

**ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ**

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка 2D пригодницької гри для персональних
комп'ютерів»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

*Кваліфікаційна робота містить результати власних досліджень.
Використання ідей, результатів і текстів інших авторів мають посилання
на відповідне джерело*

_____ Анна ВІРИЧ
(підпис)

Виконала: здобувачка вищої освіти групи ПД-42

_____ Анна ВІРИЧ

Керівник:

_____ Ірина ЩЕРБИНА

канд. техн. наук, доц.

Рецензент:

Київ 2026

**ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ**

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення
Ступінь вищої освіти Бакалавр
Спеціальність 121 Інженерія програмного забезпечення
Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри

Інженерії програмного забезпечення

_____ Ірина ЗАМРІЙ

« ____ » _____ 2026 р.

**ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ**

Вірич Анні Максимівні

1. Тема кваліфікаційної роботи: «Розробка 2D пригодницької гри для персональних комп'ютерів»
керівник кваліфікаційної роботи канд. тех. наук, доц. Ірина ЩЕРБИНА,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 року № 51.
2. Строк подання кваліфікаційної роботи «29» травня 2026 року.
3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про ретро-платформери, опис підходів до розробки 2D-платформерів, технічна документація з описом Unity, C#.
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
 1. Дослідження предметної галузі та ринку 2D платформерів.
 2. Вибір технічних засобів та проєктування комп'ютерної гри «Roots of Silence».
 3. Розробка та тестування комп'ютерної гри.
5. Перелік графічного матеріалу: *презентація*
 1. Аналіз аналогів.
 2. Вимоги до програмного забезпечення.

3. Програмні засоби реалізації.
4. Флоу-діаграма ігрового процесу.
5. Діаграма розгортання.
6. Діаграма класів: гравець та вороги.
7. Діаграма класів: менеджери та оточення.
8. Діаграма станів класу Него.
9. Діаграма прецедентів.
10. Екранні форми.
11. Апробація результатів дослідження.

6. Дата видачі завдання «23» лютого 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-техічної літератури	23.02 – 28.02.2026	
2	Аналіз та дослідження аналогів	01.03 – 07.03.2026	
3	Аналіз предметної галузі 2D-платформерів	08.03 – 11.03.2026	
4	Підбір графічних та музичних матеріалів для розроблюваної гри	12.03 – 15.03.2026	
5	Проектування 2D пригодницької гри «Roots of Silence»	16.03 – 22.03.2026	
6	Програмна реалізація та тестування комп'ютерної гри	22.03 – 17.04.2026	
7	Оформлення кваліфікаційної роботи та розробка демонстраційних матеріалів	18.04 – 10.05.2026	
9	Попередній захист роботи	12.05 – 22.05.2026	
10	Подання роботи	23.05 – 30.05.2026	

Здобувачка вищої освіти

(підпис)

Анна ВІРИЧ

Керівник

кваліфікаційної роботи

(підпис)

Ірина ЩЕРБИНА

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 67 стор., 3 табл., 16 рис., 10 джерел.

Кваліфікаційна робота присвячена розробці 2D пригодницької гри для персональних комп'ютерів.

Мета роботи – підвищення зручності взаємодії користувача з продуктом (2D платформером), забезпечивши баланс між ретро-естетикою та сучасними стандартами якості.

Об'єкт дослідження – процес створення 2D пригодницьких ігор для персональних комп'ютерів.

Предмет дослідження – засоби та технології розробки 2D платформерів, що охоплюють ігрові рушії, графічні та аудіо інструменти, а також програмні методи реалізації ігрової логіки.

Короткий зміст роботи: у роботі розроблено 2D пригодницьку гру для ПК «Roots of Silence», поєднавши теоретичне дослідження жанру платформерів із практичною реалізацією програмного продукту. Обґрунтовано вибір технологій (Unity, C#), спроектовано архітектуру системи та реалізовано ігрові механіки, персона-жів, локацію й інтерфейс. Проведене тестування підтвердило стабільність роботи та відповідність гри поставленим вимогам. Результатом є завершений ігровий продукт, придатний для подальшого розвитку й використання як основа для майбутніх проєктів.

Сфера використання гри охоплює галузь цифрових розваг, освіти та інформаційних технологій. Гра може використовуватись як засіб дозвілля для широкого кола користувачів, а також як приклад програмного продукту для демонстрації можливостей сучасних технологій розробки ігор.

КЛЮЧОВІ СЛОВА: КОМП'ЮТЕРНА ГРА, 2D ПЛАТФОРМЕР, РЕТРО-ПЛАТФОРМЕР, ІНДІ-ГРА, ІГРОВИЙ РУШІЙ, UNITY, C#, ГЕЙМДИЗАЙН, РІВНЕВИЙ ДИЗАЙН, МАШИНА СТАНІВ, СПРАЙТОВА АНІМАЦІЯ, ПІКСЕЛЬНА ГРАФІКА, ІГРОВИЙ ПЕРСОНАЖ, СИСТЕМА ЗДОРОВ'Я, ІГРОВИЙ ІНТЕРФЕЙС, ФУНКЦІОНАЛЬНЕ ТЕСТУВАННЯ, UML-ДІАГРАМА, ДІАГРАМА КЛАСІВ, ДІАГРАМА СТАНІВ.

ЗМІСТ

ВСТУП.....	9
1 ДОСЛІДЖЕННЯ ТЕМИ	12
1.1 Дослідження ринку 2D-платформерів	12
1.2 Взаємодія технічних засобів та функціональних можливостей сучасних 2D ігор	17
1.3 Ключові складові ретро-платформера	21
1.4 Формування власної концепції 2D платформера	25
2 ВИБІР ТЕХНІЧНИХ ЗАСОБІВ ТА ПРОЄКТУВАННЯ КОМП'ЮТЕРНОЇ ГРИ “ROOTS OF SILENCE”	28
2.1 Вибір технічних засобів та обґрунтування їх використання.....	28
2.2 Розробка флоу-діаграми ігрового процесу	30
2.3 Побудова діаграми класів для комп'ютерної гри “Roots of Silence”	32
2.4 Розробка діаграми станів класу Hero.....	39
2.5 Розробка діаграми прецедентів комп'ютерної гри «Roots of Silence».....	43
3 РОЗРОБКА КОМП'ЮТЕРНОЇ ГРИ.....	47
3.1 Розробка персонажів	47
3.2 Розробка локації та програмування ходу гри.....	52
3.3 Розробка меню та інтерфейсів	56
3.4 Тестування розробленої гри	59
ВИСНОВКИ	72
ПЕРЕЛІК ПОСИЛАНЬ	76
ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ.....	78
ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ	85

ВСТУП

Індустрія комп'ютерних ігор - одна з найпопулярніших сфер сучасного цифрового світу. Особливе місце в ній займають 2D платформери - жанр, що сформувався ще у 80–90-х роках і став основою для розвитку багатьох ігрових напрямів. Сьогодні, завдяки сучасним інструментам розробки, з'являється можливість відтворити атмосферу класичних ретро-ігор, поєднавши її з новими технічними можливостями та якістю. Створення 2D пригодницької гри для ПК базується на використанні сучасних ігрових рушіїв, мов програмування та систем анімації, що дозволяє реалізувати як класичні механіки платформерів, так і нові елементи, які розширюють ігровий досвід.

Актуальність даної теми обумовлена зростанням популярності інді-розробки та підвищеним інтересом до ігор із ретро-естетикою, які поєднують простоту візуального виконання з глибиною геймплею. У сучасних умовах цифровізації розробка комп'ютерних ігор виступає не лише як форма розваги, але й як окремий напрям програмної інженерії, що потребує комплексного підходу, поєднання технічних знань та творчого мислення. Крім того, розвиток доступних інструментів розробки значно знижує поріг входження у галузь, що робить подібні проєкти актуальними як з практичної, так і з освітньої точки зору.

Метою роботи є підвищення зручності взаємодії користувача з продуктом (2D платформером), забезпечивши баланс між ретро-естетикою та сучасними стандартами якості. Для досягнення мети необхідно вирішити такі завдання:

- Провести аналіз літературних джерел та існуючих прикладів 2D платформерів, визначити їхні ключові особливості та вплив на сучасну ігрову індустрію.
- Здійснити огляд сучасних інструментів та технологій для створення 2D ігор, визначити їх переваги та недоліки.
- Сформулювати вимоги до розробки гри, включаючи технічні та візуальні

аспекти.

- Визначити програмні засоби для реалізації гри (ігровий рушій, мова програмування, бібліотеки).
- Розробити архітектуру гри та реалізувати основні ігрові механіки.
- Провести тестування гри, оцінити її функціональність та відповідність поставленим вимогам.

Кожне із зазначених завдань є взаємопов'язаним та спрямоване на досягнення комплексного результату, що охоплює як теоретичні, так і практичні аспекти розробки.

Об'єктом дослідження є процес створення 2D пригодницьких ігор для персональних комп'ютерів.

Розгляд об'єкта дослідження дозволяє охопити загальні закономірності та підходи до розробки ігрових застосунків, включаючи етапи проектування, реалізації та тестування.

Предметом дослідження є засоби та технології розробки 2D платформерів, що охоплюють ігрові рушії, графічні та інструменти, а також програмні методи реалізації ігрової логіки.

Методи дослідження включають системно-структурний аналіз, порівняльний метод, моделювання та прототипування.

Застосування зазначених методів забезпечує наукову обґрунтованість отриманих результатів та дозволяє здійснити комплексний підхід до вирішення поставлених завдань.

Для реалізації гри застосовуються сучасні технології: ігровий рушій Unity, мова програмування C#, системи анімації та фізики, а також інструменти для тестування та оптимізації.

Вибір зазначених технологій обумовлений їх широким застосуванням у галузі, високою продуктивністю та наявністю розвиненої екосистеми інструментів і ресурсів, що значно спрощує процес розробки.

Наукова новизна роботи полягає у створенні 2D пригодницької гри, яка поєднує ностальгійні елементи класичних платформерів із сучасними

технологічними рішеннями, забезпечуючи новий рівень інтерактивності та якості, а також у системному підході до інтеграції ретро-естетики та сучасних технічних можливостей, що дозволяє отримати продукт, орієнтований як на широку аудиторію гравців, так і на подальший розвиток у межах інді-розробки.

Перший розділ присвячено дослідженню теми. У ньому здійснюється аналіз історичного розвитку жанру 2D платформерів, визначаються його ключові особливості та вплив на сучасну ігрову індустрію.

Другий розділ містить вибір технологічних засобів та проектування гри. Тут аналізуються сучасні інструменти розробки, зокрема ігровий рушій Unity та мова програмування C#, визначаються їхні переваги й недоліки. Також формуються вимоги до гри, описуються основні механіки та архітектура проєкту, включаючи сюжетні та технічні аспекти.

Третій розділ присвячено практичній реалізації гри. У ньому подано опис процесу створення ігрових сцен, персонажів та механік, наведено приклади коду, діаграми класів та скріншоти готових елементів.

Завершальним етапом є тестування гри, оцінка її функціональності та відповідності поставленим вимогам.

1 ДОСЛІДЖЕННЯ ТЕМИ

1.1 Дослідження ринку 2D-платформерів

Жанр 2D-платформерів є одним із найважливіших у розвитку індустрії відеоігор. Його становлення відбулося у 1980-ті роки, коли від простих експериментів у сфері інтерактивних розваг індустрія перейшла до створення цілісних аркадних систем. Саме цей період став визначальним для формування базових механік платформерів, які й сьогодні залишаються актуальними. Аркадні автомати, розташовані у торгових центрах, кафе та ігрових залах, забезпечили широке поширення жанру серед молоді. Просте управління, яскрава графіка та динамічний ігровий процес зробили ці ігри надзвичайно привабливими для аудиторії.

У межах жанру сформувалося кілька піджанрів - вони та їхні характерні риси представлені в таблиці 1.1.

Таблиця 1.1

Піджанри платформерів

Піджанр	Характерні риси	Приклади
Класичні платформери	Лінійне проходження, бонуси, поступове ускладнення	Super Mario Bros., Sonic
Метроїдванії	Велика нелінійна карта, відкриття нових здібностей, дослідження світу	Metroid, Castlevania
Пазл-платформери	Поєднання платформінгу з логічними завданнями, експерименти з простором і часом	Braid, Fez
Хардкорні платформери	Висока складність, швидкий темп	Super Meat Boy, Celeste

Однією з перших ігор, що заклала основу жанру, стала Donkey Kong (1981). Вона запропонувала новаторську вертикальну структуру рівнів, яка вимагала від гравця не лише рухатися вперед, а й підніматися вгору, долаючи перешкоди та ворогів. Це створювало новий вимір складності та динаміки, а також започаткувало концепцію «персонаж проти середовища». Donkey Kong визначила базові принципи платформерів - уникання небезпек і досягнення мети, що стали фундаментом для подальшого розвитку жанру.

Через кілька років з'явилася Super Mario Bros. (1985), яка закріпила жанр у його класичному вигляді. Вона запропонувала чітку структуру рівнів, систему бонусів (монети, гриби, зірки) та унікальний художній стиль, що став символом епохи. Головним досягненням гри було поєднання простоти управління з поступовим ускладненням ігрового процесу. Super Mario Bros. створила модель лінійного проходження з наростанням складності, яка стала стандартом для платформерів на десятиліття вперед і визначила очікування гравців щодо жанру.

Хоча Pac-Man (1980) не є платформером у класичному сенсі, він відіграв важливу роль у формуванні аркадної культури. Його просте управління, яскрава графіка та динамічний процес показали значення ритму, повторюваності та поступового ускладнення, які пізніше були інтегровані у платформери. Pac-Man довів, що навіть мінімалістичні механіки можуть створювати захопливий ігровий досвід, а сама аркадна культура стала середовищем, у якому платформери набули популярності.

Таким чином, ретро-ігри 1980-х років заклали фундаментальні принципи жанру: поступове ускладнення, чітка структура ігрового простору, наголос на динаміці та простоті управління. Вони сформували базові очікування аудиторії, які залишаються актуальними й сьогодні, хоча сучасні платформери значно розширили межі жанру, додаючи нові механіки, сюжетні елементи та художні рішення.

З розвитком технологій та появою інді-сцени жанр платформерів отримав нове дихання. Якщо у 1980-ті роки основою були аркадні автомати та прості механіки, то сучасні 2D-платформери поєднують ретро-естетику з інноваційними

дизайнерськими рішеннями, додаючи глибокий сюжетний вимір та нові способи взаємодії з ігровим простором. Вони демонструють, що жанр не лише зберіг свою актуальність, а й еволюціонував у напрямку більшої складності, емоційності та художньої виразності.

Одним із найяскравіших прикладів сучасних платформерів є Hollow Knight (2017). Ця гра належить до піджанру метроїдваній і пропонує гравцеві дослідження великого нелінійного світу, де поступове відкриття нових здібностей дозволяє долати раніше недоступні перешкоди. Hollow Knight вирізняється високим рівнем художньої деталізації: готична естетика, атмосферна музика та унікальний дизайн персонажів створюють цілісний і запам'ятовуваний світ. Важливим аспектом є також баланс між викликом і винагородою - гра вимагає від гравця уважності та витривалості, але водночас дарує відчуття відкриття й занурення у складну, багатопланову історію.

Іншим знаковим прикладом є Celeste (2018), що належить до хардкорних платформерів. Вона поєднує надзвичайно високий рівень складності з глибоким емоційним сюжетом. Гравець керує героїнею, яка підіймається на гору, долаючи як фізичні, так і психологічні бар'єри. Кожна механіка гри відображає внутрішню боротьбу персонажа, а складність рівнів стає метафорою особистих викликів. Celeste демонструє тенденцію сучасних платформерів до інтеграції наративу та емоційної складової, що розширює межі жанру й перетворює його на засіб художнього самовираження.

Ще одним важливим прикладом є Braid (2008), який став піонером інді-ренесансу у жанрі платформерів. Його головною особливістю є механіка контролю часу, що перетворює класичний платформінг на інтелектуальну головоломку. Гравець отримує можливість змінювати хід подій, що додає новий рівень складності та глибини. Braid показав, що сучасні інді-розробники можуть поєднувати класичну форму з інноваційними ідеями, створюючи унікальний ігровий досвід, який виходить за межі традиційного розуміння жанру.

Таким чином, аналіз ретро-платформерів 1980-х років та сучасних інді-проектів показує, що жанр пройшов довгий шлях еволюції - від простих аркадних

експериментів до багатогранних ігор із глибоким наративом та інноваційними механіками. Класичні зразки заклали фундаментальні принципи, тоді як нові проекти розширили межі жанру, інтегрувавши емоційні та художні аспекти. Для більшої наочності та систематизації результатів дослідження доцільно представити порівняння ключових характеристик старих і нових платформерів у вигляді таблиці 1.2, що дозволяє чітко простежити зміни та тенденції розвитку жанру.

Таблиця 1.2

Порівняння ретро-платформерів та сучасних платформерів

Ознака	Ретро-платформери (1980-ті)	Сучасні платформери (2000-ні й далі)
Графіка	Піксельна, обмежена технічними ресурсами	Висока деталізація, ретро стиль як естетика
Механіки	Простота, поступове ускладнення	Інноваційні механіки (контроль часу, сюжет)
Структура	Лінійні рівні	Нелінійність, єдина карта, інтеграція сюжету
Цільова аудиторія	Масова, орієнтована на аркадні автомати	Інді-гравці, хардкорна спільнота, ностальгійні фанати
Вплив на жанр	Закладення фундаментальних принципів	Розширення меж жанру, поєднання ретро й новизни

Одним із визначальних чинників відродження та активного розвитку жанру 2D платформера у XXI столітті стало формування повноцінної інді-сцени - середовища незалежних розробників, що отримали можливість створювати та розповсюджувати ігрові продукти без залучення великих видавців. Цей процес став можливим завдяки збігу кількох технологічних та ринкових тенденцій, що докорінно змінили структуру ігрової індустрії.

Фундаментальну роль у цьому процесі відіграла демократизація інструментів розробки. Поява та широке розповсюдження безкоштовних або доступних за ціною

ігрових рушіїв - Unity, Godot, GameMaker Studio - усунула технічний бар'єр, який раніше робив розробку комерційних ігор прерогативою великих студій із значними ресурсами. Сучасний незалежний розробник має доступ до повноцінного набору інструментів для створення фізики, анімації, звуку та рівневого дизайну, не вдаючись до власноруч написаних низькорівневих систем. Це суттєво скоротило час та вартість розробки, відкривши простір для творчих експериментів, які великі студії з комерційних міркувань рідко собі дозволяють.

Паралельно з інструментами розробки трансформувалась і система дистрибуції. Цифрові платформи - Steam, itch.io, Nintendo eShop, GOG - надали незалежним розробникам прямий вихід на глобальний ринок без необхідності виробництва фізичних носіїв, укладання угод із роздрібними мережами або залучення видавців як посередників. Це кардинально знизило поріг входу на ринок: проєкт, створений однією особою або малою командою, може бути миттєво доступний аудиторії по всьому світу. Відсутність витрат на виробництво та логістику фізичних копій також дозволила встановлювати гнучку цінову політику, що додатково розширило аудиторію інді-ігор.

Важливим доповненням до цифрової дистрибуції стали платформи краудфандингу - Kickstarter та Indiegogo. Вони сформували принципово нову модель фінансування: розробники отримали можливість залучати кошти безпосередньо від майбутньої аудиторії ще до початку або в процесі розробки, мінімізуючи залежність від зовнішніх інвесторів та видавців. Ця модель не лише вирішує питання фінансування, але й формує спільноту навколо проєкту на ранніх етапах його існування, що забезпечує стартову аудиторію та органічне поширення інформації про гру.

Соціальні мережі та відеоплатформи - YouTube, TikTok, Twitter/X, Discord - відіграли не менш значущу роль у популяризації інді-ігор. Вони стали каналами органічного маркетингу, де окремий трейлер, геймплейний кліп або огляд від популярного стримера здатні забезпечити грі широку аудиторію без традиційних рекламних витрат. Феномен «вірусного» поширення контенту перетворив соціальні мережі на рівноправний інструмент просування поряд із

платною рекламою.

Сукупність цих чинників - доступні інструменти розробки, цифрова дистрибуція, краудфандинг та соціальні мережі - створила екосистему, в якій жанр 2D платформера переживає стійке відродження. Саме незалежні студії, вільні від комерційного тиску великих видавців, стали головними хранителями та реформаторами жанрових традицій, поєднуючи ретро-естетику з сучасними наративними та геймплейними підходами. Успіх таких проєктів, як *Hollow Knight*, *Celeste* та *Shovel Knight*, є переконливим свідченням того, що інді-сцена є не периферією ігрової індустрії, а її повноцінним та творчо активним сегментом.

1.2 Взаємодія технічних засобів та функціональних можливостей сучасних 2D ігор

2D платформи епохи 1980-1990-х років формувались в умовах жорстких технічних обмежень апаратного забезпечення. Обчислювальні потужності консолей того часу - таких як Nintendo Entertainment System (NES), Sega Mega Drive та аркадних автоматів - були надзвичайно скромними за сучасними стандартами, що безпосередньо визначало як художні рішення, так і підходи до проєктування ігрового процесу.

Типова роздільна здатність екрану становила 256×240 пікселів для платформи NES або 320×224 для Sega Mega Drive. Обмежений обсяг відеопам'яті та продуктивність графічних чипів унеможлилювали виведення деталізованих зображень, що породило характерну піксельну естетику, яка згодом стала самостійним художнім напрямом. Більшість консолей підтримували палітру від 16 до 64 кольорів, доступних одночасно на екрані, що накладало суттєві обмеження на художників, змушуючи їх виражати форму, глибину та настрій виключно засобами контрасту та силуету. Персонажі та об'єкти складались з мінімальної кількості пікселів: розмір спрайту головного героя рідко перевищував 16×16 або 32×32 пікселі, а анімаційні послідовності обмежувались 2-4 кадрами, що вимагало від художників максимальної виразності кожного окремого кадру.

Ігрові картриджі мали вкрай обмежений розмір - від кількох десятків до кількох сотень кілобайтів, що змушувало розробників до агресивної оптимізації: повторне використання тайлів, процедурна генерація елементів рівнів та компресія графічних даних стали стандартними практиками галузі. Апаратні обмеження дозволяли виводити на екран не більше 8-64 спрайтів одночасно залежно від платформи, а перевищення цього ліміту призводило до характерного мерехтіння об'єктів. Відсутність апаратного графічного прискорення означала, що всі обчислення виконувались центральним процесором, що суттєво обмежувало складність алгоритмів рендерингу.

Попри зазначені обмеження, саме вони стимулювали появу багатьох винахідливих технічних рішень: прокрутка паралаксу для створення ілюзії глибини, маніпуляція рядками розгортки для спотворення зображення, а також повторне використання обмеженого набору тайлів для побудови різноманітних рівнів. Таким чином, технічні обмеження не лише звужували можливості розробників, але й ставали рушієм творчого пошуку, що заклав фундамент сучасної ігрової індустрії.

Анімація та система управління персонажами є центральними функціональними елементами 2D платформерів, оскільки саме вони визначають динаміку ігрового процесу, емоційну виразність персонажів та ступінь залученості гравця. Основним методом анімування персонажів слугувала покадрова техніка, за якої кожен рух представлявся послідовністю 2-4 унікальних спрайтів. Незважаючи на мінімальну кількість кадрів, досвідчені художники досягали переконливої виразності шляхом ретельного підбору ключових поз. Класичним прикладом є анімація бігу Маріо у Super Mario Bros., реалізована лише трьома кадрами, проте й досі сприймається як органічна та жива.

Для створення ілюзії безперервного руху широко застосовувались замкнуті анімаційні цикли, де останній кадр плавно переходив у перший. Цей підхід дозволяв мінімальним набором спрайтів передавати тривалі дії - біг, ходьбу, стрибки - без необхідності зберігати розгорнуті анімаційні послідовності. З метою економії пам'яті розробники активно застосовували горизонтальне дзеркальне

відображення спрайтів: персонаж, що рухається вліво, використовував той самий набір спрайтів, що й при русі вправо, проте дзеркально відображений апаратними засобами, що дозволяло вдвічі скоротити обсяг анімаційних даних.

Більшість контролерів того часу оперували дискретними сигналами, що обумовило ступінчастий характер управління без аналогового регулювання швидкості. Попри це, розробники досягали відчуття плавності за допомогою програмної симуляції інерції та прискорення.

Параметри гравітації, висоти стрибка та горизонтальної швидкості підбирались вручну, оскільки повноцінні фізичні рушії були відсутні, а якість цього налаштування безпосередньо визначала ігровий досвід.

Поведінка персонажа описувалась через набір взаємовиключних станів - стояння, ходьба, стрибок, падіння, отримання шкоди - перехід між якими відбувався за чітко визначеними умовами. Ця концепція машини станів, закладена ще у перших платформерах, залишається актуальним архітектурним підходом у сучасній розробці ігор.

З розвитком апаратного забезпечення та появою доступних ігрових рушіїв ситуація кардинально змінилась. Сучасні інді-розробники працюють в умовах, що принципово відрізняються від реалій епохи NES чи Sega: обсяг оперативної та відеопам'яті вимірюється гігабайтами, потужність процесорів і графічних карт дозволяє виконувати мільярди операцій за секунду, а розповсюдження гри через цифрові платформи усуває будь-які обмеження на розмір дистрибутиву.

Це відкрило перед 2D інді-іграми якісно новий рівень художніх та технічних можливостей. Графіка більше не обмежена ні роздільною здатністю, ні палітрою: сучасні 2D ігри можуть використовувати повнокольорові ілюстрації у стилі акварелі, детальні рукомальовані фони, динамічне освітлення та складні партикл-ефекти, що раніше були прерогативою виключно тривимірних ігор. Такі проекти, як *Hollow Knight*, *Cuphead* або *Ori and the Blind Forest*, демонструють, що 2D графіка здатна досягати рівня кінематографічної якості без жодних технічних компромісів.

Анімація у сучасних інді-іграх також вийшла далеко за межі покадрових

спрайтів. Широке розповсюдження отримала кісткова анімація (skeletal animation), за якої персонаж складається з ієрархічно пов'язаних сегментів, що деформуються у реальному часі. Такий підхід дозволяє створювати плавні, органічні рухи з довільною кількістю проміжних положень без необхідності малювати кожен кадр вручну. Інструменти на кшталт Spine або Dragonbones зробили кісткову анімацію доступною навіть для одноосібних розробників.

Фізичні рушії, інтегровані у сучасні ігрові середовища - Unity, Godot, Game Maker Studio - надають розробникам повноцінні системи моделювання гравітації, зіткнень, м'яких тіл та рідин без необхідності реалізовувати ці підсистеми вручну. Система управління звуком дозволяє відтворювати десятки аудіошарів одночасно, реалізовувати просторовий звук та динамічну адаптивну музику, що змінюється залежно від ігрової ситуації.

Водночас показово, що багато сучасних інді-розробників свідомо обирають естетику піксельної графіки - не через технічну необхідність, а як художній стиль. Ігри на кшталт Celeste, Shovel Knight або Undertale демонструють, що піксель-арт у сучасному виконанні є не поступкою обмеженням, а усвідомленим творчим вибором, збагаченим сучасними технологіями: динамічним освітленням, шейдерними ефектами та повноцінними фізичними системами. Таким чином, сучасні 2D інді-проекти успадкували художню мову класичних платформерів, позбавившись при цьому технічних обмежень, що її породили, і перетворивши колишні вимушені рішення на свідому естетику.

Візуалізацію еволюції платформерів зображено на рисунку 1.1.

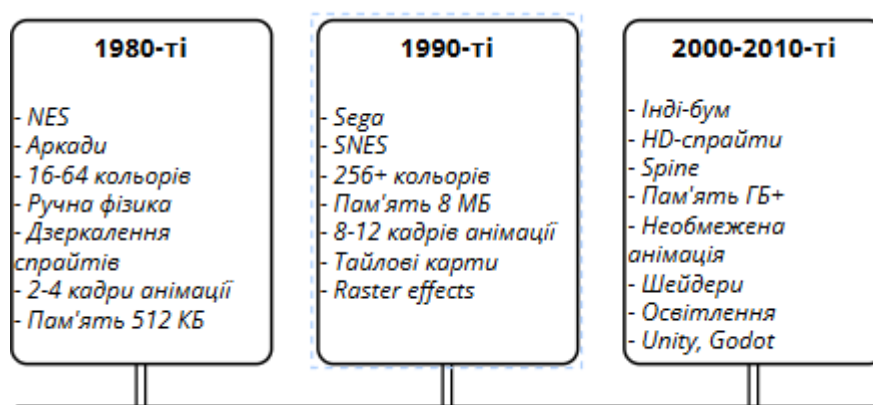


Рис. 1.1 Еволюція технічних можливостей 2D ігрової графіки

1.3 Ключові складові ретро-платформера

Розробка ретро-платформера передбачає відтворення не лише візуальної естетики класичних ігор, але й сукупності геймплейних, технічних та художніх принципів, що визначали ігровий досвід епохи 1980-1990-х років. Аналіз класичних зразків жанру дозволяє виокремити чотири ключові складові, без яких неможлива автентична реалізація платформера у ретро-стилі: система управління персонажем, рівневий дизайн, аудіосупровід та загальна структура ігрового процесу.

Основним принципом управління у класичних платформерах була інтуїтивна простота у поєднанні з глибиною майстерності. Схема введення, як правило, обмежувалась мінімальним набором дій - переміщення по горизонталі, стрибок та атака, - однак ця мінімалістичність не означала відсутності складності. Навпаки, обмежений набір команд дозволяв гравцеві швидко засвоїти базові механіки, залишаючи водночас простір для вдосконалення навичок через точний таймінг, комбінування дій та ситуативне прийняття рішень.

Відчуття управління - так званий *game feel* - досягалось через ретельне ручне налаштування фізичних параметрів: швидкості горизонтального переміщення, висоти та тривалості стрибка, гравітаційного прискорення та інерції гальмування. Саме ці числові значення, підібрані без жодного фізичного рушія, визначали, чи відчувається персонаж живим та чуйним або ж важким і некерованим. У класичних платформерах на кшталт *Super Mario Bros.* параметри руху були відпрацьовані до рівня, при якому управління сприймалось як природне продовження намірів гравця, - і саме цей стандарт є орієнтиром для сучасних ретро-проектів. У технічному сенсі поведінка персонажа реалізовувалась через машину станів із чітко розмежованими станами стояння, бігу, стрибка, падіння, атаки та отримання шкоди. Така архітектура забезпечувала передбачувану та відтворювану реакцію системи на дії гравця, що є обов'язковою умовою справедливого геймплею.

Рівневий дизайн є, мабуть, найбільш визначальною складовою класичного

платформера, оскільки саме він формує темп гри, відчуття прогресу та загальну складність ігрового досвіду. Незважаючи на суворі технічні обмеження, розробники класичної епохи виробили низку принципів, що залишаються актуальними й у сучасних ретро-проектах. Центральним з них є поступове ускладнення, реалізоване через концепцію «навчи - застосуй - перевір»: початкові рівні виконують функцію навчального простору, де гравець у безпечних умовах знайомиться з базовими механіками, тоді як наступні рівні поступово вводять нові елементи - рухомі платформи, нових противників, нові типи перешкод, - нарощуючи складність таким чином, щоб гравець відчував виклик, але не безсилля.

Важливою складовою рівневого дизайну є баланс між складністю та винагородою. Важкі ділянки рівня мають компенсуватись відчутною нагородою у вигляді бонусних предметів, додаткових життів або посилень, що мотивує гравця до дослідження та прийняття ризиків. Структура ігрового простору у класичних платформах, як правило, лінійна з елементами дослідження: головний маршрут є очевидним та доступним, тоді як бічні відгалуження пропонують необов'язкові винагороди для допитливих гравців. Система чекпоінтів регулює рівень напруження: їхня наявність знижує ціну помилки та робить гру доступнішою, тоді як їхня відсутність підвищує ставки та посилює відчуття досягнення при успішному проходженні. Фінальний бос локації виконує функцію підсумкової перевірки навичок, накопичених протягом рівня, і є невід'ємним структурним елементом жанру.

Музика та звукові ефекти є не менш важливою складовою ретро-платформера, оскільки саме аудіосупровід формує атмосферу та підтримує емоційний зв'язок між грою та гравцем. Через апаратні обмеження композитори класичної епохи працювали з чіптюн-синтезаторами, що мали лише 3-5 незалежних звукових каналів. Ця обмеженість породила характерну естетику - прості, ритмічні та надзвичайно запам'ятовувані мелодії, які стали культурним символом цілого покоління. Кожна локація, як правило, отримувала власну музичну тему, що задавала її емоційний тон: динамічна мелодія підкреслювала швидкість та азарт, тривожна - небезпеку та напруження. Головна тема Super Mario Bros. чи музика з

Sonic the Hedgehog є показовими прикладами того, як мелодія може стати невіддільною частиною ігрового образу.

Звукові ефекти виконували не лише естетичну, але й функціональну роль: звук стрибка, підбору предмета, удару чи отримання шкоди слугував миттєвим звуковим зворотним зв'язком, що дозволяв гравцеві реагувати на події навіть без візуального контролю. Таким чином, аудіосупровід у класичному платформері функціонував як паралельний інформаційний канал, що підтримував ігровий ритм та емоційне занурення. Саме тому сучасні ретро-проекти, як правило, свідомо відтворюють чіптюн-естетику - не через технічну необхідність, а як стилістичний маркер жанрової автентичності.

Таким чином, ретро-платформер є результатом ретельно вивіреної взаємодії чотирьох складових: інтуїтивного управління з глибиною майстерності, продуманого рівневого дизайну з поступовим ускладненням, збалансованої структури винагород та атмосферного аудіосупроводу, - вся ця структура представлена на рисунку 1.2.

Відтворення кожної з цих складових є необхідною умовою для створення ігрового досвіду, що відповідає жанровим очікуванням гравця та передає дух класичної епохи.

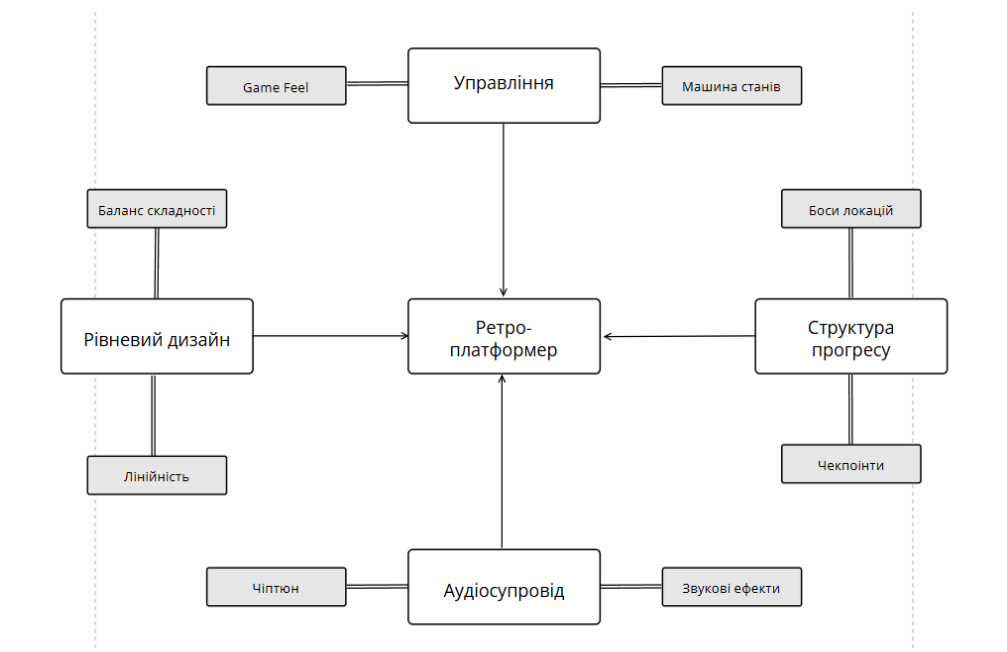


Рис. 1.2 Структура ключових складових ретро-платформера

Поряд із технічними та геймплейними складовими, важливу роль у формуванні привабливості ретро-платформерів відіграє соціокультурний феномен ностальгії. Цей чинник не є суто емоційним переживанням окремого гравця - він функціонує як ринковий, культурний та міжпоколінний механізм, що визначає стійкий попит на ігри у ретро-стилі навіть у контексті сучасного технологічно насиченого ігрового ринку.

З психологічної точки зору ностальгія являє собою емоційно забарвлене звернення до минулого досвіду, пов'язаного з позитивними спогадами. Для значної частини сучасної аудиторії гравців 2D платформери стали першим досвідом взаємодії з відеоіграми взагалі. Повернення до знайомих візуальних образів, характерних мелодій та ігрових механік активує механізм афективної пам'яті, породжуючи відчуття емоційної близькості та безпеки, асоційованих із дитячими або юнацькими роками. Саме ця прив'язаність до особистого досвіду формує стійкий емоційний зв'язок між гравцем та ігровим продуктом, що виходить за межі суто розважального інтересу.

На рівні ринку ностальгійний попит став потужним комерційним стимулом. Сучасні студії - як великі видавці, так і незалежні розробники - свідомо використовують естетику класичних платформерів: піксельну графіку, чіптюн-саундтреки та механіки, характерні для ігор 1980–1990-х років. Цей підхід дозволяє одночасно апелювати до ностальгійних почуттів досвідченої аудиторії та пропонувати їй продукт, відповідний сучасним стандартам якості та зручності. Успіх таких проєктів, як *Shovel Knight*, *Celeste* та *Undertale*, є переконливим свідченням того, що ретро-естетика здатна бути не лише художнім стилем, але й ефективною ринковою стратегією.

Окремого значення набуває міжпоколінний вимір ностальгійного феномену. Молодші гравці, які не мали безпосереднього досвіду з класичними іграми епохи NES чи Sega, знайомляться з цією культурною спадщиною через сучасні ремейки, духовні наступники та інді-проєкти, що відтворюють ретро-стилістику. Таким чином, 2D платформер функціонує як культурний міст між різними поколіннями гравців: для одних він є поверненням до витоків власного ігрового досвіду, для

інших - відкриттям самобутньої художньої та геймплейної традиції. Цей міжпоколінний діалог підтримує актуальність жанру в довгостроковій перспективі та робить ностальгію не просто емоцією, а структурним чинником культурної стійкості ретро-платформера як явища.

1.4 Формування власної концепції 2D платформера

У процесі дослідження класичних та сучасних 2D платформерів було здійснено системний аналіз їхніх ключових механік, структурних особливостей та художніх рішень. На основі цього аналізу було свідомо обрано піджанр класичного платформера як концептуальну основу проєкту. Класичний платформер вирізняється серед суміжних піджанрів - метроїдваній, пазл-платформерів - чіткою акцентуацією на точності управління, рівневій структурі та безпосередньому відчутті руху персонажа. Саме ці елементи формують ядро жанрового досвіду, що залишається актуальним і впізнаваним для сучасного гравця попри десятиліття еволюції індустрії. Додатково слід підкреслити, що проведений аналіз дозволив не лише систематизувати наявні підходи до розробки ігор даного жанру, але й виявити ключові закономірності їхнього функціонування. Це, у свою чергу, створює підґрунтя для обґрунтованого вибору дизайнерських рішень, які відповідають очікуванням цільової аудиторії. Актуальність класичного платформера пояснюється його здатністю забезпечувати чіткий та зрозумілий ігровий досвід, що базується на механічній майстерності гравця.

Вибір піджанру зумовлений кількома чинниками. По-перше, класичний платформер є жанром із найбільш усталеними та дослідженими дизайнерськими конвенціями, що дозволяє спиратись на перевірену часом основу при формуванні власного ігрового досвіду. По-друге, піджанр передбачає відносну стриманість у наборі механік, що є важливим з точки зору масштабу індивідуального розробницького проєкту: замість широти функціональності акцент робиться на глибині та відпрацьованості базових систем. По-третє, саме класичний платформер є найбільш впізнаваною точкою перетину між ретро-естетикою та сучасними

очікуваннями гравців, що забезпечує проєкту чітке жанрове позиціонування. Крім того, вибір даного піджанру дозволяє оптимально поєднати технічні можливості реалізації з художньою концепцією гри, забезпечуючи баланс між складністю розробки та якістю кінцевого продукту. Такий підхід є доцільним у межах індивідуального проєкту, де ресурси є обмеженими, а ефективність використання наявних інструментів відіграє ключову роль.

Концепція гри ґрунтується на трьох основних принципах: поступове ускладнення в межах єдиної локації, відмова від чекпоінтів та акцент на унікальній візуальній складовій. Зазначені принципи формують цілісну систему проєктних рішень, яка визначає як геймплейні, так і художні аспекти гри, забезпечуючи узгодженість між різними складовими ігрового досвіду.

Важливим елементом є поступове ускладнення ігрового процесу в межах однієї безперервної локації. На відміну від класичної структури, де складність нарастає через зміну рівнів, у даному проєкті вона реалізується шляхом поступового ускладнення завдань і перешкод у межах єдиного простору. Це створює ефект «подорожі», де гравець рухається крізь цілісний світ, що постійно змінюється та кидає нові виклики. Такий підхід забезпечує відчуття безперервності та занурення, а також формує динаміку розвитку, яка підтримує інтерес протягом усього проходження. З дизайнерської точки зору це рішення є більш вимогливим, оскільки потребує ретельного контролю кривої складності без можливості «обнулити» напруження через зміну локації чи рівня. Додатково варто зазначити, що реалізація єдиної локації сприяє формуванню цілісного сприйняття ігрового світу, де кожен елемент має своє місце та функціональне призначення. Це дозволяє підвищити рівень занурення гравця та забезпечити логічну послідовність розвитку подій.

Концепція також передбачає відмову від використання чекпоінтів. Це рішення має на меті підвищення напруженості та концентрації гравця. Відсутність можливості швидко повернутися після помилки змушує більш уважно ставитися до кожної дії, що формує особливий характер гри. Такий підхід наближає її до платформерів класичної епохи, де складність була невід'ємною частиною ігрового

досвіду, а успішне проходження важкої ділянки мало реальну емоційну вагу. Водночас він створює унікальну атмосферу виклику, яка може стати відмінною рисою проєкту на тлі сучасних ігор, що нерідко надають пріоритет доступності над напруженням.

З науково-практичної точки зору, таке рішення дозволяє дослідити вплив підвищеної складності на поведінку користувача та рівень його залученості, що є важливим аспектом сучасного геймдизайну.

Третім ключовим аспектом є візуальна складова та дизайн простору. Замість процедурної генерації обрано ручне створення локації, що дозволяє приділити увагу кожній деталі та сформувати цілісну художню концепцію. Єдина карта гри розробляється як послідовність зон із власною атмосферою, стилістичними особливостями та унікальною просторовою композицією. Такий підхід забезпечує естетичну виразність і запам'ятовуваність гри, адже гравець отримує не лише механічний виклик, а й повноцінний художній досвід. Візуальна унікальність сприятиме формуванню емоційного зв'язку між гравцем та ігровим світом, що є одним із ключових чинників тривалої залученості. Крім того, ручний дизайн рівнів дозволяє точніше контролювати розташування ігрових елементів, забезпечуючи баланс між складністю та справедливістю ігрового процесу.

Таким чином, сформована концепція поєднує класичні механіки жанру з авторським підходом до дизайну. Вибір піджанру класичного платформера як основи, поступове ускладнення в межах єдиної локації, відсутність чекпоінтів і акцент на унікальному візуалі утворюють основу майбутнього проєкту та визначають його стиль. У результаті гра позиціонується як продукт, що поєднує ретро-ностальгію з сучасними вимогами до художньої цілісності та геймдизайну.

2 ВИБІР ТЕХНІЧНИХ ЗАСОБІВ ТА ПРОЄКТУВАННЯ КОМП'ЮТЕРНОЇ ГРИ “ROOTS OF SILENCE”

2.1 Вибір технічних засобів та обґрунтування їх використання

Розробка будь-якого програмного продукту розпочинається з вибору технічного стеку - сукупності інструментів, платформ та мов програмування, що визначають архітектуру проєкту, продуктивність розробки та кінцеві характеристики продукту. У контексті створення 2D пригодницької гри для персонального комп'ютера вибір технічних засобів є принциповим рішенням, що впливає на всі наступні етапи реалізації.

Як основний інструмент розробки було обрано ігровий рушій Unity версії 2022 LTS. Це рішення обґрунтовується сукупністю технічних, практичних та економічних чинників.

Unity є одним із найпоширеніших рушіїв у сфері інді-розробки та займає лідируючі позиції серед інструментів для створення 2D ігор. Серед ключових технічних переваг Unity для розробки 2D платформера слід виокремити такі.

По-перше, Unity має повноцінну вбудовану підтримку 2D-розробки: спеціалізований 2D-рендерер, систему спрайтів із підтримкою спрайт-атласів, інструменти Tilemap для побудови рівнів із тайлів, а також вбудований фізичний рушій Box2D, що забезпечує реалістичну симуляцію фізики у двовимірному просторі. Усі ці підсистеми інтегровані в єдине середовище розробки та не потребують підключення сторонніх бібліотек.

По-друге, Unity має потужну систему анімації - Animator та Animation, - що дозволяє створювати як покадрові спрайтові анімації, так і складні анімаційні дерева з переходами між станами, керованими параметрами. Це є критично важливим для реалізації виразної анімації персонажа у платформері.

По-третє, рушій надає зручний редактор сцен із візуальним інтерфейсом, що суттєво прискорює процес побудови рівнів, налаштування об'єктів та їхніх

компонентів. Компонентна архітектура Unity забезпечує гнучке та розширюване проєктування ігрових об'єктів, а система префабів дозволяє ефективно повторно використовувати елементи гри.

По-четверте, Unity підтримує крос-платформенну компіляцію, що дозволяє без суттєвих змін у кодовій базі збирати проєкт під різні операційні системи - Windows, macOS, Linux, - що є важливою перевагою з точки зору охоплення потенційної аудиторії.

Нарешті, Unity має безкоштовну ліцензію Personal, доступну для розробників із річним доходом нижче встановленого порогу, що робить його економічно доцільним вибором для індивідуального проєкту.

Серед альтернативних рушіїв розглядались Godot Engine та GameMaker Studio 2. Godot є перспективним рушієм з відкритим вихідним кодом та зростаючою спільнотою, однак поступається Unity за зрілістю екосистеми та обсягом доступної документації. GameMaker Studio 2 є традиційно сильним інструментом саме для 2D-ігор, проте його платна ліцензія та більш вузька область застосування роблять його менш доцільним вибором у контексті даного проєкту.

Мовою програмування для реалізації ігрової логіки було обрано C# - основну скриптову мову платформи Unity. C# є строго типізованою об'єктно-орієнтованою мовою із сучасним синтаксисом, що підтримує всі ключові парадигми програмування, необхідні для розробки ігрової логіки: успадкування, інкапсуляцію, поліморфізм, інтерфейси та узагальнене програмування. Статична типізація дозволяє виявляти значну частину помилок на етапі компіляції, що підвищує надійність коду та спрощує його підтримку.

З практичної точки зору C# у контексті Unity надає розробнику зручний доступ до всіх підсистем рушія через добре задокументований API. Система компонентів Unity органічно поєднується з об'єктно-орієнтованим підходом C#: кожен компонент є класом, що наслідує базовий клас MonoBehaviour і отримує доступ до стандартних подій ігрового циклу - Awake, Start, Update, FixedUpdate та інших. Це забезпечує чіткість архітектури та передбачуваність поведінки системи.

Особливої уваги заслуговує підхід до формування графічної складової

проєкту. Замість самостійного створення спрайтів та тайлсетів із нуля було прийнято рішення використовувати готові графічні ресурси, отримані з двох основних джерел - Unity Asset Store та itch.io.

Unity Asset Store є офіційним маркетплейсом екосистеми Unity, що містить тисячі ресурсів, сумісних безпосередньо з рушієм: спрайти персонажів, тайлсети, анімовані ефекти, аудіопакети та готові компоненти. Значна частина ресурсів доступна безкоштовно або за символічну плату, а їхня технічна сумісність із форматами Unity гарантується самою платформою, що суттєво скорочує час на імпорт та налаштування.

Itch.io, у свою чергу, є платформою інді-розробників, на якій поряд із готовими іграми широко представлені відкриті графічні паки у стилі піксель-арт, розповсюджені за ліцензіями, що дозволяють їх використання у власних проєктах. Ця платформа вирізняється значною різноманітністю авторських стилів та естетик, що дозволяє підібрати ресурси, максимально відповідні художній концепції конкретного проєкту.

Такий підхід до роботи з графічними ресурсами є поширеною та обґрунтованою практикою в інді-розробці. Він дозволяє сфокусувати розробницькі ресурси на реалізації ігрової логіки, дизайні рівнів та загальній концепції, не витрачаючи значний час на графічне виробництво, яке є окремою спеціалізованою дисципліною. При цьому важливою умовою є дотримання ліцензійних вимог кожного використаного ресурсу - зокрема, перевірка дозволу на використання у комерційних або навчальних проєктах.

Таким чином, обраний технічний стек - Unity як рушій, C# як мова програмування, Unity Asset Store та itch.io як джерела графічних ресурсів - є обґрунтованим і збалансованим вибором для реалізації 2D пригодницької гри для ПК в умовах індивідуального розробницького проєкту.

2.2 Розробка флоу-діаграми ігрового процесу

Для наочного відображення логіки ігрового процесу розробленого проєкту

було побудовано флоу-діаграму (яку Ви можете побачити на рис. 2.1), що ілюструє повний цикл взаємодії гравця з грою - від моменту запуску застосунку до завершення проходження. Діаграма охоплює всі ключові стани системи та переходи між ними, включаючи розгалуження, що виникають залежно від дій гравця.

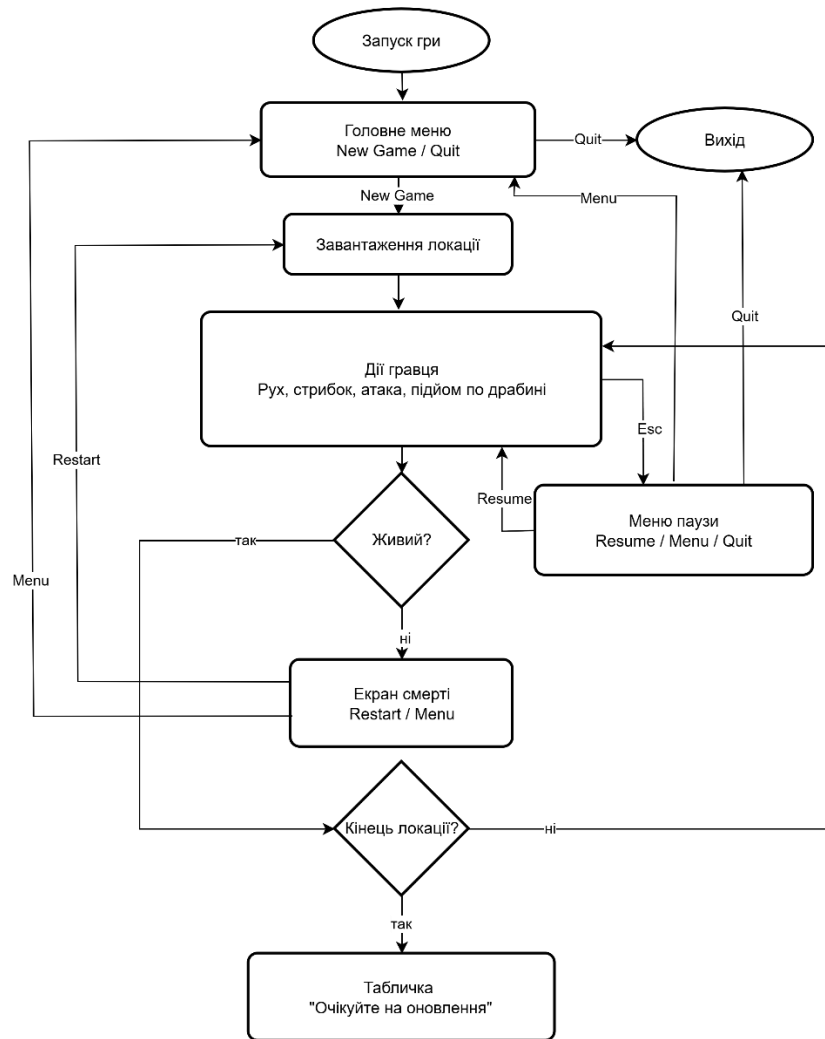


Рис. 2.1 Флоу-діаграма ігрового процесу

Ігровий процес розпочинається із запуску застосунку, після чого система переходить до відображення головного меню. Головне меню містить дві функціональні кнопки: «New Game», що ініціює завантаження ігрової локації, та «Quit», що завершує роботу застосунку. Така мінімалістична структура головного меню відповідає традиціям класичних платформерів, де інтерфейс не відволікав

гравця від безпосереднього занурення в гру.

Після натискання «New Game» система переходить до завантаження локації, а потім - до основного стану ігрового процесу. У цьому стані гравець має у своєму розпорядженні чотири базові дії: горизонтальний рух, стрибок, атаку та підйом по драбині. Під час активної гри гравець у будь-який момент може викликати меню паузи натисканням клавіші Escape. Меню паузи містить три опції: «Resume» - повернення до активного ігрового процесу, «Menu» - перехід до головного меню, та «Quit» - завершення роботи застосунку. Наявність меню паузи є важливим елементом зручності, що дозволяє гравцеві перервати сесію без втрати контексту.

Ігровий процес завершується настанням однієї з двох умов. Першою є загибель персонажа: у разі отримання критичної шкоди система переходить до екрану смерті, що надає гравцеві дві опції - «Restart», яка перезапускає локацію з самого початку, та «Menu», яка повертає до головного меню. Відсутність чекпоінтів означає, що рестарт завжди відбувається з початкової точки локації, що є свідомим концептуальним рішенням, спрямованим на підвищення напруженості та відтворення духу класичних платформерів.

Другою умовою є досягнення кінця локації. На відміну від традиційного екрану завершення, у даному проєкті фінальна точка локації позначається внутрішньоігровим об'єктом - табличкою з написом «Очікуйте на оновлення». Це рішення органічно вписує інформацію про поточний стан розробки безпосередньо в ігровий світ, не порушуючи занурення гравця переходом до окремого інтерфейсного екрану.

Таким чином, флоу-діаграма відображає логічно завершену та збалансовану структуру ігрового процесу з чітко визначеними точками входу, розгалуження та виходу, що забезпечує передбачувану та зручну взаємодію між гравцем і системою.

2.3 Побудова діаграми класів для комп'ютерної гри “Roots of Silence”

Діаграма класів (Class Diagram) є одним із найважливіших та найчастіше використовуваних типів діаграм у стандарті UML. Вона належить до групи

структурних діаграм і призначена для відображення статичної архітектури програмної системи: класів, їхніх атрибутів, методів та відношень між ними. На відміну від поведінкових діаграм, що описують динаміку системи в часі, діаграма класів показує «зріз» архітектури у стані спокою - як саме організовано код, а не як він виконується. Для розробника діаграма класів є основним інструментом проектування архітектури перед написанням коду, а для читача роботи - найбільш інформативним засобом розуміння структури програмного продукту. Основними елементами діаграми класів є самі класи, що зображуються у вигляді прямокутників із трьома секціями - назва, атрибути та методи, - а також відношення між ними. У стандарті UML виділяють кілька типів відношень між класами. Асоціація позначає загальний зв'язок між двома класами, за якого один клас використовує або знає про інший. Залежність є більш слабким зв'язком і позначає, що один клас використовує інший лише тимчасово, наприклад, як параметр методу. Композиція позначає відношення «частина-ціле», за якого дочірній об'єкт не може існувати без батьківського. Наслідування позначає відношення «є», за якого дочірній клас успадковує атрибути та методи батьківського. У даному проєкті найбільш поширеними є відношення асоціації та залежності, що відповідає компонентній архітектурі Unity, де класи взаємодіють через посилання та виклики методів, а не через ієрархію наслідування. Програмна архітектура проєкту реалізована засобами об'єктно-орієнтованого програмування мовою C# у середовищі Unity та організована за компонентною моделлю, характерною для цього рушія. Кожен ігровий об'єкт є носієм одного або кількох скриптів-компонентів, що наслідують базовий клас `MonoBehaviour` та отримують доступ до стандартного ігрового циклу через методи `Awake()`, `Start()`, `Update()` та `FixedUpdate()`. Наслідування від `MonoBehaviour` є спільною рисою всіх класів проєкту, тому на діаграмі це відношення не зображується явно для уникнення надлишкового ускладнення схеми.

Для наочного відображення структури класів та зв'язків між ними побудовано діаграму класів, що охоплює чотири функціональні групи: класи гравця, класи ворогів, класи оточення та системні менеджери. З огляду на значну

кількість класів у проєкті для зручності сприйняття діаграму розділено на дві частини: гравець та вороги (див. рис. 2.2), менеджери та оточення (див. рис. 2.3).

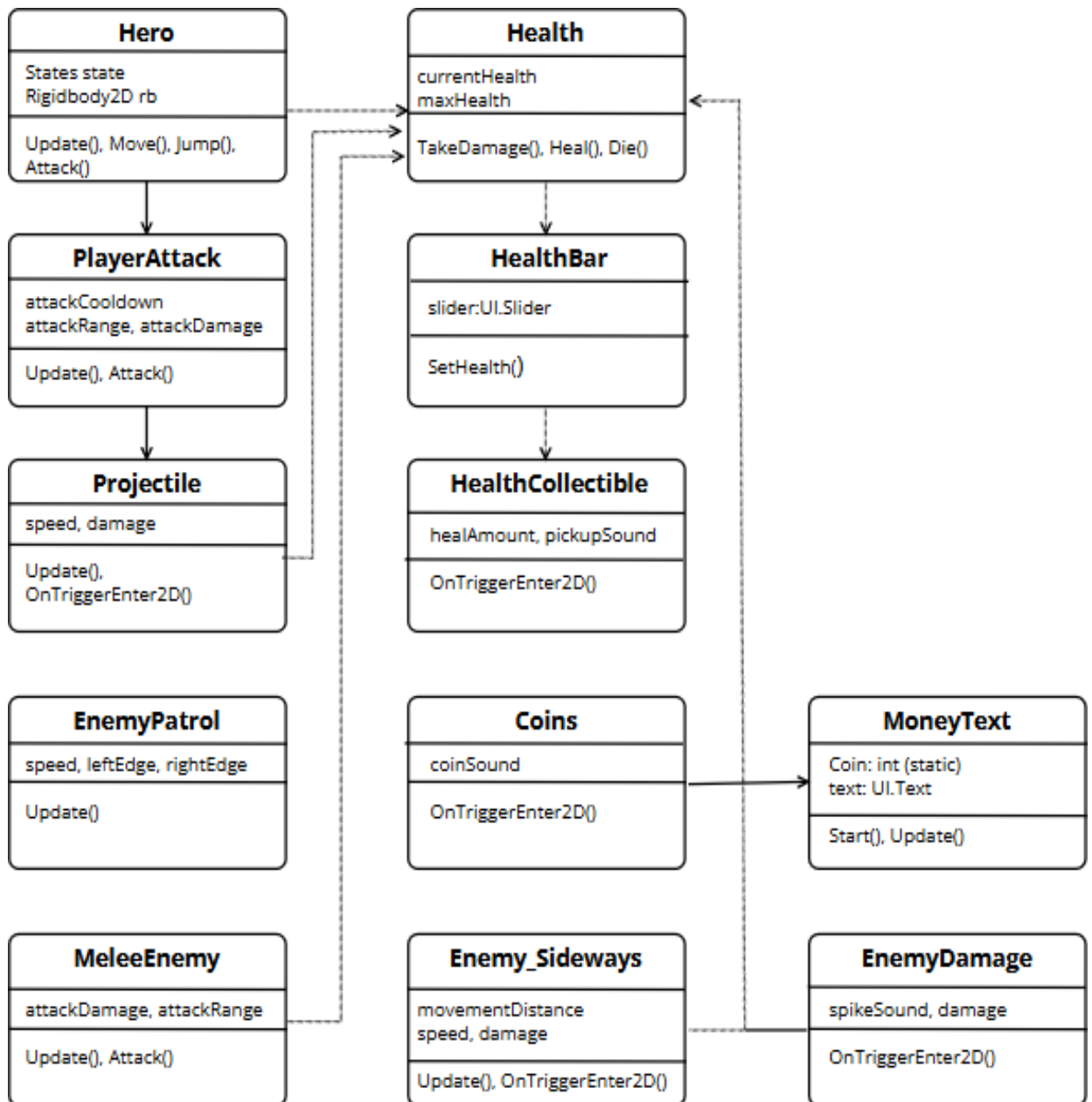


Рис. 2.2 Діаграма класів: гравець та вороги

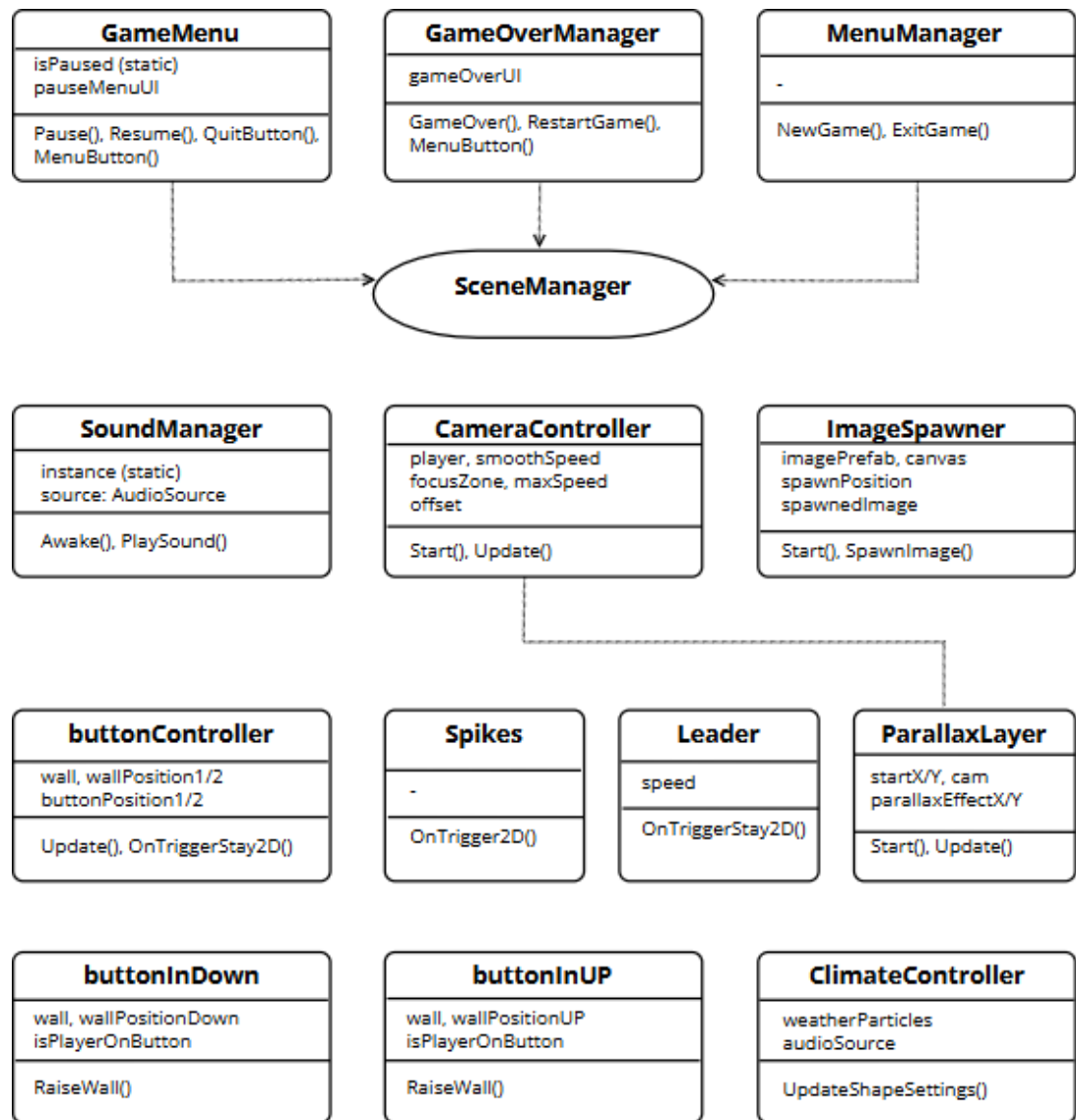


Рис. 2.3 Діаграма класів: менеджери та оточення

Центральним класом проєкту є Неро, що реалізує повну логіку управління персонажем гравця і є точкою інтеграції більшості інших підсистем. Клас зберігає поточний стан персонажа через перелічення States - власний тип даних, що містить усі можливі стани машини станів, - а також посилання на компонент Rigidbody2D, що забезпечує взаємодію з фізичним рушієм Box2D. Клас Неро містить три ключові методи: Move(), що обробляє горизонтальне переміщення та застосовує відповідний імпульс до Rigidbody2D, Jump(), що додає вертикальний імпульс при натисканні клавіші стрибка, та Attack(), що ініціює перехід до стану атаки та делегує логіку пошкодження асоційованому класу PlayerAttack. Неро знаходиться

в асоціації з класом `PlayerAttack`, який відповідає за логіку атаки: зберігає параметри дальності, пошкодження та затримки між ударами, а також ініціює створення об'єктів класу `Projectile` через метод `Instantiate()` рушія Unity. Об'єкти класу `Projectile` є снарядами, що рухаються у просторі після створення та при зіткненні з ворогом викликають метод `TakeDamage()` класу `Health` через компонент виявлення колізій `OnTriggerEnter2D()`.

Клас `Health` є ключовим компонентом системи пошкоджень і використовується як гравцем, так і ворогами, що робить його одним із найбільш пов'язаних класів у системі. Він зберігає поточне значення здоров'я `currentHealth` та максимальне значення `maxHealth` і надає три основні методи: `TakeDamage()`, що зменшує поточне здоров'я на задану величину та перевіряє умову загибелі, `Heal()`, що збільшує поточне здоров'я на задану величину в межах максимального значення, та `Die()`, що ініціює процес загибелі персонажа. Метод `Die()` викликає `GameOverManager`, що ініціює відображення екрану смерті з відповідними опціями навігації. Візуальне відображення здоров'я гравця забезпечується класом `HealthBar`, що асоційований з `Health` та керує компонентом `UI.Slider`, встановлюючи його значення пропорційно відношенню `currentHealth` до `maxHealth`. Збір аптечок реалізований класом `HealthCollectible`, що при зіткненні з гравцем через `OnTriggerEnter2D()` викликає метод `Heal()` класу `Health` та відтворює відповідний звуковий ефект через `SoundManager`. Клас `Coins` побудований за аналогічним принципом: при підборі гравцем він збільшує статичну змінну `Coin` у класі `MoneyText`, що автоматично оновлює відповідний текстовий елемент інтерфейсу `UI.Text`. Використання статичної змінної для зберігання лічильника забезпечує глобальний доступ до її значення з будь-якої точки програми без необхідності передавати посилання між об'єктами.

Група ворогів представлена чотирма класами із різним рівнем складності поведінки. `EnemyPatrol` реалізує базову поведінку патрулювання між двома крайніми точками, що задаються як публічні поля `leftEdge` та `rightEdge`: при досягненні крайньої точки ворог змінює напрямок руху та дзеркально відображає свій спрайт. `MeleeEnemy` розширює базову поведінку патрулювання активною

бойовою логікою: при входженні гравця у зону ураження `attackRange` ворог ініціює атаку, що завдає шкоди компоненту `Health` гравця зі значенням `attackDamage` з дотриманням кулдауну між ударами. `Enemy_Sideways` моделює ворога, що здійснює маятниковоподібний горизонтальний рух на фіксовану відстань `movementDistance` та наносить шкоду гравцеві при прямому зіткненні через `OnTriggerEnter2D()`. `EnemyDamage` є спрощеним класом пасивної пастки - шипів чи аналогічних об'єктів, - що завдає шкоди гравцеві при контакті та відтворює відповідний звуковий ефект через `SoundManager`, не маючи власної логіки переміщення.

Оточення та інтерактивні елементи рівня представлені шістьма класами, що реалізують різноманітні механіки взаємодії з ігровим простором. Класи `buttonController`, `buttonInDown` та `buttonInUP` реалізують механіку інтерактивних кнопок, що керують переміщенням стін у локації: при входженні гравця у тригерну зону кнопки відповідна стіна плавно переміщується до заданої позиції через метод `Vector3.MoveTowards()`.

Відмінність між трьома класами полягає у напрямку та поведінці переміщення стіни: `buttonController` реалізує перемикання між двома позиціями, `buttonInDown` опускає стіну при натисканні, `buttonInUP` піднімає. Клас `Spikes` обробляє зіткнення з шипами та перезапускає сцену через `SceneManager.LoadScene()` - на відміну від класу `Health`, загибель від шипів у даній реалізації не проходить через стандартний механізм здоров'я, а викликає миттєвий перезапуск. Клас `Leader` реалізує механіку драбини: при перебуванні гравця у тригерній зоні він модифікує компонент `Rigidbody2D` персонажа, вимикаючи гравітацію через параметр `gravityScale` та дозволяючи вертикальний рух за клавішами введення.

При виході із зони гравітація відновлюється. `ClimateController` керує погодними ефектами через систему частинок `ParticleSystem`, налаштовуючи параметри форми та колізій частинок дощу, а також відтворює фонову музику та звуки оточення через компонент `AudioSource`. `ParallaxLayer` забезпечує ефект паралакс-скролінгу фонових шарів відносно позиції основної камери: кожен шар

рухається з власним коефіцієнтом `parallaxEffectX`, що визначає частку переміщення шару відносно переміщення камери.

Системні менеджери утворюють сервісний шар архітектури, що координує роботу решти компонентів. `GameMenu` керує меню паузи: при натисканні `Escape` встановлює статичну змінну `isPaused` та призупиняє ігровий час через `Time.timeScale = 0`, а також надає методи `Resume()`, `MenuButton()` та `QuitButton()` для навігації з меню паузи. `GameOverManager` відображає екран смерті та надає гравцеві опції рестарту через `SceneManager.LoadScene()` або повернення до головного меню. `MenuManager` керує головним меню та реалізує методи `NewGame()` та `ExitGame()` для ініціації ігрової сесії та завершення застосунку. `SoundManager` реалізований за патерном `Singleton` зі статичним полем `instance`, що забезпечує глобальну точку доступу до єдиного екземпляра класу, та надає централізований метод `PlaySound()`, що приймає аудіокліп як параметр і відтворює його через вбудований `AudioSource`.

Централізація аудіологіки в одному класі усуває необхідність підключати `AudioSource` до кожного об'єкта окремо та забезпечує єдину точку керування звуком у всій системі. `CameraController` забезпечує плавне слідування камери за персонажем гравця через лінійну інтерполяцію між поточною та цільовою позицією з налаштовуваними параметрами `smoothSpeed` та `focusZone`. `ImageSpawner` відповідає за динамічне створення UI-зображень на компоненті `Canvas` під час виконання гри через метод `SpawnImage()`, що інстанціює заданий префаб у визначеній позиції канвасу.

Таким чином, архітектура проєкту дотримується ключових принципів об'єктно-орієнтованого проєктування. Принцип єдиної відповідальності реалізовано через чітке функціональне розмежування класів: кожен клас виконує одну конкретну задачу і не містить логіки, що виходить за її межі. Принцип слабого зв'язування досягається через взаємодію компонентів переважно через посилення та виклики публічних методів, а не через пряму маніпуляцію внутрішнім станом.

Використання патерну `Singleton` для `SoundManager` забезпечує глобальний

доступ до сервісного класу без порушення інкапсуляції. Усе це в сукупності забезпечує модульність коду, спрощує його відлагодження та створює основу для подальшого розширення проєкту новими класами та механіками без необхідності суттєвої переробки існуючої архітектури.

2.4 Розробка діаграми станів класу Hero

Для детального моделювання поведінки головного персонажа гравця в часі побудовано UML-діаграму станів класу Hero, що зображена на рисунку 2.4. Діаграма станів (State Machine Diagram) є одним із стандартних засобів уніфікованої мови моделювання UML і належить до групи поведінкових діаграм. На відміну від діаграми класів, що описує статичну структуру системи, діаграма станів відображає динамічну поведінку об'єкта в часі: вона показує, у яких станах може перебувати об'єкт, які події спричиняють переходи між станами та які дії виконуються при кожному переході.

Цей тип діаграм є особливо доцільним для моделювання об'єктів із чітко вираженим життєвим циклом та реактивною поведінкою, що залежить від зовнішніх подій, - саме таким об'єктом є ігровий персонаж, поведінка якого визначається діями гравця та фізичним станом середовища. Концепція машини станів (State Machine) є фундаментальним архітектурним патерном у розробці ігор і широко застосовується для керування поведінкою персонажів, ворогів та інших ігрових об'єктів. Її ключова перевага полягає у тому, що об'єкт у будь-який момент часу перебуває рівно в одному стані, що унеможливлює виникнення суперечливих або невизначених поведінкових сценаріїв.

Наприклад, без машини станів персонаж теоретично міг би одночасно виконувати анімацію бігу та стрибка, що призводило б до некоректного візуального результату та непередбачуваної фізичної поведінки. Машина станів усуває цю проблему на архітектурному рівні, забезпечуючи детермінованість поведінки системи.

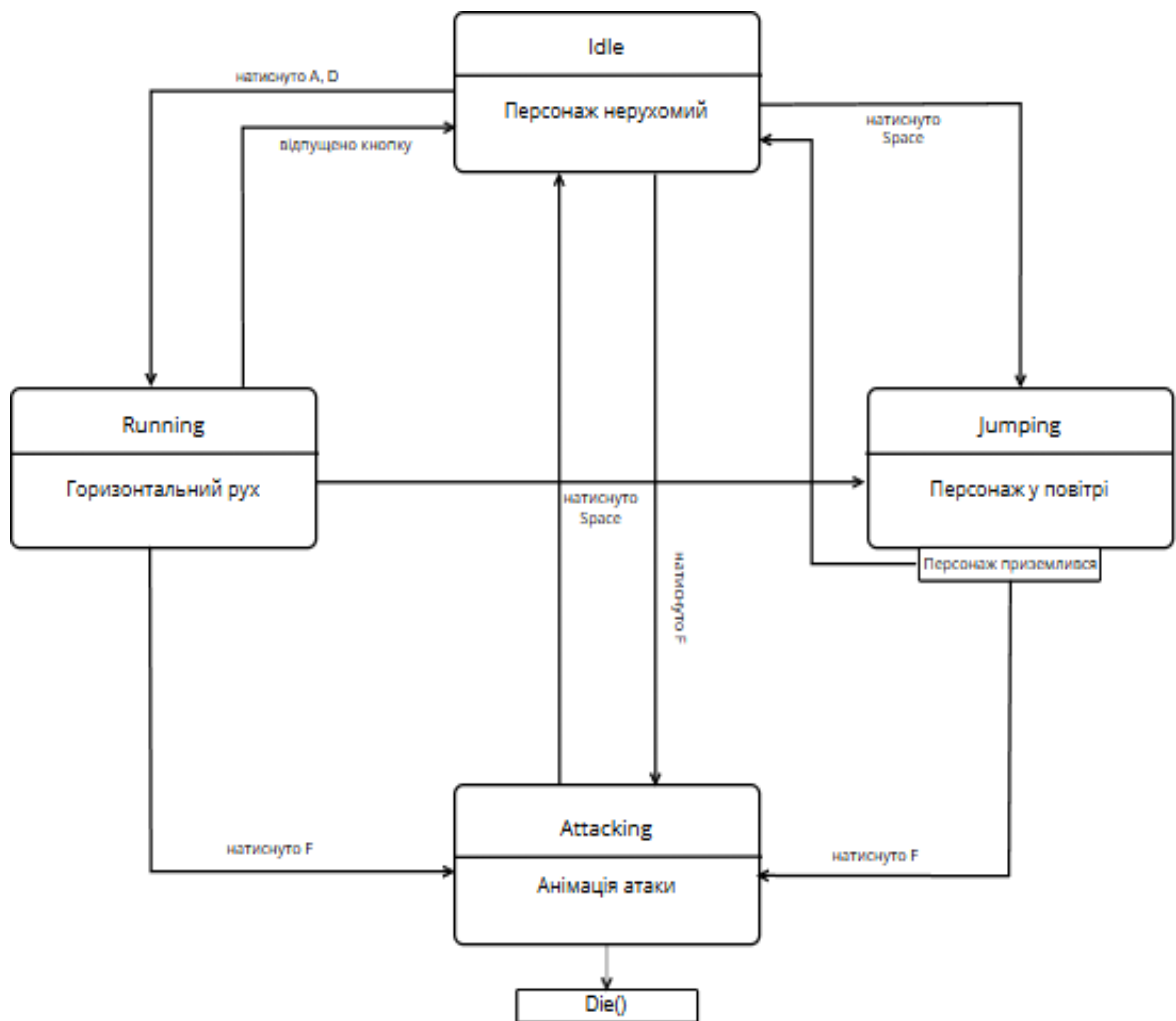


Рис. 2.4 UML-діаграма станів класу Hero

У даному проєкті машина станів реалізована програмно у класі Hero через перелічення States, що містить усі можливі стани персонажа. Метод Update(), що викликається рушієм Unity щоразу перед відтворенням нового кадру, перевіряє поточні умови - стан клавіш введення та фізичний стан персонажа - та оновлює значення поточного стану відповідно. Компонент Animator отримує оновлений стан та автоматично переходить до відповідного анімаційного кліпу згідно з налаштованим графом переходів.

Клас Hero підтримує чотири основні стани: Idle, Running, Jumping та Attacking. Кожен стан відповідає певній поведінці персонажа та набору анімацій, що відтворюються через компонент Animator рушія Unity.

Стан Idle є початковим станом персонажа після завантаження сцени та

активується щоразу, коли гравець не виконує жодних активних дій. Це базовий стан спокою, до якого система повертається після завершення будь-якої іншої дії. У стані Idle компонент Rigidbody2D не отримує горизонтального імпульсу, швидкість персонажа по горизонталі дорівнює нулю, а Animator відтворює анімацію спокою - незначні коливання або дихальні рухи, що підкреслюють живу природу персонажа. Перехід зі стану Idle до Running відбувається при натисканні клавіші руху ліворуч або праворуч, що фіксується методом `Input.GetKey()`, який повертає значення `true` протягом усього часу утримання клавіші. Зворотний перехід з Running до Idle настає у момент відпускання клавіші руху, коли `Input.GetKey()` повертає `false`.

Стан Running активується при утриманні клавіші горизонтального руху та забезпечує переміщення персонажа відповідно до напрямку натиснутої клавіші. У цьому стані метод `Move()` застосовує горизонтальний імпульс до компонента Rigidbody2D, а Animator відтворює анімацію бігу. При зміні напрямку руху спрайт персонажа автоматично дзеркально відображається по горизонтальній осі через зміну параметра `localScale` об'єкта, що забезпечує коректну орієнтацію персонажа відносно напрямку переміщення.

Стан Jumping активується при натисканні клавіші Space як зі стану Idle, так і зі стану Running - тобто персонаж може стрибнути як із місця, так і під час бігу без будь-яких додаткових обмежень. Натискання Space фіксується методом `Input.GetKeyDown()`, що реагує лише на момент першого натискання клавіші, а не на її утримання, що унеможливорює безперервний стрибок при тривалому натисканні. Технічно перехід реалізується через застосування вертикального імпульсу до Rigidbody2D за допомогою методу `AddForce()` з параметром `ForceMode2D.Impulse`, що забезпечує миттєве додавання імпульсу на відміну від поступового прикладання сили.

Повернення до стану Idle відбувається у момент приземлення персонажа, що визначається компонентом `GroundCheck` - спеціальним дочірнім об'єктом із тригерним колайдером, що розміщений у нижній точці персонажа та перевіряє наявність зіткнення з шаром землі через метод `Physics2D.OverlapCircle()`.

Стан `Attacking` є унікальним серед усіх станів тим, що він доступний з будь-якого іншого стану без обмежень: гравець може виконати атаку як стоячи, так і під час бігу або перебуваючи у повітрі. Ця особливість є свідомим дизайнерським рішенням, що надає гравцеві максимальну гнучкість у бойових ситуаціях. Перехід до стану `Attacking` ініціюється натисканням клавіші `F`, що фіксується методом `Input.GetKeyDown()`. Безпосередньо механікою атаки керує асоційований клас `PlayerAttack`, що обробляє параметри пошкодження та кулдауну між ударами, тоді як клас `Hero` відповідає лише за зміну стану та активацію відповідної анімації.

Після завершення анімаційного кліпу атаки система автоматично повертається до стану `Idle` незалежно від того, з якого стану було виконано перехід - це означає, що навіть якщо гравець атакував під час бігу, після атаки персонаж перейде до `Idle`, а не повернеться до `Running`, що є стандартною поведінкою для жанру платформера.

Діаграма також відображає кінцевий стан персонажа - смерть, що настає при виклику методу `Die()` класу `Health` у разі вичерпання здоров'я або контакту з водою. Кінцевий стан є виходом із системи станів: об'єкт `Hero` деактивується через `SetActive(false)`, усі компоненти фізики та введення зупиняються, а `GameOverManager` отримує керування та відображає екран смерті. На діаграмі кінцевий стан позначається стандартним UML-символом - зафарбованим колом у колі, - що відповідає нотації завершення життєвого циклу об'єкта.

Таким чином, машина станів класу `Hero` забезпечує чітку, передбачувану та детерміновану поведінку персонажа, де кожен стан є взаємовиключним, а переходи між ними визначаються конкретними подіями введення або умовами фізичного середовища.

Така архітектура суттєво спрощує відлагодження коду - оскільки у будь-який момент відомо, у якому стані перебуває персонаж і чому, - унеможливорює одночасну активацію суперечливих станів та забезпечує коректну та узгоджену роботу системи анімацій із програмною логікою гри.

2.5 Розробка діаграми прецедентів комп'ютерної гри «Roots of Silence»

Для формального опису функціональних вимог до гри з точки зору взаємодії користувача з системою побудовано UML-діаграму прецедентів (Use Case Diagram), що зображена на рисунку 2.5.

Діаграма прецедентів (Use Case Diagram) є одним із стандартних засобів уніфікованої мови моделювання UML і належить до групи поведінкових діаграм. Її основне призначення полягає у формальному описі функціональних вимог до системи з точки зору зовнішнього користувача, без деталізації внутрішньої програмної реалізації.

Діаграма прецедентів відповідає на питання «що система повинна робити» і є першим кроком у процесі аналізу вимог, що передуює проектуванню архітектури. У контексті ігрового проєкту цей тип діаграм дозволяє чітко та однозначно визначити повний перелік взаємодій між гравцем і грою, що забезпечує узгодженість між задекларованими функціональними вимогами та реалізованим функціоналом.

Основними елементами діаграми прецедентів є актори, прецеденти, системна межа та відношення між ними. Актор представляє зовнішню сутність, що взаємодіє з системою. Прецедент описує певний сценарій використання системи, що надає акторові відчутний результат. Системна межа позначає межі застосунку, відокремлюючи внутрішню логіку системи від зовнішніх суб'єктів.

Відношення між прецедентами можуть бути двох основних типів: *include* - обов'язкове включення одного прецеденту до іншого, та *extend* - опціональне розширення базового прецеденту додатковим сценарієм за певних умов.

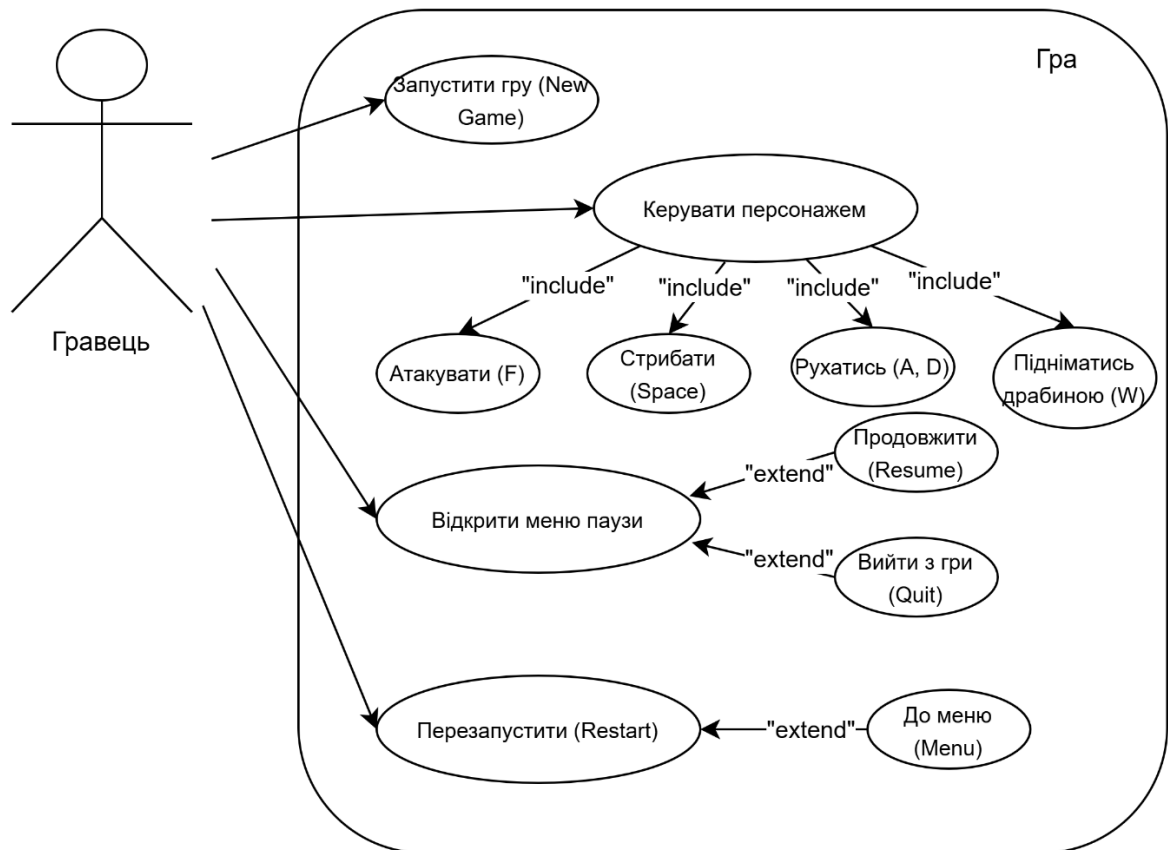


Рис. 2.5 UML-діаграма прецедентів (Use Case)

Єдиним актором у системі є гравець - користувач, що безпосередньо взаємодіє з грою через клавіатуру. Такий вибір є обґрунтованим: гра не передбачає мережевої взаємодії, серверної логіки або адміністративних функцій, тому всі сценарії використання системи ініціюються виключно однією зовнішньою сутністю - людиною за клавіатурою. Усі прецеденти згруповані всередині системної межі, що позначає програмний застосунок «Roots of Silence».

Центральним прецедентом є «Керувати персонажем», який об'єднує чотири обов'язкові підпрецеденти, пов'язані відношенням include. Відношення include означає, що виконання батьківського прецеденту завжди передбачає виконання включеного - тобто управління персонажем є неможливим без реалізації кожної з чотирьох базових дій. Таким чином, управління персонажем включає: переміщення горизонтально клавішами ліво та право, стрибок клавішею Space, атаку клавішею F та підйом по драбині при перебуванні у відповідній тригерній зоні. Ці чотири дії

є фундаментальними механіками гри та реалізуються у класі Hero через методи Move(), Jump(), Attack() та взаємодію з класом Leader відповідно. Виокремлення кожної дії як самостійного підпрецеденту дозволяє чітко задокументувати кожну механіку та простежити її відповідність конкретним методам класу Hero на рівні реалізації.

Прецедент «Запустити гру» є точкою входу до ігрового процесу та відповідає натисканню кнопки «New Game» у головному меню. Активація цього прецеденту ініціює завантаження ігрової локації через Unity SceneManager та переводить систему зі стану головного меню до стану активного ігрового процесу. Прецедент також передбачає альтернативний сценарій у головному меню - вихід із застосунку через кнопку «Quit», що реалізується методом ExitGame() класу MenuManager та завершує роботу застосунку без переходу до ігрової сцени.

Прецедент «Відкрити меню паузи» активується натисканням клавіші Escape під час активного ігрового процесу та розгалужується на три прецеденти, пов'язані відношенням extend. Відношення extend означає, що зазначені прецеденти є опціональними розширеннями базового сценарію, що виконуються лише за певних умов - у даному випадку залежно від вибору гравця в меню паузи. Гравець може обрати одну з трьох опцій: продовжити поточну ігрову сесію через «Resume», повернутись до головного меню через «Menu» або завершити роботу застосунку через «Quit». Клас GameMenu реалізує цю логіку через методи Pause(), Resume(), MenuButton() та QuitButton() з відповідним керуванням параметром Time.timeScale для зупинки та відновлення ігрового часу. Використання відношення extend, а не include, є принциповим: жодна з трьох опцій не є обов'язковою складовою базового прецеденту паузи - гравець вільно обирає одну з них залежно від ситуації.

Прецедент «Перезапустити» активується при загибелі персонажа - настанні однієї з умов смерті. Система відображає екран смерті з написом «You Died», після чого гравець може обрати один із двох варіантів: «Restart» для повторного проходження локації з початкової позиції або «Menu» для повернення до головного меню. Обидва варіанти є опціональними розширеннями базового прецеденту загибелі та реалізуються класом GameOverManager через методи RestartGame()

та `MenuButton()` відповідно.

Важливо зазначити, що прецедент «Перезапустити» не є результатом свідомого вибору гравця, а настає як системна реакція на ігрову подію - вичерпання здоров'я або контакт з водою, - що відрізняє його від прецедентів навігації у меню.

Таким чином, діаграма прецедентів відображає повну множину функціональних можливостей гравця, що налічує дев'ять прецедентів, організованих у три смислові рівні: запуск та вихід із гри, активний ігровий процес із базовими механіками управління та реакція системи на події паузи і загибелі. Відношення `include` і `extend` дозволяють структурувати прецеденти ієрархічно, чітко розмежовуючи обов'язкові та опціональні сценарії, і уникнути дублювання при описі складних взаємозалежних сценаріїв використання.

Побудована діаграма прецедентів є формальним підтвердженням повноти реалізованого функціоналу та слугує сполучною ланкою між задекларованими вимогами до продукту та його програмною реалізацією.

3 РОЗРОБКА КОМП'ЮТЕРНОЇ ГРИ

3.1 Розробка персонажів

Візуальна концепція персонажів гри побудована на протистоянні двох сил: духу-охоронця лісу як головного героя та жаб-злих духів як ворогів. Ця тематична основа визначила художні рішення щодо зовнішнього вигляду, кольорової палітри та анімаційних особливостей кожного персонажа.

Додатково варто зазначити, що обране протистояння має не лише естетичний, але й концептуально-смісловий характер, оскільки формує цілісну систему образів, яка сприймається гравцем на інтуїтивному рівні. Використання архетипічного конфлікту «захисник - загроза» дозволяє підвищити рівень занурення у ігровий процес та забезпечує логічну узгодженість між візуальними та геймплейними елементами. Такий підхід широко застосовується у сучасному геймдизайні як засіб підсилення емоційного впливу на користувача.

Загальний візуальний стиль гри виконано у техніці піксельної графіки на тлі похмурого темно-синього лісового середовища, що створює атмосферу містичного підземного світу. Доцільність використання піксельної графіки обумовлена не лише стилістичними міркуваннями, але й технічними аспектами реалізації. Зокрема, даний підхід дозволяє оптимізувати використання обчислювальних ресурсів, що є важливим фактором для забезпечення стабільної роботи застосунку на різних апаратних платформах.

Крім того, обмеження роздільної здатності спрайтів сприяє підвищенню візуальної узагальненості образів, що позитивно впливає на швидкість їх сприйняття користувачем. Темна кольорова гама середовища виконує роль фону, який не відволікає увагу від основних ігрових об'єктів, а також підсилює відчуття ізольованості та небезпеки.

Головний персонаж гри є духом-охоронцем лісу - містичною істотою, покликаною очистити ліс від нечисті. Візуально персонаж представлений у вигляді

невеликої фігури у темно-червоному плащі з жовтими очима, що світяться, підкреслюючи його надприродну природу. Темна кольорова гама персонажа - насичені відтінки малинового та бордового - контрастує з холодним синьо-зеленим тлом локації, що забезпечує його чітку візуальну впізнаваність на будь-якій ділянці рівня. Спрайт персонажа отримано з платформи itch.io та адаптовано до вимог проєкту засобами редактора Unity.

Окрім цього, при розробці образу персонажа було враховано принципи візуальної ієрархії та силуетної читабельності, відповідно до яких ключові ігрові об'єкти повинні легко розпізнаватися незалежно від складності сцени. Контраст між персонажем і фоном забезпечує швидку локалізацію об'єкта на екрані, що є критично важливим для динамічних ігор.

Також варто відзначити, що використання світлових акцентів (зокрема очей) сприяє формуванню фокусної точки, яка привертає увагу користувача (див. на рис. 3.1).



Рис. 3.1 Спрайт духу-охоронця лісу

Анімаційний набір головного героя складається з трьох станів, що відповідають машині станів класу Hero. Анімація спокою (Idle) відтворюється у стані очікування та підкреслює живу природу персонажа через незначні коливання або дихальні рухи. Анімація бігу (Run) активується при горизонтальному переміщенні та забезпечує відчуття динаміки руху. Анімація стрибка (Jump)

відображає фазу польоту персонажа у повітрі. Усі анімації реалізовані у вигляді покадрових спрайтових послідовностей та керуються компонентом Animator рушія Unity через параметри, що відповідають поточному стану машини станів персонажа.

З технічної точки зору, використання машини станів дозволяє формалізувати логіку поведінки персонажа та забезпечити чітке розмежування між різними режимами його функціонування.

Це, у свою чергу, спрощує подальшу підтримку та розширення програмного коду. Плавність переходів між анімаційними станами досягається шляхом інтерполяції параметрів, що мінімізує візуальні артефакти та підвищує загальну якість анімації.

Система ворогів представлена трьома різновидами жаб-злих духів, що відрізняються розміром, силою та кольоровою гамою. Така градація ворогів забезпечує поступове ускладнення ігрового процесу відповідно до принципу наростання складності, описаного у теоретичній частині роботи.

Крім того, поділ ворогів на типи дозволяє реалізувати диференційований підхід до балансування ігрового процесу. Кожен наступний тип ворога виконує роль логічного продовження попереднього, що сприяє поступовому формуванню навичок гравця та зменшує ймовірність різких стрибків складності.

Перший тип - мала синя жаба - є найслабшим ворогом у грі. Її невеликий розмір та блакитна кольорова гама вписуються у загальну холодну палітру лісового середовища, що може ускладнювати її своєчасне виявлення гравцем.

Цей ворог призначений для ознайомлення гравця з базовою механікою бою на ранніх ділянках локації. З точки зору геймдизайну, даний тип виконує функцію навчального елемента, який дозволяє користувачеві без значних ризиків освоїти основні принципи взаємодії з противниками, включаючи таймінг атак та уникнення шкоди.



Рис. 3.2 Спрайт ворога першого типу

Другий тип - середня зелена жаба - є ворогом проміжного рівня складності. Більший розмір спрайту та зелена кольорова гама відрізняють її від першого типу, сигналізуючи гравцеві про підвищену небезпеку. Вищі показники здоров'я та шкоди від атаки вимагають від гравця більш обережного підходу та точнішого таймінгу атак. Додатково варто зазначити, що введення такого типу ворога сприяє переходу від базового до більш усвідомленого рівня гри, де гравець змушений аналізувати ситуацію та планувати свої дії наперед.



Рис. 3.3 Спрайт ворога другого типу

Третій тип - велика фіолетова жаба - є найсильнішим ворогом у грі. Великий розмір спрайту та насичена фіолетова кольорова гама роблять цього ворога візуально домінуючим і одразу сигналізують про максимальний рівень загрози. Зіткнення з цим типом ворога вимагає від гравця повного володіння ігровими механіками та стратегічного підходу до бою. У контексті ігрового процесу цей ворог виконує роль перевірки сформованих навичок гравця, виступаючи своєрідним підсумковим елементом складності на відповідному етапі проходження.



Рис. 3.4 Спрайт ворога третього типу

Анімаційний набір усіх трьох типів ворогів є ідентичним за структурою та включає п'ять станів: спокій (Idle), стрибок (Jump), атаку (Attack), отримання шкоди (Hurt) та смерть (Death). Наявність анімації отримання шкоди та смерті є важливим елементом зворотного зв'язку з гравцем - вони підтверджують успішність атаки та формують відчуття задоволення від перемоги над противником. Спрайти ворогів також отримано з платформи itch.io та інтегровано у проєкт із налаштуванням відповідних анімаційних контролерів у середовищі Unity. Загалом, використання повного набору анімаційних станів сприяє підвищенню інформативності поведінки ворогів та зменшує необхідність додаткових текстових або графічних підказок інтерфейсу.

Таким чином, розробка персонажів ґрунтується на принципі візуальної читабельності: кожен персонаж має унікальну кольорову гаму та силует, що дозволяє гравцеві миттєво ідентифікувати рівень загрози та приймати тактичні рішення без необхідності звертатись до додаткових підказок інтерфейсу. Додатково слід підкреслити, що реалізація зазначених принципів сприяє зниженню когнітивного навантаження на користувача та підвищує ефективність взаємодії з ігровим середовищем. У результаті формується цілісний користувацький досвід, який поєднує естетичну привабливість, функціональність та зручність сприйняття, що є одним із ключових критеріїв якості сучасних ігрових застосунків.

3.2 Розробка локації та програмування ходу гри

Ігрова локація проєкту являє собою єдиний безперервний рівень, виконаний у стилістиці темного містичного лісу. На відміну від класичної структури платформерів, де прогрес визначається зміною окремих рівнів, у даному проєкті складність наростає плавно в межах одного просторового середовища: по мірі просування гравця вперед змінюється щільність ворогів, характер перешкод та конфігурація платформ, зберігаючи при цьому єдину художню атмосферу.

Геометрія рівня побудована методом ручного розміщення індивідуальних спрайтів у редакторі сцен Unity. Кожен елемент платформи, стіни, підлоги та декоративного оточення розміщувався вручну з налаштуванням позиції, масштабу та відповідного компонента колайдера. Такий підхід є більш трудомістким порівняно з автоматизованою системою Tilemap, однак надає повний контроль над кожною деталлю рівня та дозволяє досягти точного художнього результату без обмежень тайлової сітки. Ручне розміщення спрайтів також відповідає загальній концепції проєкту, що передбачає авторський підхід до дизайну локації та приділяє увагу кожному елементу ігрового простору.

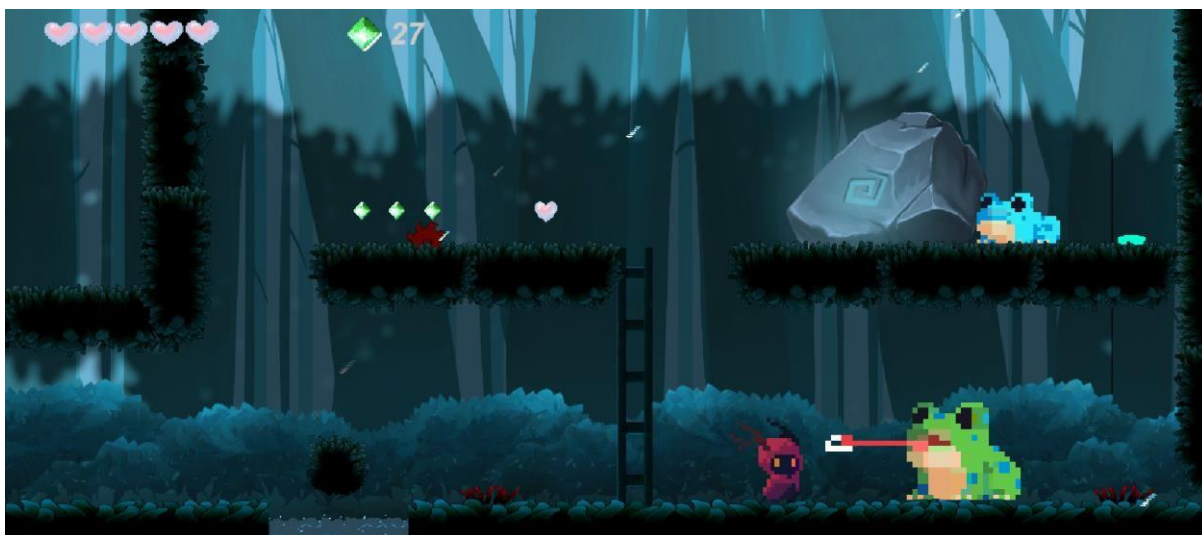


Рис. 3.5 Демонстрація відрізка ігрової локації

Локація містить кілька типів перешкод, що у сукупності формують різноманітні сценарії загибелі та вимагають від гравця різних реакцій. Статичні шипи розміщені на підлозі та стелях окремих ділянок і наносять шкоду при контакті через компонент `EnemyDamage`. Рухома пила є динамічною перешкодою, що переміщується по заданій траєкторії та вимагає від гравця точного таймінг проходження небезпечної зони. Вода є особливим типом смертельної зони: при падінні у водойму персонаж гине миттєво незалежно від поточного рівня здоров'я, що реалізовано через тригерну зону з негайним перезапуском сцени засобами `SceneManager`.

Інтерактивні елементи рівня представлені кнопками, що керують підйомом стін і відкривають прохід до подальших ділянок локації. Механіка реалізована класами `buttonController`, `buttonInDown` та `buttonInUP`: при вході гравця у тригерну зону кнопки відповідна стіна плавно переміщується до заданої позиції, звільняючи прохід. Ця механіка виконує подвійну функцію - структурує просування гравця по локації та вносить елемент дослідження, оскільки гравець має знайти та активувати кнопку перед проходженням певної ділянки.

Для вертикального переміщення між платформами в локації розміщено шість драбин, реалізованих через клас `Leader`. При перебуванні гравця у зоні драбини компонент `Rigidbody2D` персонажа переходить у спеціальний режим: гравітація

вимикається, що дозволяє вільно рухатись вертикально за допомогою клавіш руху.

По всій локації розкидані два типи підбираємих предметів. Кристали слугують основною нагородою та заохоченням до дослідження: їхня кількість відображається у верхній частині екрану через клас MoneyText і оновлюється в реальному часі при підборі через клас Coins. Серця є предметами відновлення здоров'я: кожне серце поповнює запас здоров'я персонажа на одну одиницю через виклик методу Heal() класу Health, що відображається на індикаторі здоров'я HealthBar у верхній частині інтерфейсу. Розміщення обох типів предметів підпорядковане принципу балансу складності та винагороди: серця та кристали розташовані поблизу небезпечних ділянок, стимулюючи гравця до виправданого ризику.

Завершення поточної версії локації позначене табличкою з написом «Очікуйте на оновлення», що вбудована безпосередньо в ігровий простір як внутрішньоігровий об'єкт. Такий підхід дозволяє інформувати гравця про поточний стан розробки без переривання ігрового занурення через окремий екран завершення.

Слідування камери за персонажем реалізовано у класі CameraController з алгоритмом плавної інтерполяції на основі параметрів smoothSpeed та focusZone. Камера не переміщується миттєво до позиції гравця, а поступово наближається до неї з налаштовуваною швидкістю, що створює природне та комфортне візуальне відчуття руху.

Ефект глибини ігрового простору забезпечується системою паралаксу, реалізованою класом ParallaxLayer. Фон локації складається з шести незалежних шарів, кожен із яких переміщується з власним коефіцієнтом швидкості відносно руху камери. Ближні шари рухаються швидше, далекі - повільніше, що створює переконливу ілюзію тривимірної глибини у двовимірному просторі. Параметри parallaxEffectX та parallaxEffectY кожного шару налаштовуються індивідуально в редакторі Unity.

Атмосфера локації підсилюється двома типами ефектів оточення. Дощ реалізовано через систему частинок ParticleSystem, керовану класом

ClimateController: частинки генеруються у верхній частині екрану та падають вниз, імітуючи атмосферні опади. Одночасно з частинками відтворюється звук дощу через компонент AudioSource, що підсилює ефект присутності та занурення у середовище. Туман на окремих ділянках локації реалізований як статичний спрайтовий шар з налаштованою прозорістю, що накладається поверх ігрового простору та візуально ізолює певні зони, підкреслюючи їхню небезпечність або таємничість.

Навчання гравця реалізоване через систему внутрішньоігрових табличок, розміщених безпосередньо в просторі локації у ключових точках проходження. Кожна табличка оформлена у єдиному піксельному стилі з темним фоном та світлою рамкою, що відповідає загальній візуальній мові гри, і містить коротке текстове повідомлення українською мовою. Такий підхід до навчання є свідомим дизайнерським рішенням: підказки не виводяться через окремий інтерфейсний екран і не переривають ігровий процес, а органічно вписуються у світ гри як частина її простору.

Таблички розміщені послідовно по ходу рівня та охоплюють усі ключові механіки: перша знайомить гравця зі схемою управління персонажем («Керування духом»), друга попереджає про небезпечні зони середовища («Червоне - небезпека. Вода - вірна смерть»), третя пояснює систему нагород («Збирайте алмази для проходження гри»), четверта інформує про клавішу атаки («F - атака»), п'ята пояснює механіку інтерактивних кнопок («Активуй кнопку, щоб пройти далі»). Завершує послідовність табличка наприкінці локації з написом «Очікуйте на оновлення», що сигналізує про межу поточної версії гри.

Реалізація табличок як статичних спрайтових об'єктів у сцені є економічним та ефективним рішенням: вони не потребують окремої програмної логіки, завжди знаходяться на своїх місцях та сприймаються гравцем як природна частина ігрового світу, що відповідає принципу ненав'язливого навчання через середовище, характерному для жанру платформера.

Загибель персонажа настає за двома сценаріями. У першому сценарії здоров'я гравця вичерпується внаслідок накопичених пошкоджень від ворогів, шипів або

рухомої пили: при досягненні нульового значення здоров'я клас Health викликає метод Die(), що передає управління класу GameOverManager, який відображає екран смерті з опціями Restart та Menu. У другому сценарії гравець падає у воду, що призводить до миттєвої загибелі незалежно від поточного рівня здоров'я. Відсутність чекпоінтів означає, що у будь-якому з цих випадків рестарт завжди відбувається з початкової точки локації, що відповідає концептуальному рішення, описаному у розділі формування концепції проєкту.

3.3 Розробка меню та інтерфейсів

Усі елементи користувацького інтерфейсу гри реалізовані засобами стандартної системи Unity UI на основі компонента Canvas та підпорядковані єдиній художній концепції проєкту - темній, атмосферній естетиці містичного лісу. Інтерфейс навмисно виконано у мінімалістичному стилі: кількість елементів зведена до необхідного мінімуму, щоб не відволікати гравця від візуального середовища гри.

Головне меню гри «Roots of Silence» відображається на тлі статичного ілюстративного зображення темного лісу у холодній синьо-сірій кольоровій гамі, що одразу занурює гравця в атмосферу проєкту ще до початку гри. У лівій частині екрану розміщена назва гри, виконана великим курсивним шрифтом із засічками, що надає їй виразного та стилізованого вигляду.

У правій частині розташовані дві кнопки з темним напівпрозорим фоном: «New Game», що ініціює завантаження ігрової локації через SceneManager, та «Quit», що завершує роботу застосунку.

Логіка головного меню реалізована у класі MenuManager.



Рис. 3.6 Головне меню гри

Під час гри у верхній частині екрану відображається мінімалістичний HUD, що містить два елементи. Зліва розміщений індикатор здоров'я у вигляді горизонтального ряду піксельних сердець рожевого кольору - кожне серце відповідає одній одиниці здоров'я персонажа. При отриманні шкоди відповідне серце зникає, при підборі аптечки - відновлюється. Управління станом індикатора здійснюється класом HealthBar через компонент UI.Slider. Праворуч від сердець розміщений лічильник зібраних кристалів: зелена піксельна іконка кристала супроводжується числовим значенням, що оновлюється в реальному часі через клас MoneyText при кожному підборі. Обидва елементи виконані у піксельному стилі, що відповідає загальній графічній концепції гри і органічно вписується у візуальне середовище.



Рис. 3.7 HUD

Меню паузи викликається натисканням клавіші Escape під час гри та відображається поверх поточного ігрового кадру без переходу до окремої сцени. Фоном слугує розмитий, напівпрозорий синій блок, крізь який залишається видимим ігровий світ разом із персонажем. По центру екрану вертикально розміщені три текстові кнопки без фонових блоків: «Resume» - для повернення до гри, «Menu» - для переходу до головного меню, та «Quit» - для завершення застосунку. Мінімалістичний підхід до оформлення паузи підкреслює, що гравець лише тимчасово зупинив сесію, а не перейшов до повноцінного окремого екрану. Логіка паузи реалізована у класі GameMenu: при виклику меню встановлюється статична змінна isPaused та зупиняється ігровий час через `Time.timeScale = 0`.

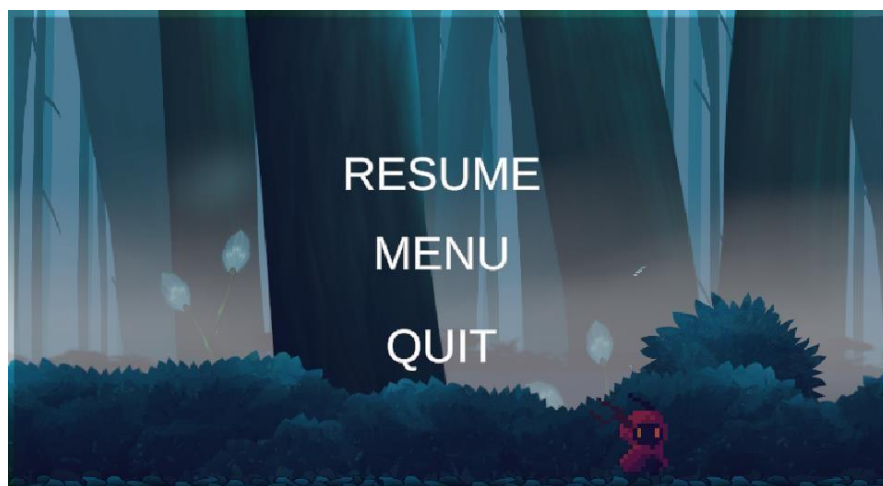


Рис. 3.8 Меню паузи

Екран смерті є найбільш виразним елементом інтерфейсу з художньої точки зору. При загибелі персонажа поверх ігрового зображення з'являється темний напівпрозорий прямокутний блок, у центрі якого великим червоним шрифтом із засічками виведено напис «YOU DIED». Насичений червоний колір тексту на темному тлі формує різкий емоційний контраст із холодною синьою палітрою ігрового середовища, підкреслюючи серйозність події. По обидва боки від центрального напису розміщені дві кнопки з темним фоном: «Menu» ліворуч та «Restart» праворуч. Такий симетричний розподіл кнопок є навмисним дизайнерським рішенням - гравець у момент емоційного напруження одразу бачить обидві опції без необхідності шукати їх поглядом. Логіка екрану смерті реалізована

у класі `GameOverManager`, що активується викликом методу `GameOver()` з класу `Health` при досягненні нульового значення здоров'я або при контакті з водою.



Рис. 3.9 Екран смерті

Таким чином, система інтерфейсів проєкту побудована на принципі мінімалізму та художньої узгодженості: кожен екран підпорядкований загальній темній атмосфері гри, не перевантажений зайвими елементами та надає гравцеві рівно стільки інформації, скільки необхідно для прийняття рішень у кожній конкретній ситуації.

3.4 Тестування розробленої гри

Тестування є обов'язковим етапом розробки програмного забезпечення, що передуює його публічному випуску або захисту як проєктного продукту. Метою тестування є виявлення дефектів, перевірка відповідності реалізованого функціоналу специфікованим вимогам та підтвердження стабільності роботи застосунку в різних умовах використання. У контексті ігрового проєкту тестування набуває додаткового виміру: окрім технічної коректності, перевіряється ігровий досвід - відгук управління, логіка прогресії та узгодженість усіх ігрових систем між собою. Відсутність належного тестування навіть технічно коректного коду може призвести до випуску продукту з критичними порушеннями ігрового балансу, некоректною поведінкою персонажа або нестабільною роботою інтерфейсів, що безпосередньо впливає на якість ігрового досвіду та загальне сприйняття продукту

гравцем. Таким чином, тестування є не факультативним, а невід'ємним компонентом процесу розробки, що забезпечує відповідність між задекларованими вимогами та реалізованим функціоналом.

Процес тестування програмного забезпечення в загальному випадку охоплює кілька рівнів: від перевірки окремих функцій і методів до оцінки поведінки системи в цілому в умовах реального використання. Вибір конкретного методу або поєднання методів тестування залежить від масштабу проєкту, наявних ресурсів, технологічного стеку та характеру самого продукту. У сучасній практиці програмної інженерії виділяють кілька основних методів тестування, кожен із яких має власну область застосування, переваги та обмеження.

Модульне тестування (Unit Testing) - метод, за якого окремі структурні одиниці коду - методи, класи або функції - перевіряються ізольовано від решти системи з використанням автоматизованих тест-скриптів. Ідея методу полягає у тому, що кожна функціональна одиниця перевіряється незалежно: для цього визначаються вхідні дані, що подаються на вхід методу, та очікуваний результат, із яким порівнюється фактичний вихід.

Перевагою модульного тестування є висока швидкість виконання тестів та можливість їхнього автоматичного запуску після кожної зміни кодової бази, що дозволяє миттєво виявляти регресії - ситуації, коли нові зміни порушили вже перевірену функціональність. Цей підхід широко застосовується у промисловій розробці програмного забезпечення, зокрема у методологіях TDD (Test-Driven Development), де тести пишуться до написання самого коду, та BDD (Behavior-Driven Development), де тести формулюються у термінах бізнес-поведінки системи.

Однак для ігрового проєкту на Unity модульне тестування має суттєве структурне обмеження: переважна більшість ігрових компонентів наслідують базовий клас `MonoBehaviour`, що робить їх нерозривно пов'язаними з ігровим циклом рушія. Методи таких компонентів - `Start()`, `Update()`, `OnTriggerEnter2D()` та інші - викликаються не безпосередньо з коду, а рушієм у певні моменти ігрового циклу, що унеможливорює їхнє ізольоване тестування без запущеного екземпляра Unity. Ізольоване тестування таких компонентів потребує застосування спеціалізованих

фреймворків, таких як Unity Test Framework у режимі Play Mode, що суттєво ускладнює процес та значно збільшує обсяг підготовчої роботи.

Крім того, принципове обмеження модульного тестування полягає у тому, що воно перевіряє лише ізольовану поведінку окремих методів, але не здатне виявити помилки, що виникають на рівні взаємодії між компонентами - наприклад, некоректну передачу даних між класами Health та HealthBar або неправильну послідовність подій між GameOverManager та SceneManager.

Інтеграційне тестування (Integration Testing) - метод, що перевіряє коректність взаємодії між кількома компонентами або підсистемами системи, що були розроблені та протестовані окремо. Якщо модульне тестування перевіряє, чи правильно працює окремий «цеглинка», то інтеграційне тестування перевіряє, чи коректно ці «цеглинки» з'єднані між собою.

У контексті ігрового проєкту інтеграційне тестування могло б охоплювати, наприклад, перевірку коректності взаємодії між класами Health та GameOverManager при вичерпанні здоров'я, між класами Coins та MoneyText при підборі кристала, або між класами GameMenu та SceneManager при переході між сценами. Кожен із цих сценаріїв передбачає взаємодію кількох компонентів, що не може бути перевірена на рівні ізольованого модульного тесту.

Однак інтеграційне тестування в Unity також потребує значної інфраструктури автоматизованих тестів і, як правило, доцільне у великих командних проєктах із чітко розподіленими підсистемами та тривалим циклом підтримки. Для індивідуального проєкту обмеженого масштабу витрати на написання, налаштування та підтримку інтеграційних тестів суттєво перевищують практичну користь від їхнього застосування, тим більше що більшість відповідних сценаріїв може бути перевірена в рамках функціонального тестування методом чорного ящика.

Регресійне тестування (Regression Testing) - метод, що полягає у повторному виконанні раніше пройдених тестів після внесення змін до кодової бази з метою підтвердження того, що нові зміни не порушили вже перевірену функціональність. Цей метод відповідає на питання: «чи не зламали ми щось нові змінами?».

Регресійне тестування є надзвичайно важливим у великих та тривалих проєктах із великою командою розробників, де зміни в одній підсистемі можуть непередбачувано впливати на поведінку інших. Наприклад, зміна у класі Health може порушити логіку GameOverManager або HealthBar, навіть якщо самі ці класи не змінювались.

Ефективне регресійне тестування базується на автоматизованому наборі тестів, що може бути запущений після кожного коміту до репозиторію - наприклад, у рамках конвеєра CI/CD (Continuous Integration / Continuous Deployment). Однак це знову повертає до проблем, властивих модульному та інтеграційному підходам у контексті Unity: без наявної автоматизованої тестової бази регресійне тестування зводиться до ручного повторного проходження всіх перевірок після кожної зміни, що є трудомістким і практично нездійсненним для одного розробника в умовах обмеженого часу.

Навантажувальне тестування (Performance Testing) - метод, що оцінює поведінку системи під інтенсивним навантаженням та вимірює кількісні показники продуктивності: частоту кадрів, час відгуку, споживання оперативної та відеопам'яті, завантаженість процесора тощо. Цей метод є критично важливим для великих ігрових проєктів із відкритим світом, великою кількістю одночасних об'єктів або складними алгоритмами штучного інтелекту. У контексті двовимірної інді-гри обмеженого масштабу, що не передбачає одночасної обробки тисяч об'єктів або розрахунків ресурсоємних алгоритмів, навантажувальне тестування у повному обсязі не є доцільним.

Тестування зручності використання (Usability Testing) - метод, що оцінює якість ігрового досвіду з точки зору реального кінцевого користувача: зрозумілість інтерфейсу, інтуїтивність управління, навчальна крива та загальна задоволеність продуктом. Однак повноцінне usability-тестування передбачає залучення репрезентативної групи тестових учасників, розробку сценаріїв спостереження, збір та статистичний аналіз відгуків, що виходить за межі ресурсних і часових можливостей індивідуального академічного проєкту та потребує відповідної методологічної підготовки.

З огляду на проведений аналіз усіх розглянутих методів для тестування даного проєкту було обрано метод функціонального тестування методом чорного ящика (Black-Box Testing) у поєднанні з ручним дослідницьким тестуванням (Exploratory Testing). Такий вибір обґрунтовується комплексом взаємопов'язаних аргументів технічного, методологічного та практичного характеру.

Метод чорного ящика передбачає перевірку функціональності системи без звернення до внутрішньої реалізації коду: тестувальник взаємодіє із застосунком як звичайний користувач, перевіряючи відповідність вхідних дій та вихідних результатів заздалегідь визначеним очікуванням. Цей підхід є найбільш природним для ігрового застосунку, оскільки кінцева цінність продукту визначається виключно якістю взаємодії користувача з системою, а не деталями її програмної реалізації. Гравець не має доступу до внутрішнього стану системи і взаємодіє з нею виключно через ігровий інтерфейс та клавіатуру, тому тестування з тієї самої позиції є найбільш релевантним підходом до оцінки якості продукту.

Важливою перевагою методу чорного ящика є також його незалежність від технічних деталей реалізації: навіть суттєва зміна внутрішньої логіки класу не потребує перегляду тест-кейсів, якщо зовнішня поведінка системи залишається незмінною.

Функціональне тестування на основі формалізованих тест-кейсів забезпечує систематичне та повне охоплення всіх задокументованих функціональних вимог до продукту: кожна задекларована функція гри отримує відповідний тест-кейс із чітко визначеними передумовами, послідовністю дій та критерієм успішності. Така структурованість дозволяє об'єктивно підтвердити або спростувати відповідність реалізованого функціоналу специфікації, а формалізована таблиця тест-кейсів слугує документальним свідченням проведеного тестування, що є важливою складовою академічного проєкту.

Дослідницьке тестування органічно доповнює структурований підхід, компенсуючи його основний методологічний недолік - неможливість передбачити на етапі складання тест-кейсів усі можливі сценарії використання, особливо нетипові та граничні. Вільне дослідження застосунку без заздалегідь визначеного

сценарію дозволяє виявити граничні випадки, нестандартні послідовності дій та непередбачені комбінації ігрових ситуацій, що не були охоплені формальними тестами. Саме в рамках дослідницького тестування найчастіше виявляються дефекти, пов'язані зі взаємодією між підсистемами - наприклад, некоректна поведінка при одночасному виконанні кількох дій гравцем або рідкісні послідовності переходів між меню.

Поєднання цих двох підходів забезпечує оптимальний баланс між систематичністю та гнучкістю тестування в умовах індивідуальної розробки, не потребуючи при цьому значної інфраструктури автоматизованих тестів і залишаючись практично здійсненним для одного розробника без залучення додаткових ресурсів.

Перед початком тестування було підготовлено тестову документацію у вигляді таблиці тест-кейсів, що охоплювала всі задокументовані функціональні вимоги до гри. Структура кожного тест-кейсу включає унікальний ідентифікатор для однозначної ідентифікації тесту, короткий опис сценарію, що перевіряється, конкретну послідовність дій тестувальника, очікуваний результат відповідно до специфікації, фактичний результат, отриманий під час виконання тесту, та підсумковий статус виконання - «Пройдено» або «Не пройдено».

Тестування проводилось на цільовій платформі - персональному комп'ютері під керуванням операційної системи Windows - у збірці гри, скомпільованій засобами Unity у режимі Development Build. Цей режим, на відміну від фінальної Release Build, зберігає налагоджувальну інформацію та дозволяє відстежувати консольні повідомлення, попередження та виключення рушія в реальному часі через вбудовану консоль Unity, що суттєво спрощує локалізацію та діагностику виявлених дефектів.

Усього було сформовано 24 тест-кейси відповідно до основних підсистем гри: головне меню, управління персонажем, система здоров'я та смерті, екран смерті, меню паузи, інтерактивні елементи рівня, поведінка ворогів та продуктивність. Повна таблиця тест-кейсів із результатами виконання наведена у таблиці 3.1.

Таблиця 3.1

Тест-кейси

ID	Сценарій	Дії	Очікуваний результат	Фактичний результат	Статус
ТС-01	Запуск нової гри	Натиснути «New Game» у головному меню	Ігрова сцена завантажується, персонаж з'являється у початковій позиції	Відповідає очікуванню	Пройдено
ТС-02	Вихід з гри	Натиснути «Quit» у головному меню	Застосунок завершує роботу	Відповідає очікуванню	Пройдено
ТС-03	Відображення фону меню	Запустити застосунок, переглянути головне меню	Фонове зображення відображається коректно, назва гри та кнопки видимі	Відповідає очікуванню	Пройдено
ТС-04	Горизонтальний рух	Утримувати клавіші A та D	Персонаж рухається у відповідному напрямку, анімація бігу активується, спрайт дзеркально відображається при зміні напрямку	Відповідає очікуванню	Пройдено
ТС-05	Стрибок	Натиснути Space на платформі та під час бігу	Персонаж отримує вертикальний імпульс, анімація стрибка активується, приземлення без проникнення у геометрію	Відповідає очікуванню	Пройдено

Продовження таблиці 3.1

Тест-кейси

ID	Сценарій	Дії	Очікуваний результат	Фактичний результат	Статус
ТС-06	Отримання шкоди від ворога	Підійти до ворога та допустити контакт	Здоров'я зменшується на відповідну величину, HealthBar оновлюється	Відповідає очікуванню	Пройдено
ТС-07	Отримання шкоди від шипів	Наступити на шипи	Здоров'я зменшується через EnemyDamage, HealthBar оновлюється	Відповідає очікуванню	Пройдено
ТС-08	Отримання шкоди від пили	Доторкнутись до рухомої пили	Здоров'я зменшується, HealthBar оновлюється	Відповідає очікуванню	Пройдено
ТС-09	Миттєва смерть від води	Впасти у водойму	Персонаж гине миттєво незалежно від рівня здоров'я, сцена перезапускається через SceneManager	Відповідає очікуванню	Пройдено
ТС-10	Смерть від вичерпання НР	Отримувати шкоду до нуля здоров'я	Викликається Die(), з'являється екран смерті «You Died», ігровий процес зупиняється	Відповідає очікуванню	Пройдено
ТС-11	Підбір аптечки-серця	Підійти до серця на рівні	НР збільшується на 1, HealthBar оновлюється, звук відтворюється через SoundManager	Відповідає очікуванню	Пройдено

Продовження таблиці 3.1

Тест-кейси

ID	Сценарій	Дії	Очікуваний результат	Фактичний результат	Статус
ТС-12	Рестарт гри	На екрані смерті натиснути «Restart»	Сцена перезавантажується, персонаж у початковій позиції з повним НР, кристали скинуті до 0	Відповідає очікуванню	Пройдено
ТС-13	Повернення до меню зі смерті	На екрані смерті натиснути «Menu»	Застосунок повертається до головного меню, повторний запуск гри працює коректно	Відповідає очікуванню	Пройдено
ТС-14	Виклик паузи	Натиснут и Escape під час гри	Time.timeScale = 0, меню паузи відображається поверх гри, об'єкти заморожені	Відповідає очікуванню	Пройдено
ТС-15	Відновлення гри	У меню паузи натиснути «Resume»	Time.timeScale = 1, меню зникає, гра відновлюється з того самого місця	Відповідає очікуванню	Пройдено
ТС-16	Перехід до меню з паузи	У меню паузи натиснути «Menu»	Time.timeScale скидається до 1, відбувається перехід до головного меню без заморозки	Дефект: Time.timeScale не скидався - виправлено	Пройдено
ТС-17	Вихід з паузи	У меню паузи натиснути «Quit»	Застосунок завершує роботу	Відповідає очікуванню	Пройдено
ТС-18	Механіка кнопок та стін	Наступити на кнопку-тригер	Стіна плавно переміщується до заданої позиції, прохід відкривається	Відповідає очікуванню	Пройдено

Продовження таблиці 3.1

Тест-кейси

ID	Сценарій	Дії	Очікуваний результат	Фактичний результат	Статус
ТС-19	Підбір кристалів	Підійти до кристала на рівні	Лічильник MoneyText збільшується на 1, звук відтворюється через SoundManager	Відповідає очікуванню	Пройдено
ТС-20	Патрулювання ворога	Спостерігати за EnemyPatrol	Ворог коректно змінює напрямок між крайніми точками, спрайт дзеркально відображається	Відповідає очікуванню	Пройдено
ТС-21	Атака ворога	Підійти до ворога на відстань атаки	Ворог ініціює атаку з коректним значенням шкоди та кулдауном між ударами	Відповідає очікуванню	Пройдено
ТС-22	Загибель ворога	Атакувати ворога до нуля HP	Анімація смерті відтворюється, об'єкт деактивується, залишкових колайдерів немає	Відповідає очікуванню	Пройдено
ТС-23	Частота кадрів	Запустити гру, пройти повну локацію з увімкненим Unity Profiler	Стабільні 60 FPS без критичних просідань	Середнє 60 FPS, пікові просідання до 54–56 FPS при дощових ефектах	Пройдено
ТС-24	Споживання пам'яті	Провести ігрову сесію 30 хв з моніторингом через Unity Profiler	Споживання RAM не перевищує 400 МБ, відсутні витоки пам'яті	Максимум 350 МБ, витоків не виявлено	Пройдено

Блок тестування головного меню (ТС-01 - ТС-03) підтвердив стабільність та передбачуваність базових сценаріїв взаємодії користувача із застосунком на початковому етапі роботи. Зокрема, перевірка запуску нової гри засвідчила коректну ініціалізацію ігрової сцени, включно з правильним завантаженням усіх необхідних об'єктів та компонентів. Функція завершення роботи застосунку спрацювала без затримок і побічних ефектів, що є критично важливим для забезпечення коректного завершення процесів. Також було підтверджено належний рівень візуального оформлення головного меню: елементи інтерфейсу відображаються без артефактів, що позитивно впливає на перше враження користувача.

Блок тестування управління персонажем (ТС-04 - ТС-05) дозволив оцінити якість реалізації базових механік геймплею. Горизонтальний рух персонажа функціонує плавно та передбачувано, без ривків або затримок, що свідчить про коректну обробку вводу користувача. Окрему увагу приділено синхронізації анімацій із рухом, що забезпечує цілісність візуального сприйняття. Механіка стрибка була протестована в різних умовах, включаючи рух, що дозволило переконатися у відсутності фізичних аномалій, таких як «залипання» або некоректна взаємодія з поверхнями рівня.

Блок тестування системи здоров'я та смерті (ТС-06 - ТС-11) охопив широкий спектр сценаріїв, пов'язаних із отриманням шкоди та реакцією гри на критичні стани персонажа. Проведене тестування підтвердило узгодженість між логікою обробки шкоди та візуальним відображенням стану здоров'я. Всі джерела шкоди працюють відповідно до заданих параметрів, що забезпечує передбачуваний ігровий процес. Особливо важливою є перевірка крайніх випадків, зокрема миттєвої смерті, яка була реалізована без конфліктів із загальною логікою гри. Крім того, механіка відновлення здоров'я була протестована на предмет коректності інтеграції з аудіосистемою та UI, що підтвердило цілісність ігрового досвіду.

Блок тестування екрану смерті (ТС-12 - ТС-13) продемонстрував коректну обробку стану завершення ігрової сесії. Було підтверджено, що після смерті персонажа гра переходить у відповідний стан без зависань або втрати контролю.

Обидва варіанти подальших дій користувача працюють стабільно та не спричиняють накопичення помилок або залишкових даних. Це свідчить про правильну реалізацію механізмів очищення стану гри перед повторним запуском або поверненням до меню.

Блок тестування меню паузи (ТС-14 - ТС-17) дозволив перевірити коректність призупинення ігрового процесу та його подальшого відновлення. Загалом функціональність паузи реалізована відповідно до очікувань: усі динамічні процеси зупиняються, а інтерфейс залишається доступним для взаємодії. Виявлений дефект у сценарії переходу до головного меню під час паузи вказав на важливість контролю глобальних параметрів часу гри. Його усунення підтвердило стабільність архітектури після внесення змін, а повторне тестування довело відсутність регресій у суміжних функціях.

Блок тестування інтерактивних елементів рівня (ТС-18 - ТС-19) показав, що механіки взаємодії з оточенням працюють узгоджено та без конфліктів. Зокрема, реакція об'єктів на дії гравця відбувається своєчасно та відповідає ігровій логіці. Це є важливим аспектом, оскільки подібні елементи безпосередньо впливають на геймплей та залученість користувача. Також підтверджено, що система підрахунку ресурсів інтегрована з інтерфейсом та аудіосупроводом без помилок.

Блок тестування поведінки ворогів (ТС-20 - ТС-22) дозволив оцінити коректність реалізації штучного інтелекту на базовому рівні. Поведінка ворогів є передбачуваною та відповідає заданим сценаріям, що важливо для балансу складності гри. Перевірка бойових механік показала, що система атак працює стабільно, без надмірної частоти або затримок. Крім того, було підтверджено коректне завершення життєвого циклу ворогів, що запобігає накопиченню невидимих або неактивних об'єктів у сцені.

Блок тестування продуктивності (ТС-23 - ТС-24) був спрямований на оцінку стабільності роботи застосунку в умовах тривалого використання. Отримані результати свідчать про оптимізоване використання ресурсів системи. Частота кадрів залишається на прийнятному рівні навіть у динамічних сценах, що забезпечує комфортний ігровий процес. Аналіз використання пам'яті показав

відсутність витоків, що є критичним показником якості програмної реалізації та свідчить про правильне управління ресурсами.

Загалом результати функціонального тестування демонструють високий рівень стабільності та готовності програмного продукту до використання.

Усі основні ігрові механіки, інтерфейсні елементи та системні компоненти працюють узгоджено, що підтверджує відповідність застосунку поставленим вимогам. Виявлений під час тестування дефект був оперативно усунений і не вплинув на загальну якість фінальної збірки.

Отже, розроблена гра «Roots of Silence» може вважатися успішно протестованою та готовою до подальшого використання або демонстрації.

ВИСНОВКИ

У процесі виконання кваліфікаційної роботи бакалавра на тему «Розробка 2D пригодницької гри для ПК» було досягнуто поставленої мети та вирішено всі визначені завдання. Отримані результати свідчать про комплексний характер виконаної роботи, яка охоплює як теоретичні аспекти досліджуваної предметної області, так і практичну реалізацію програмного продукту. Послідовне виконання поставлених завдань забезпечило досягнення логічної цілісності дослідження та відповідність кінцевого результату початково сформульованим вимогам.

У першому розділі проведено комплексне дослідження теми двовимірних платформерів. Здійснено аналіз ринку 2D-платформерів, досліджено його поточний стан, тенденції розвитку та місце інді-ігор у сучасній ігровій індустрії. Розглянуто взаємодію технічних засобів та функціональних можливостей сучасних 2D ігор: проаналізовано еволюцію від технічно обмежених ігор класичної епохи до можливостей сучасних інді-проектів. Визначено ключові складові ретро-платформера - систему управління персонажем, рівневий дизайн із принципом поступового ускладнення, аудіосупровід та роль ностальгії як соціокультурного чинника популярності жанру. На основі проведеного дослідження сформовано власну концепцію гри «Roots of Silence», що ґрунтується на трьох принципах: єдина безперервна локація, відсутність чекпоінтів та акцент на авторському ручному дизайні.

Додатково слід зазначити, що проведений аналіз дозволив не лише узагальнити існуючі підходи до розробки ігор даного жанру, але й виявити їхні сильні та слабкі сторони, що стало підґрунтям для формування обґрунтованих проєктних рішень. Особлива увага приділялась питанням користувацького досвіду та залученості гравця, що є ключовими факторами успішності сучасних ігрових продуктів. Таким чином, теоретичне дослідження виконало роль фундаменту для подальшої практичної реалізації.

У другому розділі здійснено вибір технічних засобів та проектування комп'ютерної гри. Проведено порівняльний аналіз доступних ігрових рушіїв та обґрунтовано вибір Unity як основного інструменту розробки у поєднанні з мовою програмування C#, платформами Unity Asset Store та itch.io як джерелами графічних ресурсів. Розроблено флоу-діаграму ігрового процесу, що відображає повний цикл взаємодії гравця з грою від запуску до завершення. Побудовано діаграму класів, що описує архітектуру програмного коду гри з чотирма функціональними групами: класи гравця, класи ворогів, класи оточення та системні менеджери.

Здійснений вибір технологічного стеку базується на критеріях продуктивності, масштабованості та доступності інструментів розробки, що забезпечує ефективність реалізації програмного продукту. Використання об'єктно-орієнтованого підходу при проектуванні архітектури системи дозволило забезпечити модульність та гнучкість коду, що є важливим для подальшої підтримки та розширення функціональності гри.

Розроблено діаграму станів класу Hero, що моделює поведінку головного персонажа через машину станів із чотирма основними станами та визначеними переходами між ними. Побудовано діаграму прецедентів, що формально описує повну множину функціональних можливостей гравця в межах системи. Застосування зазначених інструментів моделювання дозволило формалізувати поведінку системи та забезпечити узгодженість між різними її компонентами ще на етапі проектування. Це, у свою чергу, сприяло зменшенню кількості помилок під час реалізації та підвищенню якості кінцевого програмного продукту.

У третьому розділі виконано безпосередню розробку комп'ютерної гри. Розроблено систему персонажів: головного героя - духа-охоронця лісу - та трьох типів ворогів-жаб з різними параметрами складності, кольоровою гамою та анімаційними наборами. Реалізовано єдину ігрову локацію з ручним розміщенням спрайтів, інтерактивними елементами - кнопками, рухомими стінами, шипами, рухомою пилою, водоймами та драбинами, - навчальною системою внутрішньоігрових табличок та атмосферними ефектами дощу.

Розроблено повний набір ігрових інтерфейсів: головне меню з назвою гри та кнопками навігації, HUD із індикатором здоров'я та лічильником кристалів, меню паузи з трьома опціями та виразний екран смерті «You Died».

Проведено функціональне тестування методом чорного ящика з формуванням 24 тест-кейсів, що охоплюють усі задокументовані функціональні вимоги. За результатами тестування всі тест-кейси отримали статус «Пройдено», один виявлений дефект усунено до фінальної збірки. У процесі розробки особлива увага приділялась узгодженості між візуальними, функціональними та технічними компонентами гри. Реалізація інтерактивних елементів була спрямована на створення цілісного ігрового середовища, яке стимулює дослідження та взаємодію. Проведене тестування підтвердило стабільність роботи системи та відповідність реалізованого функціоналу поставленим вимогам, що є важливим показником якості програмного забезпечення.

Таким чином, у результаті виконання кваліфікаційної роботи бакалавра розроблено повноцінну двовимірну пригодницьку гру «Roots of Silence» для персонального комп'ютера, що поєднує класичні механіки жанру платформера з авторською художньою концепцією містичного лісового світу та відповідає всім сформульованим функціональним вимогам. Отриманий результат демонструє можливість створення якісного ігрового продукту в межах індивідуального проєкту з використанням сучасних інструментів розробки. Поєднання теоретичних знань і практичних навичок дозволило реалізувати програмний продукт, що відповідає сучасним вимогам до подібного типу застосунків.

Практична цінність роботи полягає у створенні завершеного ігрового продукту, придатного для подальшого розширення контенту та розвитку як повноцінного комерційного інді-проєкту. Крім того, результати роботи можуть бути використані як основа для подальших досліджень у галузі розробки комп'ютерних ігор, а також як приклад реалізації проєкту подібного типу в освітніх цілях. Це підкреслює універсальність і прикладний характер отриманих результатів.

Робота пройшла апробацію. За її результатами було опубліковано наступні тези доповідей:

1. Вірич А.М., Щербина І.С.. Розробка 2D пригодницької гри з елементами головоломки. Матеріали Всеукраїнської науково-технічної конференції "Виклики та рішення в програмній інженерії", 26 листопада 2025 року, Київ, ДУІКТ. Збірник тез. – К.: ДУІКТ, 2025. С.384-385.

2. Вірич А.М., Щербина І.С. Дослідження ефекту ностальгії як чиннику емоційної залученості у класичних 2D платформах. Матеріали Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 23 квітня 2026 року, Київ, ДУІКТ. Збірник тез. – К.: ДУІКТ, 2026. С.122-123.

ПЕРЕЛІК ПОСИЛАНЬ

1. Unity Technologies. Unity Manual: 2D Game Development. Unity Documentation, 2023. URL:
<https://docs.unity3d.com/6000.4/Documentation/Manual/Unity2D.html>
2. Unity Technologies. Unity Scripting API: MonoBehaviour. Unity Documentation, 2023. URL:
<https://docs.unity3d.com/ScriptReference/MonoBehaviour.html>
- Microsoft. C# Language Reference. Microsoft Learn, 2024. URL:
<https://learn.microsoft.com/en-us/dotnet/csharp/language-reference>
3. Marcus Toftedahl, Henrik Engström. “A Taxonomy of Game Engines and the Tools that Drive the Industry”. Proceedings of the 13th International Conference on the Foundations of Digital Games, 2019. URL:
https://www.researchgate.net/publication/337165844_A_Taxonomy_of_Game_Engines_and_the_Tools_that_Drive_the_Industry
4. Peter D. McDonald. Run and Jump: The Meaning of the 2D Platformer, New York, 2024. 240 pages.
5. Alan Dennis, Barbara Haley Wixom, David Tegarden. Systems Analysis and Design: An Object-Oriented Approach with UML, 6th edition, Hoboken, 2021. 656 pages.
6. Richard Carrillo. The Role of a Great Game Designer, London, 2021. 198 pages.
7. Ian Schreiber, Brenda Romero. Game Balance, Boca Raton, 2021. 672 pages.
8. B. Shamile. Software Development with UML Diagrams, 2022. 320 pages.
9. Raph Koster. A Theory of Fun for Game Design, Scottsdale, 2022. 288 pages.
10. Andrew Troelsen, Philip Japikse. Pro C# 7: With .NET and .NET Core, 8th edition, New York, 2020. 1328 pages.

11. Brenda Romero. The Game Balance Concept, Boca Raton, 2024. 832 pages.
12. Chris Scullion. Jumping for Joy: The History of Platform Video Games: Including Every Mario and Sonic Platformer, Barnsley, 2022. 168 pages.
13. Dan Peel. The Ultimate 80's Retro Gaming Collection, Manchester, 2021. 160 pages.
14. Jeffrey Richter. CLR via C#, 4th edition, Redmond, 2012. 826 pages.
15. Dorothy Graham, Rex Black, Erik van Veenendaal. Foundations of Software
16. Testing: ISTQB Certification, 4th edition, Boston, 2020. 288 pages.
17. Glenford J. Myers, Tom Badgett, Corey Sandler. The Art of Software Testing, 3rd edition, Hoboken, 2020. 272 pages.
18. Святослав Куликов. Тестування програмного забезпечення. Базовий курс, 3-є видання, Київ, 2024. 312 с.
19. Rex van der Spuy. Beginning Game Programming with Unity, 4th edition, New York, 2021. 380 pages.
20. Tracy Fullerton. Game Design Workshop: A Playcentric Approach to Creating Innovative Games, 5th edition, Boca Raton, 2023. 556 pages.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ



ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ



Розробка 2D пригодницької гри для персональних комп'ютерів

Виконала студентка 4 курсу
групи ПД-42
Вірич Анна Максимівна
Керівник роботи
доц. каф., канд. техн. наук, доц. Щербина Ірина Сергіївна

Київ – 2026

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: підвищити зручність та ефективність взаємодії користувача з продуктом (2D платформером), забезпечивши баланс між ретро-естетикою та сучасними стандартами якості.

Об'єкт дослідження: процес створення 2D пригодницьких ігор для персональних комп'ютерів.

Предмет дослідження: засоби та технології розробки 2D платформерів, що охоплюють ігрові рушії, графічні та аудіо інструменти, а також програмні методи реалізації ігрової логіки.

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Провести аналіз літературних джерел та існуючих прикладів 2D платформерів, визначити їхні ключові особливості та вплив на сучасну ігрову індустрію.
2. Здійснити огляд сучасних інструментів та технологій для створення 2D ігор, визначити їх переваги та недоліки.
3. Сформулювати вимоги до розробки гри, включаючи технічні та візуальні аспекти.
4. Визначити програмні засоби для реалізації гри (ігровий рушій, мова програмування, бібліотеки).
5. Розробити архітектуру гри та реалізувати основні ігрові механіки.
6. Провести тестування гри, оцінити її функціональність та відповідність поставленим вимогам.

2

АНАЛІЗ АНАЛОГІВ

Критерій/Гра	Hollow Knight	Ori and the Blind Forest	Celeste	Roots of Silence
Низькі системні вимоги	Оперативна пам'ять (RAM): 4 GB. Відеокарта: GeForce 9800 GTX+	Оперативна пам'ять (RAM): 4 GB. Відеокарта: GeForce 240 GT	Оперативна пам'ять (RAM): 2 GB. Відеокарта: Intel HD 4000	Оперативна пам'ять (RAM): 4 GB. Відеокарта: GeForce 240 GT
Низький поріг входження для новачків	-	-	-	+
Наявність звукового супроводу	+	+	+	+
Наявність заохочень (збір очок)	+	+	+	+
Нестатичні вороги (патрулювання)	+	+	+	+

3

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги	Нефункціональні вимоги
Рух персонажа: біг, стрибок, рух драбиною	Частота кадрів ≥ 60 fps
Система здоров'я гравця та ворогів	Input lag ≤ 100 мс
Коректна взаємодія гравця з ігровими об'єктами (кнопки, вороги, пастки, алмази, аптечки)	Час відгуку інтерфейсу користувача ≤ 100 мс
Коректна поведінка ворогів (патрулювання, визначення гравця, атака)	Кількість критичних помилок = 0 впродовж всієї ігрової сесії
Музичний та звуковий супровід	Підтримка ОС Windows

4

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ



Мова
програмування C#



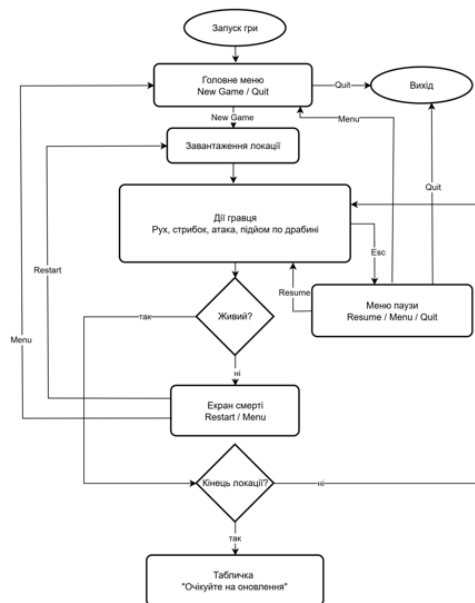
Застосунок Paint



Рушій Unity

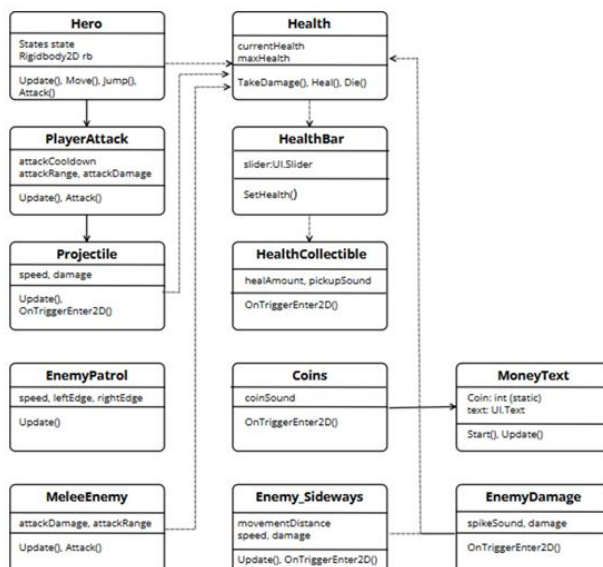
5

ФЛОУ-ДІАГРАМА ІГРОВОГО ПРОЦЕСУ



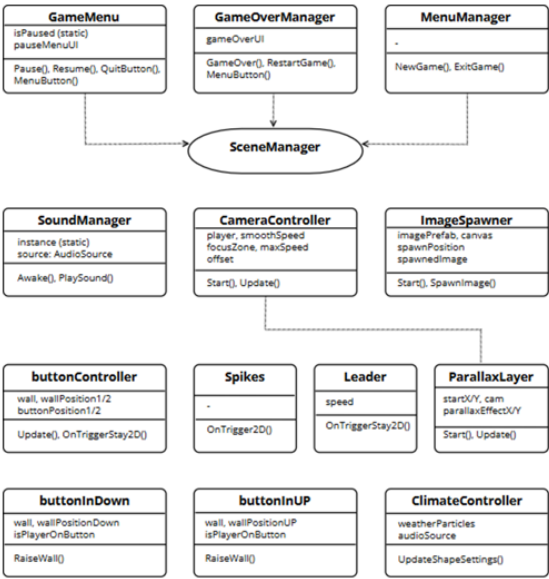
6

ДІАГРАМА КЛАСІВ: ГРАВЕЦЬ ТА ВОРОГИ

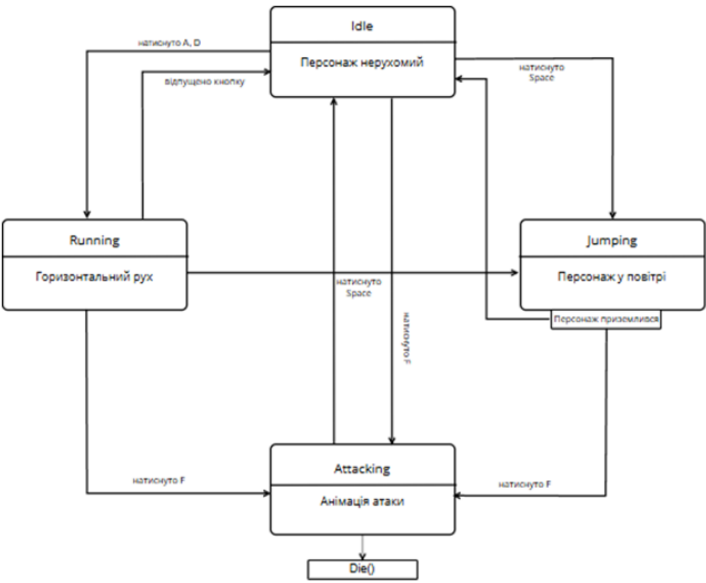


7

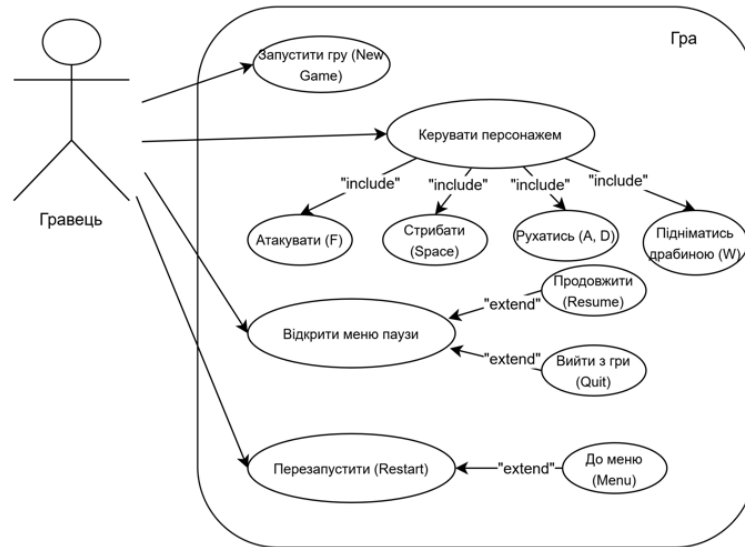
ДІАГРАМА КЛАСІВ: МЕНЕДЖЕРИ ТА ОТОЧЕННЯ



ДІАГРАМА СТАНІВ КЛАСУ HERO

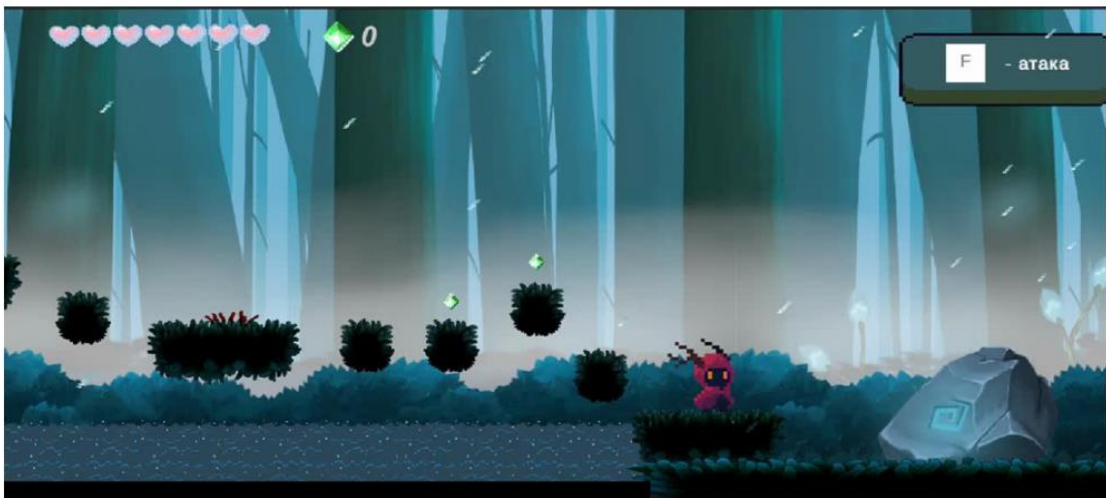


ДІАГРАМА ПРЕЦЕДЕНТІВ



10

ЕКРАННІ ФОРМИ



Демонстрація частини геймплею

11

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

Тези доповідей:

1. Вірич А.М., Щербина І.С.. Розробка 2D пригодницької гри з елементами головоломки. Матеріали всеукраїнської науково-технічної конференції "Виклики та рішення в програмній інженерії", 26 листопада 2025 року, Київ, ДУІКТ. Збірник тез. – К.: ДУІКТ, 2025. С.384-385.
2. Вірич А.М., Щербина І.С. Дослідження ефекту ностальгії як чиннику емоційної залученості у класичних 2D платформерах. Матеріали всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно- комунікаційних технологіях», 23 квітня 2026 року, Київ, ДУІКТ. Збірник тез. (подано до друку).

12

ВИСНОВКИ

1. Проведено аналіз літературних джерел та існуючих прикладів 2D платформерів, визначено їхні ключові особливості та вплив на сучасну ігрову індустрію.
2. Здійснено огляд сучасних інструментів та технологій для створення 2D ігор, визначено їхні переваги та недоліки.
3. Сформовано вимоги до розробки гри, що охоплюють технічні та візуальні аспекти.
4. Визначено програмні засоби для реалізації гри, включаючи вибір Unity та C#.
5. Розроблено архітектуру гри та реалізовано основні ігрові механіки.
6. Проведено тестування розробленої гри методом чорного ящика, сформована таблиця з 24 тест-кейсів.

13

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

```

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

public class EnemyDamage : MonoBehaviour
{
    [SerializeField] private AudioClip spikeSound;
    [SerializeField] protected float damage;

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.tag == "Player")
        {
            SoundManager.instance.PlaySound(spikeSound);
            collision.GetComponent<Health>().TakeDamage(damage);
        }
    }
}

using UnityEngine;

public class Enemy_Sideways : MonoBehaviour
{
    [SerializeField] private float movementDistance;
    [SerializeField] private float speed;
    [SerializeField] private float damage;
    private bool movingLeft;
    private float leftEdge;
    private float rightEdge;

    private void Awake()
    {
        leftEdge = transform.position.x - movementDistance;
        rightEdge = transform.position.x + movementDistance;
    }

    private void Update()
    {
        if (movingLeft)
        {
            if (transform.position.x > leftEdge)
            {
                transform.position = new
                Vector3(transform.position.x - speed * Time.deltaTime,
                transform.position.y, transform.position.z);
            }
            else
            {
                movingLeft = false;
            }
        }
        else
        {
            if (transform.position.x < rightEdge)
            {
                transform.position = new
                Vector3(transform.position.x + speed * Time.deltaTime,
                transform.position.y, transform.position.z);
            }
            else
            {
                movingLeft = true;
            }
        }
    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.tag == "Player")
        {
            SoundManager.instance.PlaySound(spikeSound);
            collision.GetComponent<Health>().TakeDamage(damage);
        }
    }
}

using UnityEngine;

public class Projectile : MonoBehaviour
{
    [SerializeField] private float speed;
    private float direction;
    private float lifetime;

    private BoxCollider2D boxCollider;

    private void Awake()
    {
        boxCollider = GetComponent<BoxCollider2D>();
    }

    private void Update()
    {
        // рухаємо фаєрбол вперед
        float movementSpeed = speed * Time.deltaTime *
        direction;
        transform.Translate(movementSpeed, 0, 0);

        // таймер життя (щоб не залишався вічно)
        lifetime += Time.deltaTime;
        if (lifetime > 5)
        {
            Debug.Log("Projectile lifetime expired, disabling");
            gameObject.SetActive(false);
        }
    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
        Debug.Log("Projectile collided with " + collision.name);

        if (collision.CompareTag("Enemy"))
        {
            // наносимо шкоду ворогу
            Health enemyHealth =
            collision.GetComponent<Health>();
            if (enemyHealth != null)
            {
                enemyHealth.TakeDamage(1);
            }

            // вимикаємо фаєрбол тільки при зіткненні з Enemy
            gameObject.SetActive(false);
        }
    }

    public void SetDirection(float _direction)
    {
        direction = _direction;
    }
}

```

```

        movingLeft = true;
    }
}

private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.tag == "Player")
    {
        SoundManager.instance.PlaySound(spikeSound);
        collision.GetComponent<Health>().TakeDamage(damage);
    }
}

using UnityEngine;

public class Projectile : MonoBehaviour
{
    [SerializeField] private float speed;
    private float direction;
    private float lifetime;

    private BoxCollider2D boxCollider;

    private void Awake()
    {
        boxCollider = GetComponent<BoxCollider2D>();
    }

    private void Update()
    {
        // рухаємо фаєрбол вперед
        float movementSpeed = speed * Time.deltaTime *
        direction;
        transform.Translate(movementSpeed, 0, 0);

        // таймер життя (щоб не залишався вічно)
        lifetime += Time.deltaTime;
        if (lifetime > 5)
        {
            Debug.Log("Projectile lifetime expired, disabling");
            gameObject.SetActive(false);
        }
    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
        Debug.Log("Projectile collided with " + collision.name);

        if (collision.CompareTag("Enemy"))
        {
            // наносимо шкоду ворогу
            Health enemyHealth =
            collision.GetComponent<Health>();
            if (enemyHealth != null)
            {
                enemyHealth.TakeDamage(1);
            }

            // вимикаємо фаєрбол тільки при зіткненні з Enemy
            gameObject.SetActive(false);
        }
    }

    public void SetDirection(float _direction)
    {
        direction = _direction;
    }
}

```

```

{
    lifetime = 0;
    direction = _direction;
    gameObject.SetActive(true);
    boxCollider.enabled = true;

    float xScale = Mathf.Abs(transform.localScale.x) *
(_direction > 0 ? 1 : -1);
    transform.localScale = new Vector3(xScale,
transform.localScale.y, transform.localScale.z);

    Debug.Log($"Projectile direction set to {_direction},
localScale: {transform.localScale}");
}
} using UnityEngine;
using UnityEngine.U2D;

public class PlayerAttack : MonoBehaviour
{
    [SerializeField] private float attackCooldown;
    [SerializeField] private Transform firePoint;
    [SerializeField] private GameObject[] fireballs;
    private AudioClip fireballSound;

    private Animator anim;
    private Hero hero;
    private float cooldownTimer = Mathf.Infinity;

    private void Awake()
    {
        anim = GetComponent<Animator>();
        hero = GetComponent<Hero>();
        Debug.Log("Awake(): Animator і Hero отримані");
    }

    private void Update()
    {
        Debug.Log("Update(): таймер = " + cooldownTimer);

        if (Input.GetKeyDown(KeyCode.F) && cooldownTimer
> attackCooldown)
        {
            Debug.Log("Update(): натиснута F, викликаємо
Attack()");
            Attack();
        }

        cooldownTimer += Time.deltaTime;
    }

    private void Attack()
    {
        Debug.Log("Attack(): старт атаки");

        hero.State = States.attack;
        Debug.Log("Attack(): стан героя = attack");

        cooldownTimer = 0;

        int fireballIndex = FindFireball();
        Debug.Log("Attack(): знайдений фаєрбол з індексом "
+ fireballIndex);

        if (fireballs[fireballIndex] == null)
        {
            Debug.LogError("Attack(): fireballs[" + fireballIndex +
"] = NULL!");
            return;
        }

        fireballs[fireballIndex].transform.position =
firePoint.position;
        Debug.Log("Attack(): позиція фаєрболу встановлена у
firePoint");

        float direction = hero.GetDirection();
        Debug.Log("Attack(): напрямок героя = " + direction);

        Projectile proj =
fireballs[fireballIndex].GetComponent<Projectile>();
        if (proj == null)
        {
            Debug.LogError("Attack(): у fireball немає
компонента Projectile!");
            return;
        }

        proj.SetDirection(direction);
        Debug.Log("Attack(): викликано SetDirection для
фаєрболу");
    }

    private int FindFireball()
    {
        for (int i = 0; i < fireballs.Length; i++)
        {
            if (fireballs[i] != null &&
!fireballs[i].activeInHierarchy)
            {
                Debug.Log("FindFireball(): повертаю індекс " + i);
                return i;
            }
        }
        Debug.LogWarning("FindFireball(): всі фаєрболи
активні або null, повертаю 0");
        return 0;
    }
} using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Hero : MonoBehaviour
{
    [SerializeField] private float speed = 3f;
    [SerializeField] private float jumpForce = 15f;
    private bool isGrounded = false;

    private Rigidbody2D rb;
    private Animator anim;
    private SpriteRenderer sprite;

    [Header("SFX")]
    [SerializeField] private AudioClip jumpSound;

    public static Hero Instance { get; set; }

    public States State
    {
        get { return (States)anim.GetInteger("state"); }
        set { anim.SetInteger("state", (int)value); }
    }

    private Vector3 initialScale;

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

```

```

// Тепер шукаємо Animator на дочірньому об'єкті
Square
Transform square = transform.Find("Square");
if (square != null)
{
    anim = square.GetComponent<Animator>();
    sprite = square.GetComponent<SpriteRenderer>();
}
else
{
    Debug.LogError("Square child not found! Please check
hierarchy.");
}

// Зберігаємо початковий розмір персонажа
initialScale = new Vector3(2.5f, 2.5f, 1f);
transform.localScale = initialScale;

var checkPointKey = "CheckPoint";
var hasLastPoint = PlayerPrefs.HasKey(checkPointKey);

if (!hasLastPoint)
    return;

var lastPoint = PlayerPrefs.GetFloat(checkPointKey);
var position = transform.position;

transform.position = new Vector3(lastPoint, position.y,
position.z);
}

private void FixedUpdate()
{
    CheckGround();
}

private void Update()
{
    if (isGrounded) State = States.idle;
    if (Input.GetButton("Horizontal"))
        Run();
    if (isGrounded && Input.GetButtonDown("Jump"))
        Jump();

    // Логуювання для перевірки розміру персонажа, лише
    якщо розмір змінився
    if (transform.localScale != initialScale)
    {
        Debug.Log($"Hero localScale:
{transform.localScale}");
    }
}

private void Run()
{
    if (isGrounded) State = States.run;

    Vector3 dir = transform.right *
Input.GetAxis("Horizontal");

    transform.position =
Vector3.MoveTowards(transform.position, transform.position
+ dir, speed * Time.deltaTime);

    // Використовуємо sprite.flipX для перевертання
персонажа
    if (dir.x < 0)
    {

```

```

        sprite.flipX = true;
    }
    else if (dir.x > 0)
    {
        sprite.flipX = false;
    }
}

private void Jump()
{
    rb.AddForce(transform.up * jumpForce,
ForceMode2D.Impulse);
    SoundManager.instance.PlaySound(jumpSound);
}

private void CheckGround()
{
    Collider2D[] collider =
Physics2D.OverlapCircleAll(transform.position, 0.8f);
    isGrounded = collider.Length > 1;

    if (!isGrounded) State = States.jump;
}

public SpriteRenderer GetSprite()
{
    return sprite;
}

public float GetDirection()
{
    return transform.localScale.x > 0 ? 1f : -1f;
}

}

public enum States
{
    idle,
    run,
    jump,
    attack,
    hit,
    death
}

using UnityEngine;

public class HealthCollectible : MonoBehaviour
{
    [SerializeField] private float healthValue;
    [SerializeField] private AudioClip pickupSound;

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.tag == "Player")
        {
            SoundManager.instance.PlaySound(pickupSound);

            collision.GetComponent<Health>().AddHealth(healthValue);
            gameObject.SetActive(false);
        }
    }
}

using UnityEngine;
using UnityEngine.UI;

public class HealthBar : MonoBehaviour
{
    [SerializeField] private Health playerHealth;
    [SerializeField] private Image totalhealthBar;

```

```

[SerializeField] private Image currentHealthBar;

private void Start()
{
    totalHealthBar.fillAmount = playerHealth.currentHealth /
7;
}

private void Update()
{
    currentHealthBar.fillAmount = playerHealth.currentHealth
/ 7;
}
}
using UnityEngine;
using UnityEngine.SceneManagement;

public class Health : MonoBehaviour
{
    [SerializeField] private float startingHealth = 100f;
    [SerializeField] private Animator anim;
    public float currentHealth { get; private set; }

    private bool dead;

    [Header("Components")]
    [SerializeField] private Behaviour[] components;
    private bool invulnerable;

    [Header("Death Sound")]
    [SerializeField] private AudioClip deathSound;
    [SerializeField] private AudioClip hurtSound;

    [Header("Game Over Manager")]
    [SerializeField] private GameManager
gameOverManager;

    private void Awake()
    {
        currentHealth = startingHealth;

        // Шукаємо Animator у дочірніх елементах
        anim = GetComponentInChildren<Animator>();

        if (anim == null)
            Debug.LogError("Animator не знайдено у дочірніх
елементах " + gameObject.name);
    }

    public void TakeDamage(float _damage)
    {
        currentHealth = Mathf.Clamp(currentHealth - _damage, 0,
startingHealth);

        if (currentHealth > 0)
        {
            if (anim != null)
            {
                anim.SetTrigger("hurt");
                Debug.Log("Hurt trigger sent to Animator");
            }

            if (SoundManager.instance != null && hurtSound !=
null)
                SoundManager.instance.PlaySound(hurtSound);
        }
        else
        {
            if (!dead)

```

```

{
    if (anim != null)
        anim.SetTrigger("die");

    // Деактивуємо всі прикріплені компоненти
    if (components != null)
    {
        foreach (Behaviour component in components)
            if (component != null)
                component.enabled = false;
    }

    dead = true;

    if (gameOverManager != null)
        gameOverManager.GameOver();

    if (SoundManager.instance != null && deathSound
!= null)
        SoundManager.instance.PlaySound(deathSound);
    }
}

public void AddHealth(float _value)
{
    currentHealth = Mathf.Clamp(currentHealth + _value, 0,
startingHealth);
}

private void Deactivate()
{
    gameObject.SetActive(false);
}
}
using UnityEngine;

public class MeleeEnemy : MonoBehaviour
{
    [Header("Attack Parameters")]
    [SerializeField] private float attackCooldown;
    [SerializeField] private float range;
    [SerializeField] private int damage;

    [Header("Collider Parameters")]
    [SerializeField] private float colliderDistance;
    [SerializeField] private BoxCollider2D boxCollider;

    [Header("Player Layer")]
    [SerializeField] private LayerMask playerLayer;
    private float cooldownTimer = Mathf.Infinity;

    [Header("Attack Sound")]
    [SerializeField] private AudioClip attackSound;

    //References
    private Animator anim;
    private Health playerHealth;
    private EnemyPatrol enemyPatrol;

    private void Awake()
    {
        anim = GetComponent<Animator>();
        enemyPatrol = GetComponentInParent<EnemyPatrol>();
    }

    private void Update()
    {
        cooldownTimer += Time.deltaTime;

```

```

//Attack only when player in sight?
if (PlayerInSight())
{
    if (cooldownTimer >= attackCooldown)
    {
        cooldownTimer = 0;
        anim.SetTrigger("attack");
        SoundManager.instance.PlaySound(attackSound);
    }
}

if (enemyPatrol != null)
    enemyPatrol.enabled = !PlayerInSight();
}

private bool PlayerInSight()
{
    RaycastHit2D hit =
        Physics2D.BoxCast(boxCollider.bounds.center +
transform.right * range * transform.localScale.x *
colliderDistance,
        new Vector3(boxCollider.bounds.size.x * range,
boxCollider.bounds.size.y, boxCollider.bounds.size.z),
0, Vector2.left, 0, playerLayer);

    if (hit.collider != null)
        playerHealth = hit.transform.GetComponent<Health>();

    return hit.collider != null;
}
private void OnDrawGizmos()
{
    Gizmos.color = Color.red;
    Gizmos.DrawWireCube(boxCollider.bounds.center +
transform.right * range * transform.localScale.x *
colliderDistance,
        new Vector3(boxCollider.bounds.size.x * range,
boxCollider.bounds.size.y, boxCollider.bounds.size.z));
}

private void DamagePlayer()
{
    if (PlayerInSight())
        playerHealth.TakeDamage(damage);
}
}
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;

public class MenuManager : MonoBehaviour
{
    public void ContinueGame()
    {
        SceneManager.LoadScene("The_Witch");
    }
    public void NewGame()
    {
        PlayerPrefs.DeleteAll();
        SceneManager.LoadScene(1);
    }

    public void ExitGame()
    {
        Application.Quit();
    }
}

```

```

using UnityEngine;

public class EnemyPatrol : MonoBehaviour
{
    [Header("Patrol Points")]
    [SerializeField] private Transform leftEdge;
    [SerializeField] private Transform rightEdge;

    [Header("Enemy")]
    [SerializeField] private Transform enemy;

    [Header("Movement parameters")]
    [SerializeField] private float speed;
    private Vector3 initScale;
    private bool movingLeft;

    [Header("Idle Behaviour")]
    [SerializeField] private float idleDuration;
    private float idleTimer;

    [Header("Enemy Animator")]
    [SerializeField] private Animator anim;

    private void Awake()
    {
        initScale = enemy.localScale;
    }
    private void OnDisable()
    {
        anim.SetBool("moving", false);
    }

    private void Update()
    {
        if (movingLeft)
        {
            if (enemy.position.x >= leftEdge.position.x)
                MoveInDirection(-1);
            else
                DirectionChange();
        }
        else
        {
            if (enemy.position.x <= rightEdge.position.x)
                MoveInDirection(1);
            else
                DirectionChange();
        }
    }

    private void DirectionChange()
    {
        anim.SetBool("moving", false);
        idleTimer += Time.deltaTime;

        if (idleTimer > idleDuration)
            movingLeft = !movingLeft;
    }

    private void MoveInDirection(int _direction)
    {
        idleTimer = 0;
        anim.SetBool("moving", true);

        //Make enemy face direction
        enemy.localScale = new Vector3(Mathf.Abs(initScale.x) *
_direction,
            initScale.y, initScale.z);
    }
}

```

```

        //Move in that direction
        enemy.position = new Vector3(enemy.position.x +
Time.deltaTime * _direction * speed,
        enemy.position.y, enemy.position.z);
    }
}
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Leader : MonoBehaviour
{
    [SerializeField]
    float speed = 3;

    void OnTriggerStay2D(Collider2D other)
    {
        other.GetComponent<Rigidbody2D>().gravityScale = 0;
        if (other.gameObject.CompareTag("Player"))
        {
            if (Input.GetKey(KeyCode.W) ||
Input.GetKey(KeyCode.UpArrow))
            {
                other.GetComponent<Rigidbody2D>().velocity =
new Vector2(0, speed);
            }
            else if (Input.GetKey(KeyCode.S) ||
Input.GetKey(KeyCode.DownArrow))
            {
                other.GetComponent<Rigidbody2D>().velocity =
new Vector2(0, -speed);
            }
            else
            {
                other.GetComponent<Rigidbody2D>().velocity =
new Vector2(0, 0);
            }
        }
    }

    void OnTriggerExit2D(Collider2D other)
    {
        other.GetComponent<Rigidbody2D>().gravityScale = 1;
    }
}
using UnityEngine;
using UnityEngine.SceneManagement;

public class GameManager : MonoBehaviour
{
    public GameObject gameOverUI;

    // Викликайте цей метод, коли персонаж помирає
    public void GameOver()
    {
        gameOverUI.SetActive(true);
        Time.timeScale = 0f; // Зупиняє час у гри
    }

    // Метод для перезапуску гри
    public void RestartGame()
    {
        Time.timeScale = 1f; // Повертає час у нормальний
режим

SceneManager.LoadScene(SceneManager.GetActiveScene().na
me); // Перезапускає поточну сцену
    }
}

```

```

// Метод для виходу з гри
public void MenuButton()
{
    SceneManager.LoadScene("Menu");
    Time.timeScale = 1f;
}
}
using UnityEngine;

public class CameraController : MonoBehaviour
{
    public Transform player;
    public float smoothSpeed = 0.125f;
    public Vector2 focusZone = new Vector2(3f, 3f);
    public float maxSpeed = 10f;

    private Vector3 offset;

    void Start()
    {
        offset = transform.position - player.position;
    }

    void Update()
    {
        float distanceX = Mathf.Abs(transform.position.x -
player.position.x);

        bool shouldScroll = distanceX > focusZone.x;

        if (shouldScroll)
        {
            int direction = (player.position.x > transform.position.x)
? 1 : -1;

            float scrollSpeed = Mathf.Min(distanceX / 32f,
maxSpeed) * direction;

            Vector3 targetPosition = new Vector3(player.position.x,
transform.position.y, transform.position.z);
            targetPosition.x += scrollSpeed *
Time.fixedDeltaTime;

            transform.position = Vector3.Lerp(transform.position,
targetPosition, smoothSpeed * Time.deltaTime);
        }
    }
}

```