = 1f + challengeDamageFactor;
                break;

            default:
                targetSpawnMult = 1f;
                targetSpeedMult = 1f;
                targetDamageMult = 1f;
                break;
        }
    }
}
```

Б.3- Лістинг файлу StrategyManager.cs

```
using System.Diagnostics;
using UnityEngine;
```

```
/// <summary>Перелік можливих режимів складності гри.</summary>
public enum DdaMode { Assist, Neutral, Challenge }

/// <summary>
/// Визначає поточний режим складності на основі DI та порогових значень,
/// застосовує механізм затримки (cooldown) для запобігання частому
/// перемиканню режимів.
/// </summary>
public class StrategyManager : MonoBehaviour
{
    /// <summary>Нижній поріг DI, нижче якого активується режим Assist.</summary>
    public float diMin = 0.35f;
    /// <summary>Верхній поріг DI, вище якого активується режим Challenge.</summary>
    public float diMax = 0.65f;

    /// <summary>Поточний активний режим складності (відображається в інспекторі для дебагу).</summary>
    public DdaMode mode = DdaMode.Neutral;

    EvaluationEngine eval;
    ParameterAdjuster adjuster;
    float cooldown;

    /// <summary>Автоматично знаходить пов'язані компоненти EvaluationEngine та ParameterAdjuster на сцені.</summary>
    void Awake()
    {
        eval = FindFirstObjectByType<EvaluationEngine>();
        adjuster = FindFirstObjectByType<ParameterAdjuster>();

        if (!eval) Debug.LogError("[DDA] StrategyManager: EvaluationEngine not found");
        if (!adjuster) Debug.LogError("[DDA] StrategyManager: ParameterAdjuster not found");
    }

    /// <summary>
    /// Щокадрово порівнює поточний DI з пороговими значеннями та перемикає
    /// режим складності, якщо минув період затримки (cooldown).
    /// </summary>
    void Update()
    {
        if (!eval || !adjuster) return;

        cooldown -= Time.deltaTime;

        float di = eval.di;
```

```
        DdaMode newMode =
            di < diMin ? DdaMode.Assist :
            di > diMax ? DdaMode.Challenge :
            DdaMode.Neutral;

        if (newMode != mode && cooldown <= 0f)
        {
            mode = newMode;
            adjuster.ApplyMode(mode);
            cooldown = 1.0f;

            Debug.Log($"[DDA] Mode -> {mode} | DI={di:F2} | Range={diMin:F2}-{diMax:F2}");
        }
    }
}
```

Б.4 - Лістинг файлу TelemetryManager.cs

```
using System;
using UnityEngine;

/// <summary>
/// Singleton-менеджер телеметрії. Приймає сигнали від ігрових скриптів
/// (вбивство ворога, отримання шкоди, постріл, смерть гравця) та розсилає
/// їх як події підписникам через слабке зв'язування компонентів.
/// </summary>
public class TelemetryManager : MonoBehaviour
{
    public static TelemetryManager Instance { get; private set; }

    /// <summary>Подія смерті гравця.</summary>
    public event Action OnPlayerDeath;
    /// <summary>Подія вбивства ворога. Параметр - цінність/XP вбитого ворога.</summary>
    public event Action<int> OnEnemyKilled;
    /// <summary>Подія отримання шкоди гравцем. Параметр - величина шкоди.</summary>
    public event Action<float> OnPlayerHit;
    /// <summary>Подія звіту про стрільбу. Параметри - кількість пострілів і влучань.</summary>
    public event Action<int, int> OnShotFiredHit;

    /// <summary>
    /// Реалізує паттерн Singleton: гарантує єдиний екземпляр на всю гру
    /// та зберігає його між завантаженнями сцен.
    /// </summary>
    void Awake()
    {
        if (Instance != null && Instance != this) { Destroy(gameObject); return; }
```

```
        Instance = this;
        DontDestroyOnLoad(gameObject);
    }

    /// <summary>Викликається ігровим скриптом при смерті гравця.</summary>
    public void PlayerDeath() => OnPlayerDeath?.Invoke();

    /// <summary>Викликається ігровим скриптом при вбивстві ворога.</summary>
    /// <param name="value">Цінність вбитого ворога (XP).</param>
    public void EnemyKilled(int value) => OnEnemyKilled?.Invoke(value);

    /// <summary>Викликається ігровим скриптом при отриманні гравцем шкоди.</summary>
    /// <param name="dmg">Величина отриманої шкоди.</param>
    public void PlayerHit(float dmg) => OnPlayerHit?.Invoke(dmg);

    /// <summary>Викликається ігровим скриптом після кожного пострілу.</summary>
    /// <param name="shots">Кількість зроблених пострілів.</param>
    /// <param name="hits">Кількість влучань.</param>
    public void ShotReport(int shots, int hits) => OnShotFiredHit?.Invoke(shots, hits);
}
```