

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «бакалавр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)

на тему:

«2D гра жанру Metroidvania у стилі PixelArt: ігрові механіки»

Виконав студент групи ІПЗ-22-2

МОЖАРОВСЬКИЙ Ярослав Сергійович

Керівник роботи:

ЛОБАНЧИКОВА Надія Миколаївна

Рецензент:

Єфремов Юрій Миколайович

Житомир – 2026

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»

Зав. кафедри інженерії
програмного забезпечення
Тетяна Вакалюк
«28» лютого 2026 р.

ЗАВДАННЯ

на кваліфікаційну роботу

Здобувач вищої освіти: **МОЖАРОВСЬКИЙ Ярослав Сергійович**

Керівник роботи: **ЛОБАНЧИКОВА Надія Миколаївна**

Тема роботи: **«2D гра жанру Metroidvania у стилі PixelArt: ігрові механіки»**,
затверджена наказом закладу вищої освіти від «27» лютого 2026 р., №92/с.

Вихідні дані для роботи: об'єктом дослідження є процес створення двовимірних комп'ютерних ігор із дослідницькими та бойовими механіками, характерними для жанру Metroidvania. Предметом дослідження є програмні засоби та алгоритмічні підходи до побудови систем керування персонажем, фізики ігрового світу, бойових механік, розвитку здібностей персонажа та взаємодії з об'єктами віртуального середовища.

Консультанти з бакалаврської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Лобанчикова Н. М.	01.03.2026	01.03.2026
2	Лобанчикова Н. М.	01.04.2026	01.04.2026
3	Лобанчикова Н. М.	01.05.2026	01.05.2026

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи бакалавра містить 140 сторінок, з них 71 сторінка основного тексту, 34 рис., 7 таблиць, 5 додатків та 33 використаних джерел.

Метою роботи є створення двовимірної гри жанру Metroidvania у стилі PixelArt із реалізацією основних ігрових механік. У роботі досліджено особливості жанру та можливості рушія Unity. Для розробки застосовано засоби ООП, компонентний підхід і механізми фізичного моделювання. Результатом роботи є ігровий застосунок, що реалізує керування персонажем, бойову систему, розвиток здібностей, взаємодію з ігровим середовищем та інші елементи геймплею.

Об'єктом дослідження є процес створення двовимірних комп'ютерних ігор із дослідницькими та бойовими механіками, характерними для жанру Metroidvania.

Предметом дослідження є програмні засоби та алгоритмічні підходи до побудови систем керування персонажем, фізики ігрового світу, бойових механік, розвитку здібностей персонажа та взаємодії з об'єктами віртуального середовища.

КЛЮЧОВІ СЛОВА: 2D, UNITY, МЕТРОЇДВАНІЯ, НЕЛІНІЙНИЙ ІГРОВИЙ ПРОЦЕС, ПІКСЕЛЬНА ГРАФІКА.

					ІПЗ.КР.Б – 121 – 26 – ПЗ			
Змн.	Арк.	№ докум.	Підпис	Дата	2D гра жанру Metroidvania у стилі PixelArt: ігрові механіки Пояснювальна записка	Літ.	Арк.	Аркушів
Розроб.		Я. С. Можаровський						
Керівник		Н. М. Лобанчикова					3	75
Рецензент		Ю. М. Єфремов				Житомирська політехніка, група ІПЗ-22-2		
Зав. каф.		Т. А. Вакалюк						

ABSTRACT

The bachelor's qualification thesis consists of more than 140 pages, of which 71 pages are the main text, 34 figures, 7 tables, 5 appendixes, and 33 references.

The purpose of the work is to develop a 2D Metroidvania game in a PixelArt style with the implementation of core gameplay mechanics. The study examines the distinctive features of the Metroidvania genre and the capabilities of the Unity game engine. The development process is based on OOP principles, a component-based architecture, and physics simulation techniques. As a result, a game application has been created. The project includes character control mechanics, a combat system, character progression features, and interaction with the game environment. The developed solution may serve as a foundation for further development of two-dimensional games.

The object of study is the process of creating two-dimensional computer games with exploration and combat mechanics characteristic of the Metroidvania genre.

The subject of study is the software tools and algorithmic approaches to building character control systems, game world physics, combat mechanics, character ability progression, and interaction with virtual environment objects.

KEYWORDS: 2D, METROIDVANIA, NON-LINEAR GAMEPLAY, PIXELART, UNITY.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ.....	6
ВСТУП.....	7
РОЗДІЛ 1. АНАЛІЗ ПРИНЦИПІВ ПОБУДОВИ ІГРОВИХ МЕХАНІК ТА ПРОГРАМНИХ ЗАСОБІВ РОЗРОБКИ METROIDVANIA-ІГОР.....	10
1.1. Постановка задачі.....	10
1.2. Аналіз аналогів програмного продукту.....	12
1.3. Вибір архітектури системи.....	16
1.4. Обґрунтування вибору інструментальних засобів та вимоги до апаратного забезпечення.....	19
Висновки до першого розділу.....	22
РОЗДІЛ 2. ПРОЄКТУВАННЯ ТА РОЗРОБКА ІГРОВОГО ЗАСТОСУНКУ ЖАНРУ METROIDVANIA.....	23
2.1. Проєктування структури програмного продукту та варіантів використання.....	23
2.2. Розробка системи ігрових об'єктів та взаємодії між ними.....	31
2.3. Проєктування та реалізація основних ігрових механік.....	42
Висновки до другого розділу.....	49
РОЗДІЛ 3. ОПИС РОБОТИ ПРОГРАМНОГО ПРОДУКТУ, ЙОГО ВСТАНОВЛЕННЯ ТА ТЕСТУВАННЯ.....	51
3.1 Порядок встановлення та запуску програмного продукту.....	51
3.2 Структура інтерфейсу та особливості ігрового процесу.....	55
3.3. Тестування роботи програмного продукту.....	62
Висновки до третього розділу.....	68
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	73
ДОДАТКИ.....	76

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API (Application Programming Interface) – програмний інтерфейс взаємодії застосунків.

CPU (Central Processing Unit) – центральний процесор.

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

GUI (Graphical User Interface) – графічний інтерфейс користувача.

HUD (Heads-Up Display) – елементи інтерфейсу, що відображаються під час ігрового процесу.

IDE (Integrated Development Environment) – інтегроване середовище розробки.

PixelArt – стиль комп'ютерної графіки, побудований на використанні окремих пікселів як основних елементів зображення.

Sprite – двовимірне графічне зображення, що використовується як елемент ігрової сцени.

UI (User Interface) – інтерфейс користувача.

UX (User Experience) – користувацький досвід взаємодії з програмним продуктом.

ООП – об'єктно-орієнтоване програмування.

ОС – операційна система.

ПК – персональний комп'ютер.

ПЗ – програмне забезпечення.

ШІ – штучний інтелект.

ВСТУП

Актуальність теми. Цифровий розважальний сектор економіки на сьогодні демонструє найвищі темпи еволюції в межах індустрії інформаційно-комунікаційних технологій. Магістральним драйвером цієї трансформації виступає розширення і пришвидшення технічного забезпечення обчислювальних архітектур, що додатково до оптимізації архітектурних середовищ програмних платформ та невинним розвитком інструментарію девелопменту, створює сприятливий базис для інтенсифікації виробництва інтерактивного контенту з різною диференціацією щодо ресурсних витрат і масштабів реалізації [1]. Особливе місце серед них займають двовимірні ігри (або 2D), які завдяки поєднанню відносно невисоких вимог до апаратного забезпечення проєктувальників та користувачів, разом із широкими можливостями художнього оформлення залишаються популярними. Крім того у геймерів з досвідом, такі проєкти викликають теплі ностальгічні відчуття. Одним із напрямів розвитку 2D ігор є жанр *Metroidvania*, для якого характерними є дослідження великого взаємопов'язаного світу рівня чи рівнів, поступове відкриття нових підрівнів чи локацій при отриманні спец здібностей персонажем та поєднання ігроладу платформерів з більш просунутими бойових механік [2]. Популярність цього жанру пояснюється високим рівнем залучення гравця до процесу, заохочення до дослідження ігрового середовища, що забезпечує багаторазову реграбельність без відчуття одноманітності. Разом із цим створення подібних ігрових продуктів потребує реалізації складної системи взаємопов'язаних механік, які забезпечують природну і захопливу взаємодію персонажа з ігровим світом.

Важливою складовою сучасних ігор є не лише візуальне оформлення, а й якість програмної реалізації базових ігрових систем [3]. Від коректної роботи механізмів переміщення, бойової системи і системи шкоди, фізичних процесів, та розвитку персонажа безпосередньо залежить загальне враження гри для користувача. Саме тому дослідження підходів до побудови таких систем та їх практична реалізація засобами сучасних ігрових рушіїв є актуальним завданням.

Одним із найбільш поширених інструментів розробки комп'ютерних (ПК) ігор є рушій Unity, використання якого дозволяє поєднати інструменти візуального проєктування, фізичного моделювання, анімації та програмування в єдиному середовищі розробки [4], що робить його доцільним вибором для створення двовимірної гри жанру Metroidvania.

Метою кваліфікаційної роботи є розробка 2D-гри жанру Metroidvania у стилі PixelArt з особливими акцентами на реалізацією основних ігрових механік.

Завдання роботи:

- проаналізувати сучасні засоби розробки двовимірних ігор в цілому та особливості жанру Metroidvania;
- обґрунтувати вибір програмних засобів для реалізації проєкту;
- реалізувати систему керування персонажем та механіки переміщення;
- розробити бойову систему та механізми взаємодії персонажа з противниками;
- реалізувати систему очок здоров'я, отримання пошкоджень та використання здібностей;
- створити механізм поступового відкриття нових можливостей персонажа;
- забезпечити взаємодію користувача з об'єктами ігрового середовища;
- провести тестування працездатності створеної гри.

Об'єктом дослідження є процес розробки двовимірних ігор на ПК та інших платформах із механіками нелінійного проходження та взаємодії з об'єктами рівня.

Предметом дослідження є програмна реалізація систем керування персонажем, бойової взаємодії, фізичних процесів, розвитку здібностей та інтеракцій з елементами ігрового середовища у 2D-грі жанру Metroidvania.

Методи дослідження. У ході виконання кваліфікаційної роботи використано такі методи та засоби проєктування програмного забезпечення, як: аналіз джерел і подібних програмних рішень; принципи об'єктно-орієнтованого програмування (ООП); компонентний підхід до проєктування програмних систем;

засоби фізичного моделювання рушія Unity; а також окремі методи побудови ігрових алгоритмів і систем взаємодії між об'єктами.

Інформаційну базу дослідження становлять наукові публікації, технічна документація Unity, матеріали розробників ігор, спеціалізовані інформаційні ресурси та результати власних практичних досліджень у сфері розробки комп'ютерних ігор.

Практичне значення отриманих результатів полягає у створенні працездатного програмного застосунку, який може використовуватися як демонстраційний приклад реалізації ігрових механік у середовищі Unity, а також слугувати основою для подальшого розвитку повноцінного ігрового проєкту з дистрибуцією на цифрових майданчиках або навчальних розробок у сфері геймдеву.

Апробація. За темою кваліфікаційної роботи було опубліковано 1 тезу конференції: Можаровський Я.С., Лобанчикова Н.М., Концидайло А.М. Специфіка тестування ігрового балансу та механік у Metroidvania. Матеріали Всеукраїнської науково-практичної on-line конференції здобувачів вищої освіти і молодих учених «Тенденції розвитку технологій в автоматизації, електроенергетиці, приладобудуванні та робототехніці» (присвяченої Дню науки 11–15 травня 2026 року). – Житомир: Державний університет «Житомирська політехніка», 2026. – С. 111. [1].

РОЗДІЛ 1. АНАЛІЗ ПРИНЦИПІВ ПОБУДОВИ ІГРОВИХ МЕХАНІК ТА ПРОГРАМНИХ ЗАСОБІВ РОЗРОБКИ METROIDVANIA-ІГОР

1.1. Постановка задачі

Жанр Metroidvania залишається одним із найбільш популярних напрямів розвитку двовимірних ПК ігор. Його характерними особливостями є наявність взаємопов'язаного ігрового світу рівня чи рівнів, можливість нелінійного проходження локацій, поступове відкриття нових можливостей та здібностей персонажа та активне використання бойових механік. Створення програмного продукту такого типу потребує комплексного підходу до втілення у життя ігрових систем, які повинні функціонувати узгоджено та забезпечувати цілісний ігровий процес [5,6].

Цифровий розважальний сектор ігрової індустрії демонструє стрімке зростання, серед якого особливе місце посідають двовимірні інтерактивні проекти, що поєднують у собі складні механіки взаємодії та глибокий геймплей. Доступ до заблокованих локацій, таємних кімнат та альтернативних маршрутів відкривається гравцеві нелінійно – лише в міру знаходження специфічних предметів, розблокування унікальних фізичних здібностей або отримання нових видів озброєння головного героя.

Розробка програмного продукту такого класу є комплексною інженерно-програмною проблемою. Вона вимагає не просто створення окремих ізольованих ігрових систем, а побудови цілісної, масштабованої та відмовостійкої архітектури програмного забезпечення, в якій кожна підсистема функціонує в синергії з іншими в реальному часі під керуванням єдиного ігрового циклу. Будь-який розсинхрон чи архітектурна помилка на етапі проектування взаємодії між модулем введення, фізичним рушієм та логікою станів персонажа здатна критично погіршити користувацький досвід (UX).

Об'єктом дослідження у межах цієї кваліфікаційної роботи є процес програмної розробки двовимірної комп'ютерної гри з механіками нелінійного проходження, платформінгу, дослідження лабіринтоподібних рівнів та

динамічної бойової системи. Предметом дослідження виступають алгоритмічні підходи, патерни проєктування та спеціалізовані програмні засоби, за допомогою яких реалізується система керування персонажем, машина станів, бойова система, стихійні та магнітні здібності, а також логіка штучного інтелекту противників у грі жанру Metroidvania.

Метою роботи є проєктування, програмна реалізація, оптимізація та тестування застосунку «Fiat Lux» у стилі PixelArt, з особливим фокусом на створення гнучких, чистих та модульних ігрових механік.

Для досягнення поставленої мети необхідно послідовно вирішити такі завдання:

1. Провести аналіз особливостей жанру Metroidvania та обґрунтувати вибір ігрового рушія Unity як платформи реалізації проєкту.
2. Спроектувати загальну архітектуру гри та реалізувати систему керування персонажем із підтримкою різних пристроїв введення, механіками переміщення (ходьба, стрибки, ривки) та машиною станів.
3. Розробити бойову систему з підтримкою атак у різних напрямках, механізмами отримання пошкоджень та поведінкою противників на основі моделі станів.
4. Реалізувати систему здоров'я, ресурсів та прогресії персонажа через поступове відкриття нових здібностей і можливостей взаємодії з об'єктами ігрового середовища.
5. Провести тестування розробленого програмного продукту та оцінити відповідність реалізованих можливостей визначеним цілям.

Результатом виконання зазначених завдань має стати програмний продукт у вигляді 2D-гри, з втіленими ігровими механіками типовими для жанру Metroidvania та оформленням у стилі піксельної графіки. Реалізовані рішення повинні забезпечувати можливість подальшого розширення функціональних можливостей гри, додавання нових рівнів та локацій, противників, здібностей та ігрових сценаріїв на фундаменті існуючої архітектури програмного забезпечення (ПЗ).

1.2. Аналіз аналогів програмного продукту

Перед початком безпосередньо процесу розробки програмного продукту раціонально провести аналіз існуючих представників жанру Metroidvania та з піксельною графікою. Вивчення аналогічних рішень дозволяє визначити набір основних функціональних можливостей, які користувачі очікуватимуть від подібних ігор, а також виявити підходи до реалізації механік дослідження світу, бойової взаємодії та розвитку персонажа. Крім того, можна навчитися на помилках чи гарно втіленому кодові застосунку. Для даного аналізу у межах роботи було обрано три популярні проєкти, що належать до жанру Metroidvania та мають значну популярність серед гравців, зокрема Hollow Knight, Nine Sols та Blasphemous. Зазначені ігри демонструють різні підходи до побудови ігрового світу, можливостей головного героя, балансу, художнього оформлення та підходів до втілення взаємодії користувача з віртуальним середовищем.

Hollow Knight (рис. 1.1) є однією з найбільш відомих та впливових ігор жанру Metroidvania, розробленою незалежною студією Team Cherry і випущеною у 2017 році. Гра розгортається у підземному королівстві комах Халлоунест, де гравець керує безіменним лицарем, що досліджує занедбані руїни, населені зараженими істотами. Основою ігрового процесу виступає дослідження великого взаємопов'язаного світу, розділеного на окремі тематичні регіони — від засіяних грибами лісів до кристалічних шахт і давніх царських залів. Доступ до нових територій відкривається після отримання спеціальних здібностей, таких як ривок, вистріл магією, подвійний стрибок, удар об землю або можливість пересування по вертикальних поверхнях. Бойова система гри побудована на використанні ближніх атак цвяхом та магічних спец здібностей, які працюють у чотирьох напрямках. Значна увага приділяється точності керування персонажем та швидкості реакції гравця. Система «чарів» (Charms) надає гравцеві широкі можливості кастомізації: невеличкі талісмани, що займають певну кількість ячок, кардинально змінюють стиль бою, стрибків чи відновлення душі. Механіка збору «душі» під час бою та витрачання її на зцілення або магічні атаки формує

динамічний цикл ризику й винагороди, що є одним із ключових елементів геймплею. Бос-файти є особливо відзначуваним елементом гри — кожен із десятків босів має унікальні паттерни поведінки та вимагає від гравця адаптації тактики. Наративна подача реалізована через фрагментарне дослідження: діалоги персонажів, описи предметів і натяки в архітектурі локацій поступово складаються в цілісну картину трагічної історії цивілізації комах.

Перевагами гри є: велика кількість секретних зон з різноманітними «смаколиками», гнучке налаштування персонажа під свій стиль гри та, збалансована крива складності гри. Разом із тим високий рівень складності окремих елементів може ускладнювати адаптацію нових користувачів до ігрового процесу.

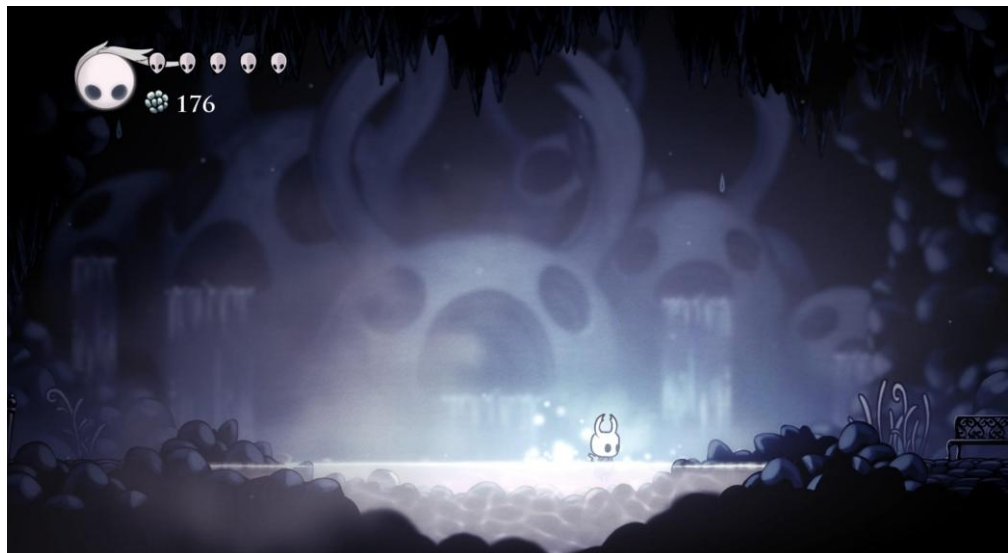


Рис. 1.1 – Фрагмент ігроладу гри Hollow Knight[7]

Nine Sols (рис. 1.2) поєднує класичне дослідження взаємопов’язаного світу Metroidvania із динамічною бойовою системою, заснованою на механіці відбиття атак, в унікальному візуальному стилі «таопунку». Ігровий процес вимагає високої точності керування та вивчення таймінгів ворогів: успішне відбиття атак накопичує енергію, яку гравець використовує для потужних контратак за допомогою вибухових талісманів. Дослідження локацій розвивається поступово через розблокування нових здібностей пересування, таких як: ривки, подвійні стрибки, біг по стінах, гак-кішка, що дозволяє долати заплутані платформінгові зони. Система розвитку персонажа реалізована через дерево навичок у

центральному хабі, де покращуються бойові комбінації, характеристики здоров'я та ефективність зброї.

Перевагами гри є надзвичайно чуйне і точне керування, унікальний художній стиль ручної анімації та бойова система, яка переосмислює стандартні для жанру Metroidvania сутички. Недоліком можна вважати надто жорстке прив'язування успіху гравця до однієї конкретної механіки – відбиття атак, що практично нівелює альтернативні стилі ведення бою та може викликати відчуття монотонності під час тривалого проходження.



Рис. 1.2 – Фрагмент ігрового ладу гри Nine Sols[8]

lasphemous (рис. 1.3) є представником жанру Metroidvania з підвищеною увагою до бойової взаємодії та високої складності проходження в атмосфері похмурого релігійного лору. Ігровий процес базується на виснажливих зіткненнях, використанні комбінацій атак меча, таймінгах для парирування та смертельних добиваннях ворогів, де кожен тип супротивників вимагає індивідуального тактичного підходу. Проторольова система розвитку реалізована через пошук предметів розарію, молитов-заклинань та серцевин зброї, які змінюють характеристики героя, тоді як просування світом часто обмежується не фізичними навичками, а наявністю реліквій, що відкривають приховані шляхи, матеріалізують нові платформи чи захищають від смертоносних пасток навколишнього середовища.

Перевагою гри є детально пропрацьована, безкомпромісна бойова система, високий рівень реграбельності завдяки налаштуванню пасивних бонусів, а також глибоке, атмосферне художнє оформлення у стилі похмурого PixelArt. Недоліком можна вважати високий поріг входження для недосвідчених користувачів, затягнутий бектрекінг через специфіку навігації та відчутні штрафи за смерть персонажа.

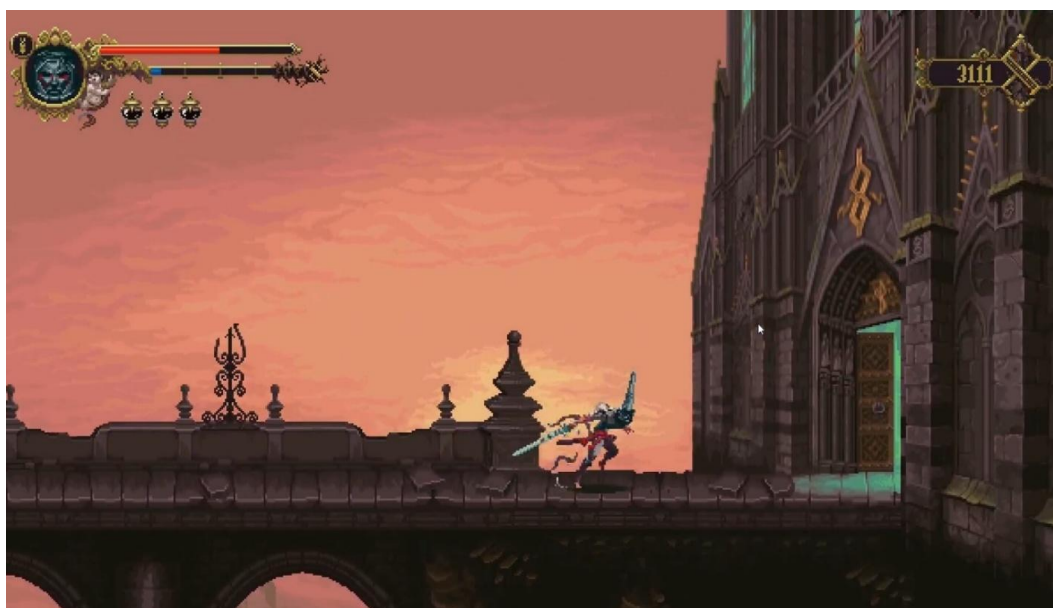


Рис. 1.3 – Фрагмент ігрового ладу гри Blasphemous[9]

Для порівняння функціональних можливостей проаналізованих аналогів складено табл. 1.1.

Таблиця 1.1

Порівняльна характеристика аналогів

Функція	Hollow Knight	Nine Sols	Blasphemous	Власна розробка
Дослідження світу	+	+	+	+
Система здібностей	+	+	+	+
Бойова система	+	+	+	+
Прогресія персонажа	+	+	+	+
Взаємодія з об'єктами	+	+	±	+
Маніпуляція фізичними об'єктами	—	±	—	+
PixelArt-оформлення	±	—	+	+

Проведений аналіз показав, що всі розглянуті ігри використовують характерні для жанру механіки дослідження світу та поступового відкриття нових можливостей персонажа. Водночас кожен проєкт робить акцент на окремих аспектах ігрового процесу: Hollow Knight – на дослідженні світу, Ori and the Blind Forest – на мобільності персонажа та платформерному ігровому процесу, а Blasphemous – на бойовій взаємодії.

На відміну від розглянутих аналогів, у межах кваліфікаційної роботи передбачається реалізація власної системи взаємодії з фізичними об'єктами, що доповнює традиційні механіки жанру Metroidvania. Крім того, розроблюваний програмний продукт поєднуватиме систему розвитку здібностей, дослідження локацій та бойову взаємодію в межах єдиної програмної архітектури з можливостями для подальшого розширення функціоналу для персонажа та рівнів.

1.3. Вибір архітектури системи

Одним із ключових етапів у створенні ігрового програмного продукту є ідентифікація архітектури ПЗ. Від прийнятих рішень залежить зручність підтримки гри, можливість зміни та подальшого розширення функціоналу гри, а також ефективність взаємодії між окремими елементами ігрової системи. При втіленні розробки найрізноманітніших програмних рішень використовуються різні архітектурні підходи, серед яких найбільш поширеними є монолітна, багаторівнева та компонентно-орієнтована архітектури [10]. Монолітний підхід передбачає розміщення всієї логіки програми в межах невеликої кількості жорстко пов'язаних модулів. Перевагою такого рішення є простота реалізації, однак при ускладненні ПЗ та при збільшенні кількості функцій підтримка та зміна коду значно ускладнюються, після певного рівня змін, нерідко легше почати розробку з нуля, обравши інший тип архітектури. Багаторівнева архітектура широко застосовується у веб-орієнтованих системах та корпоративних програмних продуктах. Вона забезпечує чітке розмежування представлення, обробки та їх зберігання даних. Проте для 2D-гри використання такого підходу не є доцільним через особливості жанру для організації даних та відсутність

необхідності в складній багаторівневій структурі [11]. Таким чином, методом виключення, для реалізації ігрового застосунку обрано компонентно-орієнтовану архітектуру, яка є базовим підходом до побудови застосунків у середовищі Unity [12]. Основна ідея полягає у формуванні поведінки об'єкта шляхом поєднання незалежних компонентів, кожен із яких відповідає за окрему функціональність. На рис. 1.4 наведено загальну структуру обраної архітектури адаптовану до розроблюваного проєкту.

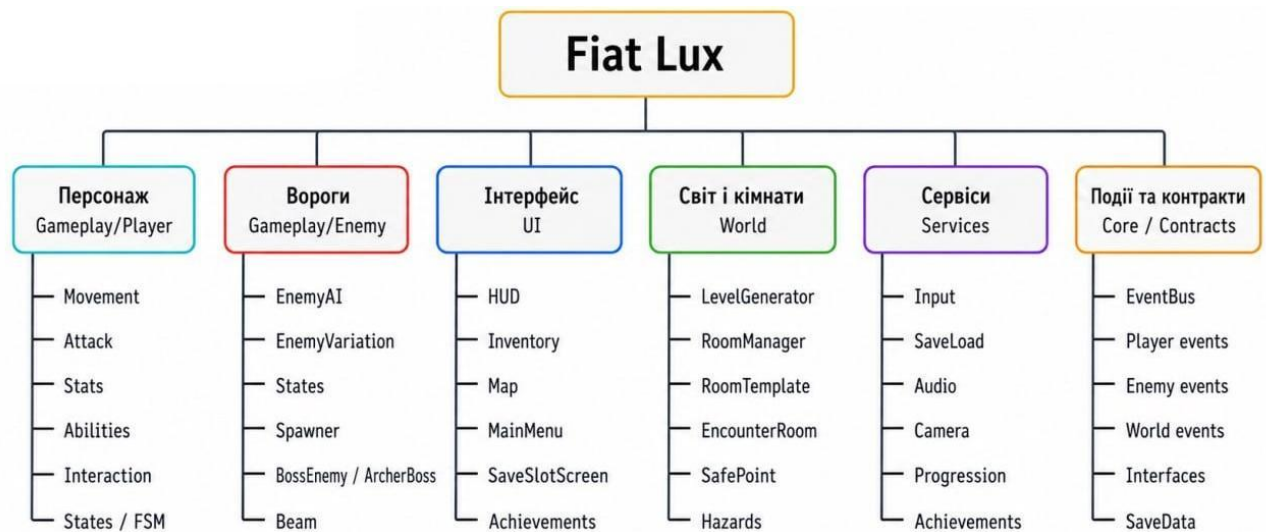


Рис. 1.4 – Загальна структура програмного продукту

Обраний тип архітектури передбачає побудову на основі взаємодії декількох незалежних компонентів, кожен із яких відповідає за виконання окремого набору функцій. Центральне місце в системі займає модуль керування персонажем, який забезпечує обробку ввідних команд користувача та взаємодій основних систем гри. Коректність переміщення та взаємодії об'єктів забезпечується модулем фізичної взаємодії, що працює спільно з бойовою системою та механізмами отримання пошкоджень. Окремі компоненти відповідають за контроль стану персонажа, облік здоров'я та використання спец здібностей, які відкриваються при знаходженні на мапі відповідних посилень. Для взаємодії з елементами ігрового середовища передбачено спец модуль, що забезпечує роботу з інтерактивними об'єктами та ігровими подіями. Поведінка противників реалізується окремою підсистемою (модулем), яка має скриптовані

дії при появі персонажа у його зоні активації. Додатково в архітектурі передбачено модуль GUI, що забезпечує відображення інформації про стан здоров'я персонажа, енергія для використання доступних здібностей, їх статус розблокування та інші елементи ігрового процесу [16].

Кожен із зазначених модулів виконує власний набір функцій та взаємодіє з іншими компонентами через визначені програмні зв'язки. Такий підхід дозволяє уникнути надмірної залежності між частинами системи та значно спрощує внесення змін у процесі подальшого розвитку проєкту. У розроблюваній грі основна логіка персонажа розподілена між декількома незалежними компонентами. Зокрема, окремі класи відповідають за пересування персонажа, обробку атак, систему здоров'я, отримання пошкоджень і використання здібностей. Аналогічний підхід використовується і для інших об'єктів ігрового світу. На рисунку 1.5 наведено взаємодію основних компонентів персонажа.

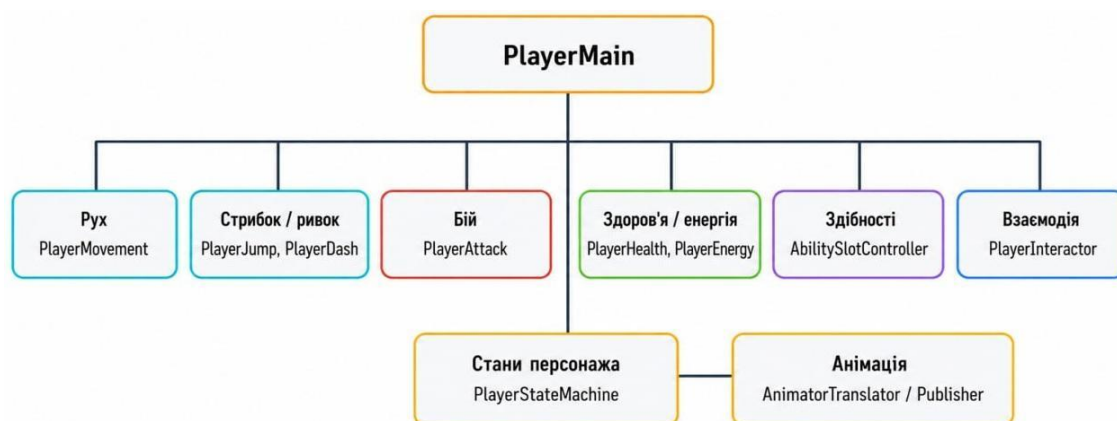


Рис. 1.5 – Взаємодія компонентів персонажа

Така структура забезпечує високу flexibility системи [17]. До прикладу, додавання нової здібності або нового типу противника не потребує зміни вже реалізованих механізмів пересування чи бойової взаємодії, а потребує лише налаштувань самого противника. У більшості випадків достатньо створити новий програмний компонент та підключити його до існуючої структури. Ще однією перевагою обраної архітектури є можливість повторного використання окремих модулів. Реалізовані компоненти можуть бути використані в інших проєктах без істотних змін програмного коду, що позитивно впливає на швидкість подальшої

розробки. Отже, для реалізації двовимірної гри жанру Metroidvania обрано компонентно-орієнтовану архітектуру, яка найбільш повно відповідає особливостям рушія Unity, який планується використовувати, та забезпечує модульність, масштабованість і зручність підтримки програмного забезпечення.

1.4. Обґрунтування вибору інструментальних засобів та вимоги до апаратного забезпечення

Ефективність розробки будь якого достатньо великого проєкту великою мірою залежить від правильного вибору інструментальних засобів. Для створення сучасної гри необхідно забезпечити можливість реалізації графічної складової, фізичної взаємодії об'єктів, обробки ввідних дій користувача, керування анімаціями та можливості удосконалення і розширення у майбутньому. Тому під час виконання роботи особлива увага приділялася вибору програмних засобів, які забезпечують необхідний набір можливостей та відповідають поставленим завданням.

Для реалізації програмного продукту обрано ігровий рушій Unity [15-17]. Даний рушій належить до найбільш поширених платформ розробки комп'ютерних і консольних ігор та інтерактивних застосунків. Однією з його переваг є повна підтримка двовимірної графіки, фізичного моделювання практично будь-якої, анімаційних систем та сучасних засобів програмування. Використання Unity дозволяє організувати процес розробки в межах єдиного програмного середовища та забезпечує можливість подальшого перенесення проєкту на різні програмні платформи [18].

Рушій має потужну вбудовану підсистему 2D Core, яка включає компонентні інструменти Sprite Editor, Sprite Atlas (для оптимізації текстурної пам'яті та зменшення викликів отрисовки) та 2D Pixel Perfect, що є критично важливим для збереження автентичності та візуальної чіткості стилю PixelArt без розмиття пікселів при рендерингу в різних роздільних здатностях екрана. Інтегрований фізичний рушій Unity Physics 2D забезпечує високу точність та продуктивність обчислень при роботі з двовимірними колайдерами та

динамічними твердими тілами. Це дозволило безшовно реалізувати такі складні фізичні механіки проєкту, як інерція руху, гравітаційне прискорення падіння, детекція заземлення, пружний відскок персонажа при атаці вниз та імпульсний вплив магнітних сил на об'єкти в ігровому просторі.

Як основну мову програмування використано C#. Вибір цієї мови обумовлений глибокою її інтеграцією з Unity, підтримкою об'єктно-орієнтованого підходу та широкими можливостями побудови складних програмних систем [19]. Використання C# дозволяє створювати модульний програмний код, розділяти логіку окремих механік та спрощує підтримку проєкту в процесі його розвитку [20]. Використання мови C# дозволило створити строгу типізацію об'єктів, реалізувати подієво-орієнтовану модель зв'язку через делегати та події, гнучко керувати колекціями даних за допомогою LINQ, а також розмежувати бізнес-логіку механік від безпосереднього відображення інтерфейсу.

Для написання та налагодження програмного коду використано середовище розробки Visual Studio Code версії 1.123.0. Воно забезпечує підсвічування синтаксису, автоматичне завершення коду, засоби пошуку помилок та інструменти відлагодження, що позитивно впливає на швидкість і якість розробки [21]. VS Code забезпечує інтелектуальне автодоповнення коду IntelliSense, миттєву навігацію за структурою класів, а також пряме підключення дебаггера до потоку Unity Editor, що дозволяє відслідковувати значення змінних у реальному часі під час геймплейних сесій.

Графічне оформлення гри створюється у стилі піксель арт. Такий підхід дозволяє досягти виразного художнього вигляду без використання складних тривимірних моделей та високих вимог до апаратних ресурсів. Крім того, піксельна графіка добре поєднується з жанром Metroidvania та сприяє створенню цілісного візуального стилю проєкту і виглядає стильно і вінтажно [22].

Для реалізації анімацій використовуються вбудовані засоби Unity Animator, що забезпечують плавне відтворення рухів персонажа, бойових дій та взаємодії з навколишнім середовищем [23]. Використання стандартних механізмів рушія

дозволяє зменшити складність програмної реалізації та забезпечити підвищену сумісність між окремими компонентами системи.

Для систематизації та наочного представлення обраного технологічного стеку проєкту, перелік та цільове призначення всіх використаних утиліт було зведено в єдину структуру, яку демонструє табл. 1.2.

Таблиця 1.2

Інструментальні засоби розробки

Засіб	Призначення
Рушій Unity	Розробка ігрового застосунку
Мова C#	Реалізація програмної логіки
Visual Studio Code	Написання та налагодження коду
Unity Animator	Керування анімаціями
Unity Physics 2D	Фізична взаємодія об'єктів
Git	Контроль версій проєкту

Оскільки програмний продукт є двовимірною грою з використанням фізичного моделювання, анімації та систем штучної поведінки ігрових об'єктів, його функціональна спроможність безпосередньо детермінована рівнем ресурсної бази, що має бути достатньою для виконання покладених на неї обчислювальних завдань [24]. Для контролю версій проєкту використовувався широкий інструментарій Git [25]. Для стабільного запуску гри, відсутності фризів під час динамічного підвантаження кімнат та забезпечення плавного ігрового процесу, обчислювальний пристрій гравця має відповідати набору параметрів, які містить табл. 1.3.

Таблиця 1.3

Рекомендовані вимоги до апаратного забезпечення користувача

Компонент	Рекомендовані вимоги
Операційна система	Windows 11 (64-bit)
Процесор (CPU)	Intel Core i5 і більш сучасні
Оперативна пам'ять	8 ГБ
Відеоадаптер	NVIDIA GeForce GTX 1050 або і більш сучасний
Вільне місце на диску	256 МБ
Роздільна здатність екрана	1920×1080

Відтак, для архітектурної реалізації програмного комплексу було

детерміновано стековий набір інструментальних засобів, що забезпечує оптимізацію циклів розробки, верифікації та ітеративного вдосконалення ігрового контенту. Інтеграція ігрового рушія Unity, синтаксису мови C# та спеціалізованих середовищ для налагодження коду створює надійний базис для імплементації складних ігрових механік. При цьому обрана конфігурація апаратних засобів гарантує стабільність функціонування продукту в межах стандартизованих обчислювальних параметрів, характерних для більшості актуальних архітектур персональних комп'ютерів.

Висновки до першого розділу

У першому розділі проведено дослідження особливостей жанру Metroidvania та визначено основні вимоги до розроблюваної гри. Сформульовано необхідні до виконання задачі, встановлено перелік функціональних можливостей майбутньої гри та окреслено основні механіки, необхідні для реалізації повноцінного ігрового процесу.

У ході аналізу сучасних аналогів було розглянуто ігри Hollow Knight, Ori and the Blind Forest та Blasphemous. Проведене порівняння дозволило визначити характерні риси жанру, серед яких дослідження взаємопов'язаного світу, поступове відкриття нових здібностей персонажа, наявність бойової системи та елементів прогресії. Отримані результати стали основою для формування функціональних вимог до розробки запланованої гри.

На основі аналізу існуючих архітектурних підходів було обґрунтовано вибір компонентно-орієнтованої архітектури, що найбільш повно корелює з особливостям ігрового рушія Unity та забезпечує модульність, масштабованість і зручність подальшої підтримки ПЗ. Також виконано обґрунтування вибору інструментальних засобів розробки та визначено вимоги до апаратного забезпечення гравця.

Отримані результати створюють необхідну теоретичну та технологічну базу для подальшого проєктування структури ігрового застосунку, розробки основних ігрових механік та реалізації програмного застосунку в наступних розділах роботи.

РОЗДІЛ 2. ПРОЄКТУВАННЯ ТА РОЗРОБКА ІГРОВОГО ЗАСТОСУНКУ ЖАНРУ METROIDVANIA

2.1. Проєктування структури програмного продукту та варіантів використання

Після визначення функціональних вимог до програмного продукту наступним етапом є проєктування його загальної структури та визначення способів взаємодії користувача з основними модулями гри. На цьому етапі формується уявлення про склад застосунку, взаємозв'язки між окремими підсистемами, відповідальність кожного програмного компонента та перелік дій, які можуть бути виконані під час ігрового процесу. Для гри жанру Metroidvania особливо важливим є правильне виділення функціональних блоків, оскільки ігровий процес поєднує керування персонажем, дослідження рівнів, бойову взаємодію, використання здібностей, прогресію та поступове відкриття нових зон.

Розроблюваний програмний продукт являє собою двовимірну гру жанру Metroidvania під назвою Fiat Lux. У межах ігрового процесу гравець керує персонажем, переміщується локаціями, долає перешкоди, взаємодіє з об'єктами середовища, використовує зброю та здібності, вступає в бій із ворогами та поступово відкриває нові можливості. Особливістю жанру є те, що частина ігрового світу на початкових етапах залишається недоступною. Доступ до таких ділянок відкривається після отримання нової здібності, зброї або іншого елемента прогресії [26]. Таким чином, структура програмного продукту має підтримувати не лише базові механіки руху й бою, а й систему умовного доступу до об'єктів і зон.

На етапі проєктування було визначено, що архітектура застосунку повинна бути модульною. Це означає, що кожна основна механіка реалізується у вигляді окремої підсистеми, яка має власну відповідальність і взаємодіє з іншими підсистемами через чітко визначені зв'язки. Такий підхід дає змогу змінювати або розширювати окремі частини гри без необхідності суттєво переробляти всю програмну структуру. Наприклад, додавання нової зброї не повинно вимагати

зміни логіки руху персонажа, а створення нового типу ворога не повинно впливати на роботу системи здібностей.

До основних підсистем програмного продукту належать: підсистема керування персонажем, підсистема станів персонажа, бойова підсистема, підсистема здібностей, підсистема ворогів, підсистема статусних ефектів, підсистема інтерактивних об'єктів світу, підсистема прогресії та підсистема користувацького інтерфейсу. Кожна з них виконує окрему роль у забезпеченні ігрового процесу. У сукупності ці підсистеми формують цілісну архітектуру гри, у якій дії користувача перетворюються на зміну стану персонажа, ворогів, об'єктів світу та загального прогресу проходження.

Підсистема керування персонажем відповідає за обробку введення користувача та перетворення команд на конкретні дії ігрового об'єкта. До таких дій належать рух ліворуч і праворуч, стрибок, ривок, атака, використання здібностей та взаємодія з предметами. Ця підсистема є однією з центральних, оскільки саме через неї гравець безпосередньо впливає на стан гри. Вона повинна забезпечувати швидку реакцію на введення, коректну зміну напрямку руху, перевірку доступності певних дій і передачу команд до інших компонентів.

Підсистема станів персонажа використовується для організації поведінки гравця залежно від поточної ситуації. Персонаж може перебувати у стані простою, бігу, стрибка, падіння, ривка, атаки, лікування або смерті. Використання машини станів дозволяє уникнути надмірної кількості умов у одному класі та розділити логіку кожної дії на окремі компоненти. Наприклад, поведінка персонажа під час стрибка відрізняється від поведінки під час ривка чи атаки, тому кожен стан має власні правила входу, оновлення та завершення [27].

Бойова підсистема забезпечує нанесення пошкоджень, вибір активної зброї, визначення напрямку атаки, перевірку потрапляння по ворогу та застосування відкидання. У грі передбачено використання зброї ближнього та дальнього бою, що потребує різних способів обробки атаки. Для ближнього бою важливо визначити зону удару навколо персонажа, а для дальнього – створити снаряд, який рухається у заданому напрямку. Бойова підсистема також взаємодіє з

підсистемою енергії, прогресії та ворогів, оскільки результат атаки може впливати на стан противника й характеристики персонажа.

Підсистема здібностей відповідає за використання спеціальних дій, що відкриваються в процесі проходження. До таких здібностей належать вогонь, лід, притягування та відштовхування металевих об'єктів. Здібності виконують подвійну роль: з одного боку, вони можуть застосовуватися у бойових ситуаціях, а з іншого – використовуються для взаємодії з об'єктами світу та відкриття нових шляхів. Наприклад, вогонь може впливати на об'єкти, що горять, лід – заморожувати воду або цілі, а магнітні здібності – змінювати положення металевих об'єктів.

Підсистема ворогів відповідає за створення та керування поведінкою противників. Вороги мають власні параметри здоров'я, пошкодження, швидкості руху, зони виявлення гравця та логіку атаки. Для реалізації поведінки ворогів використовується поділ на стани, такі як патрулювання, пошук, переслідування та атака. Це дозволяє задавати для кожного типу противника окрему модель поведінки, не змінюючи загальні принципи роботи системи. Наприклад, ближній ворог намагається наблизитися до персонажа, дальній – атакує з відстані, а літаючий може використовувати простір над платформами.

Підсистема статусних ефектів забезпечує тимчасовий вплив на об'єкти гри. До таких ефектів належать горіння, замороження та інші дії, що змінюють поведінку або стан цілі. Винесення статусів в окрему підсистему дозволяє застосовувати однакові ефекти до різних об'єктів: ворогів, перешкод або інших інтерактивних елементів. Наприклад, ефект горіння може завдавати пошкодження протягом часу, а замороження може обмежувати рух цілі або змінювати її взаємодію з іншими системами.

Підсистема інтерактивних об'єктів світу включає об'єкти, з якими гравець може взаємодіяти під час проходження. До них належать небезпечні зони, контрольні точки, предмети розблокування, перешкоди, вода та металеві об'єкти. Небезпечні зони можуть повертати персонажа до останньої безпечної точки або

відкидати його назад [28]. Контрольні точки зберігають позицію для відродження. Предмети розблокування відкривають нову зброю або здібності. Саме ця підсистема підтримує характерну для Metroidvania структуру світу, у якій доступ до певних ділянок залежить від уже отриманих можливостей.

Підсистема прогресії відповідає за збереження інформації про відкриту зброю, отримані здібності та покращені характеристики персонажа. У межах гри прогресія проявляється не лише у збільшенні бойових можливостей, а й у розширенні доступу до нових ділянок рівня. Це означає, що прогресія безпосередньо пов'язана з навігацією, бойовою системою та інтерактивними об'єктами. Завдяки цьому гравець поступово отримує нові інструменти для подолання перешкод і повернення до раніше недоступних зон.

Підсистема користувацького інтерфейсу забезпечує відображення важливої інформації для гравця. До неї належать індикатори здоров'я, енергії, доступної зброї, активних здібностей, повідомлення про отримання нових предметів і стан персонажа. Інтерфейс не повинен перевантажувати екран, але має своєчасно інформувати користувача про критичні зміни, наприклад втрату здоров'я, нестачу енергії або відкриття нової можливості.

Для визначення функціональних можливостей програмного продукту було виконано моделювання варіантів використання. Діаграма варіантів використання дозволяє описати взаємодію між акторами та основними функціями системи на високому рівні. У межах розроблюваної гри основним актором є гравець. Також у моделі можна виділити допоміжних акторів, які не є користувачами у звичайному розумінні, але беруть участь у формуванні ігрового процесу: ворогів, об'єкти світу та ігрову систему. Гравець ініціює основні дії, вороги реагують на його появу та атаки, об'єкти світу змінюють стан під впливом здібностей, а ігрова система забезпечує виконання правил, перевірку умов і зміну стану сцени.

На діаграмі варіантів використання гравець пов'язаний із такими основними сценаріями, як керування персонажем, переміщення, стрибок, ривок, атака, використання здібностей, взаємодія з об'єктами, отримання пошкоджень, смерть і відродження, а також відкриття нових можливостей. Частина цих сценаріїв є

базовими та доступними з початку гри, а частина активується лише після отримання відповідних предметів або здібностей. Наприклад, рух і стрибок є базовими діями, тоді як використання певної стихійної або магнітної здібності залежить від прогресу персонажа.

Окрему групу варіантів використання становлять бойові сценарії. До них належать атака ближньою зброєю, атака дальньою зброєю, визначення напрямку атаки, нанесення пошкодження, відкидання цілі та знищення ворога. Ці сценарії можуть бути пов'язані між собою відношеннями включення, оскільки виконання атаки передбачає перевірку напрямку, визначення цілі та застосування результату. Якщо атака досягає ворога, система змінює його запас здоров'я, а в разі вичерпання здоров'я переводить ворога у стан смерті.

Група сценаріїв, пов'язаних зі здібностями, включає використання вогню, льоду, притягування та відштовхування металевих об'єктів. Кожна здібність передбачає перевірку запасу енергії та доступності використання. Якщо енергії достатньо, система активує відповідний ефект. Якщо ресурс відсутній, дія не виконується. Такі сценарії демонструють зв'язок між підсистемою здібностей, підсистемою енергії та об'єктами світу.

Сценарії прогресії описують отримання нових можливостей у процесі проходження. Гравець може підбирати предмети, які відкривають зброю, здібності або покращення характеристик. Після цього система оновлює стан прогресу персонажа, а нові можливості стають доступними для подальшого використання. Ці сценарії мають важливе значення для жанру Metroidvania, оскільки саме вони забезпечують поступове відкриття світу та мотивують гравця повертатися до раніше відвіданих ділянок.

Сценарії взаємодії зі світом включають потрапляння в небезпечну зону, активацію контрольної точки, взаємодію з об'єктами та зміну стану середовища під впливом здібностей. Наприклад, якщо персонаж потрапляє в небезпечну зону, система може повернути його до останньої безпечної точки. Якщо гравець активує контрольну точку, вона зберігається як позиція для подальшого відродження. Якщо персонаж застосовує лід до води або магнітну здібність до металевого

об'єкта, відповідний об'єкт змінює свій стан і впливає на подальше проходження.

Для більш детального подання функціональних можливостей програмного продукту доцільно подати опис основних варіантів використання у вигляді таблиці.

Таблиця 2.1

Опис основних варіантів використання програмного продукту

Назва варіанту використання	Актор	Передумова	Основний сценарій	Результат
Керувати персонажем	Гравець	Ігрова сцена запущена	Гравець натискає клавіші руху, система змінює напрям і швидкість персонажа	Персонаж переміщується рівнем
Виконати стрибок	Гравець	Персонаж перебуває на землі	Гравець натискає кнопку стрибка, система перевіряє стан персонажа та застосовує імпульс	Персонаж переходить у стан стрибка
Виконати ривок	Гравець	Ривок доступний	Гравець натискає кнопку ривка, система перевіряє cooldown і виконує швидке переміщення	Персонаж швидко зміщується в обраному напрямку
Атакувати ворога	Гравець	Активна зброя доступна	Гравець натискає кнопку атаки, система визначає напрям, активну зброю та ціль	Ворог отримує пошкодження
Використати здібність	Гравець	Здібність розблокована, енергії достатньо	Гравець активує здібність, система витрачає енергію та застосовує ефект	Ціль або об'єкт світу змінює стан
Отримати пошкодження	Ворог / небезпечна зона	Персонаж потрапляє під атаку або в небезпечну зону	Система зменшує здоров'я персонажа та перевіряє його стан	Персонаж продовжує гру або переходить у стан смерті
Відродитися	Ігрова система	Здоров'я персонажа вичерпано	Система визначає останню безпечну точку та переносить персонажа	Персонаж повертається до гри

Відкрити нову здібність	Гравець	Предмет розблокування доступний	Гравець взаємодіє з предметом, система оновлює прогрес	Нова здібність стає доступною
Відкрити нову зброю	Гравець	Предмет зброї доступний	Гравець підбирає предмет, система додає зброю до доступних	Гравець може використовувати нову зброю
Активувати контрольну точку	Гравець	Персонаж досягає SafePoint	Система запам'ятовує контрольну точку	SafePoint стає точкою повернення
Взаємодіяти з об'єктом світу	Гравець	Об'єкт має відповідний інтерфейс взаємодії	Гравець застосовує дію або здібність до об'єкта	Об'єкт змінює стан
Патрулювати територію	Ворог	Ворог активний у сцені	Ворог рухається заданою ділянкою та перевіряє наявність гравця	Ворог залишається в патрулі або переходить до переслідування
Переслідувати гравця	Ворог	Гравця виявлено	Ворог змінює стан і рухається до гравця	Ворог скорочує дистанцію до цілі
Атакувати гравця	Ворог	Гравець перебуває в зоні атаки	Ворог виконує атаку	Гравець отримує пошкодження

Подані варіанти використання показують, що більшість дій у грі пов'язана з безпосередньою взаємодією гравця з персонажем, ворогами та об'єктами середовища. При цьому значна частина сценаріїв має умови виконання. Наприклад, атака потребує наявності активної зброї, використання здібності – достатньої кількості енергії, а доступ до деяких зон – попереднього відкриття відповідної можливості. Це підтверджує необхідність використання модульної структури, у якій кожна підсистема відповідає за власну частину перевірок і змін стану.

На рис. 2.1 подано діаграму варіантів використання (англ. Use Case Diagram) програмного продукту Fiat Lux. Вона відображає основні функціональні

можливості гри та зв'язки між діями гравця, ворогів, об'єктів світу й ігрової системи.

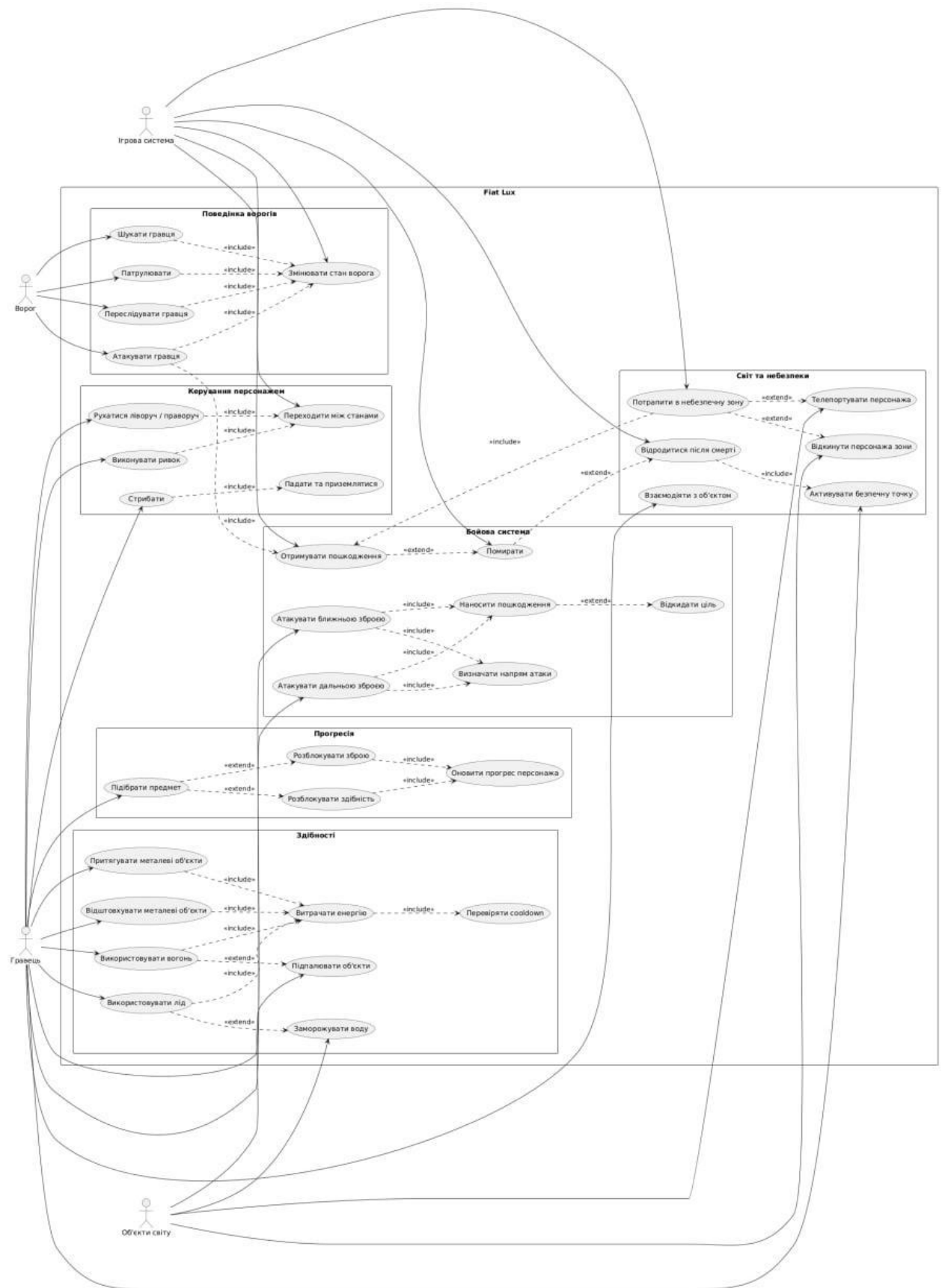


Рис. 2.1 – Діаграма варіантів використання гри Fiat Lux

Аналіз діаграми варіантів використання показує, що центральне місце в системі займає персонаж гравця. Через нього користувач взаємодіє з більшістю інших компонентів застосунку. Усі основні дії гравця або змінюють стан персонажа, або ініціюють взаємодію з іншими об'єктами. Наприклад, команда атаки впливає на ворога, команда використання здібності може впливати на об'єкт світу, а потрапляння в небезпечну зону змінює позицію персонажа або його запас здоров'я.

Важливою особливістю моделі є наявність умовних сценаріїв. Частина дій не може бути виконана без попереднього виконання інших сценаріїв. Наприклад, використання певної здібності можливе лише після її розблокування, а перехід до деяких ділянок рівня залежить від наявності відповідного інструмента взаємодії. Такі зв'язки добре відповідають жанру *Metroidvania*, у якому прогрес гравця визначається не лише проходженням рівня вперед, а й поверненням до раніше недоступних зон після отримання нових можливостей.

Отже, у межах підрозділу було визначено загальну структуру програмного продукту, основні підсистеми гри та сценарії взаємодії користувача з ігровим застосунком. Результати проектування варіантів використання стали основою для подальшого створення об'єктно-орієнтованої моделі системи, опису взаємодії ігрових об'єктів та реалізації основних механік *Fiat Lux*.

2.2. Розробка системи ігрових об'єктів та взаємодії між ними

Для реалізації ігрового застосунку *Fiat Lux* було спроектовано систему взаємопов'язаних ігрових об'єктів, яка охоплює персонажа гравця, його стани, бойову систему, здібності, ворогів, статусні ефекти, небезпечні зони та інтерактивні об'єкти світу. Основною метою проектування цієї системи було створення такої структури, у якій кожна механіка має окрему відповідальність, але може взаємодіяти з іншими механіками через чітко визначені компоненти. Це важливо для гри жанру *Metroidvania*, оскільки в ній одночасно працюють системи руху, бою, дослідження світу, відкриття нових можливостей та повернення до раніше недоступних зон.

Центральним об'єктом системи є персонаж гравця (див. рис. 2.2). Його робота організована навколо класів PlayerMain, PlayerContext та PlayerStateMachine.

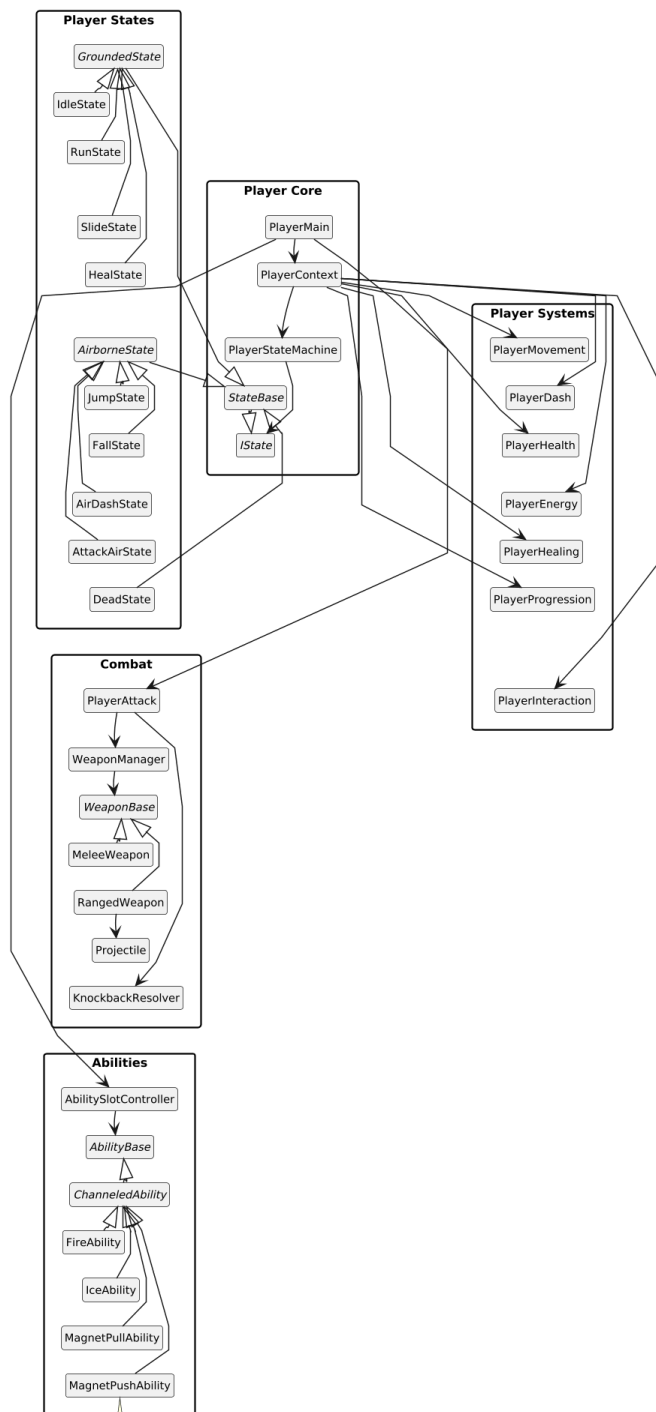


Рис. 2.2 – Діаграма класів персонажа, станів, бою та здібностей

Клас PlayerMain відповідає за початкову ініціалізацію персонажа та об'єднання основних компонентів у єдиний керований об'єкт. PlayerContext виступає контейнером залежностей і зберігає посилання на компоненти руху, ривка, здоров'я, енергії, атаки, лікування, прогресії та взаємодії з об'єктами світу.

Завдяки цьому окремі стани персонажа не звертаються напряму до всіх компонентів сцени, а отримують потрібні об'єкти через контекст. Використання `PlayerContext` спрощує підтримку коду, оскільки вся інформація, необхідна для роботи станів персонажа, зосереджена в одному місці [29]. Наприклад, стан стрибка може звернутися до компонента руху, стан атаки – до бойової системи, а стан лікування – до компонента здоров'я і лікування. Такий підхід зменшує кількість прямих залежностей між класами та робить систему більш придатною для подальшого розширення.

Для керування поведінкою персонажа використовується `PlayerStateMachine`. Машина станів відповідає за перехід між основними режимами поведінки: наземним, повітряним та станом смерті. Кожен із цих режимів може мати власні вкладені стани. До наземних станів належать простій, біг, ковзання, атака на землі та лікування. До повітряних станів належать стрибок, падіння, повітряний ривок та атака в повітрі. Така структура дозволяє відокремити логіку кожної дії та уникнути надмірної кількості умов у одному класі [30].

Модель станів персонажа є важливою для підтримки коректної поведінки під час швидкої зміни дій. Наприклад, персонаж може перейти зі стану бігу до стрибка, зі стрибка до падіння, з падіння до приземлення, а також виконати ривок або атаку залежно від введення користувача. Кожен стан має власні правила входу, оновлення та виходу, що дозволяє контролювати доступність дій у конкретний момент часу.

Бойова система побудована навколо класів `PlayerAttack`, `WeaponManager` та `WeaponBase`. `PlayerAttack` приймає команду атаки, перевіряє можливість її виконання, визначає напрям удару та звертається до активної зброї. `WeaponManager` відповідає за зберігання доступної зброї, вибір поточної зброї та перемикання між відкритими типами. Базовий клас `WeaponBase` задає спільну структуру зброї, а конкретні класи `MeleeWeapon` і `RangedWeapon` реалізують логіку ближнього та дальнього бою.

Зброя ближнього бою використовується для ураження ворогів у зоні удару біля персонажа. Після виконання атаки система перевіряє, чи потрапив ворог у

відповідну область, після чого йому завдається пошкодження. Додатково може бути застосовано відкидання, яке розраховується через KnockbackResolver. Це дозволяє винести фізичну реакцію на удар в окремий компонент і використовувати її не тільки для зброї, а й для інших механік, наприклад небезпечних зон.

Зброя дальнього бою реалізується через створення снаряда Projectile, який рухається у заданому напрямку та завдає пошкодження при зіткненні з ціллю. Такий підхід дозволяє реалізувати різні типи атак без дублювання загальної логіки бойової системи. WeaponManager при цьому не повинен знати внутрішню реалізацію конкретної зброї, а лише викликає її метод атаки [31].

Окрему роль у системі відіграє підсистема здібностей. Вона побудована на базовому класі AbilityBase, від якого успадковуються конкретні здібності: FireAbility, IceAbility, MagnetPullAbility та MagnetPushAbility. Кожна здібність має власну вартість енергії, час перезарядки та ефект. Перед виконанням здібності система перевіряє, чи достатньо в персонажа енергії. Якщо ресурсу вистачає, енергія витрачається, після чого виконується відповідний ефект.

Здібності в Fiat Lux виконують не лише бойову, а й навігаційну функцію. Вогонь може впливати на об'єкти, що здатні горіти, лід – заморожувати воду або цілі, а магнітні здібності – притягувати або відштовхувати металеві об'єкти. Саме завдяки цьому система здібностей підтримує жанрову особливість Metroidvania: нові можливості персонажа відкривають доступ до нових ділянок світу.

Система енергії реалізована через компонент PlayerEnergy. Він зберігає поточну та максимальну кількість енергії, відповідає за її поступове відновлення та перевіряє можливість витрати ресурсу. Такий компонент відокремлює ресурсну логіку від самих здібностей. У результаті AbilityBase не змінює значення енергії напряму, а звертається до PlayerEnergy, який приймає рішення про можливість виконання дії.

Підсистема здоров'я персонажа відповідає за отримання пошкоджень, перевірку стану життя та перехід до смерті. Якщо запас здоров'я зменшується до нуля, машина станів переводить персонажа у DeadState. Після цього може бути

виконано відродження біля останньої активованої безпечної точки. Такий підхід пов'язує систему здоров'я, машину станів і об'єкти світу в єдиний сценарій обробки смерті та повернення до гри.

Друга частина діаграми класів описує систему світу, статусних ефектів і ворогів (див. рис. 2.3). До об'єктів світу належать *BurnableObstacle*, *FreezableWater*, *Metallic*, *ItemUnlocker*, *SafePoint* та класи небезпечних зон. Ці об'єкти реагують на дії гравця та змінюють стан середовища. Наприклад, *BurnableObstacle* може реагувати на вогонь, *FreezableWater* – на лід, а *Metallic* – на магнітні здібності.

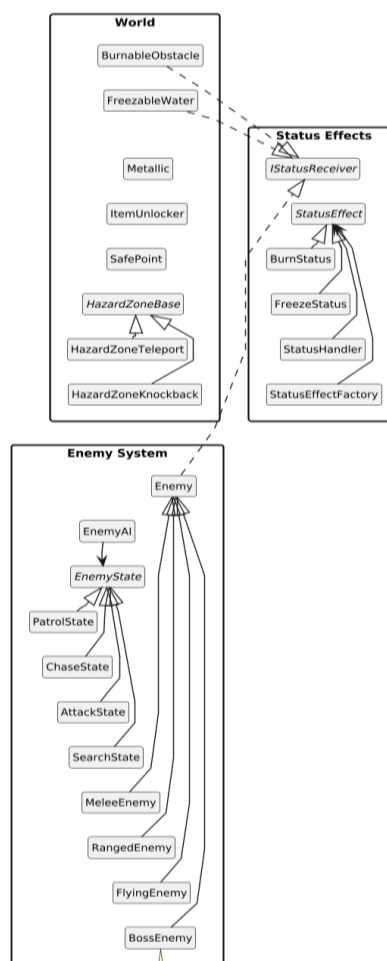


Рис. 2.3 – Діаграма класів світу, статусних ефектів та ворогів

SafePoint використовується як контрольна точка для повернення персонажа після смерті або потрапляння в небезпечну зону. Коли гравець активує *SafePoint*, він зберігається як остання безпечна позиція. Якщо персонаж потрапляє в

HazardZoneTeleport, система переносить його до цієї точки. Це дозволяє уникнути повного перезапуску рівня та підтримує безперервність ігрового процесу.

Небезпечні зони реалізовані через базовий клас HazardZoneBase та похідні класи HazardZoneTeleport і HazardZoneKnockback. Базовий клас задає спільну логіку визначення контакту з персонажем, а конкретні реалізації визначають наслідки цього контакту. HazardZoneTeleport переносить персонажа до безпечної точки, а HazardZoneKnockback може відштовхувати його назад. Такий поділ дозволяє легко додавати нові типи небезпек.

Підсистема статусних ефектів реалізована окремо від здібностей і ворогів. Її основними елементами є IStatusReceiver, StatusEffect, StatusHandler, BurnStatus, FreezeStatus та StatusEffectFactory. Інтерфейс IStatusReceiver визначає, що об'єкт може приймати статусні ефекти. StatusHandler зберігає активні ефекти, оновлює їх протягом часу та видаляє після завершення дії. Базовий клас StatusEffect описує спільні властивості ефекту, такі як тип, тривалість, кількість накладань і сила впливу.

BurnStatus відповідає за ефект горіння, який може завдавати пошкодження протягом часу. FreezeStatus відповідає за замороження, що може обмежувати рух або змінювати поведінку цілі. StatusEffectFactory використовується для створення потрібного статусу без прямої залежності здібностей від конкретних класів ефектів. Завдяки цьому новий статус можна додати без суттєвої зміни існуючих здібностей.

Система ворогів складається з базового класу Enemy, класу EnemyAI, набору станів поведінки та конкретних типів противників. Клас Enemy відповідає за здоров'я, отримання пошкоджень, реакцію на горіння або замороження та смерть. EnemyAI відповідає за поведінку противника, пошук гравця, пересування, зміну станів і виконання атак. Такий поділ дозволяє відокремити фізичний стан ворога від його поведінкової логіки.

Поведінка ворогів організована через стани PatrolState, SearchState, ChaseState та AttackState. У стані патрулювання ворог рухається заданою ділянкою та перевіряє, чи бачить гравця. Якщо гравця виявлено, ворог переходить

до бойового стану. У стані переслідування він намагається скоротити дистанцію до цілі, а в стані атаки виконує удар або дальню атаку. Така модель дозволяє зробити поведінку ворогів зрозумілою та розширюваною.

Конкретні типи ворогів відрізняються способом руху та атаки. *MeleeEnemy* орієнтований на ближній бій і намагається наблизитися до персонажа. *RangedEnemy* атакує з дистанції та може підтримувати безпечну відстань. *FlyingEnemy* використовує повітряний простір і має інші правила пересування. *BossEnemy* має складнішу поведінку та може використовувати декілька атакувальних станів. Усі ці типи використовують спільні принципи системи ворогів, але реалізують власну спеціалізовану логіку.

Для опису даних і зв'язків між основними сутностями було побудовано ER-подібну схему (див. рис. 2.4). Вона відображає ключові об'єкти ігрової логіки: *Player*, *Enemy*, *Weapon*, *Ability*, *StatusEffect*, *EnemyState*, *PlayerState*, *SafePoint*, *HazardZone*, *UnlockItem* та *WorldObject*. Така схема дозволяє показати, які дані зберігаються в системі та як вони пов'язані між собою під час виконання ігрового процесу.

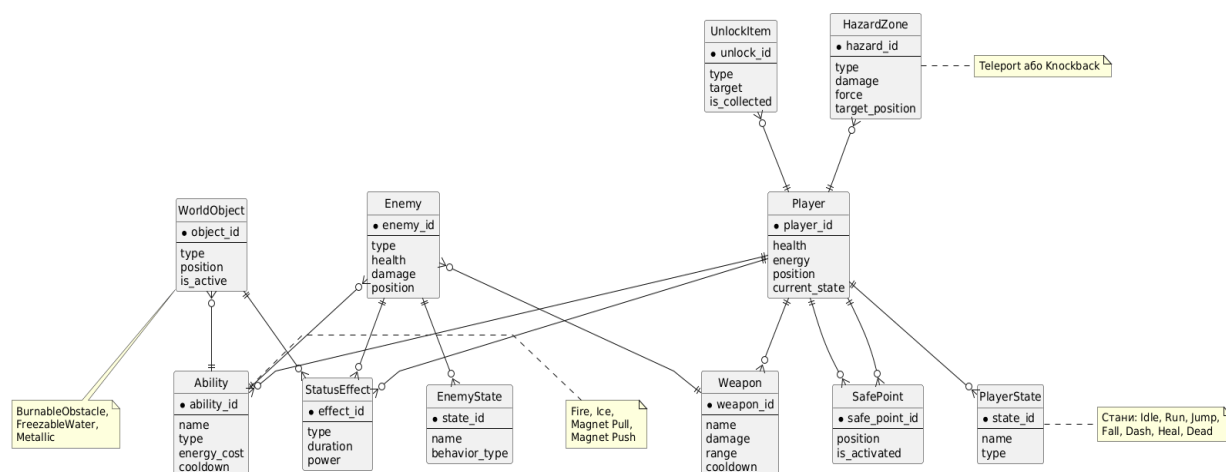


Рис. 2.4 – ER-подібна схема ігрових сутностей Fiat Lux

Сутність *Player* містить основні характеристики персонажа: здоров'я, енергію, позицію та поточний стан. Вона пов'язана зі зброєю, здібностями, контрольними точками та станами. Сутність *Weapon* описує зброю персонажа, її пошкодження, дальність і час перезарядки. Сутність *Ability* описує здібності, їхню

вартість енергії, тип і cooldown. Така модель показує, що персонаж є центральною сутністю, навколо якої формуються основні ігрові взаємодії.

Сутність Enemy містить інформацію про тип ворога, здоров'я, пошкодження, позицію та поточний стан поведінки. Ворог може отримувати статусні ефекти та реагувати на атаки гравця. Сутність EnemyState описує можливі стани поведінки, такі як патрулювання, пошук, переслідування та атака. Завдяки цьому поведінка ворога може змінюватися залежно від відстані до гравця, видимості цілі та результатів бойової взаємодії.

Сутності WorldObject, HazardZone, UnlockItem та SafePoint описують об'єкти ігрового світу. WorldObject є загальним представленням об'єктів середовища, які можуть бути активними або неактивними, мати позицію та тип. HazardZone описує небезпечні зони, що можуть завдавати пошкодження, телепортувати або відкидати персонажа. UnlockItem відповідає за відкриття нових можливостей, а SafePoint визначає позицію, до якої персонаж може повернутися після смерті або помилки.

Взаємодія між ігровими об'єктами відбувається в межах основного ігрового циклу (див. рис. 2.5).

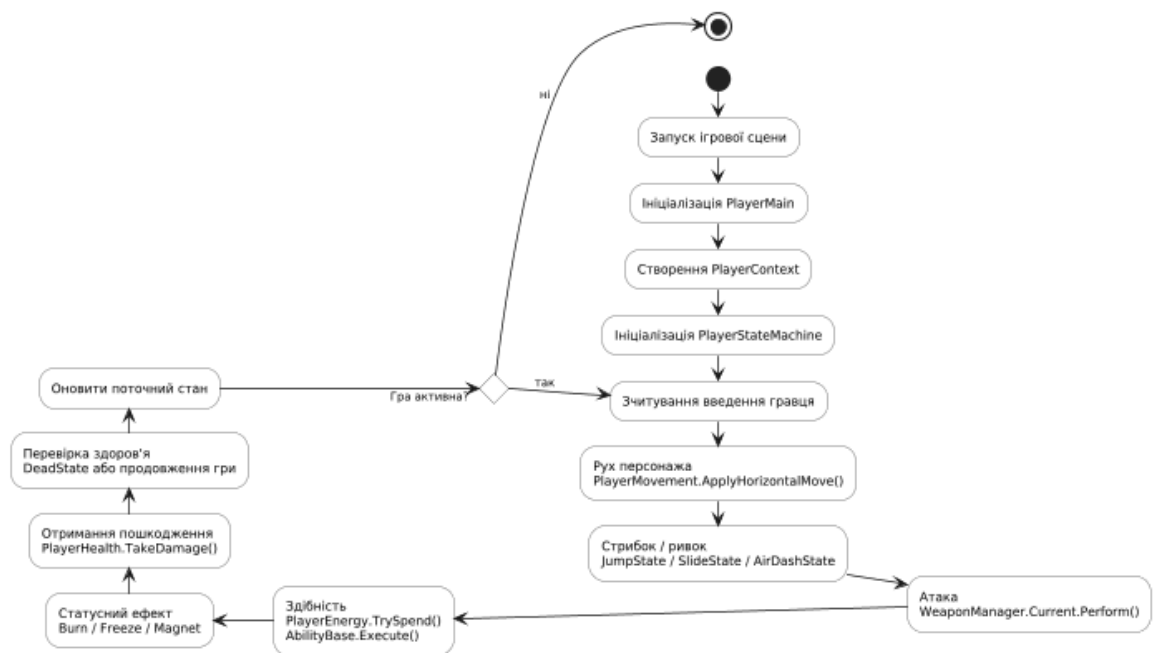


Рис. 2.5 – Діаграма активностей основного ігрового циклу

Після запуску сцени ініціалізується PlayerMain, створюється PlayerContext і запускається PlayerStateMachine. Далі система постійно зчитує введення гравця та перевіряє, які дії потрібно виконати. Якщо користувач задає напрям руху, компонент PlayerMovement змінює швидкість персонажа. Якщо натиснуто кнопку стрибка, перевіряється перебування персонажа на землі, після чого активується відповідний стан.

У разі атаки система визначає напрям удару, звертається до WeaponManager, отримує активну зброю та викликає її метод виконання атаки. Якщо ворог перебуває в зоні ураження, йому завдається пошкодження, після чого може бути застосовано відкидання. Якщо здоров'я ворога зменшується до нуля, викликається логіка смерті. Така послідовність дозволяє чітко відокремити введення користувача, вибір зброї, перевірку цілі та наслідки атаки (див. рис. 2.6).

Під час використання здібності спочатку перевіряється доступність відповідної дії та запас енергії. Якщо енергії достатньо, вона витрачається, а здібність виконує свій ефект. Якщо ціль підтримує отримання статусів, до неї застосовується BurnStatus, FreezeStatus або інший ефект. Якщо ціллю є об'єкт світу, наприклад металевий об'єкт або вода, змінюється стан цього об'єкта. Таким чином, одна й та сама підсистема здібностей може впливати як на ворогів, так і на середовище.

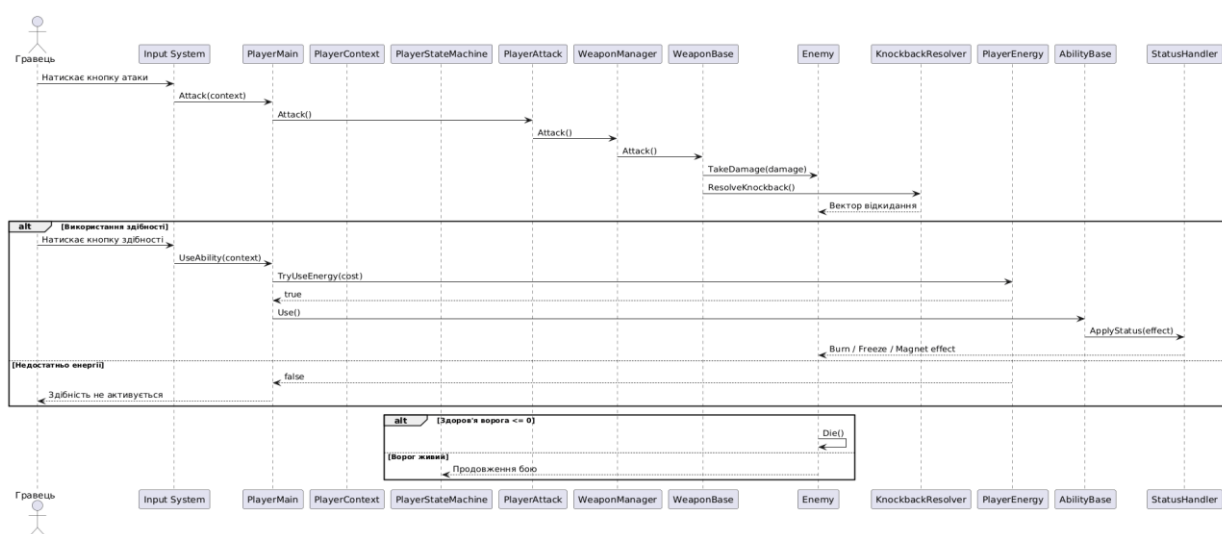


Рис. 2.6 – Діаграма послідовності атаки та використання здібності

Діаграма послідовності деталізує порядок викликів між компонентами. Коли гравець натискає кнопку атаки, система введення передає команду до PlayerMain, після чого PlayerAttack звертається до WeaponManager. Активна зброя виконує атаку, визначає результат і передає пошкодження ворогу через метод TakeDamage. Після цього KnockbackResolver розраховує напрям і силу відкидання. Якщо запас здоров'я ворога вичерпано, виконується метод Die.

У випадку використання здібності команда передається до відповідного об'єкта AbilityBase. Перед виконанням викликається перевірка енергії через PlayerEnergy. Якщо енергії достатньо, запускається логіка здібності, а результат передається цільовому об'єкту. Якщо об'єкт має StatusHandler, до нього додається статусний ефект. Якщо енергії недостатньо, здібність не активується, а стан цілі не змінюється.

Для узагальнення призначення основних класів програмної системи доцільно подати їх у вигляді таблиці.

Таблиця 2.2

Призначення основних класів системи Fiat Lux

Клас	Підсистема	Призначення
PlayerMain	Персонаж	Ініціалізація основних компонентів персонажа
PlayerContext	Персонаж	Зберігання посилань на підсистеми персонажа
PlayerStateMachine	Стани	Керування переходами між станами
PlayerMovement	Рух	Горизонтальний рух, стрибок, перевірка землі
PlayerDash	Рух	Виконання ривка та ковзання
PlayerHealth	Здоров'я	Обробка пошкоджень і перевірка смерті
PlayerEnergy	Енергія	Витрата й відновлення енергії
PlayerAttack	Бій	Обробка команди атаки
WeaponManage	Бій	Вибір і перемикання активної зброї
WeaponBase	Бій	Базова структура зброї
MeleeWeapon	Бій	Реалізація ближньої атаки
RangedWeapon	Бій	Реалізація дальньої атаки

Projectile	Бій	Рух снаряда та пошкодження при зіткненні
KnockbackResolver	Бій	Розрахунок відкидання цілі
FireAbility	Здібності	Застосування вогняного ефекту
IceAbility	Здібності	Застосування льодяного ефекту
MagnetPullAbility	Здібності	Притягування металевих об'єктів
MagnetPushAbility	Здібності	Відштовхування металевих об'єктів
StatusHandler	Статуси	Зберігання та оновлення активних статусів
BurnStatus	Статуси	Ефект горіння
FreezeStatus	Статуси	Ефект замороження
Enemy	Вороги	Здоров'я, пошкодження та смерть ворога
EnemyAI	Вороги	Поведінка та зміна станів ворога
AttackState	Вороги	Виконання атаки
PatrolState	Вороги	Патрулювання території
SafePoint	Світ	Контрольна точка повернення
HazardZoneTeleport	Світ	Перенесення персонажа до SafePoint
ItemUnlocker	Прогресія	Відкриття зброї або здібності
BurnableObstacle	Світ	Об'єкт, що реагує на вогонь
FreezableWater	Світ	Об'єкт, що реагує на лід
FreezableWater	Світ	Об'єкт, що реагує на магнітні здібності

Таким чином, система ігрових об'єктів Fiat Lux побудована за модульним принципом. Персонаж керується через контекст і машину станів, бойова система

використовує окремі класи зброї, здібності взаємодіють зі світом через ресурс енергії та статусні ефекти, а вороги мають власну поведінкову модель [28]. Така структура забезпечує зрозумілий розподіл відповідальності між класами, полегшує тестування окремих механік і створює основу для подальшого розширення гри новими ворогами, здібностями, небезпечними зонами та інтерактивними об'єктами.

2.3. Проєктування та реалізація основних ігрових механік

Основу ігрового процесу Fiat Lux становлять механіки керування персонажем, переміщення між платформами, атаки, використання здібностей, взаємодії з ворогами, об'єктами світу та системою прогресії. Оскільки гра належить до жанру Metroidvania, реалізовані механіки повинні не лише забезпечувати базове проходження рівнів, а й підтримувати поступове відкриття нових можливостей. Саме тому окремі дії персонажа, такі як використання здібностей, перемикання зброї або взаємодія з об'єктами, пов'язані з поточним прогресом гравця.

Однією з головних механік є керування станами персонажа. Для цього використовується `PlayerStateMachine`, яка відповідає за зміну поточного стану персонажа залежно від умов гри. Машина станів дозволяє відокремити логіку простою, руху, стрибка, падіння, атаки, лікування та смерті. Завдяки цьому кожна дія персонажа реалізується незалежно, а перехід між діями відбувається контролювано:

```
public void ChangeState(IState next)
{
    _current?.Exit();
    _current = next;
    _current.Enter();
    OnStateChanged?.Invoke(_current);
}
public void Update()
{
    if (!_ctx.Health.IsAlive && _current != Dead)
    {
        ChangeState(Dead);
        return;
    }
}
```

```

    }
    _current?.Update();
}

```

У наведеному фрагменті метод `ChangeState()` завершує попередній стан, призначає новий і викликає його метод `Enter()`. Метод `Update()` перевіряє, чи персонаж живий. Якщо запас здоров'я вичерпано, виконується перехід у `DeadState`. Таким чином, логіка смерті персонажа не змішується з логікою руху або атаки, а обробляється як окремий стан.

Механіка руху персонажа реалізована через компонент `PlayerMovement`. Він відповідає за горизонтальне переміщення, перевірку перебування персонажа на землі, виконання стрибка та обробку відкидання. Під час руху враховується напрям введення, швидкість персонажа та можливі обмеження, наприклад стан ривка або `knockback`. Це дозволяє уникнути конфліктів між різними фізичними діями:

```

public void ApplyHorizontalMove(float input)
{
    if (!canMoveDuringKnockback) return;
    if (isDashing) return;
    rb.linearVelocity = new Vector2(
        input * moveSpeed * _speedMultiplier,
        rb.linearVelocity.y
    );
}
public void RequestJump()
{
    if (isGrounded && !isDashing)
        jumpRequested = true;
}

```

Метод `ApplyHorizontalMove()` змінює горизонтальну швидкість персонажа лише тоді, коли персонаж не перебуває у стані ривка або відкидання. Метод `RequestJump()` створює запит на стрибок тільки за умови, що персонаж стоїть на землі та не виконує ривок. Такий підхід запобігає некоректному накладанню дій, наприклад стрибку під час активного `dash`.

Окремою складовою руху є механіка ривка. Вона використовується для швидкого переміщення персонажа, ухилення від атак і подолання певних перешкод. У грі передбачено наземний та повітряний варіанти ривка. Залежно від поточного стану персонажа система переводить його у `SlideState` або `AirDashState`.

Це дозволяє зберігати різну поведінку для руху на землі та в повітрі.

Механіка атаки реалізована через `PlayerAttack`, `WeaponManager` і конкретні класи зброї. Після натискання кнопки атаки система перевіряє, чи завершився `cooldown`, визначає активну зброю, напрям удару та виконує атаку. Якщо атака влучає у ворога, йому завдається пошкодження, а для ближньої зброї додатково застосовується відкидання:

```
public void RequestAttack()
{
    if (Time.time < _nextAttackTime) return;
    if (_ctx == null) return;
    var weapon = weaponManager.Current;
    if (weapon == null) return;
    Vector2Int dir = AttackDirectionResolver.Resolve(_ctx);
    var result = weapon.Perform(_ctx, dir, _comboIndex);
    if (weapon.IsMelee && weapon is MeleeWeapon melee)
    {
        KnockbackResolver.Apply(_ctx, result, melee.KnockbackCfg);

        if (result.HitSomething)
            _ctx.Energy.Restore(melee.EnergyPerMeleeHit);
    }
    EventBus.Publish(new PlayerAttackedEvent(
        dir, _comboIndex, result.HitSomething
    ));
}
```

У цьому фрагменті спочатку перевіряється можливість виконання атаки. Далі визначається активна зброя та напрям удару. Метод `weapon.Perform()` виконує конкретну реалізацію атаки, а результат використовується для подальшої обробки. Якщо використовується зброя ближнього бою, застосовується `knockback`, а при успішному влучанні персонаж може відновити частину енергії.

Для керування зброєю використовується `WeaponManager`. Він зберігає список доступних видів зброї та дозволяє перемикатися лише на ті, які були відкриті в процесі проходження. Це підтримує механіку прогресії, оскільки нова зброя стає доступною не одразу, а після отримання відповідного предмета:

```
public bool TrySwitchTo(WeaponType type)
{
    if (!progression.IsWeaponUnlocked(type)) return false;

    for (int i = 0; i < weapons.Length; i++)
    {

```

```

        if (weapons[i].Type == type)
        {
            _currentIndex = i;
            ApplyActiveVisual();
            OnWeaponChanged?.Invoke(Current);
            return true;
        }
    }
    return false;
}

```

Метод TrySwitchTo() спочатку перевіряє, чи розблоковано потрібний тип зброї. Якщо зброя доступна, вона стає активною, оновлюється її візуальне представлення та викликається подія зміни зброї. Завдяки цьому інші системи, наприклад інтерфейс, можуть реагувати на зміну активного озброєння.

Система здібностей реалізована через базовий клас AbilityBase. Кожна здібність має власну вартість енергії та cooldown. Перед виконанням здібності перевіряється, чи не перебуває вона на перезарядці, а також чи достатньо енергії у персонажа. Якщо умови виконані, запускається відповідний ефект здібності:

```

public void TryPerform(PlayerContext ctx)
{
    if (_onCooldown) return;
    if (!ctx.Energy.TrySpend(energyCost))
        return;
    EventBus.Publish(new PlayerAbilityActivatedEvent(
        GetType().Name, isStart: true
    ));
    StartCoroutine(Execute(ctx));
}

```

У наведеному фрагменті здібність не активується, якщо вона перебуває на cooldown або якщо в персонажа недостатньо енергії. Це дозволяє контролювати частоту використання сильних дій і робить систему здібностей збалансованою. Конкретна логіка здібності реалізується в методі Execute(), який перевизначається в похідних класах.

Витрата та відновлення енергії винесені в окремий компонент PlayerEnergy. Це дозволяє відокремити ресурсну логіку від конкретних здібностей. Здібність лише надсилає запит на витрату енергії, а PlayerEnergy самостійно перевіряє, чи можна виконати дію:

```

public bool TrySpend(float cost)
{

```

```

        if (_energy < cost)
        {
            OnEnergyDepleted?.Invoke();
            return false;
        }
        _energy -= cost;
        OnEnergySpent?.Invoke(cost);
        OnEnergyChanged?.Invoke(_energy, _maxEnergy);
        return true;
    }

```

Якщо поточного запасу енергії менше за вартість здібності, метод повертає false і повідомляє систему про нестачу ресурсу. Якщо енергії достатньо, її кількість зменшується, після чого викликаються події зміни енергії. Це дозволяє оновлювати інтерфейс та інші системи без прямої залежності від PlayerEnergy.

Окрему роль відіграє система статусних ефектів. Вона дозволяє накладати на об'єкти тимчасові стани, наприклад горіння або замороження. Статуси зберігаються в StatusHandler, який оновлює їх щокадрово та видаляє після завершення дії:

```

public void ApplyStatus(StatusEffect status)
{
    float res = resistance.GetResistance(status.GetType());
    int effectiveStacks = 0;
    for (int i = 0; i < status.StacksToAdd; i++)
        if (Random.value > res) effectiveStacks++;
    if (effectiveStacks <= 0) return;
    var type = status.GetType();
    if (!_active.TryGetValue(type, out var existing))
    {
        _active[type] = status;
        status.AddStack(effectiveStacks);
        status.OnApply(gameObject);
    }
    else
    {
        int prev = existing.Stacks;
        existing.AddStack(effectiveStacks);
        existing.OnStackChanged(gameObject, prev);
    }
}

```

Метод ApplyStatus() враховує опір цілі до конкретного статусу та визначає кількість ефективних накладань. Якщо статус ще не активний, він додається до словника активних ефектів. Якщо такий статус уже існує, збільшується кількість його накладань. Це дозволяє реалізувати накопичувальні ефекти та різну стійкість

об'єктів до впливу здібностей.

Механіка ворогів складається з отримання пошкоджень, зміни станів поведінки та виконання атак. Базовий клас Enemy відповідає за здоров'я, реакцію на пошкодження, горіння, замороження та смерть. Якщо здоров'я ворога зменшується до нуля, викликається метод Die():

```
public void TakeDamage(float damage)
{
    currentHealth -= damage;
    OnDamaged?.Invoke();

    if (currentHealth <= 0f)
        Die();
}
```

Метод TakeDamage() зменшує поточний запас здоров'я ворога та викликає подію отримання пошкодження. Після цього виконується перевірка на смерть. Завдяки такому підходу інші компоненти, наприклад EnemyAI або система винагороди, можуть реагувати на пошкодження чи знищення ворога через події.

Поведінка ворога реалізована через стани. У стані патрулювання ворог рухається у визначеному напрямку, перевіряє наявність гравця та реагує на межі платформи або перешкоди. Якщо гравця виявлено, ворог переходить до бойового стану:

```
public override void Update()
{
    ai.MoveInPatrolDirection();

    if (ai.CanSeePlayer())
    {
        ai.PublishPlayerSpotted();
        ai.SetState(ai.CreateCombatState());
        return;
    }
    if (ai.IsAtPatrolBoundary())
        ai.FlipDirection();
}
```

У цьому фрагменті ворог спочатку виконує патрулювання. Якщо гравець потрапляє в зону видимості, викликається подія виявлення гравця і ворог переходить у бойовий стан. Якщо ворог досягає межі патрулювання, він змінює напрям руху. Такий алгоритм забезпечує базову, але зрозумілу поведінку

противника.

Для перевірки видимості гравця використовується метод `CanSeePlayer()`. Він враховує дистанцію до гравця, кут огляду та наявність перешкод між ворогом і персонажем:

```
public virtual bool CanSeePlayer()
{
    if (Player == null) return false;
    Vector2 toPlayer =
        (Vector2)Player.position - (Vector2)transform.position;
    float distance = toPlayer.magnitude;
    if (distance > detectionRange) return false;

    if (!_aggressiveVision)
    {
        float angle = Vector2.Angle(
            new Vector2(FacingDirection, 0f), toPlayer
        );
        if (angle > fovAngle * 0.5f) return false;
    }
    return true;
}
```

Метод `CanSeePlayer()` дозволяє ворогу реагувати лише на ті ситуації, коли гравець справді перебуває в межах виявлення. Спочатку перевіряється відстань до персонажа, після чого враховується напрям погляду ворога. Це робить поведінку противників більш природною, оскільки вони не реагують на гравця миттєво з будь-якої точки сцени.

Механіка небезпечних зон використовується для обробки ситуацій, коли персонаж потрапляє в небезпечну ділянку рівня. Одним із варіантів такої зони є `HazardZoneTeleport`, яка повертає персонажа до останньої безпечної точки `SafePoint`:

```
protected override void OnPlayerHit(GameObject player)
{
    SafePoint target = SafePoint.Last;
    if (target == null)
        return;
    Vector2 destination = target.transform.position;
    if (player.TryGetComponent<Rigidbody2D>(out var rb))
    {
        rb.linearVelocity = Vector2.zero;
        rb.position = destination;
    }
    else
```

```

    {
        player.transform.position = destination;
    }
}

```

У наведеному фрагменті система визначає останню активовану безпечну точку. Якщо вона існує, персонаж переноситься до її позиції. Перед переміщенням фізична швидкість персонажа скидається до нуля, щоб уникнути некоректного руху після телепортації. Якщо `Rigidbody2D` відсутній, змінюється позиція `Transform`.

Таким чином, основні ігрові механіки *Fiat Lux* реалізовані як набір взаємопов'язаних, але незалежних систем. Машина станів керує поведінкою персонажа, `PlayerMovement` і `PlayerDash` відповідають за переміщення, бойова система обробляє атаки та зброю, `AbilityBase` і `PlayerEnergy` забезпечують роботу здібностей, `StatusHandler` реалізує статусні ефекти, а `EnemyAI` керує поведінкою ворогів. Такий підхід робить архітектуру гри гнучкою, зрозумілою та придатною для подальшого розширення.

Висновки до другого розділу

У другому розділі було виконано проєктування та опис програмної структури ігрового застосунку *Fiat Lux* у жанрі *Metroidvania*. На основі визначених функціональних вимог було сформовано загальну архітектуру системи, виділено основні модулі програмного продукту та описано їхню роль у забезпеченні ігрового процесу. До ключових складових системи належать модулі керування персонажем, бойової взаємодії, використання здібностей, поведінки ворогів, статусних ефектів, взаємодії з об'єктами світу та прогресії гравця.

Було розглянуто варіанти використання програмного продукту, що дозволило визначити основні сценарії взаємодії користувача з грою. Гравець може керувати персонажем, переміщуватися між ділянками рівня, виконувати стрибки та ривки, атакувати ворогів, використовувати здібності, взаємодіяти з об'єктами середовища, відкривати нову зброю та здібності, а також проходити небезпечні

зони. Це дало змогу сформувавши логічну модель поведінки системи та визначити взаємозв'язки між її компонентами.

У процесі проєктування системи ігрових об'єктів було побудовано об'єктно-орієнтовану модель застосунку. Центральним елементом цієї моделі є персонаж гравця, робота якого організована через PlayerMain, PlayerContext та PlayerStateMachine. Використання машини станів дозволило розділити логіку руху, стрибка, падіння, ривка, атаки, лікування та смерті на окремі стани. Такий підхід підвищує зрозумілість структури програми та спрощує подальше розширення функціональності.

Окремо було спроектовано бойову систему, систему здібностей, поведінку ворогів та взаємодію з об'єктами світу. Бойова система базується на використанні активної зброї, визначенні напрямку атаки, нанесенні пошкодження та застосуванні відкидання. Система здібностей використовує запас енергії персонажа та дозволяє взаємодіяти з ворогами й об'єктами середовища через вогонь, лід і магнітні ефекти. Поведінка ворогів реалізована через стани патрулювання, пошуку, переслідування та атаки, що забезпечує більш гнучку реакцію противників на дії гравця.

Також у розділі було описано основні ігрові механіки, що формують ігровий процес Fiat Lux. До них належать переміщення персонажа, стрибок, ривок, атака, використання здібностей, отримання пошкоджень, смерть і відродження, відкриття нових можливостей та взаємодія з небезпечними зонами. Наведені фрагменти програмного коду підтверджують практичну реалізацію цих механік у межах проєкту Unity.

Отже, у другому розділі було сформовано проєктну та програмну основу гри Fiat Lux. Розроблена структура забезпечує модульність, розширюваність і логічну узгодженість основних ігрових систем. Отримані результати стали підґрунтям для подальшого опису роботи програмного продукту, порядку його запуску, особливостей інтерфейсу та тестування реалізованих функцій.

РОЗДІЛ 3. ОПИС РОБОТИ ПРОГРАМНОГО ПРОДУКТУ, ЙОГО ВСТАНОВЛЕННЯ ТА ТЕСТУВАННЯ

3.1 Порядок встановлення та запуску програмного продукту

Розроблений програмний продукт є двовимірним ігровим застосунком у жанрі Metroidvania, створеним за допомогою ігрового рушія Unity. Гра призначена для запуску на персональному комп'ютері під керуванням операційної системи Windows. Програмний продукт може запускатися двома способами: через готову збірку гри або безпосередньо з середовища Unity Editor під час розробки й тестування.

Для запуску готової збірки користувачу необхідно отримати папку з виконуваним файлом гри та службовими файлами, що формуються Unity під час побудови проєкту. Зазвичай така папка містить виконуваний файл із розширенням .exe, каталог із ресурсами гри та додаткові службові файли. Усі ці елементи повинні зберігатися в одній директорії, оскільки виконуваний файл використовує ресурси, сцени, бібліотеки та налаштування, розташовані поруч із ним. Мінімальні вимоги до запуску програмного продукту наведено в табл. 3.1.

Таблиця 3.1

Мінімальні вимоги до запуску ігрового застосунку

Компонент	Мінімальні вимоги
Операційна система	Windows 10 або новіша
Процесор	Двоядерний процесор з частотою від 2.0 ГГц
Оперативна пам'ять	4 ГБ
Відеоадаптер	Інтегрований або дискретний відеоадаптер з підтримкою DirectX
Вільне місце на диску	Не менше 256 МБ
Пристрої введення	Клавіатура; за потреби геймпад

Перед запуском програмного продукту необхідно переконатися, що комп'ютер відповідає мінімальним вимогам. Оскільки застосунок є 2D-грою, він

не потребує високопродуктивного апаратного забезпечення, однак для стабільної роботи необхідна наявність сучасної версії операційної системи Windows, відеоадаптера з підтримкою базових графічних можливостей, а також достатнього обсягу оперативної пам'яті для завантаження сцен і ресурсів гри.

Для встановлення гри користувачу не потрібно виконувати складну процедуру інсталяції. Достатньо розпакувати архів із готовою збіркою у вибрану папку на комп'ютері. Після цього запуск виконується подвійним натисканням на виконуваний файл гри. Такий підхід є типовим для невеликих ігрових застосунків, створених у Unity, оскільки всі необхідні ресурси вже входять до складу збірки. Структуру теки зі збіркою відображено на рис. 3.1.

```

Build/
├─ 2D Metroidvania.exe
├─ UnityCrashHandler64.exe
├─ UnityPlayer.dll
├─ MonoBleedingEdge/
├─ D3D12/
├─ 2D Metroidvania_BurstDebugInformation_DoNotShip/
│   └─ Data/
│       └─ Plugins/
│           └─ x86_64/
│               └─ lib_burst_generated.txt
└─ 2D Metroidvania_Data/
    ├─ Managed/
    ├─ Plugins/
    ├─ Resources/
    ├─ app.info
    ├─ boot.config
    ├─ globalgamemangers
    ├─ globalgamemangers.assets
    ├─ globalgamemangers.assets.resS
    ├─ level0
    ├─ level1
    ├─ level2
    ├─ resources.assets
    ├─ resources.assets.resS
    ├─ RuntimeInitializeOnLoads.json
    ├─ ScriptingAssemblies.json
    ├─ sharedassets0.assets
    ├─ sharedassets0.assets.resS
    ├─ sharedassets0.resource
    ├─ sharedassets1.assets
    ├─ sharedassets1.assets.resS
    ├─ sharedassets2.assets
    └─ sharedassets2.assets.resS
    
```

Рис. 3.1 – Структура теки з готовою збіркою ігрового застосунку

Після запуску виконуваного .exe - файлу відкривається головне вікно гри (див рис. 3.2). На початковому екрані користувач може перейти до початку гри, завантаження збереження або налаштувань, якщо такі елементи передбачені поточною збіркою. Основна взаємодія з грою виконується за допомогою клавіатури. Персонаж може рухатися, стрибати, атакувати, виконувати ривок, використовувати здібності та взаємодіяти з об'єктами ігрового світу.



Рис. 3.2 – Початковий екран ігрового застосунку

Другим способом запуску є відкриття проєкту через Unity Editor (див. рис. 3.3).

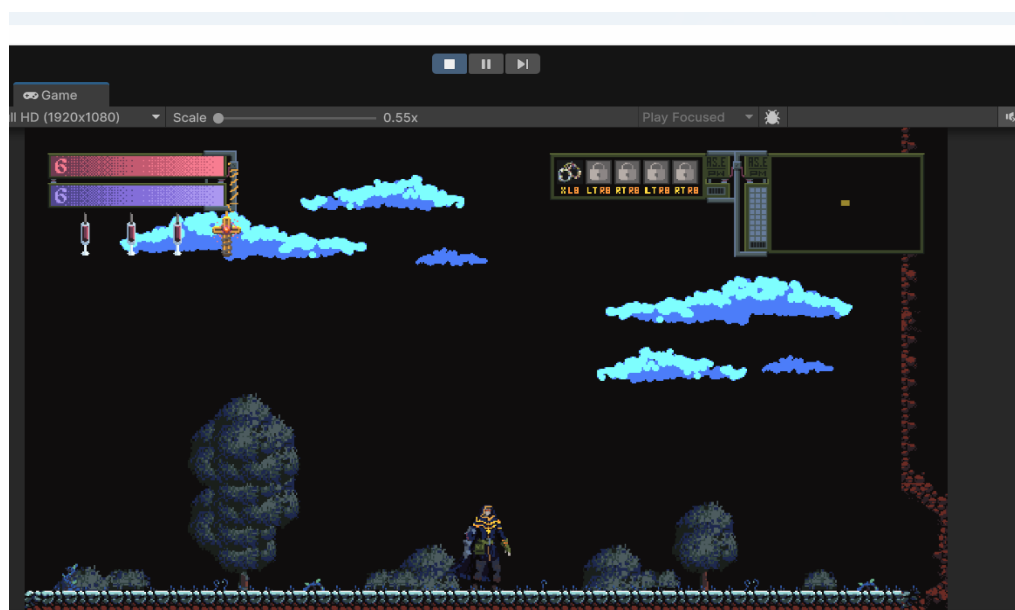


Рис.3.3 – Запуск ігрового застосунку в середовищі Unity Editor

Цей спосіб використовується розробниками під час налагодження, тестування і внесення змін до ігрових механік. Для цього необхідно відкрити Unity Hub, додати папку проєкту до списку проєктів і запустити її у відповідній версії Unity. Після відкриття проєкту можна вибрати потрібну сцену та натиснути кнопку Play у верхній частині редактора.

Запуск через Unity Editor дає змогу перевіряти роботу окремих сцен, змінювати параметри компонентів, переглядати об'єкти ігрового світу, тестувати взаємодію персонажа з ворогами, платформами, здібностями та елементами інтерфейсу. Такий спосіб є основним під час розробки, оскільки дозволяє швидко знаходити помилки та перевіряти зміни без створення нової збірки після кожного редагування.

Під час запуску гри завантажуються необхідні сцени, ресурси, налаштування персонажа, інтерфейс користувача та компоненти ігрової логіки. Основна логіка роботи персонажа активується після створення об'єкта гравця на сцені. Компонент PlayerMain ініціалізує контекст персонажа, автомат станів, систему здоров'я, енергії, атаки та інші пов'язані модулі. Після цього гра переходить у режим обробки введення користувача та оновлення ігрового стану. Окрему роль під час запуску відіграють сцени та службові об'єкти. Вони відповідають за створення ігрового середовища, завантаження кімнат, роботу камери, інтерфейсу, збережень і подій між компонентами. Завдяки цьому після запуску користувач одразу отримує готове середовище для взаємодії з грою: персонаж може переміщуватися рівнем, вступати в бій, використовувати здібності та взаємодіяти з об'єктами.

Таким чином, встановлення і запуск програмного продукту не потребує складних дій від користувача. Готова збірка запускається безпосередньо через виконуваний файл, а для розробки й тестування може використовуватися Unity Editor. Обидва способи запуску забезпечують доступ до основного функціоналу гри, однак запуск через редактор додатково надає можливість аналізувати внутрішню структуру сцен, компонентів і параметрів ігрових об'єктів.

3.2 Структура інтерфейсу та особливості ігрового процесу

Інтерфейс ігрового застосунку Fiat Lux призначений для відображення поточного стану персонажа, доступних ресурсів, активних здібностей, зброї та важливих ігрових подій. Оскільки програмний продукт належить до жанру Metroidvania, інтерфейс не повинен перевантажувати екран, адже основна увага гравця зосереджена на переміщенні, боях, дослідженні рівнів і взаємодії з об'єктами. Тому елементи інтерфейсу розміщуються так, щоб не перекривати персонажа, ворогів і важливі частини рівня.

Основним екраном користувача (див. рис. 3.5) є ігрова сцена, на якій відображається персонаж, платформи, інтерактивні об'єкти, вороги та службові елементи інтерфейсу. Під час гри користувач отримує інформацію про кількість здоров'я, запас енергії, доступні лікування, активну зброю та здібності. Ці дані необхідні для прийняття рішень у процесі проходження: коли атакувати, коли ухилятися, коли використати лікування або здібність.

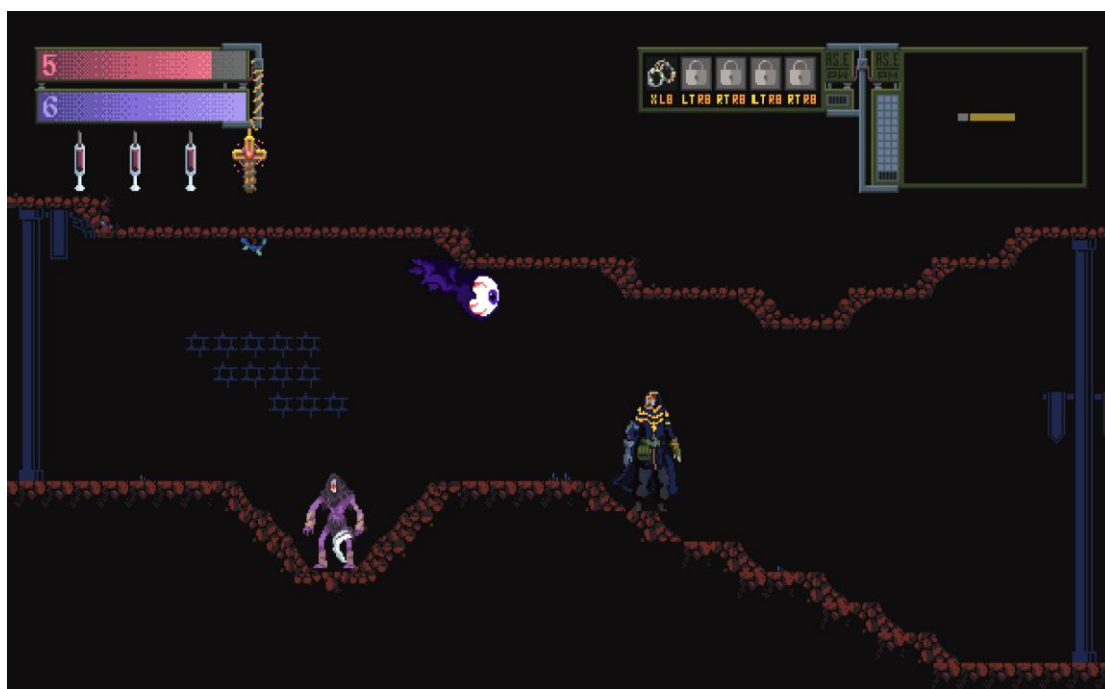


Рис. 3.5 – Основний ігровий екран застосунку Fiat Lux

Одним із головних елементів інтерфейсу є індикатор здоров'я персонажа. Він відображає поточний запас життєвих очок і дозволяє гравцю оцінювати ризик під час бою або проходження небезпечних ділянок рівня. Здоров'я змінюється

після отримання пошкоджень від ворогів, снарядів, небезпечних зон або інших перешкод. Після зміни здоров'я відповідна подія передається до інтерфейсу, завдяки чому HUD оновлюється без прямої залежності від внутрішньої логіки персонажа.

Поряд із системою здоров'я використовується система лікування (див. рис. 3.6). Лікування має обмежену кількість зарядів і виконується поступово протягом певного часу. Такий підхід не дозволяє гравцю миттєво відновити весь запас здоров'я в небезпечній ситуації, тому використання лікування потребує вибору безпечного моменту. У межах інтерфейсу доцільно показувати кількість доступних зарядів лікування або відповідну піктограму.

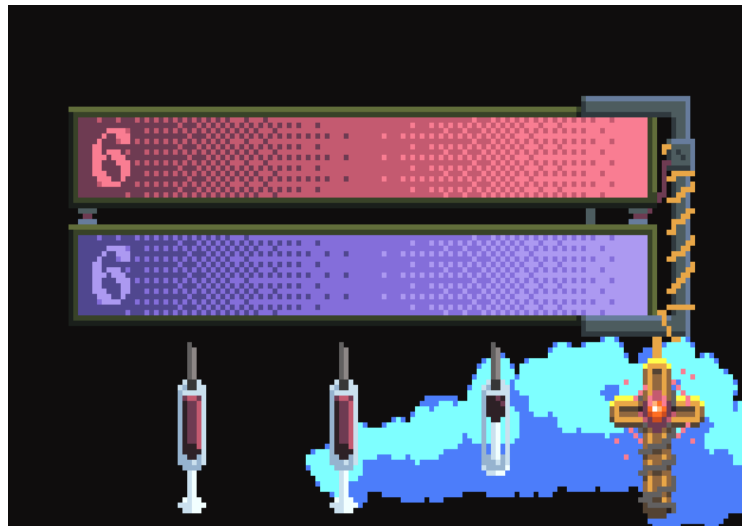


Рис. 3.6 – Відображення здоров'я та лікування персонажа

Наступним важливим елементом є індикатор енергії. Енергія використовується для виконання здібностей персонажа та частково відновлюється під час успішних бойових дій. Це створює зв'язок між атакою і використанням спеціальних можливостей: гравець не може безперервно застосовувати здібності, а повинен підтримувати ресурс через активну участь у бою. Такий підхід робить ігровий процес більш збалансованим.

Окрім здоров'я та енергії, інтерфейс відображає активну зброю і доступні здібності (див. рис. 3.7). У грі передбачено поступове відкриття особливих здібностей і зброї персонажа, тому не всі елементи доступні з початку проходження і відкриваються при піднятті необхідного елементу апгрейду. Після

розблокування нової зброї або здібності відповідні елементи можуть з'являтися в інтерфейсі, повідомляючи гравця про зміну стану прогресії.



Рис. 3.7 – Вікно відображення активної зброї та здібностей персонажа

Основою ігрового процесу є керування персонажем. Гравець може рухатися ліворуч і праворуч, стрибати (див. рис. 3.8), падати з платформ, виконувати ривок у повітрі та підкат на землі. Рух персонажа реалізовано з використанням фізичної системи Unity, зокрема Rigidbody2D та CapsuleCollider2D. Завдяки цьому переміщення враховує зіткнення з поверхнями, гравітацію та взаємодію з іншими фізичними об'єктами.

Для зручності керування реалізовано буферизацію стрибка. Якщо гравець натискає кнопку стрибка трохи раніше моменту приземлення, команда не втрачається, а виконується після контакту із землею. Також використовується короткий інтервал після сходження з платформи, протягом якого персонаж ще може виконати стрибок. Ці механіки роблять платформерну частину гри більш комфортною та зменшують кількість ситуацій, коли дія не спрацьовує через мінімальну похибку в часі натискання.



Рис. 3.8 – Переміщення персонажа і виконання стрибка

Окремою механікою переміщення є ривок (див. рис. 3.9). Він дозволяє швидко змінити позицію персонажа, подолати перешкоду або уникнути атаки ворога. Під час ривка персонаж на короткий час отримує підвищену швидкість, а гравітація тимчасово змінюється, щоб рух був різким і контрольованим. Для наземного переміщення використовується підкат, під час якого змінюється форма колайдера персонажа. Це дає змогу проходити нижчі ділянки рівня та використовувати підкат як частину бойового ухилення.



Рисунок 3.9 – Анімація виконання ривка або підкату персонажем

Бойова система є однією з ключових частин ігрового процесу (див. рис. 3.10).

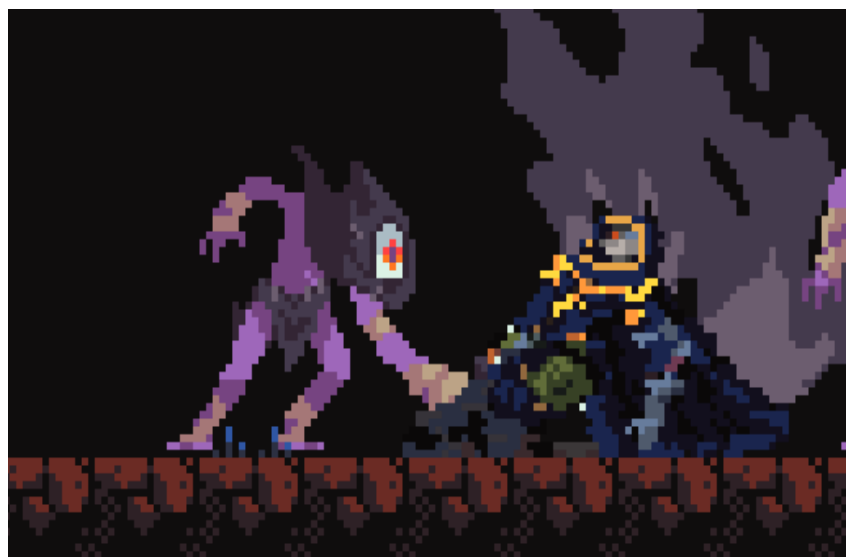


Рис. 3.10 – Бойова взаємодія персонажа з ворогом

Персонаж може атакувати ворогів у кількох напрямках: ліворуч, праворуч, вгору або вниз. Напрямок атаки визначається на основі введення гравця та

поточного напрямку персонажа. Для ближньої атаки використовуються окремі зони ураження, які перевіряють наявність ворогів у відповідному напрямку. Якщо ворог потрапляє в зону атаки, йому передається пошкодження через спільний інтерфейс отримання шкоди.

У бойовій системі також реалізовано комбо-атаки. Послідовні удари можуть мати різні множники шкоди, що заохочує гравця підтримувати темп бою. Якщо атака влучає у ворога, персонаж може відновити частину енергії. Це пов'язує ближній бій із системою здібностей і створює додатковий стимул до активної гри.

Характерною механікою є атака вниз із відскоком, яка використовується під час бою та переміщення. Якщо персонаж атакує ворога або спеціальний об'єкт знизу вниз, він отримує імпульс у протилежному напрямку. Це дозволяє не лише завдавати шкоди, а й використовувати ворогів або об'єкти як тимчасові платформи для подолання перешкод. Така механіка добре відповідає жанру *Metroidvania*, оскільки поєднує бойову систему з дослідженням рівня.

Особливе місце в ігровому процесі займають здібності персонажа. У грі передбачено магнітні та елементарні здібності, які відкриваються поступово. Магнітна взаємодія (див. рис. 3.11) дозволяє впливати на металеві об'єкти. Залежно від типу об'єкта результат може відрізнятись: легкі об'єкти отримують імпульс, важкі переміщуються між заданими точками, а статичні можуть використовуватися для зворотного ривка персонажа. Це розширює способи проходження рівнів і додає просторові головоломки.



Рис. 3.11 – Використання магнітної здібності персонажа

Елементарні здібності використовуються для взаємодії з об'єктами світу та ворогами. Наприклад, вогонь може впливати на горючі перешкоди, а лід – на

об'єкти, які підтримують замороження (див. рис. 3.12). Такі механіки роблять здібності не лише бойовими інструментами, а й засобами дослідження ігрового середовища. Гравець отримує нові можливості поступово, тому після розблокування здібностей може повертатися до раніше недоступних ділянок.

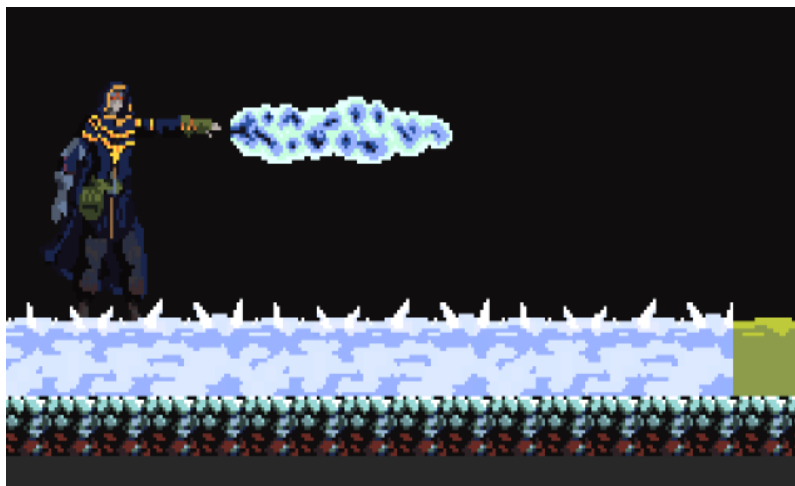


Рис. 3.12 – Взаємодія здібності з об'єктом ігрового світу

Важливою частиною ігрового процесу є система прогресії. Вона визначає, які можливості вже доступні персонажу: ривок, зброя, магнітні здібності, елементарні здібності, покращення здоров'я, енергії або лікування. Прогресія дозволяє поступово ускладнювати ігровий процес і відкривати нові сценарії проходження. Для гравця це проявляється через отримання нових предметів, здібностей або покращень, а на рівні може бути підказка щодо використання здібностей (див. рис. 3.13).



Рис. 3.13 – Повідомлення про розблокування нової можливості

Вороги в грі реагують на появу персонажа в зоні виявлення (див. рис. 3.14), можуть патрулювати, переслідувати гравця, атакувати його та отримувати пошкодження. Для звичайних ворогів використовується логіка станів, завдяки

якій поведінка змінюється залежно від ситуації. Якщо гравець перебуває поза зоною видимості, ворог може патрулювати. Якщо персонаж потрапляє в зону виявлення, ворог переходить до переслідування або атаки.



Рис. 3.14 – Поведінка ворога під час виявлення персонажа

Складнішою бойовою ситуацією є взаємодія з босом (див. рис. 3.15). Бос має кілька фаз, які залежать від кількості здоров'я, та використовує різні типи атак. Під час бою гравець повинен не лише атакувати, а й уникати небезпечних зон, снарядів і променів. Наявність фаз робить бій поступово складнішим і вимагає від гравця адаптації до нових патернів поведінки.

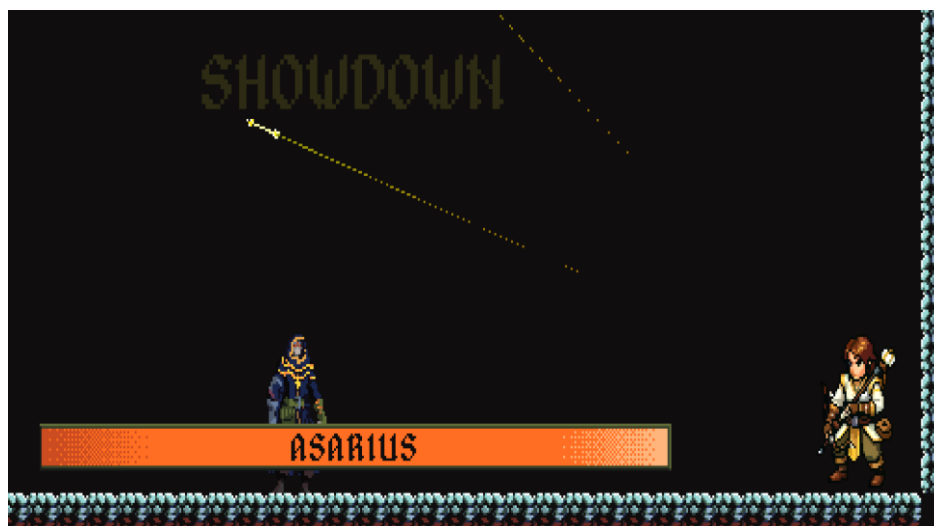


Рис. 3.15 – Бій із босом

Отже, структура інтерфейсу та ігрового процесу застосунку Fiat Lux побудована навколо швидкого зчитування стану персонажа й активної взаємодії з ігровим світом. Інтерфейс повідомляє гравцю про здоров'я, ресурси, зброю та здібності, а ігрові механіки забезпечують переміщення, бій, прогресію та

дослідження рівнів. Завдяки поєднанню руху, атак, здібностей і поведінки ворогів формується основний ігровий цикл: дослідження локації, зіткнення з перешкодами, бій, отримання нових можливостей і відкриття нових шляхів проходження.

3.3. Тестування роботи програмного продукту

Тестування ігрового застосунку Fiat Lux виконувалося з метою перевірки коректності роботи основних ігрових механік, стабільності запуску, правильності відображення інтерфейсу, реакції персонажа на введення користувача, роботи бойової системи, здібностей, ворогів і системи прогресії. Оскільки програмний продукт є інтерактивною грою, основну увагу було приділено функціональному тестуванню типових сценаріїв проходження [32].

Перевірка виконувалася у середовищі Unity Editor та у готовій збірці гри. Тестування в Unity Editor використовувалося під час розробки для швидкої перевірки окремих сцен, компонентів і параметрів об'єктів. Тестування готової збірки дозволило переконатися, що програмний продукт запускається поза середовищем розробки та зберігає основну функціональність [33].

Під час тестуванні перевірялися такі групи функцій:

- запуск програмного продукту;
- відображення головного меню та ігрового інтерфейсу;
- керування персонажем;
- рух, стрибок, ривок і підкат;
- бойова система;
- отримання пошкоджень і лікування;
- використання здібностей;
- взаємодія з об'єктами світу;
- поведінка ворогів;
- бій із босом;
- розблокування нових можливостей персонажа.

Функціональне тестування виконувалося шляхом послідовного проходження тестових сценаріїв. Для кожного сценарію визначався очікуваний результат, після чого поведінка гри перевірялася вручну під час запуску сцени. Результати тестування наведено в табл. 3.2.

Таблиця 3.2

Результати функціонального тестування ігрового застосунку

№	Тестовий сценарій	Очікуваний результат	Результат
1	Запуск виконуваного файлу гри	Застосунок відкривається без помилок	Виконано
2	Перехід до ігрової сцени	Завантажується рівень, персонаж з'являється на сцені	Виконано
3	Відображення HUD	На екрані видно здоров'я, ресурси та активні елементи інтерфейсу	Виконано
4	Натискання клавіш руху	Персонаж рухається ліворуч або праворуч	Виконано
5	Виконання стрибка	Персонаж відштовхується від поверхні та переходить у повітряний стан	Виконано
6	Виконання ривка	Персонаж швидко зміщується в напрямку руху або погляду	Виконано
7	Виконання підкату	Персонаж виконує наземний ривок зі зміною пози	Виконано
8	Атака ближнього бою	Ворог у зоні ураження отримує пошкодження	Виконано
9	Атака вниз по ворогу	Персонаж отримує відскок після влучання	Виконано
10	Отримання пошкодження персонажем	Здоров'я зменшується, інтерфейс оновлюється	Виконано
11	Використання лікування	Здоров'я поступово відновлюється	Виконано
12	Використання магнітної здібності	Металевий об'єкт реагує відповідно до свого типу	Виконано
13	Використання елементальної здібності	Об'єкт світу або ворог реагує на ефект здібності	Виконано
14	Виявлення персонажа ворогом	Ворог переходить від патрулювання до переслідування або атаки	Виконано

15	Перемога над ворогом	Ворог зникає або переходить у стан смерті	Виконано
16	Бій із босом	Бос виконує атаки та змінює поведінку відповідно до фази	Виконано
17	Розблокування здібності	Нова можливість стає доступною персонажу	Виконано
18	Перезапуск сцени або повторний запуск гри	Основні елементи гри завантажуються повторно без критичних помилок	Виконано

Спочатку доцільно перевірити правильність завантаження та ініціалізації першої ігрової сцени. Перевірка пройдено успішно, багів, помилок чи відхилень не було виявлено (див. рис. 3.16).

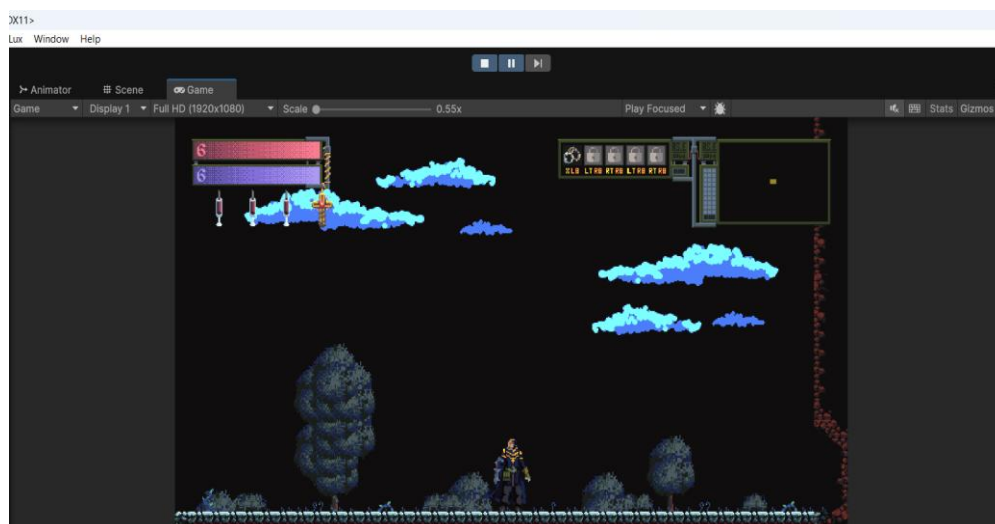


Рис. 3.16 – Перевірка завантаження ігрової сцени

Окремо перевірялося керування персонажем (див. рис. 3.17).

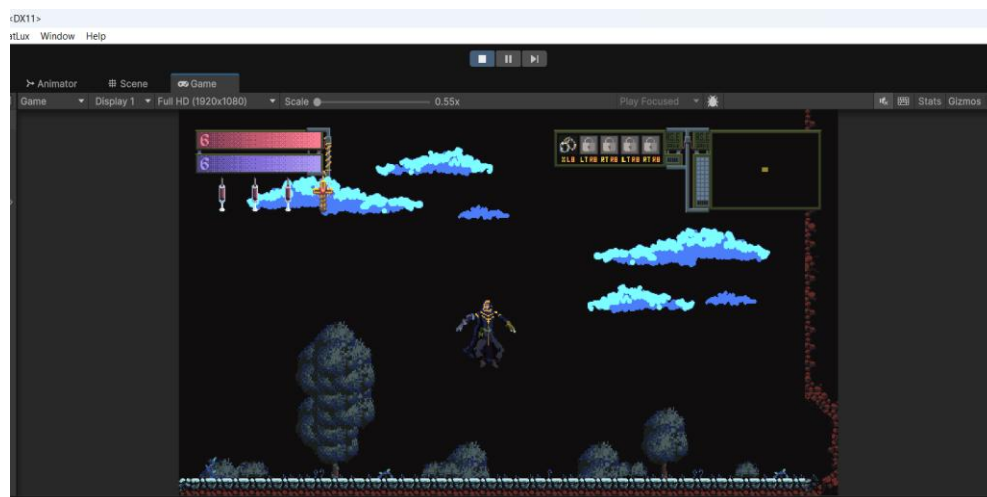


Рис. 3.17 – Тестування руху, стрибка та ривка персонажа

Під час тестування було встановлено, що персонаж коректно реагує на введення користувача: рухається в заданому напрямку, змінює напрям погляду, виконує стрибок, ривок і підкат. Також було перевірено, що під час ривка персонаж тимчасово отримує змінену швидкість, а після завершення дії повертається до звичайного фізичного стану.

Під час тестування бойової системи (рис. 3.18) перевірялося нанесення пошкоджень ворогам, визначення напрямку атаки, робота комбо та відскок після атаки вниз. Було підтверджено, що атака спрацьовує лише тоді, коли ворог перебуває в зоні ураження. Після успішного удару ворог отримує пошкодження, а в разі атаки вниз персонаж отримує імпульс відскоку.



Рис. 3.18 – Тестування бойової системи

Також було перевірено систему здоров'я та лікування (див. рис. 3.19).



Рис. 3.19 – Тестування отримання пошкодження та лікування

Після контакту з небезпечним об'єктом або атакою ворога здоров'я персонажа зменшується, а інтерфейс оновлює відповідний індикатор. Після активації лікування здоров'я відновлюється поступово, а кількість доступних зарядів зменшується. Це підтверджує правильну роботу взаємодії між системою здоров'я, лікуванням та інтерфейсом користувача. Для здібностей перевірялося, чи правильно вони реагують на введення користувача та чи взаємодіють з об'єктами світу (див. рис. 3.20).



Рис. 3.20 – Тестування використання здібностей персонажа

Магнітна здібність тестувалася на металевих об'єктах різних типів. Легкі об'єкти повинні отримувати імпульс, важкі – переміщуватися відповідно до заданих обмежень, а статичні – створювати умови для зворотного ривка персонажа. Елементарні здібності перевірялися через взаємодію з відповідними об'єктами та ворогами.

Поведінка ворогів тестувалася через наближення персонажа до зони виявлення (див. рис. 3.21). У результаті перевірки було підтверджено, що ворог переходить із пасивного стану до переслідування або атаки. Також перевірялося отримання ворогом пошкодження, його реакція на відкидання, смерть і зникнення зі сцени після завершення відповідної анімації або логіки.



Рис. 3.21 – Тестування поведінки ворога

Окремим етапом було протестовано бій із босом (див. рис. 3.22).

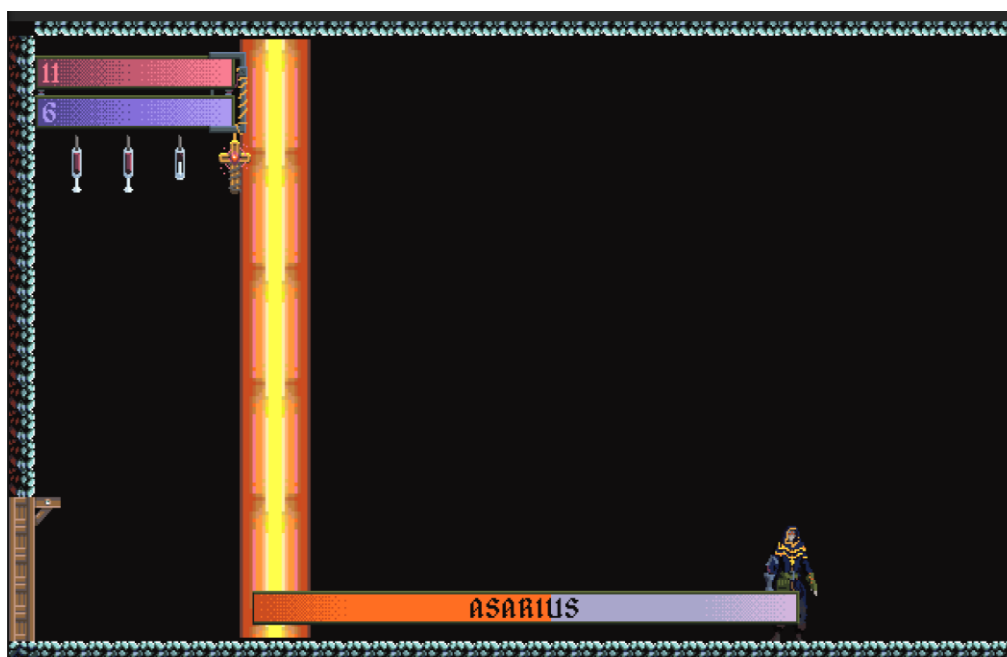


Рис. 3.22 – Тестування бою з босом

Перевірялося, чи починається бос-зустріч після входу гравця до відповідної кімнати, чи оновлюється індикатор здоров'я босу, чи виконує бос атаки та чи змінює фазу залежно від залишку здоров'я. Під час тестування також перевірялися небезпечні зони атак, снаряди, промені та вікна вразливості після завершення атаки.

Для перевірки системи прогресії було протестовано сценарії розблокування здібностей, зброї та покращень (див. рис. 3.23). Після отримання відповідного предмета або активації події нова можливість ставала доступною персонажу. Також перевірялося, що розблоковані можливості відображаються в інтерфейсі та можуть бути використані в ігровому процесі.



Рис. 3.23 – Тестування розблокування нової можливості

Крім функціонального тестування, було виконано перевірку зручності ігрового процесу. Оцінювалося, чи достатньо швидко персонаж реагує на введення, чи зрозуміло відображаються елементи HUD, чи не перекриває інтерфейс важливі об'єкти сцени, чи можна вчасно реагувати на атаки ворогів. За результатами перевірки основні механіки працюють стабільно та забезпечують очікувану взаємодію користувача з грою.

Також перевірялося, чи не виникають критичні помилки під час переходу між типовими ігровими ситуаціями: рух, атака, отримання пошкодження, лікування, використання здібності, взаємодія з ворогом і повторний запуск сцени. У процесі тестування не було виявлено помилок, які унеможливають проходження основних сценаріїв.

Підсумкові результати тестування свідчать, що реалізований програмний продукт виконує основні функції, необхідні для 2D гри жанру Metroidvania. Персонаж коректно реагує на дії користувача, інтерфейс відображає актуальний стан гри, бойова система дозволяє взаємодіяти з ворогами, здібності впливають на об'єкти світу, а система прогресії забезпечує поступове відкриття нових можливостей.

Висновки до третього розділу

У третьому розділі було описано порядок встановлення, запуску та використання ігрового застосунку Fiat Lux, а також проведено перевірку працездатності реалізованих

механік. Розглянуто два основні способи запуску програмного продукту: через готову збірку гри та через середовище Unity Editor. Встановлено, що готова збірка не потребує складної процедури інсталяції, а запуск виконується безпосередньо через виконуваний файл.

Було описано структуру інтерфейсу ігрового застосунку та його основні елементи: індикатор здоров'я, систему лікування, енергію, активну зброю, здібності та повідомлення про розблокування нових можливостей. Інтерфейс забезпечує гравця необхідною інформацією під час проходження та не перевантажує екран зайвими елементами.

Також було розглянуто особливості ігрового процесу, зокрема рух персонажа, стрибок, ривок, підкат, бойову систему, використання здібностей, взаємодію з об'єктами світу, поведінку ворогів і бій із босом. Реалізовані механіки формують основний ігровий цикл: дослідження рівня, подолання перешкод, бій із противниками, отримання нових можливостей і відкриття нових шляхів проходження.

У межах тестування було перевірено запуск гри, відображення інтерфейсу, керування персонажем, роботу руху, атак, лікування, здібностей, прогресії, ворогів і босу. За результатами тестування встановлено, що основні функції програмного продукту працюють коректно, а реалізовані механіки відповідають очікуваній поведінці. Критичних помилок, які унеможливають проходження основних сценаріїв гри, не виявлено.

Отже, розроблений ігровий застосунок є працездатним, має зрозумілий порядок запуску, функціональний інтерфейс і набір взаємопов'язаних ігрових механік. Реалізована частина програмного продукту відповідає поставленим завданням і може бути використана як основа для подальшого розширення гри новими рівнями, ворогами, здібностями, ефектами та елементами інтерфейсу.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було досягнуто поставленої мети, яка полягала у розробці 2D-гри жанру Metroidvania у стилі PixelArt з акцентом на реалізацію основних ігрових механік. Результатом роботи став ігровий застосунок Fiat Lux, створений засобами рушія Unity, у якому реалізовано керування персонажем, переміщення, бойову систему, систему здоров'я та ресурсів, розвиток здібностей, взаємодію з об'єктами середовища та перевірку працездатності основних функцій. Розроблений програмний продукт демонструє цілісний підхід до створення двовимірної гри, де окремі механіки взаємодіють між собою в межах єдиного ігрового процесу.

Першим завданням було проаналізувати особливості жанру Metroidvania та обґрунтувати вибір ігрового рушія Unity як платформи реалізації проєкту. У межах цього завдання визначено характерні риси жанру – нелінійне дослідження взаємопов'язаного світу, поступове відкриття нових зон і здібностей, поєднання платформерних та бойових механік. Аналіз аналогів (Hollow Knight, Ori and the Blind Forest, Blasphemous) дав змогу сформулювати перелік необхідних функціональних можливостей. Обрання рушія Unity обґрунтовано його розвиненою підтримкою 2D-графіки, фізичного моделювання, анімаційних систем та компонентного підходу, а використання мови C# забезпечило реалізацію логіки гри відповідно до принципів об'єктно-орієнтованого програмування.

Другим завданням було спроектувати загальну архітектуру гри та реалізувати систему керування персонажем. Для організації взаємодії між підсистемами обрано компонентно-орієнтовану архітектуру, яка дозволяє розширювати функціональність без суттєвих змін у вже реалізованих компонентах. Систему керування персонажем побудовано на основі машини станів, що забезпечує коректну обробку введення користувача та стабільну зміну станів руху: ходьба, стрибок, падіння, ривок. Реалізований підхід забезпечив природну реакцію персонажа на дії гравця та можливість поєднання рухових механік з іншими підсистемами гри.

Третім завданням було розробити бойову систему та механізми взаємодії з противниками. Реалізовано систему атак із урахуванням напрямку персонажа, нанесення та отримання пошкоджень, механіку відкидання. Поведінку противників побудовано на основі моделі станів, що включає патрулювання, пошук, переслідування та атаку. Такий підхід забезпечив базову реактивність ворогів на дії гравця та створив фундамент для подальшого ускладнення їхньої поведінки.

Четвертим завданням була реалізація системи здоров'я, ресурсів та прогресії персонажа. Створено механізми обліку здоров'я, отримання шкоди, відновлення та смерті, а також систему енергії для використання здібностей, що вводить елемент управління ресурсами в ігровий процес. Систему прогресії реалізовано через поступове відкриття нових здібностей, зброї та покращень – ривку, магнітних і елементальних ефектів, підвищення характеристик. Розблоковані можливості пов'язані з доступом до нових частин рівня та підтримують збереження стану прогресії. Реалізовано також взаємодію з об'єктами ігрового середовища: небезпечні зони, безпечні точки, інтерактивні фізичні об'єкти та магнітна механіка впливу на металеві предмети, що робить середовище повноцінною частиною ігрового процесу.

П'ятим завданням було проведення тестування розробленого програмного продукту. Перевірено запуск застосунку, роботу інтерфейсу, керування персонажем, переміщення, стрибки, ривки, атаки, лікування, використання здібностей, прогресію, взаємодію з ворогами, босом та об'єктами середовища. За результатами тестування встановлено, що основні функції гри працюють коректно, а реалізовані механіки відповідають визначеним вимогам. Критичних помилок, які унеможлилювали б проходження основних ігрових сценаріїв, не виявлено.

Таким чином, у кваліфікаційній роботі послідовно виконано всі поставлені завдання: проаналізовано предметну область та обґрунтовано вибір інструментів, спроектовано архітектуру та реалізовано систему керування персонажем, розроблено бойову систему, реалізовано механіки здоров'я, ресурсів і прогресії та

проведено тестування. Розроблений ігровий застосунок Fiat Lux відповідає визначеній меті та демонструє практичну реалізацію основних принципів побудови 2D-гри жанру Metroidvania у стилі PixelArt. Отримані результати мають практичне значення, оскільки можуть слугувати основою для подальшого розвитку повноцінного ігрового продукту та прикладом реалізації взаємопов'язаних ігрових систем у середовищі Unity.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Можаровський Я.С., Лобанчикова Н.М., Концидайло А.М. Специфіка тестування ігрового балансу та механік у Metroidvania. Матеріали Всеукраїнської науково-практичної on-line конференції здобувачів вищої освіти і молодих учених «Тенденції розвитку технологій в автоматизації, електроенергетиці, приладобудуванні та робототехніці» (присвяченої Дню науки 11-15 травня 2026 року). – Житомир: Державний університет «Житомирська політехніка», 2026. – С. 111. URL: <https://conf.ztu.edu.ua/tezy-dopovidey-naukovo-praktychnoyi-on-line-konferentsiyi-zdobuvachiv-vyshchoyi-osvity-i-molodykh-uchenykh-prysvyachenoyi-dnyu-nauky-11-15-travnya-2026-roku/>.
2. Fullerton T. Game Design Workshop: A Playcentric Approach to Creating Innovative Games. 5th ed. Boca Raton: CRC Press, 2024. 538 p.
3. Novak J. Game Development Essentials: An Introduction. 4th ed. Boston: Cengage Learning, 2023. 432 p.
4. Unity. Level up your code with game programming patterns. URL: <https://unity.com/resources/level-up-your-code-with-game-programming-patterns> (дата звернення: 10.06.2026).
5. Lemarchand R. A Playful Production Process: For Game Designers (and Everyone). – Cambridge: MIT Press, 2022. – 392 p.
6. Pressman R. S., Maxim B. R. Software Engineering: A Practitioner's Approach. 10th ed. New York: McGraw-Hill Education, 2023. 976 p.
7. Перший аналог. Hollow Knight 2017. URL: <https://steamdb.info/app/367520/>.
8. Другий аналог. Nine Sols 2024. URL: <https://steamdb.info/app/1809540/>.
9. Третій аналог. Blasphemous 2019. URL: <https://steamdb.info/app/774361/>.
10. Weber P., Störrle H. The Quest to Understand the Potential of Hidden Game Mechanics. In: Proceedings of Mensch und Computer 2024. Gesellschaft für Informatik, 2024. URL: <https://dl.gi.de/bitstreams/0c30831d-d49c-4b07-a3ec-e4fe76dba0c9/download> (дата звернення: 10.06.2026).

11. Fowler M. Refactoring: Improving the Design of Existing Code. 2nd ed. Boston: Addison-Wesley Professional, 2022. 448 p.
12. Jackson S. Mastering Unity 2D Game Development. – Birmingham: Packt Publishing, 2022. – 474 p.
13. Moore M. Basics of Game Design. 3rd ed. Boca Raton: CRC Press, 2023. 600 p.
14. Kramarzewski A., De Nucci E. Practical Game Design: A Modern and Comprehensive Guide to Video Game Design. 2nd ed. Birmingham: Packt Publishing, 2023. 446 p.
15. Borromeo N. A., Gomila Salas J. G. Hands-On Unity Game Development. 4th ed. Birmingham: Packt Publishing, 2024. 822 p.
16. Smith M., Ferns S., Murphy S. Unity Cookbook: Over 160 Recipes to Craft Your Own Masterpiece in Unity 2023. 5th ed. Birmingham: Packt Publishing, 2023. 780 p.
17. Unity Learn. Beginner Gameplay Scripting. URL: <https://learn.unity.com> (дата звернення: 10.06.2026).
18. Buttfield-Addison P., Manning J., Nugent T. Unity Development Cookbook: Real-Time Solutions from Game Development to AI. 2nd ed. Sebastopol: O'Reilly Media, 2023. 432 p.
19. Н. В. Олефіренко, А. Р. Курганський, Л. П. Остапенко, Н. О. Пономарьова. Об'єктно-орієнтоване програмування мовою C#: навч. посіб. для здобувачів першого (бакалавр.) рівня вищ. пед. освіти, пед. кадрів закл. заг. серед. освіти. Нац. пед. ун-т ім. Г. С. Сковороди. – Харків: [б. в.], 2024. – 253 с.
20. Object-oriented programming in C#. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/tutorials/oop> (дата звернення: 10.06.2026).
21. Visual Studio Code Documentation. Microsoft. URL: <https://code.visualstudio.com/docs> (дата звернення: 10.06.2026).
22. Adams E. Game Mechanics: Advanced Game Design. 2nd ed. Berkeley: New Riders, 2022. 432 p.

					ІІЗ.КР.Б – 121 – 26 – ІІЗ	
						74

23. Unity Technologies. Animation System Documentation. URL: <https://docs.unity3d.com/Manual/AnimationSection.html> (дата звернення: 10.06.2026).
24. Ferrone H. Learning C# by Developing Games with Unity. 7th ed. Birmingham: Packt Publishing, 2024. 822 p.
25. Git Documentation. Git. URL: <https://git-scm.com/doc> (дата звернення: 10.06.2026).
26. Bond J.G. Introduction to Game Design, Prototyping, and Development. 3rd ed. Boston: Addison-Wesley, 2023. 928 p.
27. Macklin C. The Game Design Reader: A Rules of Play Anthology. Cambridge: MIT Press, 2023. 560 p.
28. Unity Manual. Unity Documentation. URL: <https://docs.unity3d.com/Manual/index.html> (дата звернення: 10.06.2026).
29. Unity Technologies. 2D Game Kit Documentation. URL: <https://learn.unity.com/project/2d-game-kit> (дата звернення: 10.06.2026).
30. Unity 2D Physics. Unity Documentation. URL: <https://docs.unity3d.com/Manual/Physics2DReference.html> (дата звернення: 10.06.2026).
31. C# documentation. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/dotnet/csharp/> (дата звернення: 10.06.2026).
32. Forgács I., Kovács A. Modern Software Testing Techniques: A Practical Guide for Developers and Testers. Berkeley: Apress, 2024. 409 p.
33. Van Veenendaal E., Black R., Graham D. Foundations of Software Testing: ISTQB Certification. 5th ed. Andover: Cengage Learning, 2024. 288 p.

ДОДАТКИ

					ІІЗ.КР.Б – 121 – 26 – ІІЗ	
						76

ДОДАТОК А

Програмний код підсистеми персонажа

Лістинг А.1 – Код класу PlayerMain

```
using UnityEngine;
using UnityEngine.InputSystem;
using FiatLux.Core.Events;
using FiatLux.Contracts.Events.Player;

namespace FiatLux.Gameplay.Player
{
    public class PlayerMain : MonoBehaviour
    {
        [Header("Health")]
        [SerializeField] public float maxHealth = 6f;

        [Header("Energy")]
        [SerializeField] public float maxEnergy = 6f;
        [SerializeField] public float energyRegen = 5f;

        [Header("Ability Energy Costs")]
        [SerializeField] public float firstAbilityCost = 20f;
        [SerializeField] public float secondAbilityCost = 30f;

        private PlayerContext _ctx;
        private PlayerStateMachine _fsm;
        private PlayerFsmAnimatorTranslator _fsmTranslator;
        private PlayerAnimatorPublisher _animatorPublisher;

        private void Awake()
        {
            _ctx = new PlayerContext(this);
            _fsm = new PlayerStateMachine(_ctx);

            _ctx.Health.Initialize(maxHealth);
            _ctx.Energy.Initialize(maxEnergy, energyRegen);
            _ctx.Attack.Initialize(_ctx);

            _ctx.Dash.OnDashStarted += () =>
            _ctx.Health.GrantInvincibility(_ctx.Dash.DashInvincibilityDuration);
            _ctx.Dash.OnSlideStarted += () =>
            _ctx.Health.GrantInvincibility(_ctx.Dash.SlideInvincibilityDuration);

            _fsmTranslator = new PlayerFsmAnimatorTranslator();
            _fsmTranslator.Init(_ctx, _fsm);

            _animatorPublisher =
            gameObject.GetComponent<PlayerAnimatorPublisher>()
```

```

        ??
gameObject.AddComponent<PlayerAnimatorPublisher>();
    _animatorPublisher.Init(_ctx);
}

private void Update()
{
    _fsm.Update();

    var dir = new Vector2(_ctx.HorizontalInput,
_ctx.VerticalInput);
    EventBus.Publish(new PlayerDirectionEvent(dir));
}

private void FixedUpdate() => _fsm.FixedUpdate();

public void Move(InputAction.CallbackContext context)
{
    Vector2 v = context.ReadValue<Vector2>();
    _ctx.HorizontalInput = v.x;
    _ctx.VerticalInput    = v.y;

    if (v.x != 0)
        _ctx.Movement.SetFacingDirection(v.x);
}

public void SwitchPreset(InputAction.CallbackContext context)
{
    if (context.performed) _ctx.SwitchPresetPressed = true;
}

public void Jump(InputAction.CallbackContext context)
{
    if (context.performed)
        _ctx.JumpPressedTime = Time.time;
}

public void Dash(InputAction.CallbackContext context)
{
    if (context.performed) _ctx.DashPressed = true;
}

public void Healing(InputAction.CallbackContext context)
{
    if (context.performed) _ctx.HealPressed = true;
}

public void NormalAttack(InputAction.CallbackContext context)
{
    if (context.performed) _ctx.AttackPressed = true;
}

```

```

    }

    public void SwitchWeapon(InputAction.CallbackContext context)
    {
        if (context.performed) _ctx.SwitchWeaponPressed = true;
    }

    public void FirstAbility(InputAction.CallbackContext context)
    {
        if (context.started) { _ctx.FirstAbilityPressedTime =
Time.time; _ctx.FirstAbilityHeld = true; }
        if (context.canceled) { _ctx.FirstAbilityHeld = false; }
    }

    public void SecondAbility(InputAction.CallbackContext context)
    {
        if (context.started) { _ctx.SecondAbilityPressedTime =
Time.time; _ctx.SecondAbilityHeld = true; }
        if (context.canceled) { _ctx.SecondAbilityHeld = false; }
    }

    public void Interact(InputAction.CallbackContext context)
    {
        if (context.started) _ctx.InteractPressed = true;
    }

    private void OnDestroy() => _fsmTranslator?.Dispose();
}
}

```

Лістинг А.2 – Код класу PlayerContext

```

using UnityEngine;
using FiatLux.Gameplay.Player.Stats;
using FiatLux.Gameplay.Player.Attack;
using FiatLux.Gameplay.Player.Movement;
using FiatLux.Gameplay.Player.Healing;
using FiatLux.Gameplay.Player.Abilities;
using FiatLux.Gameplay.Player.Unlocks;

namespace FiatLux.Gameplay.Player
{
    public class PlayerContext
    {
        public readonly Rigidbody2D      Rb;
        public readonly PlayerMovement    Movement;
        public readonly PlayerDash        Dash;
        public readonly PlayerHealth      Health;
        public readonly PlayerEnergy      Energy;
        public readonly ProjectileReservoir Reservoir;
        public readonly PlayerHealing     Healing;
        public readonly PlayerProgression Progression;
    }
}

```

```

public readonly PlayerAttack      Attack;
public readonly AbilitySlotController AbilitySlots;
public readonly PlayerInteractor Interactor;

public float HorizontalInput      { get; set; }
public float VerticalInput        { get; set; }
public bool JumpPressed => JumpBuffered;
public bool DashPressed           { get; set; }
public bool AttackPressed         { get; set; }
public bool HealPressed           { get; set; }
public bool SwitchPresetPressed { get; set; }
public bool SwitchWeaponPressed { get; set; }
public bool FirstAbilityHeld      { get; set; }
public bool SecondAbilityHeld     { get; set; }

public const float AbilityBufferDuration = 0.15f;
public float FirstAbilityPressedTime { get; set; } = -1f;
public float SecondAbilityPressedTime { get; set; } = -1f;
public bool FirstAbilityBuffered => Time.time -
FirstAbilityPressedTime <= AbilityBufferDuration;
public bool SecondAbilityBuffered => Time.time -
SecondAbilityPressedTime <= AbilityBufferDuration;
public void ConsumeFirstAbility() => FirstAbilityPressedTime = -
1f;
public void ConsumeSecondAbility() => SecondAbilityPressedTime = -
1f;
public bool InteractPressed { get; set; }

public ChanneledAbility ActiveFirstChanneled { get; set; }
public ChanneledAbility ActiveSecondChanneled { get; set; }

public const float JumpBufferDuration = 0.1f;
public float JumpPressedTime { get; set; } = -1f;
public bool JumpBuffered =>
Time.time - JumpPressedTime <= JumpBufferDuration;

public void ConsumeJump() => JumpPressedTime = -1f;

public const float CoyoteDuration = 0.1f;
public float LastGroundedTime { get; set; } = -1f;
public bool CoyoteAvailable =>
Time.time - LastGroundedTime <= CoyoteDuration;

public PlayerContext(PlayerMain main)
{
    var go = main.gameObject;

    Rb          = Require<Rigidbody2D>(go);
    Movement    = Require<PlayerMovement>(go);
    Dash        = Require<PlayerDash>(go);
    Health      = Require<PlayerHealth>(go);

```

```

        Energy      = Require<PlayerEnergy>(go);
        Reservoir   = go.GetComponent<ProjectileReservoir>();
        Healing      = go.GetComponent<PlayerHealing>();
        Progression = go.GetComponent<PlayerProgression>();
        Attack       = Require<PlayerAttack>(go);
        AbilitySlots = go.GetComponent<AbilitySlotController>();
        Interactor   = go.GetComponent<PlayerInteractor>();
    }

    private static T Require<T>(GameObject go) where T : Component
    {
        var c = go.GetComponent<T>();
        if (c == null)
            Debug.LogError($"[PlayerContext] {typeof(T).Name} не
знайдено на '{go.name}'. " +
                                "Додай компонент до об'єкта гравця.");
        return c;
    }
}

```

Лістинг А.3 – Код класу PlayerStateMachine

```

using System;
using FiatLux.Gameplay.Player.States.Grounded;
using FiatLux.Gameplay.Player.States;
using FiatLux.Gameplay.Player.States.Airborne;

namespace FiatLux.Gameplay.Player
{
    public class PlayerStateMachine
    {
        public readonly GroundedState Grounded;
        public readonly AirborneState Airborne;
        public readonly DeadState Dead;

        private readonly PlayerContext _ctx;
        private IState _current;

        public event Action<IState> OnStateChanged;
        public event Action<StateBase> OnLeafEntered;
        public event Action<StateBase> OnLeafExited;

        public PlayerStateMachine(PlayerContext ctx)
        {
            _ctx = ctx;

            Grounded = new GroundedState(ctx, this);
            Airborne = new AirborneState(ctx, this);
            Dead = new DeadState(ctx, this);

            ChangeState(Airborne);
        }
    }
}

```

```

    }

    public void ChangeState(IState next)
    {
        _current?.Exit();
        _current = next;
        _current.Enter();
        OnStateChanged?.Invoke(_current);
    }

    internal void NotifyLeafEntered(StateBase leaf) =>
    OnLeafEntered?.Invoke(leaf);
    internal void NotifyLeafExited (StateBase leaf) =>
    OnLeafExited?.Invoke(leaf);

    public void Update()
    {
        if (!_ctx.Health.IsAlive && _current != Dead)
        {
            ChangeState(Dead);
            return;
        }

        _current?.Update();
    }

    public void FixedUpdate() => _current?.FixedUpdate();
}
}

```

Лістинг А.4 – Код класу GroundedState

```

using UnityEngine;
using FiatLux.Contracts.Data;

namespace FiatLux.Gameplay.Player.States.Grounded
{
    public class GroundedState : StateBase
    {
        public readonly IdleState      Idle;
        public readonly RunState       Run;
        public readonly SlideState     Slide;
        public readonly HealState      Heal;
        public readonly AttackGroundState Attack;

        private IState _currentSub;

        public GroundedState(PlayerContext ctx, PlayerStateMachine fsm) :
        base(ctx, fsm)
        {
            Idle = new IdleState(ctx, fsm, this);

```

```

        Run    = new RunState(ctx, fsm, this);
        Slide  = new SlideState(ctx, fsm, this);
        Heal   = new HealState(ctx, fsm, this);
        Attack = new AttackGroundState(ctx, fsm, this);
    }

    protected override void OnEnter()
    {
        ctx.Dash?.OnLanded();
        EnterSub(Idle);
    }

    public override void Update()
    {
        if (!ctx.Movement.IsGrounded())
        {
            fsm.ChangeState(fsm.Airborne);
            return;
        }

        if (ctx.SwitchPresetPressed)
        {
            ctx.SwitchPresetPressed = false;

            ctx.ActiveFirstChanneled?.StopChannel(ctx);
            ctx.ActiveFirstChanneled = null;
            ctx.ActiveSecondChanneled?.StopChannel(ctx);
            ctx.ActiveSecondChanneled = null;

            ctx.ConsumeFirstAbility();
            ctx.ConsumeSecondAbility();
            ctx.FirstAbilityHeld = false;
            ctx.SecondAbilityHeld = false;

            ctx.Progression.SwitchPreset();
        }

        if (ctx.SwitchWeaponPressed && !ctx.Attack.IsAttacking)
        {
            ctx.SwitchWeaponPressed = false;
            ctx.Attack.SwitchWeaponNext();
        }
        else if (ctx.SwitchWeaponPressed)
            ctx.SwitchWeaponPressed = false;

        if (ctx.InteractPressed)
        {
            ctx.InteractPressed = false;
            ctx.Interactor?.TryInteract();
        }
    }

```

```

if (ctx.Progression.CurrentPreset == AbilityPreset.Magnetism)
{
    if (ctx.FirstAbilityBuffered)
    {
        ctx.ConsumeFirstAbility();
        ctx.AbilitySlots.GetFirstAbility()?.TryPerform(ctx);
    }
    if (ctx.SecondAbilityBuffered)
    {
        ctx.ConsumeSecondAbility();
        ctx.AbilitySlots.GetSecondAbility()?.TryPerform(ctx);
    }
}
else
{
    HandleChanneledAbility(isFirst: true);
    HandleChanneledAbility(isFirst: false);
}

if (ctx.DashPressed && ctx.Progression.DashUnlocked &&
ctx.Dash.CanDash)
{
    ctx.DashPressed = false;
    ctx.ConsumeJump();
    EnterSub(Slide);
    return;
}

if (ctx.HealPressed && ctx.Healing.CanHeal())
{
    ctx.HealPressed = false;
    EnterSub(Heal);
    return;
}

if (ctx.AttackPressed)
{
    ctx.AttackPressed = false;
    EnterSub(Attack);
    return;
}

if (ctx.JumpBuffered)
{
    ctx.ConsumeJump();
    ctx.Dash?.ForceStop();
    ctx.Movement.RequestJump();
    fsm.ChangeState(fsm.Airborne);
    return;
}
_currentSub?.Update();

```

```

    }

    public override void FixedUpdate()
    {
        if (!ctx.Dash.IsDashing)
        {
            float moveInput = ctx.Attack.IsAttacking ? 0f :
ctx.HorizontalInput;
            ctx.Movement.ApplyHorizontalMove(moveInput);
        }

        _currentSub?.FixedUpdate();
    }

    public override void Exit() => _currentSub?.Exit();

    public void EnterSub(StateBase sub)
    {
        _currentSub?.Exit();
        _currentSub = sub;
        _currentSub.Enter();
        fsm.NotifyLeafEntered(sub);
    }

    private void HandleChanneledAbility(bool isFirst)
    {
        if (ctx.AbilitySlots == null) return;

        bool pressed = isFirst ? ctx.FirstAbilityBuffered :
ctx.SecondAbilityBuffered;
        bool held     = isFirst ? ctx.FirstAbilityHeld      :
ctx.SecondAbilityHeld;

        if (!held)
        {
            if (isFirst) ctx.ConsumeFirstAbility();
            else         ctx.ConsumeSecondAbility();
        }

        var ability = isFirst
            ? ctx.AbilitySlots.GetFirstChanneled()
            : ctx.AbilitySlots.GetSecondChanneled();

        var active = isFirst ? ctx.ActiveFirstChanneled :
ctx.ActiveSecondChanneled;

        if (pressed && held && ability != null && ability.CanStart())
        {

```



```

        public readonly AirDashState    AirDash;
        public readonly AttackAirState  Attack;

        private IState _currentSub;

        public AirborneState(PlayerContext ctx, PlayerStateMachine fsm) :
        base(ctx, fsm)
        {
            Jump    = new JumpState(ctx, fsm, this);
            Fall    = new FallState(ctx, fsm, this);
            AirDash = new AirDashState(ctx, fsm, this);
            Attack  = new AttackAirState(ctx, fsm, this);
        }

        protected override void OnEnter()
        {
            ctx.ConsumeJump();
            bool goingUp = ctx.Rb.linearVelocity.y > 0 ||
            ctx.Movement.JumpRequested;
            EnterSub(goingUp ? (StateBase)Jump : Fall);
        }

        public override void Update()
        {
            if (ctx.SwitchPresetPressed)
            {
                ctx.SwitchPresetPressed = false;

                ctx.ActiveFirstChanneled?.StopChannel(ctx);
                ctx.ActiveFirstChanneled = null;
                ctx.ActiveSecondChanneled?.StopChannel(ctx);
                ctx.ActiveSecondChanneled = null;

                ctx.ConsumeFirstAbility();
                ctx.ConsumeSecondAbility();
                ctx.FirstAbilityHeld = false;
                ctx.SecondAbilityHeld = false;

                ctx.Progression.SwitchPreset();
            }
            if (ctx.SwitchWeaponPressed && !ctx.Attack.IsAttacking)
            {
                ctx.SwitchWeaponPressed = false;
                ctx.Attack.SwitchWeaponNext();
            }
            else if (ctx.SwitchWeaponPressed)
                ctx.SwitchWeaponPressed = false;

            if (ctx.InteractPressed)
            {
                ctx.InteractPressed = false;
            }
        }
    }

```

```

        ctx.Interactor?.TryInteract();
    }

    if (ctx.Progression.CurrentPreset == AbilityPreset.Magnetism)
    {
        if (ctx.FirstAbilityBuffered)
        {
            ctx.ConsumeFirstAbility();
            ctx.AbilitySlots.GetFirstAbility()?.TryPerform(ctx);
        }
        if (ctx.SecondAbilityBuffered)
        {
            ctx.ConsumeSecondAbility();
            ctx.AbilitySlots.GetSecondAbility()?.TryPerform(ctx);
        }
    }
    else
    {
        HandleChanneledAbility(isFirst: true);
        HandleChanneledAbility(isFirst: false);
    }

    if (ctx.Movement.IsGrounded())
    {
        fsm.ChangeState(fsm.Grounded);
        return;
    }

    if (ctx.DashPressed && ctx.Progression.DashUnlocked &&
    ctx.Dash.CanDash)
    {
        ctx.DashPressed = false;
        ctx.ConsumeJump();
        EnterSub(AirDash);
        return;
    }

    if (ctx.AttackPressed)
    {
        ctx.AttackPressed = false;
        EnterSub(Attack);
        return;
    }

    if (ctx.JumpBuffered && ctx.CoyoteAvailable)
    {
        ctx.ConsumeJump();
        ctx.LastGroundedTime = -1f;
        ctx.Movement.RequestJump();
        EnterSub(Jump);
        return;
    }

```

```

        }

        _currentSub?.Update();
    }

    public override void FixedUpdate()
    {
        if (!ctx.Dash.IsDashing)
            ctx.Movement.ApplyHorizontalMove(ctx.HorizontalInput);

        _currentSub?.FixedUpdate();
    }

    public override void Exit() => _currentSub?.Exit();

    public void EnterSub(StateBase sub)
    {
        _currentSub?.Exit();
        _currentSub = sub;
        _currentSub.Enter();
        fsm.NotifyLeafEntered(sub);
    }

    private void HandleChanneledAbility(bool isFirst)
    {
        if (ctx.AbilitySlots == null) return;

        bool pressed = isFirst ? ctx.FirstAbilityBuffered :
ctx.SecondAbilityBuffered;
        bool held     = isFirst ? ctx.FirstAbilityHeld      :
ctx.SecondAbilityHeld;

        if (!held)
        {
            if (isFirst) ctx.ConsumeFirstAbility();
            else         ctx.ConsumeSecondAbility();
        }

        var ability = isFirst
            ? ctx.AbilitySlots.GetFirstChanneled()
            : ctx.AbilitySlots.GetSecondChanneled();

        var active = isFirst ? ctx.ActiveFirstChanneled :
ctx.ActiveSecondChanneled;

        if (pressed && held && ability != null && ability.CanStart())
        {
            if (isFirst) ctx.ConsumeFirstAbility();

```



```

[Header("Components")]
[SerializeField] Rigidbody2D rb;
[SerializeField] SpriteRenderer spriteRenderer;
[SerializeField] private CapsuleCollider2D capsuleCollider;

[Header("Movement")]
[SerializeField] float moveSpeed = 5f;
[SerializeField] float jumpForce = 10f;
[SerializeField] float fallMultiplier = 2.5f;
private float _speedMultiplier = 1f;

[Header("Grounding")]
[SerializeField] LayerMask groundLayer;
[SerializeField] Transform groundCheck;
[SerializeField] float groundCheckRadius = 0.2f;

[Header("Reverse Dash")]
[SerializeField] private float reverseDashCooldown = 1f;
[SerializeField] private TrailRenderer tr;

public event System.Action OnJumped;
public event System.Action OnLanded;
public event System.Action OnKnockbackStarted;
public event System.Action OnKnockbackEnded;

private bool canReverseDash = true;
private bool isDashing;
private float horizontal;
private float vertical;
private bool isGrounded;
private bool jumpRequested;
private bool wasGroundedLastFrame;
private bool canMoveDuringKnockback = true;
private float facingDirection = 1f;

public float FacingDirection => facingDirection;
public Vector2 Velocity => rb.linearVelocity;
public bool JumpRequested => jumpRequested;
public void SetSpeedMultiplier(float multiplier) =>
SetChannelingSpeedMultiplier(multiplier);

private void Awake()
{
    if (rb == null) rb =
GetComponent<Rigidbody2D>();
    if (spriteRenderer == null) spriteRenderer =
GetComponent<SpriteRenderer>();
    if (capsuleCollider == null) capsuleCollider =
GetComponent<CapsuleCollider2D>();
}

```

```

private Coroutine _knockbackCoroutine;

private void FixedUpdate()
{
    if (isDashing) return;

    wasGroundedLastFrame = isGrounded;
    isGrounded = Physics2D.OverlapCircle(groundCheck.position,
groundCheckRadius, groundLayer);

    if (canMoveDuringKnockback && jumpRequested && isGrounded)
    {
        rb.linearVelocity = new Vector2(rb.linearVelocity.x,
jumpForce);
        jumpRequested = false;
        OnJumped?.Invoke();
        EventBus.Publish(new PlayerJumpedEvent());
    }

    if (rb.linearVelocity.y < 0)
        rb.linearVelocity += Vector2.up * Physics2D.gravity.y *
(fallMultiplier - 1) * Time.fixedDeltaTime;

    if (!wasGroundedLastFrame && isGrounded)
    {
        jumpRequested = false;
        float impact = Mathf.Abs(rb.linearVelocity.y);
        OnLanded?.Invoke();
        EventBus.Publish(new PlayerLandedEvent(impact));
    }
}

public void ApplyHorizontalMove(float input)
{
    if (!canMoveDuringKnockback) return;
    if (isDashing) return;
    rb.linearVelocity = new Vector2(input * moveSpeed *
_speedMultiplier, rb.linearVelocity.y);
}

public void SetFacingDirection(float dir)
{
    if (dir != 0) facingDirection = Mathf.Sign(dir);
    UpdateFacingDirection();
}

public void RequestJump()
{
    if (isGrounded && !isDashing)

```

```

        jumpRequested = true;
    }

    public bool IsGrounded() => isGrounded;
    public bool GetIsDashing() => isDashing;

    public void ApplyKnockback(Vector2 force) => ApplyKnockback(force,
0.15f);

    public void ApplyKnockback(Vector2 force, float stunDuration)
    {
        if (_knockbackCoroutine != null)
        {
            StopCoroutine(_knockbackCoroutine);
            canMoveDuringKnockback = true;
        }
        rb.linearVelocity = Vector2.zero;
        rb.AddForce(force, ForceMode2D.Impulse);
        _knockbackCoroutine =
StartCoroutine(KnockbackRoutine(stunDuration));
    }

    private IEnumerator KnockbackRoutine(float stunDuration)
    {
        canMoveDuringKnockback = false;
        OnKnockbackStarted?.Invoke();
        EventBus.Publish(new
PlayerKnockbackEvent(KnockbackEventType.Started));
        yield return new WaitForSeconds(stunDuration);
        OnKnockbackEnded?.Invoke();
        EventBus.Publish(new
PlayerKnockbackEvent(KnockbackEventType.Ended));
        canMoveDuringKnockback = true;
    }

    public void PerformReverseDash(Vector2 direction, float power,
float duration)
    {
        if (!isDashing && canReverseDash)
            StartCoroutine(ReverseDashCoroutine(direction, power,
duration));
    }

    private IEnumerator ReverseDashCoroutine(Vector2 direction, float
power, float duration)
    {
        canReverseDash = false;
        isDashing = true;
        float gravOG = rb.gravityScale;
        rb.gravityScale = 0f;
        rb.linearVelocity = direction * power;
    }

```

```

        if (tr != null) tr.emitting = true;

        yield return new WaitForSeconds(duration);

        if (tr != null) tr.emitting = false;
        rb.gravityScale = gravOG;
        isDashing = false;

        yield return new WaitForSeconds(reverseDashCooldown);
        canReverseDash = true;
    }

    public float    GetHorizontalInput()    => horizontal;
    public float    GetVerticalInput()      => vertical;
    public float    GetFacingDirection()   => facingDirection;
    public Vector2  GetVelocity()           => rb.linearVelocity;
    public void     SetVelocity(Vector2 v) => rb.linearVelocity = v;

    public void SetHorizontalInput(float input)
    {
        horizontal = input;
        if (horizontal != 0) SetFacingDirection(horizontal);
    }

    public void SetVerticalInput(float input) => vertical = input;

    public void SetChannelingSpeedMultiplier(float multiplier) =>
    _speedMultiplier = multiplier;

    private void UpdateFacingDirection()
    {
        if (spriteRenderer != null)
            spriteRenderer.flipX = facingDirection < 0;
    }

    private void OnDrawGizmosSelected()
    {
        if (groundCheck != null)
        {
            Gizmos.color = Color.green;
            Gizmos.DrawWireSphere(groundCheck.position,
groundCheckRadius);
        }
    }
}

```

Лістинг А.7 – Код класу PlayerDash

```

using UnityEngine;
using System.Collections;

```

```

using FiatLux.Core.Events;
using FiatLux.Contracts.Events.Player;

namespace FiatLux.Gameplay.Player.Movement
{
    public class PlayerDash : MonoBehaviour
    {
        [Header("Air Dash")]
        [SerializeField] private float dashPower = 24f;
        [SerializeField] private float dashTime = 0.2f;
        [SerializeField] private float dashCooldown = 0f;
        [SerializeField] private float dashInvincibilityDuration = 0.2f;

        [Header("Slide (Ground Dash)")]
        [SerializeField] private float slidePower = 24f;
        [SerializeField] private float slideTime = 0.2f;
        [SerializeField] private float slideCooldown = 1f;
        [SerializeField] private float slideInvincibilityDuration = 0.2f;

        public float DashInvincibilityDuration =>
dashInvincibilityDuration;
        public float SlideInvincibilityDuration =>
slideInvincibilityDuration;

        [SerializeField] private TrailRenderer tr;

        private Rigidbody2D _rb;
        private CapsuleCollider2D _col;
        private PlayerMovement _movement;

        public bool IsDashing { get; private set; }
        public bool CanDash { get; private set; } = true;
        private bool _dashedInAir;

        private Coroutine _activeRoutine;
        private Coroutine _cooldownRoutine;
        private Vector2 _colliderSizeOG;
        private Vector2 _colliderOffsetOG;
        private CapsuleDirection2D _colliderDirectionOG;
        private float _gravityOG;
        private bool _isSlide;

        public event System.Action OnDashStarted;
        public event System.Action OnDashEnded;
        public event System.Action OnSlideStarted;
        public event System.Action OnSlideEnded;
        public event System.Action OnDashCooldownReady;

        private void Awake()
        {
            _rb = GetComponent<Rigidbody2D>();

```

```

        _col = GetComponent<CapsuleCollider2D>();
        _movement = GetComponent<PlayerMovement>();

        _gravityOG = _rb.gravityScale;
        _colliderSizeOG = _col.size;
        _colliderOffsetOG = _col.offset;
        _colliderDirectionOG = _col.direction;
    }

    public void OnLanded() => _dashedInAir = false;

    public void PerformAirDash(float facingDirection)
    {
        if (!CanDash || _dashedInAir) return;
        _isSlide = false;
        _activeRoutine =
StartCoroutine(AirDashRoutine(facingDirection));
    }

    public void PerformSlide(float facingDirection)
    {
        if (!CanDash) return;
        _isSlide = true;
        _activeRoutine =
StartCoroutine(SlideRoutine(facingDirection));
    }

    private IEnumerator AirDashRoutine(float dir)
    {
        OnDashStarted?.Invoke();
        EventBus.Publish(new
PlayerDashEvent(DashEventType.AirDashStarted));
        CanDash = false;
        IsDashing = true;
        _dashedInAir = true;

        _gravityOG = _rb.gravityScale;
        _rb.gravityScale = 0f;
        _rb.linearVelocity = new Vector2(dir * dashPower, 0f);
        if (tr) tr.emitting = true;
        OnDashStarted?.Invoke();

        yield return new WaitForSeconds(dashTime);
        OnDashEnded?.Invoke();
        EventBus.Publish(new
PlayerDashEvent(DashEventType.AirDashEnded));
        RestoreState();
        IsDashing = false;
        OnDashEnded?.Invoke();
    }

```

```

        _cooldownRoutine =
StartCoroutine(DashCooldownRoutine(dashCooldown));
    }

    private IEnumerator SlideRoutine(float dir)
    {
        OnSlideStarted?.Invoke();
        EventBus.Publish(new
PlayerDashEvent(DashEventType.SlideStarted));
        CanDash = false;
        IsDashing = true;

        _gravityOG = _rb.gravityScale;
        _colliderSizeOG = _col.size;
        _colliderOffsetOG = _col.offset;
        _colliderDirectionOG = _col.direction;

        _col.size = new Vector2(1f, 0.5f);
        _col.offset = new Vector2(_colliderOffsetOG.x, -0.25f);
        _col.direction = CapsuleDirection2D.Horizontal;
        _rb.gravityScale = 0f;
        _rb.linearVelocity = new Vector2(dir * slidePower, 0f);
        if (tr) tr.emitting = true;
        OnSlideStarted?.Invoke();

        yield return new WaitForSeconds(slideTime);
        OnSlideEnded?.Invoke();
        EventBus.Publish(new
PlayerDashEvent(DashEventType.SlideEnded));
        RestoreState();
        IsDashing = false;
        OnSlideEnded?.Invoke();

        _cooldownRoutine =
StartCoroutine(DashCooldownRoutine(slideCooldown));
    }

    public void PerformReverseDash(Vector2 direction, float power,
float duration)
    {
        if (!IsDashing)
            StartCoroutine(ReverseDashRoutine(direction, power,
duration));
    }

    private IEnumerator ReverseDashRoutine(Vector2 dir, float power,
float duration)
    {
        IsDashing = true;
        float gravOG = _rb.gravityScale;
        _rb.gravityScale = 0f;

```

```

        _rb.linearVelocity = dir * power;
        if (tr) tr.emitting = true;

        yield return new WaitForSeconds(duration);

        if (tr) tr.emitting = false;
        _rb.gravityScale = gravOG;
        IsDashing = false;
    }

    public void ForceStop()
    {
        if (_activeRoutine == null) return;

        StopCoroutine(_activeRoutine);
        _activeRoutine = null;

        RestoreState();
        IsDashing = false;

        if (_cooldownRoutine != null) StopCoroutine(_cooldownRoutine);
        _cooldownRoutine = StartCoroutine(DashCooldownRoutine());
    }
    private IEnumerator DashCooldownRoutine(float? overrideDuration =
null)
    {
        CanDash = false;
        yield return new WaitForSeconds(overrideDuration ??
dashCooldown);
        CanDash = true;
        OnDashCooldownReady?.Invoke();
        EventBus.Publish(new
PlayerDashEvent(DashEventType.CooldownReady));
    }
    private void RestoreState()
    {
        if (tr) tr.emitting = false;
        _rb.gravityScale = _gravityOG;

        if (_isSlide)
        {
            _col.size = _colliderSizeOG;
            _col.offset = _colliderOffsetOG;
            _col.direction = _colliderDirectionOG;
        }
    }
}
}
}

```

ДОДАТОК Б

Програмний код бойової системи та зброї

Лістинг Б.1 – Код класу PlayerAttack

```
using UnityEngine;
using FiatLux.Core.Events;
using FiatLux.Contracts.Events.Player;
using FiatLux.Gameplay.Status;

namespace FiatLux.Gameplay.Player.Attack
{
    public class PlayerAttack : MonoBehaviour
    {
        [SerializeField] private WeaponManager weaponManager;

        public bool IsAttacking { get; private set; }

        private PlayerContext _ctx;
        private float _attackEndTime;
        private float _comboWindowEnd;
        private float _nextAttackTime;
        private int _comboIndex;

        public WeaponBase CurrentWeapon => weaponManager.Current;
        public void SwitchWeaponNext() => weaponManager.SwitchNext();
        public void SwitchWeaponPrev() => weaponManager.SwitchPrev();

        public void Initialize(PlayerContext ctx) => _ctx = ctx;

        public event System.Action<Vector2Int, int> OnAttackPerformed;

        private void Update()
        {
            if (IsAttacking && Time.time >= _attackEndTime)
                IsAttacking = false;

            if (_comboIndex > 0 && Time.time > _comboWindowEnd)
                _comboIndex = 0;
        }

        public void RequestAttack()
        {
            if (Time.time < _nextAttackTime) return;
            if (_ctx == null) return;

            var statusHandler = GetComponent<StatusHandler>();
            var freeze = statusHandler?.GetStatus<FreezeStatus>();
            if (freeze != null && freeze.IsMaxed) return;
        }
    }
}
```

```

var weapon = weaponManager.Current;
if (weapon == null) return;

Vector2Int dir    = AttackDirectionResolver.Resolve(_ctx);
var          result = weapon.Perform(_ctx, dir, _comboIndex);

if (weapon.IsMelee && weapon is MeleeWeapon melee)
{
    KnockbackResolver.Apply(_ctx, result, melee.KnockbackCfg);

    if (result.HitSomething)
        _ctx.Energy.Restore(melee.EnergyPerMeleeHit);
}

_nextAttackTime = Time.time + weapon.AttackCooldown;
_attackEndTime  = Time.time + weapon.AttackCooldown;
IsAttacking     = true;

if (weapon.MaxComboIndex > 0)
{
    _comboIndex = (_comboIndex + 1) % (weapon.MaxComboIndex +
1);
    _comboWindowEnd = Time.time + weapon.AttackCooldown +
weapon.ComboWindow;
}

    OnAttackPerformed?.Invoke(dir, _comboIndex);
    EventBus.Publish(new PlayerAttackedEvent(dir, _comboIndex,
result.HitSomething));
}
}
}

```

Лістинг Б.2 – Код класу WeaponManager

```

using UnityEngine;
using FiatLux.Core.Events;
using FiatLux.Contracts.Events.Player;
using FiatLux.Contracts.Data;
using FiatLux.Gameplay.Player.Stats;

namespace FiatLux.Gameplay.Player.Attack
{
    public class WeaponManager : MonoBehaviour
    {
        [SerializeField] private WeaponBase[] weapons;
        [SerializeField] private PlayerProgression progression;

        private int _currentIndex;
    }
}

```

```

public WeaponBase Current => weapons[_currentIndex];
public event System.Action<WeaponBase> OnWeaponChanged;

private void Awake()
{
    if (progression == null)
        progression = GetComponentInParent<PlayerProgression>();

    _currentIndex = FindFirstUnlocked();
    ApplyActiveVisual();

    if (progression != null)
        progression.OnWeaponUnlocked += HandleWeaponUnlocked;
}

private void OnDestroy()
{
    if (progression != null)
        progression.OnWeaponUnlocked -= HandleWeaponUnlocked;
}

private void Start()
{
    OnWeaponChanged?.Invoke(Current);
    EventBus.PublishLatest(new
PlayerWeaponChangedEvent(Current.Type, Current.IsMelee));
}

public void SwitchNext() => Switch(+1);
public void SwitchPrev() => Switch(-1);

public bool TrySwitchTo(WeaponType type)
{
    if (!progression.IsWeaponUnlocked(type)) return false;

    for (int i = 0; i < weapons.Length; i++)
    {
        if (weapons[i].Type == type)
        {
            if (i == _currentIndex) return true;
            _currentIndex = i;
            ApplyActiveVisual();
            OnWeaponChanged?.Invoke(Current);
            EventBus.Publish(new
PlayerWeaponChangedEvent(Current.Type, Current.IsMelee));
            return true;
        }
    }
    return false;
}

```

```

private void Switch(int dir)
{
    int next = FindNextUnlocked(_currentIndex, dir);
    if (next == _currentIndex) return;
    _currentIndex = next;
    ApplyActiveVisual();
    OnWeaponChanged?.Invoke(Current);
    EventBus.Publish(new PlayerWeaponChangedEvent(Current.Type,
Current.IsMelee));
}

private int FindFirstUnlocked()
{
    for (int i = 0; i < weapons.Length; i++)
        if (progression == null ||
progression.IsWeaponUnlocked(weapons[i].Type))
            return i;
    return 0;
}

private int FindNextUnlocked(int from, int dir)
{
    int n = weapons.Length;
    for (int step = 1; step <= n; step++)
    {
        int i = ((from + dir * step) % n + n) % n;
        if (progression.IsWeaponUnlocked(weapons[i].Type))
            return i;
    }
    return from;
}

private void ApplyActiveVisual()
{
    for (int i = 0; i < weapons.Length; i++)
        weapons[i].gameObject.SetActive(i == _currentIndex);
}

private void HandleWeaponUnlocked(WeaponType type)
{
}

}
}

```

Лістинг Б.3 – Код класу MeleeWeapon

```

using UnityEngine;
using System.Collections.Generic;

```

```

using FiatLux.Contracts.Data;
using FiatLux.Contracts.Interfaces;

namespace FiatLux.Gameplay.Player.Attack
{
    public class MeleeWeapon : WeaponBase
    {
        [Header("Stats")]
        [SerializeField] private float damage = 1f;
        [SerializeField] private float attackCooldown = 0.35f;
        [SerializeField] private float comboWindow = 0.4f;
        [SerializeField] private int maxComboIndex = 2;
        [SerializeField] private LayerMask enemyLayer;
        [SerializeField] private WeaponType type = WeaponType.Chain;

        [Header("Energy")]
        [SerializeField] private float energyPerMeleeHit = 1f;

        [Header("Combo damage multipliers")]
        [SerializeField] private float[] comboMultipliers = { 1f, 1.1f,
1.4f };

        [Header("Knockback")]
        [SerializeField] private KnockbackConfig knockbackConfig = new()
        {
            EnemyPush = 5f, PlayerRecoil = 3f, PogoForce = 50f,
KnockbackStunDuration = 0.25f
        };

        public float EnergyPerMeleeHit => energyPerMeleeHit;
        public KnockbackConfig KnockbackCfg => knockbackConfig;

        [Header("Attack Colliders (N/S/E/W)")]
        [SerializeField] private Collider2D northCollider;
        [SerializeField] private Collider2D southCollider;
        [SerializeField] private Collider2D eastCollider;
        [SerializeField] private Collider2D westCollider;

        public override float Damage => damage;
        public override float AttackCooldown => attackCooldown;
        public override float ComboWindow => comboWindow;
        public override int MaxComboIndex => maxComboIndex;
        public override bool IsMelee => true;
        public override WeaponType Type => type;

        private readonly List<Collider2D> _buffer = new();

        public override AttackResult Perform(PlayerContext ctx, Vector2Int
direction, int comboIndex)
        {
            var col = GetCollider(direction);

```

```

        if (col == null) return AttackResult.Miss(direction);

        var filter = new ContactFilter2D();
        filter.SetLayerMask(enemyLayer);
        filter.useTriggers = true;

        _buffer.Clear();
        col.Overlap(filter, _buffer);

        if (_buffer.Count == 0) return AttackResult.Miss(direction);

        float mult = (direction == Vector2Int.down || comboIndex >=
        comboMultipliers.Length)
            ? 1f
            : comboMultipliers[comboIndex];
        float finalDamage = damage * mult;

        foreach (var enemy in _buffer)
        {
            if (enemy.TryGetComponent<IDamageable>(out var dmg))
                dmg.TakeDamage(finalDamage);
        }

        return new AttackResult(direction, _buffer.ToArray());
    }

    private Collider2D GetCollider(Vector2Int dir)
    {
        if (dir == Vector2Int.up)    return northCollider;
        if (dir == Vector2Int.down)  return southCollider;
        if (dir == Vector2Int.right) return eastCollider;
        if (dir == Vector2Int.left)  return westCollider;
        return null;
    }
}

```

Лістинг Б.4 – Код класу RangedWeapon

```

using UnityEngine;
using FiatLux.Contracts.Data;

namespace FiatLux.Gameplay.Player.Attack
{
    public class RangedWeapon : WeaponBase
    {
        [Header("Stats")]
        [SerializeField] private float damage = 2f;
        [SerializeField] private float attackCooldown = 0.5f;
        [SerializeField] private float energyCost = 10f;
        [SerializeField] private float projectileSpeed = 14f;
    }
}

```

```

[SerializeField] private WeaponType type = WeaponType.Bow;
[Header("Refs")]
[SerializeField] private Projectile projectilePrefab;
[SerializeField] private Transform spawnPoint;

public override float Damage          => damage;
public override float AttackCooldown => attackCooldown;
public override float ComboWindow    => 0f;
public override int   MaxComboIndex  => 0;
public override bool  IsMelee        => false;

public float EnergyCost => energyCost;

public override WeaponType Type => WeaponType.Bow;

public override AttackResult Perform(PlayerContext ctx, Vector2Int
direction, int comboIndex)
{
    if (!ctx.Energy.TrySpend(energyCost))
        return AttackResult.Miss(direction);

    var proj = Instantiate(projectilePrefab, spawnPoint.position,
Quaternion.identity);
    proj.Launch(direction, projectileSpeed, damage, gameObject);

    return new AttackResult(direction,
System.Array.Empty<Collider2D>());
}
}
}

```

Лістинг Б.5 – Код класу KnockbackResolver

```

using UnityEngine;
using FiatLux.Contracts.Interfaces;

namespace FiatLux.Gameplay.Player.Attack
{
    public static class KnockbackResolver
    {
        public static void Apply(PlayerContext ctx, AttackResult result,
KnockbackConfig cfg)
        {
            foreach (var enemy in result.Targets)
            {
                Vector2 dir = ((Vector2)enemy.transform.position -
(Vector2)ctx.Rb.transform.position).normalized;

                if (enemy.TryGetComponent<IKnockbackReceiver>(out var ai))
                {
                    ai.ApplyKnockback(dir * cfg.EnemyPush);
                }
            }
        }
    }
}

```

```

        }
        else if (enemy.TryGetComponent<Rigidbody2D>(out var rb))
        {
            rb.linearVelocity = Vector2.zero;
            rb.AddForce(dir * cfg.EnemyPush, ForceMode2D.Impulse);
        }
    }

    if (!result.HitSomething) return;

    if (result.Direction == Vector2Int.down)
    {
        ctx.Rb.linearVelocity = new
Vector2(ctx.Rb.linearVelocity.x, 0f);
        ctx.Rb.AddForce(Vector2.up * cfg.PogoForce,
ForceMode2D.Impulse);
    }
    else
    {
        Vector2 recoil = -(Vector2)result.Direction *
cfg.PlayerRecoil;
        ctx.Movement.ApplyKnockback(recoil,
cfg.KnockbackStunDuration);
    }
}

[System.Serializable]
public struct KnockbackConfig
{
    public float EnemyPush;
    public float PlayerRecoil;
    public float PogoForce;
    public float KnockbackStunDuration;
}
}

```

ДОДАТОК В

Програмний код системи здібностей і статусних ефектів

Лістинг В.1 – Код класу AbilityBase

```
using UnityEngine;
using System.Collections;
using FiatLux.Core.Events;
using FiatLux.Contracts.Events.Player;

namespace FiatLux.Gameplay.Player.Abilities
{
    public abstract class AbilityBase : MonoBehaviour
    {
        [SerializeField] protected float cooldown = 1f;
        [SerializeField] protected float energyCost = 20f;

        protected bool _onCooldown;

        public bool OnCooldown => _onCooldown;
        public float EnergyCost => energyCost;

        public void TryPerform(PlayerContext ctx)
        {
            Debug.Log($"TryPerform {GetType().Name}: onCD={_onCooldown},
energy={ctx.Energy.Current}/{energyCost}");
            if (_onCooldown) { Debug.Log("→ blocked: cooldown"); return; }
            if (!ctx.Energy.TrySpend(energyCost)) { Debug.Log("→ blocked:
not enough energy"); return; }
            EventBus.Publish(new
PlayerAbilityActivatedEvent(GetType().Name, isStart: true));
            StartCoroutine(Execute(ctx));
        }
        protected IEnumerator CooldownTimer()
        {
            yield return new WaitForSeconds(cooldown);
            _onCooldown = false;
        }
        protected abstract IEnumerator Execute(PlayerContext ctx);
    }
}
```

Лістинг В.2 – Код класу ChanneledAbility

```
using UnityEngine;
using System.Collections;
using FiatLux.Core.Events;
using FiatLux.Contracts.Events.Player;
```

```

namespace FiatLux.Gameplay.Player.Abilities
{
    public abstract class ChanneledAbility : MonoBehaviour
    {
        [Header("Cooldown")]
        [SerializeField] protected float cooldown = 2f;

        [Header("Channeling")]
        [SerializeField] protected float energyPerSecond = 8f;
        [SerializeField] protected float speedMultiplier = 0.4f;
        [SerializeField] protected float damagePerSecond = 3f;
        [SerializeField] protected float damageTickRate = 0.15f;

        [Header("Hitbox Objects")]
        [SerializeField] protected Collider2D northHitbox;
        [SerializeField] protected Collider2D eastHitbox;
        [SerializeField] protected Collider2D westHitbox;
        [SerializeField] protected Collider2D southAirHitbox;
        [SerializeField] protected Collider2D southGroundHitbox;

        [SerializeField] protected LayerMask hitLayer;

        protected bool _onCooldown;
        protected bool _isChanneling;
        private float _damageTimer;
        private PlayerContext _ctx;

        public bool IsChanneling => _isChanneling;

        public bool CanStart() => !_onCooldown && !_isChanneling;

        public void StartChannel(PlayerContext ctx)
        {
            if (!CanStart()) return;
            _ctx = ctx;
            _isChanneling = true;
            _damageTimer = 0f;
            EventBus.Publish(new
PlayerAbilityActivatedEvent(GetType().Name, isStart: true));
            OnChannelStart(ctx);
        }

        public void UpdateChannel(PlayerContext ctx)
        {
            if (!_isChanneling) return;

            if (!ctx.Energy.TrySpend(energyPerSecond * Time.deltaTime))
            { StopChannel(ctx); return; }

            ctx.Movement.SetChannelingSpeedMultiplier(speedMultiplier);
        }
    }
}

```

```

        _damageTimer += Time.deltaTime;
        if (_damageTimer >= damageTickRate)
        {
            _damageTimer = 0f;
            var hits = GetCurrentHits(ctx);
            if (hits.Length > 0)
                OnDamageTick(hits, ctx, damagePerSecond *
damageTickRate);
        }

        OnChannelUpdate(ctx);
    }

    public void StopChannel(PlayerContext ctx)
    {
        if (!_isChanneling) return;
        _isChanneling = false;
        ctx.Movement.SetChannelingSpeedMultiplier(1f);
        EventBus.Publish(new
PlayerAbilityActivatedEvent(GetType().Name, isStart: false));
        OnChannelStop(ctx);
        StartCoroutine(CooldownRoutine());
    }

    protected Collider2D[] GetCurrentHits(PlayerContext ctx)
    {
        Collider2D hitbox = ResolveHitbox(ctx);
        if (hitbox == null) return System.Array.Empty<Collider2D>();

        Bounds b = hitbox.bounds;
        return Physics2D.OverlapBoxAll(b.center, b.size, 0f,
hitLayer);
    }

    private Collider2D ResolveHitbox(PlayerContext ctx)
    {
        string dir = GetDirection(ctx);

        switch (dir)
        {
            case "North": return northHitbox;

            case "East":
                SetHitboxFlip(eastHitbox, false);
                return eastHitbox;

            case "West":
                SetHitboxFlip(eastHitbox, true);
                return westHitbox;
        }
    }

```

```

        case "South":
            return ctx.Movement.IsGrounded() ? southGroundHitbox :
southAirHitbox;

        default: return eastHitbox;
    }
}

private static void SetHitboxFlip(Collider2D col, bool flipX)
{
    if (col == null) return;
    Vector3 s = col.transform.localPosition;
    col.transform.localPosition = new Vector3(
        flipX ? -Mathf.Abs(s.x) : Mathf.Abs(s.x),
        s.y, s.z
    );
}

public string GetDirection(PlayerContext ctx)
{
    if (Mathf.Abs(ctx.VerticalInput) > 0.1f)
        return ctx.VerticalInput > 0 ? "North" : "South";
    if (Mathf.Abs(ctx.HorizontalInput) > 0.1f)
        return ctx.HorizontalInput > 0 ? "East" : "West";
    return ctx.Movement.FacingDirection > 0 ? "East" : "West";
}

protected abstract void OnChannelStart (PlayerContext ctx);
protected abstract void OnChannelUpdate(PlayerContext ctx);
protected abstract void OnChannelStop (PlayerContext ctx);
protected abstract void OnDamageTick (Collider2D[] hits,
PlayerContext ctx, float damage);

private IEnumerator CooldownRoutine()
{
    {
        _onCooldown = true;
        yield return new WaitForSeconds(cooldown);
        _onCooldown = false;
    }
}
}

```

Лістинг В.3 – Код класу MagnetPushAbility

```

using UnityEngine;
using System.Collections;
using System.Collections.Generic;

```

```

using FiatLux.Core.Events;
using FiatLux.Contracts.Events.Player;
using FiatLux.Contracts.Data;
using FiatLux.Contracts.Interfaces;
using FiatLux.Gameplay.Player.Attack;

namespace FiatLux.Gameplay.Player.Abilities
{
    public class MagnetPushAbility : AbilityBase
    {
        [SerializeField] private float lightForce = 15f;
        [SerializeField] private float EnemyForceMultiplier = 3f;
        [SerializeField] private float reverseDashPower = 24f;
        [SerializeField] private float reverseDashTime = 0.2f;

        [Header("Direction Detectors")]
        [SerializeField] private GameObject northDetector;
        [SerializeField] private GameObject southDetector;
        [SerializeField] private GameObject eastDetector;
        [SerializeField] private GameObject westDetector;

        [Header("Projectile Throw")]
        [SerializeField] private Projectile projectilePrefab;
        [SerializeField] private float projectileSpeed = 14f;
        [SerializeField] private float projectileDamage = 2f;
        [SerializeField] private int maxThrowPerCast = 5;
        [SerializeField] private float throwDelay = 0.06f;
        [SerializeField] private float energyPerThrow = 0.5f;
        [SerializeField] private float playerPushbackPerShot = 1.5f;
        [Tooltip("Тривалість блокування руху під час віддачі (інакше рух одразу гасить імпульс).")]
        [SerializeField] private float pushbackStun = 0.08f;
        [Tooltip("Шанс не витратити проджектайл (покращення MagnetFrugalThrow). 0.25-0.5")]
        [SerializeField] private float frugalChance = 0.375f;

        [Header("Spawn Points (по черзі, кожен проджектайл – зі своєї точки)")]
        [SerializeField] private Transform[] northSpawnPoints;
        [SerializeField] private Transform[] southSpawnPoints;
        [SerializeField] private Transform[] eastSpawnPoints;
        [SerializeField] private Transform[] westSpawnPoints;

        private readonly Dictionary<string, List<GameObject>> _detected =
new()
        {
            { "North", new List<GameObject>() },
            { "South", new List<GameObject>() },
            { "East", new List<GameObject>() },
            { "West", new List<GameObject>() },
        };
    }
}

```

```

public event System.Action<Vector2Int> OnActivated;

private void Awake()
{
    BindDetector(northDetector, "North");
    BindDetector(southDetector, "South");
    BindDetector(eastDetector, "East");
    BindDetector(westDetector, "West");
}

private void BindDetector(GameObject go, string dir)
{
    if (go == null) return;
    var th = go.GetComponent<TriggerHandler>() ??
go.AddComponent<TriggerHandler>();
    th.InitializeForAbility(dir, this);
}

public void AddObject(string dir, GameObject obj)
{
    if (!_detected[dir].Contains(obj)) _detected[dir].Add(obj);
}

public void RemoveObject(string dir, GameObject obj) =>
_detected[dir].Remove(obj);

protected override IEnumerator Execute(PlayerContext ctx)
{
    _onCooldown = true;

    string dir = GetDirection(ctx);
    bool hasStatic = false;

    foreach (var obj in new List<GameObject>(_detected[dir]))
    {
        if (obj == null) continue;
        var metallic = obj.GetComponent<Metallic>();
        if (metallic == null) continue;

        switch (metallic.weight)
        {
            case Metallic.MetallicWeight.Light:
            {
                if
(metallic.TryGetComponent<IKnockbackReceiver>(out var ai))
                    ai.ApplyKnockback(DirToVector(dir) *
lightForce * EnemyForceMultiplier);
                else
                    metallic.ApplyMagneticForce(DirToVector(dir) *
lightForce);
            }
        }
    }
}

```

```

        break;
    }
    case Metallic.MetallicWeight.Heavy:
    {
        if
(metallic.TryGetComponent<IKnockbackReceiver>(out var ai))
            ai.ApplyKnockback(DirToVector(dir) *
lightForce * EnemyForceMultiplier * 0.5f);
        else
            metallic.ApplyMagneticForce(DirToVector(dir) *
lightForce * 0.5f);
        break;
    }
    case Metallic.MetallicWeight.Static:
        hasStatic = true;
        break;
    }
}

float playerDashPower = hasStatic ? reverseDashPower : 0f;
if (playerDashPower > 0f)
    ctx.Dash.PerformReverseDash(DirToVector(Reverse(dir)),
playerDashPower, reverseDashTime);

    OnActivated?.Invoke(DirToVector2Int(dir));
    EventBus.Publish(new PlayerAbilityUsedEvent(AbilitySlot.First,
AbilityPreset.Magnetism, DirToVector2Int(dir)));

    if (ctx.Reservoir != null && ctx.Reservoir.HasProjectile &&
projectilePrefab != null)
        yield return ThrowProjectiles(ctx, dir);

    yield return new WaitForSeconds(cooldown);
    _onCooldown = false;
}

private IEnumerator ThrowProjectiles(PlayerContext ctx, string
dir)
{
    Vector2Int dirInt = DirToVector2Int(dir);
    Vector2 dirVec = DirToVector(dir);
    Transform[] points = GetSpawnPoints(dir);
    GameObject owner = ctx.Rb != null ? ctx.Rb.gameObject :
gameObject;

    bool frugal = ctx.Progression != null &&
ctx.Progression.MagnetFrugalUnlocked;

    int thrown = 0;

```

```

        while (thrown < maxThrowPerCast &&
ctx.Reservoir.HasProjectile)
        {
            if (energyPerThrow > 0f
&& !ctx.Energy.TrySpend(energyPerThrow)) break;

            if (!ctx.Reservoir.TryConsumeForThrow()) break;

            if (frugal && Random.value < frugalChance)
                ctx.Reservoir.Refund();

            Vector3 spawnPos = ResolveSpawnPosition(points, thrown,
dir);

            var proj = Instantiate(projectilePrefab, spawnPos,
Quaternion.identity);
            proj.Launch(dirInt, projectileSpeed, projectileDamage,
owner);

            if (playerPushbackPerShot > 0f && ctx.Movement != null)
                ctx.Movement.ApplyKnockback(-dirVec *
playerPushbackPerShot, pushbackStun);

            thrown++;
            if (throwDelay > 0f) yield return new
WaitForSeconds(throwDelay);
        }
    }

    private Vector3 ResolveSpawnPosition(Transform[] points, int
index, string dir)
    {
        if (points != null && points.Length > 0)
        {
            var p = points[index % points.Length];
            if (p != null) return p.position;
        }

        return GetDetector(dir)?.transform.position ??
transform.position;
    }

    private Transform[] GetSpawnPoints(string dir) => dir switch
    {
        "North" => northSpawnPoints,
        "South" => southSpawnPoints,
    }

```

```

        "East" => eastSpawnPoints,
        "West" => westSpawnPoints,
        _      => null
    };

    private GameObject GetDetector(string dir) => dir switch
    {
        "North" => northDetector,
        "South" => southDetector,
        "East"  => eastDetector,
        "West"  => westDetector,
        _      => null
    };

    private string GetDirection(PlayerContext ctx)
    {
        if (Mathf.Abs(ctx.VerticalInput) > 0.1f)
            return ctx.VerticalInput > 0 ? "North" : "South";
        if (Mathf.Abs(ctx.HorizontalInput) > 0.1f)
            return ctx.HorizontalInput > 0 ? "East" : "West";
        return ctx.Movement.FacingDirection > 0 ? "East" : "West";
    }

    private string Reverse(string d) => d switch
    {
        "North" => "South", "South" => "North",
        "East"  => "West",  "West"   => "East", _ => d
    };

    private Vector2 DirToVector(string d) => d switch
    {
        "North" => Vector2.up,    "South" => Vector2.down,
        "East"  => Vector2.right, "West"  => Vector2.left, _ =>
Vector2.right
    };

    private Vector2Int DirToVector2Int(string d) => d switch
    {
        "North" => Vector2Int.up,
        "South" => Vector2Int.down,
        "East"  => Vector2Int.right,
        "West"  => Vector2Int.left,
        _      => Vector2Int.right
    };
}
}

```

Лістинг В.4 – Код класу StatusHandler

```

using System.Collections.Generic;
using UnityEngine;

```

```

namespace FiatLux.Gameplay.Status
{
    public class StatusHandler : MonoBehaviour, IStatusReceiver
    {
        [SerializeField] private StatusResistanceConfig resistance =
new();

        private readonly Dictionary<System.Type, StatusEffect> _active =
new();
        private readonly List<System.Type> _toRemove = new();

        private void Update()
        {
            _toRemove.Clear();

            foreach (var kvp in _active)
            {
                kvp.Value.OnTick(gameObject, Time.deltaTime);
                if (kvp.Value.IsEmpty) _toRemove.Add(kvp.Key);
            }

            foreach (var type in _toRemove)
            {
                _active[type].OnRemove(gameObject);
                _active.Remove(type);
            }
        }

        public void ApplyStatus(StatusEffect status)
        {
            float res = resistance.GetResistance(status.GetType());

            int effectiveStacks = 0;
            for (int i = 0; i < status.StacksToAdd; i++)
                if (Random.value > res) effectiveStacks++;

            if (effectiveStacks <= 0) return;

            var type = status.GetType();

            if (!_active.TryGetValue(type, out var existing))
            {
                _active[type] = status;
                status.AddStack(effectiveStacks);
                status.OnApply(gameObject);
            }
            else
            {
                int prev = existing.Stacks;
                existing.AddStack(effectiveStacks);
            }
        }
    }
}

```

```

        existing.OnStackChanged(gameObject, prev);
    }
}

public bool HasStatus<T>() where T : StatusEffect
    => _active.ContainsKey(typeof(T));

public T GetStatus<T>() where T : StatusEffect
{
    _active.TryGetValue(typeof(T), out var effect);
    return effect as T;
}

public void RemoveStatus<T>() where T : StatusEffect
{
    var type = typeof(T);
    if (!_active.TryGetValue(type, out var effect)) return;
    effect.OnRemove(gameObject);
    _active.Remove(type);
}
}
}

```

ДОДАТОК Г

Програмний код системи ворогів

Лістинг Г.1 – Код класу Enemy

```
using UnityEngine;
using FiatLux.Contracts.Interfaces;

namespace FiatLux.Gameplay.Enemies
{
    public class Enemy : MonoBehaviour, IDamageable, IBurnable, IFreezable
    {
        [SerializeField] public float AttackDamage = 2f;
        public event System.Action OnDied;
        public event System.Action OnDamaged;

        [Header("Health")]
        [SerializeField] private float maxHealth = 10f;
        [SerializeField] private float currentHealth;

        public float MaxHealth => maxHealth;
        public float CurrentHealth => currentHealth;

        public float HealthFraction => maxHealth > 0f ?
        Mathf.Clamp01(currentHealth / maxHealth) : 0f;

        public bool IsFrozen { get; private set; }
        private Rigidbody2D _rb;

        private void Awake()
        {
            _rb = GetComponent<Rigidbody2D>();
        }

        private void Start() => currentHealth = maxHealth;

        public void TakeDamage(float damage)
        {
            currentHealth -= damage;
            OnDamaged?.Invoke();
            if (currentHealth <= 0f) Die();
        }

        public virtual void OnBurn() => TakeDamage(maxHealth*0.01f);
        public virtual bool IsBurning => false;

        public virtual void OnFreeze() => SetFrozen(true);
        public virtual void OnThaw() => SetFrozen(false);
    }
}
```

```

        public void SetFrozen(bool frozen)
        {
            IsFrozen = frozen;
            if (_rb) _rb.linearVelocity = frozen ? Vector2.zero :
            _rb.linearVelocity;
        }

        public void ResetHealth() => currentHealth = maxHealth;

        private void Die()
        {
            OnDied?.Invoke();

            bool hasAnimatedAI = GetComponent<EnemyAI>() != null
                                || GetComponent<FlyingEnemy>() != null;
            if (!hasAnimatedAI)
                Destroy(gameObject);
        }
    }
}

```

Лістинг Г.2 – Код класу EnemyAI

```

using System;
using UnityEngine;
using FiatLux.Core.Events;
using FiatLux.Contracts.Interfaces;
using FiatLux.Contracts.Events.Enemy;
using FiatLux.Contracts.Events.World;
using FiatLux.Gameplay.Enemies.States;
using FiatLux.World;

namespace FiatLux.Gameplay.Enemies
{
    [RequireComponent(typeof(Enemy))]
    [RequireComponent(typeof(Rigidbody2D))]
    public abstract class EnemyAI : MonoBehaviour, ISpeedModifiable,
    IKnockbackReceiver
    {
        [Header("Detection")]
        [SerializeField] protected float detectionRange = 5f;
        [SerializeField] [Range(10f, 360f)] private float fovAngle = 90f;
        [SerializeField] protected LayerMask obstacleLayer;

        [Header("Movement")]
        [SerializeField] protected float moveSpeed = 2f;
        [SerializeField] private float patrolRayLength = 0.6f;
        [SerializeField] private LayerMask groundLayer;
    }
}

```

```

private float _speedMultiplier = 1f;

[Header("Knockback")]
[SerializeField] private float knockbackStunDuration = 0.25f;

[Header("Death")]
[SerializeField] private float deathAnimDuration = 1f;

public event Action OnPlayerSpotted;
public event Action OnPlayerLost;
public event Action OnAttackWindup;
public event Action OnAttackFired;
public event Action OnSearchStarted;

public float FacingDirection =>
Mathf.Sign(transform.localScale.x);

public Enemy Enemy { get; private set; }
public Transform Player { get; private set; }
public Vector2 LastKnownPlayerPosition { get; set; }

private EnemyState _currentState;
protected Rigidbody2D _rb;
private Animator _animator;
public Animator Animator => _animator;
private float _direction = 1f;

private float _stunEndTime = -1f;
private bool _aggressiveVision;

protected string OwnerRoomId { get; private set; } = string.Empty;
protected bool RoomActive { get; private set; }

public bool IsStunned => Time.time < _stunEndTime;
public void SetSpeedMultiplier(float multiplier) =>
_speedMultiplier = Mathf.Clamp01(multiplier);

public void SetAggressiveVision(bool active) => _aggressiveVision
= active;

protected virtual void OnEnable()
{
    var room = GetComponentInParent<Room>(true);
    OwnerRoomId = room != null ? room.RoomId : string.Empty;
}

```

```

        var active = RoomManager.Instance != null ?
RoomManager.Instance.ActiveRoom : null;
        RoomActive = active != null && active.RoomId == OwnerRoomId;

EventBus.Subscribe<ActiveRoomChangedEvent>(OnActiveRoomChanged);
    }

    protected virtual void OnDisable()
    {

EventBus.Unsubscribe<ActiveRoomChangedEvent>(OnActiveRoomChanged);
    }

    protected virtual void OnActiveRoomChanged(ActiveRoomChangedEvent
evt)
    {
        bool wasActive = RoomActive;
        RoomActive = evt.RoomId == OwnerRoomId;

        if (wasActive && !RoomActive && _rb != null)
            _rb.linearVelocity = Vector2.zero;
    }

    protected virtual void Awake()
    {
        _rb = GetComponent<Rigidbody2D>();
        _animator = GetComponentInChildren<Animator>();
        Enemy = GetComponent<Enemy>();
    }

    public static class AnimParam
    {
        public const string IsWalking = "isWalking";
        public const string Attack    = "attack";
        public const string Dash      = "dash";
        public const string BackDash  = "backDash";
        public const string Die       = "die";
    }

    protected virtual void Start()
    {
        var playerObj = GameObject.FindWithTag("Player");
        if (playerObj) Player = playerObj.transform;
        Enemy.OnDamaged += OnTookDamage;
        Enemy.OnDied    += OnDied;
        SetState(new PatrolState(this));
    }

    protected virtual void OnDestroy()

```

```

    {
        Enemy.OnDamaged -= OnTookDamage;
        Enemy.OnDied    -= OnDied;
    }

    private void OnTookDamage() => SetAggressiveVision(true);

    private void OnDied()
    {
        _currentState?.Exit();
        _currentState = null;

        _rb.linearVelocity = Vector2.zero;
        _rb.bodyType = RigidbodyType2D.Kinematic;
        if (TryGetComponent<Collider2D>(out var col)) col.enabled =
false;

        PlayAnim(AnimParam.Die);
        StartCoroutine(DestroyAfterAnim());
    }

    private System.Collections.IEnumerator DestroyAfterAnim()
    {
        yield return new WaitForSeconds(deathAnimDuration);
        Destroy(gameObject);
    }

    public void PlayAnim(string trigger)
    {
        if (_animator) _animator.SetTrigger(trigger);
    }

    public void SetAnimBool(string param, bool value)
    {
        if (_animator) _animator.SetBool(param, value);
    }

    protected virtual void Update()
    {
        if (!RoomActive) return;
        if (Enemy.IsFrozen) return;
        if (IsStunned) return;
        _currentState?.Update();
    }

    public void SetState(EnemyState newState)
    {
        _currentState?.Exit();
        _currentState = newState;
    }

```

```

        _currentState.Enter();
    }

    public virtual bool CanSeePlayer()
    {
        if (Player == null) return false;

        Vector2 toPlayer = (Vector2)Player.position -
(Vector2)transform.position;
        float distance = toPlayer.magnitude;

        if (distance > detectionRange) return false;

        if (!_aggressiveVision)
        {
            float angle = Vector2.Angle(new Vector2(FacingDirection,
0f), toPlayer);
            if (angle > fovAngle * 0.5f) return false;
        }

        if (obstacleLayer != 0)
        {
            RaycastHit2D hit = Physics2D.Raycast(transform.position,
toPlayer.normalized, distance, obstacleLayer);
            if (hit.collider != null) return false;
        }

        return true;
    }

    public void FacePlayer()
    {
        if (Player == null) return;
        float dir = Mathf.Sign(Player.position.x -
transform.position.x);
        if (dir != 0f) FaceDirection(dir);
    }

    public virtual void MoveTowardsPlayer()
    {
        if (Player == null) return;
        float dir = Mathf.Sign(Player.position.x -
transform.position.x);
        SetHorizontalVelocity(dir * moveSpeed * _speedMultiplier);
        FaceDirection(dir);
    }
}

```

```

    public void MoveTowardsPlayer(float speed)
    {
        if (Player == null) return;
        float dir = Mathf.Sign(Player.position.x -
transform.position.x);
        SetHorizontalVelocity(dir * speed * _speedMultiplier);
        FaceDirection(dir);
    }

    public virtual void MoveTowardsPosition(Vector2 target)
    {
        float dir = Mathf.Sign(target.x - transform.position.x);
        SetHorizontalVelocity(dir * moveSpeed * _speedMultiplier);
        FaceDirection(dir);
    }

    public virtual void StopMovement() => SetHorizontalVelocity(0f);

    public virtual void MoveInPatrolDirection()
    {
        SetHorizontalVelocity(_direction * moveSpeed *
_speedMultiplier);
        FaceDirection(_direction);
    }

    public virtual bool IsAtPatrolBoundary()
    {
        bool wallAhead = Physics2D.Raycast(transform.position,
                                            Vector2.right * _direction,
                                            patrolRayLength, groundLayer);
        Vector2 ahead = new Vector2(transform.position.x + _direction
* patrolRayLength,
                                    transform.position.y);
        bool noFloor = !Physics2D.Raycast(ahead, Vector2.down, 1f,
groundLayer);
        return wallAhead || noFloor;
    }

    public virtual void FlipDirection()
    {
        _direction *= -1f;
        FaceDirection(_direction);
    }

    public abstract EnemyState CreateCombatState();

    public void PublishPlayerSpotted() => OnPlayerSpotted?.Invoke();
    public void PublishPlayerLost()    => OnPlayerLost?.Invoke();

```

```

    public void PublishSearchStarted() => OnSearchStarted?.Invoke();

    public void PublishAttackWindup()
    {
        OnAttackWindup?.Invoke();
        EventBus.Publish(new EnemyAttackEvent(
            transform.position, gameObject.GetInstanceID(), name,
            EnemyAttackPhase.Windup));
    }

    public void PublishAttackFired()
    {
        OnAttackFired?.Invoke();
        EventBus.Publish(new EnemyAttackEvent(
            transform.position, gameObject.GetInstanceID(), name,
            EnemyAttackPhase.Fired));
    }

    public void PublishStateChanged(EnemyStateType state,
        EnemyStatePhase phase) =>
        EventBus.Publish(new EnemyStateChangedEvent(
            transform.position, gameObject.GetInstanceID(), name,
            state, phase));

    private void SetHorizontalVelocity(float vx) =>
        _rb.linearVelocity = new Vector2(vx, _rb.linearVelocity.y);

    public bool FacingLocked { get; private set; }
    public void LockFacing() => FacingLocked = true;
    public void UnlockFacing() => FacingLocked = false;

    private void FaceDirection(float dir)
    {
        if (FacingLocked) return;
        transform.localScale = new Vector3(
            Mathf.Abs(transform.localScale.x) * Mathf.Sign(dir),
            transform.localScale.y,
            transform.localScale.z);
    }

    public virtual void ApplyKnockback(Vector2 force)
    {
        _rb.linearVelocity = Vector2.zero;
        _rb.AddForce(force, ForceMode2D.Impulse);
        _stunEndTime = Time.time + knockbackStunDuration;
    }

    protected virtual void OnDrawGizmosSelected()
    {
        Gizmos.color = Color.yellow;
    }

```

```

        Gizmos.DrawWireSphere(transform.position, detectionRange);

        float facing = Application.isPlaying ? FacingDirection : 1f;
        Vector3 facingVec = new Vector3(facing, 0f, 0f);
        Vector3 leftBound = Quaternion.Euler(0f, 0f, fovAngle *
0.5f) * facingVec * detectionRange;
        Vector3 rightBound = Quaternion.Euler(0f, 0f, -fovAngle *
0.5f) * facingVec * detectionRange;
        Gizmos.DrawLine(transform.position, transform.position +
leftBound);
        Gizmos.DrawLine(transform.position, transform.position +
rightBound);
    }
}

```

ДОДАТОК Д

Програмний код системи прогресії та об'єктів світу

Лістинг Д.1 – Код класу PlayerProgression

```
using System.Collections.Generic;
using UnityEngine;
using FiatLux.Core.Events;
using FiatLux.Contracts.Data;
using FiatLux.Contracts.Events.Game;
using FiatLux.Contracts.Events.Player;
using FiatLux.Contracts.Events.Progression;
using FiatLux.Gameplay.Player.Unlocks;
using FiatLux.Gameplay.Player.Healing;
using FiatLux.Contracts.Events.Save;

namespace FiatLux.Gameplay.Player.Stats
{
    public class PlayerProgression : MonoBehaviour
    {
        [Header("Movement Unlocks")]
        [SerializeField] private bool dashUnlocked = true;

        [Header("Weapon Unlocks")]
        [SerializeField] private bool chainUnlocked = true;
        [SerializeField] private bool bowUnlocked = false;

        [Header("Ability Unlocks")]
        [SerializeField] private bool magnetPushUnlocked = true;
        [SerializeField] private bool magnetPullUnlocked = false;
        [SerializeField] private bool fireUnlocked = false;
        [SerializeField] private bool iceUnlocked = false;

        [Header("Magnet Upgrades")]
        [SerializeField] private bool magnetAbsorbUnlocked = false;
        [SerializeField] private bool magnetAccumulateUnlocked = false;
        [SerializeField] private bool magnetFrugalUnlocked = false;

        [Header("Active Preset")]
        [SerializeField] private AbilityPreset currentPreset =
AbilityPreset.Magnetism;
        [SerializeField] private bool elementalPresetUnlocked =
false;

        private PlayerHealth _health;
        private PlayerEnergy _energy;
        private PlayerHealing _healing;
        private ProjectileReservoir _reservoir;
```

```

        private float _healthMaxBonus;
        private float _energyMaxBonus;

        public AbilityPreset CurrentPreset          => currentPreset;
        public bool      ElementalPresetUnlocked =>
elementalPresetUnlocked;
        public bool DashUnlocked          => dashUnlocked;
        public bool ChainUnlocked          => chainUnlocked;
        public bool BowUnlocked            => bowUnlocked;
        public bool MagnetPushUnlocked => magnetPushUnlocked;
        public bool MagnetPullUnlocked => magnetPullUnlocked;
        public bool FireUnlocked           => fireUnlocked;
        public bool IceUnlocked             => iceUnlocked;
        public bool MagnetAbsorbUnlocked    => magnetAbsorbUnlocked;
        public bool MagnetAccumulateUnlocked => magnetAccumulateUnlocked;
        public bool MagnetFrugalUnlocked    => magnetFrugalUnlocked;

        public event System.Action<AbilityPreset> OnPresetChanged;
        public event System.Action<WeaponType>    OnWeaponUnlocked;
        public event System.Action<string>         OnAbilityUnlocked;

        private void Awake()
        {
            _health      = GetComponent<PlayerHealth>();
            _energy       = GetComponent<PlayerEnergy>();
            _healing      = GetComponent<PlayerHealing>();
            _reservoir    = GetComponent<ProjectileReservoir>();
        }

        private void Start()
        {
            SyncReservoir();
            PublishSnapshot();
        }

        private void SyncReservoir()
        {
            if (_reservoir != null)
                _reservoir.AccumulateEnabled = magnetAccumulateUnlocked;
        }

        private void OnEnable()
        {
            EventBus.Subscribe<AbilityUnlockRequestedEvent>(OnAbilityUnlockRequested);
            EventBus.Subscribe<AbilityRevokeRequestedEvent>(OnAbilityRevokeRequested);
            EventBus.Subscribe<WeaponUnlockRequestedEvent>(OnWeaponUnlockRequested);
            EventBus.Subscribe<CollectingSaveDataEvent>(OnCollecting);
            EventBus.Subscribe<ApplyingSaveDataEvent>(OnApplying);
        }

```

```

    }

    private void OnDisable()
    {
        EventBus.Unsubscribe<AbilityUnlockRequestedEvent>(OnAbilityUnlockRequested);

        EventBus.Unsubscribe<AbilityRevokeRequestedEvent>(OnAbilityRevokeRequested);

        EventBus.Unsubscribe<WeaponUnlockRequestedEvent>(OnWeaponUnlockRequested);
        EventBus.Unsubscribe<CollectingSaveDataEvent>(OnCollecting);
        EventBus.Unsubscribe<ApplyingSaveDataEvent>(OnApplying);
    }

    private void OnWeaponUnlockRequested(WeaponUnlockRequestedEvent e)
=> UnlockWeapon(e.WeaponType);
    private void OnAbilityUnlockRequested(AbilityUnlockRequestedEvent
e)
    {
        switch (e.UnlockType)
        {
            case AbilityUnlockType.Dash:            UnlockDash();
break;
            case AbilityUnlockType.MagnetPush:       UnlockMagnetPush();
break;
            case AbilityUnlockType.MagnetPull:       UnlockMagnetPull();
break;
            case AbilityUnlockType.Fire:             UnlockFire();
break;
            case AbilityUnlockType.Ice:              UnlockIce();
break;
            case AbilityUnlockType.UpgradeHealth:    UpgradeHealth();
EventBus.Publish(new
AbilityUnlockedEvent(AbilityUnlockType.UpgradeHealth)); break;
            case AbilityUnlockType.UpgradeEnergy:    UpgradeEnergy();
EventBus.Publish(new
AbilityUnlockedEvent(AbilityUnlockType.UpgradeEnergy)); break;
            case AbilityUnlockType.UpgradeHealPower: UpgradeHealPower(); EventBus.Publish(new
AbilityUnlockedEvent(AbilityUnlockType.UpgradeHealPower)); break;
            case AbilityUnlockType.MagnetAbsorb:     UnlockMagnetAbsorb(); break;
            case AbilityUnlockType.MagnetAccumulate: UnlockMagnetAccumulate(); break;
            case AbilityUnlockType.MagnetFrugalThrow: UnlockMagnetFrugal(); break;
        }
    }

```

```

e) private void OnAbilityRevokeRequested(AbilityRevokeRequestedEvent
{
    switch (e.Type)
    {
        case AbilityUnlockType.Dash:                RevokeDash();
break;
        case AbilityUnlockType.MagnetPush:
RevokeMagnetPush();    break;
        case AbilityUnlockType.MagnetPull:
RevokeMagnetPull();    break;
        case AbilityUnlockType.Fire:                RevokeFire();
break;
        case AbilityUnlockType.Ice:                RevokeIce();
break;
        case AbilityUnlockType.MagnetAbsorb:
RevokeMagnetAbsorb();    break;
        case AbilityUnlockType.MagnetAccumulate:
RevokeMagnetAccumulate();break;
        case AbilityUnlockType.MagnetFrugalThrow:
RevokeMagnetFrugal();    break;
    }
    PublishSnapshot();
}
public void SwitchPreset()
{
    if (!elementalPresetUnlocked) return;
    currentPreset = currentPreset == AbilityPreset.Magnetism
        ? AbilityPreset.Elemental
        : AbilityPreset.Magnetism;
    OnPresetChanged?.Invoke(currentPreset);
    EventBus.Publish(new PlayerPresetChangedEvent(currentPreset));
}

public void UnlockElementalPreset()
{
    elementalPresetUnlocked = true;
    OnAbilityUnlocked?.Invoke("ElementalPreset");
}

public bool IsWeaponUnlocked(WeaponType type) => type switch
{
    WeaponType.Bare    => true,
    WeaponType.Chain   => chainUnlocked,
    WeaponType.Bow     => bowUnlocked,
    _                  => false
};

public void UnlockWeapon(WeaponType type)
{
    switch (type)

```

```

        {
            case WeaponType.Chain: if (chainUnlocked) return;
chainUnlocked = true; break;
            case WeaponType.Bow:   if (bowUnlocked)   return;
bowUnlocked   = true; break;
            default: return;
        }
        OnWeaponUnlocked?.Invoke(type);
        EventBus.Publish(new WeaponUnlockedEvent(type));
    }

    public void UnlockDash()          { if (dashUnlocked)          return;
dashUnlocked          = true; OnAbilityUnlocked?.Invoke("Dash");
EventBus.Publish(new AbilityUnlockedEvent(AbilityUnlockType.Dash)); }
    public void UnlockMagnetPush() { if (magnetPushUnlocked) return;
magnetPushUnlocked = true; OnAbilityUnlocked?.Invoke("MagnetPush");
EventBus.Publish(new
AbilityUnlockedEvent(AbilityUnlockType.MagnetPush)); }
    public void UnlockMagnetPull() { if (magnetPullUnlocked) return;
magnetPullUnlocked = true; OnAbilityUnlocked?.Invoke("MagnetPull");
EventBus.Publish(new
AbilityUnlockedEvent(AbilityUnlockType.MagnetPull)); }
    public void UnlockFire()          { if (fireUnlocked)          return;
fireUnlocked          = true; OnAbilityUnlocked?.Invoke("Fire");
EventBus.Publish(new AbilityUnlockedEvent(AbilityUnlockType.Fire)); }
    public void UnlockIce()           { if (iceUnlocked)           return;
iceUnlocked           = true; OnAbilityUnlocked?.Invoke("Ice");
EventBus.Publish(new AbilityUnlockedEvent(AbilityUnlockType.Ice)); }

    public void UnlockMagnetAbsorb()   { if (magnetAbsorbUnlocked)
return; magnetAbsorbUnlocked   = true;
OnAbilityUnlocked?.Invoke("MagnetAbsorb");   EventBus.Publish(new
AbilityUnlockedEvent(AbilityUnlockType.MagnetAbsorb)); }
    public void UnlockMagnetAccumulate() { if
(magnetAccumulateUnlocked) return; magnetAccumulateUnlocked = true;
SyncReservoir(); OnAbilityUnlocked?.Invoke("MagnetAccumulate");
EventBus.Publish(new
AbilityUnlockedEvent(AbilityUnlockType.MagnetAccumulate)); }
    public void UnlockMagnetFrugal()   { if (magnetFrugalUnlocked)
return; magnetFrugalUnlocked   = true;
OnAbilityUnlocked?.Invoke("MagnetFrugalThrow"); EventBus.Publish(new
AbilityUnlockedEvent(AbilityUnlockType.MagnetFrugalThrow)); }

    public void RevokeDash()          { if (!dashUnlocked)          return;
dashUnlocked          = false; EventBus.Publish(new
AbilityRevokedEvent(AbilityUnlockType.Dash)); }
    public void RevokeMagnetPush() { if (!magnetPushUnlocked) return;
magnetPushUnlocked = false; EventBus.Publish(new
AbilityRevokedEvent(AbilityUnlockType.MagnetPush)); }

```

```

        public void RevokeMagnetPull() { if (!magnetPullUnlocked) return;
magnetPullUnlocked = false; EventBus.Publish(new
AbilityRevokedEvent(AbilityUnlockType.MagnetPull)); }
        public void RevokeFire()      { if (!fireUnlocked)      return;
fireUnlocked      = false; EventBus.Publish(new
AbilityRevokedEvent(AbilityUnlockType.Fire)); }
        public void RevokeIce()       { if (!iceUnlocked)       return;
iceUnlocked       = false; EventBus.Publish(new
AbilityRevokedEvent(AbilityUnlockType.Ice)); }

        public void RevokeMagnetAbsorb() { if (!magnetAbsorbUnlocked)
return; magnetAbsorbUnlocked = false; EventBus.Publish(new
AbilityRevokedEvent(AbilityUnlockType.MagnetAbsorb)); }
        public void RevokeMagnetAccumulate() { if
(!magnetAccumulateUnlocked) return; magnetAccumulateUnlocked = false;
SyncReservoir(); EventBus.Publish(new
AbilityRevokedEvent(AbilityUnlockType.MagnetAccumulate)); }
        public void RevokeMagnetFrugal() { if (!magnetFrugalUnlocked)
return; magnetFrugalUnlocked = false; EventBus.Publish(new
AbilityRevokedEvent(AbilityUnlockType.MagnetFrugalThrow)); }

    private void OnCollecting(CollectingSaveDataEvent e)
    {
        var d = e.Data;
        d.dashUnlock      = dashUnlocked;
        d.chainUnlock     = chainUnlocked;
        d.bowUnlock       = bowUnlocked;
        d.magnetPushUnlock = magnetPushUnlocked;
        d.magnetPullUnlock = magnetPullUnlocked;
        d.fireUnlock      = fireUnlocked;
        d.iceUnlock       = iceUnlocked;
        d.elementalPresetUnlock = elementalPresetUnlocked;
        d.activePreset    = (int)currentPreset;
        d.healthMaxBonus  = _healthMaxBonus;
        d.energyMaxBonus  = _energyMaxBonus;
        d.magnetAbsorbUnlock = magnetAbsorbUnlocked;
        d.magnetAccumulateUnlock = magnetAccumulateUnlocked;
        d.magnetFrugalUnlock = magnetFrugalUnlocked;
        d.projectileReserve = _reservoir != null ?
_reservoir.TotalCount : 0;
    }
    private void OnApplying(ApplyingSaveDataEvent e)
    {
        var d = e.Data;
        dashUnlocked      = d.dashUnlock;
        chainUnlocked     = d.chainUnlock;
        bowUnlocked       = d.bowUnlock;
        magnetPushUnlocked = d.magnetPushUnlock;
        magnetPullUnlocked = d.magnetPullUnlock;
        fireUnlocked      = d.fireUnlock;
        iceUnlocked       = d.iceUnlock;

```

```

        elementalPresetUnlocked = d.elementalPresetUnlock;
        currentPreset           = (AbilityPreset)d.activePreset;
        magnetAbsorbUnlocked    = d.magnetAbsorbUnlock;
        magnetAccumulateUnlocked = d.magnetAccumulateUnlock;
        magnetFrugalUnlocked    = d.magnetFrugalUnlock;
        SyncReservoir();
        _reservoir?.LoadTotal(d.projectileReserve);

        _healthMaxBonus = 0f;
        _energyMaxBonus = 0f;
        if (d.healthMaxBonus > 0f) UpgradeHealth(d.healthMaxBonus);
        if (d.energyMaxBonus > 0f) UpgradeEnergy(d.energyMaxBonus);

        OnPresetChanged?.Invoke(currentPreset);
        EventBus.Publish(new PlayerPresetChangedEvent(currentPreset));
        PublishSnapshot();
    }

    private void PublishSnapshot()
    {
        var abilities = new List<AbilityUnlockType>();
        if (dashUnlocked) abilities.Add(AbilityUnlockType.Dash);
        if (magnetPushUnlocked)
abilities.Add(AbilityUnlockType.MagnetPush);
        if (magnetPullUnlocked)
abilities.Add(AbilityUnlockType.MagnetPull);
        if (fireUnlocked) abilities.Add(AbilityUnlockType.Fire);
        if (iceUnlocked) abilities.Add(AbilityUnlockType.Ice);

        var weapons = new List<WeaponType> { WeaponType.Bare };
        if (chainUnlocked) weapons.Add(WeaponType.Chain);
        if (bowUnlocked) weapons.Add(WeaponType.Bow);

        EventBus.PublishLatest(new ProgressionSnapshotEvent(abilities,
weapons, currentPreset));
    }

    public void UpgradeHealth(float bonus = 1f)
    {
        _healthMaxBonus += bonus;
        _health?.IncreaseMax(bonus);
        OnAbilityUnlocked?.Invoke("UpgradeHealth");
    }

    public void UpgradeEnergy(float bonus = 1f)
    {
        _energyMaxBonus += bonus;
        _energy?.IncreaseMax(bonus);
        OnAbilityUnlocked?.Invoke("UpgradeEnergy");
    }
}

```

```

        public void UpgradeHealPower(float bonus = 0.5f)
        {
            _healing?.UpgradeHealPower(bonus);
            OnAbilityUnlocked?.Invoke("UpgradeHealPower");
        }
    }
}

```

Лістинг Д.2 – Код класу Metallic

```

using System.Collections;
using UnityEngine;
using FiatLux.Contracts.Interfaces;

public class Metallic : MonoBehaviour
{
    public enum MetallicWeight { Light, Heavy, Static }

    public MetallicWeight weight = MetallicWeight.Light;

    [SerializeField] private float _stopSpeedThreshold = 0.05f;

    [Header("Light – Ground Check")]
    [SerializeField] private LayerMask _groundLayer;
    [SerializeField] private float _groundCheckDistance = 0.15f;

    [Header("Heavy – Rail Settings")]
    [SerializeField] private Transform[] _waypoints;
    [SerializeField] private float _heavyMoveSpeed = 4f;
    [SerializeField] private float _heavyMoveTimeout = 3f;

    private int _currentWaypointIndex;

    private int _targetWaypointIndex = -1;

    private float _originalGravityScale = 1f;
    private Coroutine _railCoroutine;

    private void Start()
    {
        var rb = GetComponent<Rigidbody2D>();

        if (weight == MetallicWeight.Static)
        {
            if (rb == null)
            {
                rb = gameObject.AddComponent<Rigidbody2D>();
                rb.bodyType = RigidbodyType2D.Static;
            }
        }
    }
}

```

```

        return;
    }

    if (rb == null)
    {
        rb = gameObject.AddComponent<Rigidbody2D>();
        rb.bodyType = RigidbodyType2D.Dynamic;
        rb.mass = weight == MetallicWeight.Light ? 1f : 10f;
    }

    _originalGravityScale = rb.gravityScale;

    if (weight == MetallicWeight.Heavy && _waypoints != null &&
        _waypoints.Length > 0)
    {
        _currentWaypointIndex = 0;
        if (_waypoints[0] != null)
        {
            transform.position = _waypoints[0].position;
            rb.position = _waypoints[0].position;
        }
    }
}

public bool ApplyMagneticForce(Vector2 velocity)
{
    if (TryGetComponent<IKnockbackReceiver>(out _)) return false;

    var rb = GetComponent<Rigidbody2D>();
    if (rb == null) return false;

    if (weight == MetallicWeight.Heavy && _waypoints != null &&
        _waypoints.Length > 1)
    {
        if (_railCoroutine != null) StopCoroutine(_railCoroutine);
        _railCoroutine = StartCoroutine(MoveAlongRail(rb,
velocity.normalized));
        return true;
    }

    if (rb.bodyType == RigidbodyType2D.Static)
        StartCoroutine(TemporaryUnlock(rb, velocity));
    else
        rb.linearVelocity = velocity;

    return true;
}

```

```

private IEnumerator MoveAlongRail(Rigidbody2D rb, Vector2 forceDir)
{
    int targetIndex = FindTargetWaypointIndex(forceDir);
    if (targetIndex < 0) yield break;

    rb.bodyType = RigidbodyType2D.Dynamic;
    rb.gravityScale = 0f;
    _targetWaypointIndex = targetIndex;

    Vector2 targetPos = _waypoints[targetIndex].position;
    float elapsed = 0f;
    bool reachedTarget = false;

    while (elapsed < _heavyMoveTimeout && rb != null)
    {
        Vector2 toTarget = targetPos - rb.position;
        if (toTarget.magnitude < 0.05f)
        {
            reachedTarget = true;
            break;
        }

        rb.linearVelocity = toTarget.normalized * _heavyMoveSpeed;
        elapsed += Time.fixedDeltaTime;
        yield return new WaitForFixedUpdate();
    }

    if (rb != null)
    {
        rb.linearVelocity = Vector2.zero;
        rb.gravityScale = _originalGravityScale;
        rb.bodyType = RigidbodyType2D.Static;
        _targetWaypointIndex = -1;

        if (reachedTarget)
        {
            rb.position = targetPos;
            _currentWaypointIndex = targetIndex;
        }
        else
        {
            _currentWaypointIndex = FindNearestWaypointIndex();
        }
    }
}

```

```

private int FindTargetWaypointIndex(Vector2 forceDir)
{
    Vector2 currentPos = transform.position;
    int next = _currentWaypointIndex + 1;
    int prev = _currentWaypointIndex - 1;

    if (next < _waypoints.Length && _waypoints[next] != null)
    {
        Vector2 toNext = ((Vector2)_waypoints[next].position -
currentPos).normalized;
        if (Vector2.Dot(toNext, forceDir) > 0.5f) return next;
    }

    if (prev >= 0 && _waypoints[prev] != null)
    {
        Vector2 toPrev = ((Vector2)_waypoints[prev].position -
currentPos).normalized;
        if (Vector2.Dot(toPrev, forceDir) > 0.5f) return prev;
    }

    return -1;
}

private int FindNearestWaypointIndex()
{
    float minDist = float.MaxValue;
    int nearest = _currentWaypointIndex;
    for (int i = 0; i < _waypoints.Length; i++)
    {
        if (_waypoints[i] == null) continue;
        float d = Vector2.Distance(transform.position,
_waypoints[i].position);
        if (d < minDist) { minDist = d; nearest = i; }
    }
    return nearest;
}

private IEnumerator TemporaryUnlock(Rigidbody2D rb, Vector2 velocity)
{
    rb.bodyType = RigidbodyType2D.Dynamic;
    rb.linearVelocity = velocity;

    yield return new WaitForFixedUpdate();

    float threshold = _stopSpeedThreshold * _stopSpeedThreshold;
    while (rb != null && rb.linearVelocity.sqrMagnitude > threshold)
        yield return new WaitForFixedUpdate();
}

```

```

        if (weight == MetallicWeight.Light)
        {
            while (rb != null && !IsGrounded())
                yield return new WaitForFixedUpdate();
        }

        if (rb != null)
        {
            rb.linearVelocity = Vector2.zero;
            rb.bodyType = RigidbodyType2D.Static;
        }
    }

    private bool IsGrounded()
    {
        var col = GetComponent<Collider2D>();
        if (col == null) return true;

        Bounds b = col.bounds;

        RaycastHit2D[] hits = Physics2D.RaycastAll(
            new Vector2(b.center.x, b.min.y),
            Vector2.down,
            _groundCheckDistance,
            _groundLayer
        );

        foreach (var hit in hits)
            if (hit.collider != null && hit.collider.gameObject !=
gameObject)
                return true;

        return false;
    }
}

```

Лістинг Д.3 – Код класу HazardZoneTeleport

```
using UnityEngine;
```

```
namespace FiatLux.World
{
    public class HazardZoneTeleport : HazardZoneBase
    {

```

```

        protected override void OnPlayerHit(GameObject player)
        {
            SafePoint target = SafePoint.Last;

            if (target == null)
            {
                Debug.LogWarning("[HazardZoneTeleport] No SafePoint
visited yet.");
                return;
            }

            Vector2 destination = target.transform.position;

            if (player.TryGetComponent<Rigidbody2D>(out var rb))
            {
                rb.linearVelocity = Vector2.zero;
                rb.position = destination;
            }
            else
            {
                player.transform.position = destination;
            }
        }
    }
}

```

Лістинг Д.4 – Код класу SafePoint

```

using UnityEngine;

namespace FiatLux.World
{
    [RequireComponent(typeof(Collider2D))]
    public class SafePoint : MonoBehaviour
    {
        public static SafePoint Last { get; private set; }

        private void Awake()
        {
            var col = GetComponent<Collider2D>();
            col.isTrigger = true;
        }

        private void OnTriggerEnter2D(Collider2D other)
        {
            if (!other.CompareTag("Player")) return;
            Last = this;
        }
    }
}

```

```

#if UNITY_EDITOR
    private void OnDrawGizmos()
    {
        bool isLast = (Last == this);
        Gizmos.color = isLast
            ? new Color(0f, 1f, 0.2f, 0.8f)
            : new Color(0f, 0.8f, 0.4f, 0.4f);
        Gizmos.DrawSphere(transform.position, 0.3f);

        UnityEditor.Handles.color = isLast ? Color.green : new
Color(0f, 1f, 0.4f, 0.5f);
        UnityEditor.Handles.Label(
            transform.position + Vector3.up * 0.55f,
            isLast ? "★ SafePoint (last)" : "SafePoint");
    }
#endif
}

```