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

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка гри в жанрі 2D-платформер»

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

_____ Сергій СКИБА
(підпис)

Виконав: здобувач вищої освіти групи ПД-44

Сергій СКИБА

Керівник: Олександр ЯРОШЕВСЬКИЙ
асистент кафедри

Рецензент: _____

Київ 2025

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
« ____ » _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Скибі Сергію Сергійовичу

1. Тема кваліфікаційної роботи: «Розробка гри в жанрі 2D-платформер "»
керівник кваліфікаційної роботи асистент кафедри ІПЗ Олександр ЯРОШЕВСЬКИЙ, затверджені наказом Державного університету інформаційно-комунікаційних технологій від «24» лютого 2025 р. № 56.

2. Строк подання кваліфікаційної роботи «02» червня 2025 р.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про жанр 2D-платформер, особливості ігрових механік, використання рушіїв для розробки ігор, технічна документація з описом бібліотек для розробки 2D ігор на мові програмування C#

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Огляд та аналіз предметної області.
2. Проєктування архітектури гри.
3. Програмна реалізація та опис функціонування гри в жанрі 2D-платформер.
4. Тестування гри.

5. Перелік графічного матеріалу: *презентація*

1. Аналіз аналогів.
2. Вимоги до програмного забезпечення.
3. Програмні засоби реалізації.
4. Діаграма варіантів використання.
5. Діаграма активності.
6. Діаграма класів.
7. Екранні форми.
8. Демонстрація роботи застосунку.
9. Апробація результатів дослідження.

6. Дата видачі завдання«25» лютого 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	25.02.-04.03.2025	
2	Аналіз та дослідження існуючих аналогів	05.03-13.03.2025	
3	Огляд та аналіз предметної області	14.03-21.03.2025	
4	Проєктування архітектури гри	22.03-01.04.2025	
5	Програмна реалізація застосунку	02.04-25.04.2025	
6	Тестування гри	26.04-28.04.2025	
7	Оформлення роботи: вступ, висновки, реферат	29.04-11.05.2025	
8	Розробка демонстраційних матеріалів	12.05-18.05.2025	
9	Попередній захист роботи	19.05-30.05.2025	

Здобувач вищої освіти _____ Сергій СКИБА
(підпис)

Керівник
кваліфікаційної роботи _____ Олександр ЯРОШЕВСЬКИЙ
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 51 стор., 2 табл., 28 рис., 20 джерел.

Мета роботи – покращення реалізації ігрового процесу шляхом впровадження механік стрибків, атак і взаємодії з об'єктами, реалізації фізично коректного руху персонажа, налаштування адаптивної камери, а також забезпечення навігації за допомогою системи візуальних підказок.

Об'єкт дослідження – процес створення комп'ютерних ігор жанру платформер для розважальних цілей користувачів.

Предмет дослідження – програмне забезпечення, що дозволяє керувати персонажем, проходити рівні з різноманітними перешкодами та ворогами у грі жанру 2D-платформер.

Короткий зміст роботи: у роботі проаналізовано технології та методи для розробки ігор жанру 2D-платформер. Проаналізовані аналоги ігор у цьому жанрі: Super Mario Bros, Spelunky Classic, VVVVVV. Розроблено гру та програмно реалізовані ключові функціональні можливості, зокрема: управління рухом персонажу, штучний інтелект ворогів, анімації персонажів та навколишнього середовища, бойові механіки, обробка зіткнень, вкладка налаштування. Проведено функціональне та модульне тестування додатку. В роботі використано бібліотеки Unity для реалізації фізики та анімацій, а також інструменти для створення користувацького інтерфейсу та візуалізації елементів.

Сферою використання застосунку є розвага для користувачів, спрямована на розвиток реакції, просторового мислення.

КЛЮЧОВІ СЛОВА: 2D-ПЛАТФОРМЕР, UNITY, ГЕЙМДИЗАЙН, СКРИПТИ, ГРА, C#, УПРАВЛІННЯ ПЕРСОНАЖЕМ.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ПК – персональний комп’ютер

NES – Nintendo Entertainment System

IDE – Integrated Development Environment

ЦП – центральний процесор

API – Application Programming Interface

UML – Unified Modeling Language

UI – User Interface

FPS – Frame Per Second

ВСТУП

Актуальність: ігри жанру 2D-платформер залишаються актуальними в індустрії розробки ігор завдяки поєднанню простоти реалізації та великого потенціалу для творчості. Вони не вимагають високої продуктивності від апаратного забезпечення, що робить їх доступними для значної частини користувачів, включно з тими, хто використовує застарілі або бюджетні пристрої. Це дозволяє охопити широку цільову аудиторію та забезпечити стабільну взаємодію з грою на різних платформах — від ПК до мобільних телефонів.

Завдяки зрозумілому геймплею та простому управлінню, такі ігри легко засвоюються новачками, що робить їх популярними серед казуальних гравців. Одночасно, можливість створення складних ігрових рівнів, додавання ворогів, головоломок, системи прогресу чи бойових механік надає широкі можливості для залучення й більш досвідченої аудиторії. Це дозволяє адаптувати гру під різні стилі проходження: від легких пригод до складних випробувань.

Таким чином, актуальність теми полягає у поєднанні технічної доцільності, користувацької доступності та освітньої цінності, що робить розробку 2D-платформера перспективним напрямом як для дослідження, так і для практичної реалізації.

Об'єктом дослідження є процес створення комп'ютерних ігор жанру 2D-платформер для розважальних цілей користувачів.

Предметом дослідження програмне забезпечення, що дозволяє керувати персонажем, проходити рівні з різноманітними перешкодами та ворогами у грі жанру 2D-платформер.

Метою роботи покращення реалізації ігрового процесу шляхом впровадження механік стрибків, атак і взаємодії з об'єктами, реалізації фізично коректного руху персонажа, налаштування адаптивної камери, а також забезпечення навігації за допомогою системи візуальних підказок.

Методи дослідження: першим кроком проаналізовано предметну область: сутність жанру 2D-платформер, історія розвитку, цільова аудиторія, аналоги, функціональні та нефункціональні вимоги. Це дозволило визначити концепцію гри. Далі було проведено аналіз інструментів розробки таких як Unity , Godot та Unreal Engine. Це дало можливість вибрати найбільш придатне середовище для розробки проєкту. Вибір впав на ігровий рушій Unity та мову програмування C#. Також, для контролю версій обрано GitHub, а для анімації та створення об'єктів обрано редактор зображень Aseprite.

Для спрощення планування реалізації функціоналу та проєктування структури гри було використано UML-діаграми. Діаграма варіантів використання допомогла ідентифікувати ключові сценарії взаємодії користувача з системою. Діаграма класів дозволила формалізувати об'єктно-орієнтовану структуру гри, визначивши взаємозв'язки між класами, їх атрибути та методи. Діаграма активності візуалізувала логіку виконання дій у межах ігрового процесу, що сприяло виявленню можливих гілок поведінки та прийняттю рішень у межах сценаріїв.

Таким чином, використання UML-діаграм стало важливим етапом у технічному плануванні, що підвищило ефективність подальшої реалізації гри. Після проєктування структури гри та реалізації основного функціоналу було проведено тестування з метою перевірки працездатності всіх компонентів та відповідності гри заданим вимогам.

Наукова новизна роботи визначається наступними функціональними складовими: впровадження унікальної бойової системи, відображення індикатору стану здоров'я у ворогів, що покращує ігровий процес.

Практична значущість результатів полягає в можливості використання реалізованої гри для розваг користувачів.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Визначення жанру 2D-платформер

Якщо у людини спитати , що таке відеогра? То в більшості скажуть про гру, де є платформи. Платформер – це гра, де основна дія обертається навколо бігу та стрибків. Мета гравця у такій грі полягає у проходженні рівнів, долаючи різні перешкоди та противників. Сам жанр за весь час існування відеоігор залишається популярним серед гравців.

Ключовими характеристиками 2D-платформерів є:

– side-scrolling (бокова проєкція);
– стрибки – це основна механіка пересування між платформами;
– наявність різноманітних ворогів, пасток, бонусів;
– рівні в яких є початок і кінець;
– система здоров’я;
– візуальний стиль та анімації;
– реалізація фізики у грі (інерція , зіткнення тощо).

Незважаючи на просту візуальну складову, 2D-платформери часто мають складний ігровий процес, що вимагає точної координації дій, стратегічного мислення та швидкої реакції на дії у грі.

1.2 Історія розвитку жанру платформер

Історія розвитку 2D-платформерів бере початок з далеких 1980-тих років у епоху одного екрану (фіксована камера на весь рівень у грі). Тоді з’явились відеоігри, що включали переміщення персонажа між платформами. Першим таким платформером вважається Space Panic. Гра була розроблена компанією Universal.

Характеристика гри Space Panic:

– відсутність стрибків;
– можливість підніматись по драбинам;
– камера у грі була статична(направлена на весь рівень).

Але наступна гра стала поштовхом до революції у цьому жанрі. Ця гра має назву Donkey Kong, розроблена компанією Nintendo. У цій грі гравець міг керувати персонажем якого звали Jumpman(далі він стане відомий як Mario), пригати через перешкоди та лазити по драбинам. Ця гра призвела до появи франшизи з Donkey Kong.

Нова гра Mario Bros від тієї ж компанії представляє фундаментальні принципи жанру:

– рівнева структура
– плавне переміщення екрану
– стрибки
– збирання бонусів
– боротьба з ворогами

Саме ця гра дала таку популярність жанру 2D-платформери. І саме з цієї гри появилась ера side-scrolling , де камера рухається разом з персонажем. Але це була обмежена функціональність. Оскільки, дозволяло лише робити горизонтальну прокрутку камери.

Наступними героями у історії розвитку платформерів стає компанія Sega з їх грою Sonic the Hedgehod. Поки конкуренти випускали повільніші за темпом ігри, гра про Sonic(ігровий персонаж гри Sonic the Hedgehod) відзначилась високою швидкістю геймплею.

Ключові інновації у грі:

– швидкість у грі стала основою геймплею, що дозволило зробити рівні динамічнішими і більшими;
– фізика у грі дала можливість реалізувати закони інерції – поступове прискорення, можливість котитися, розганятися головному персонажу Соніку;
– унікальний дизайн рівнів(пружини, трампліни, нелінійність рівнів).

Ця гра була представником нової ери side-scrolling. Звісно ж ця гра набрала наслідувачів.

Із появою ери 3D-платформерів у 1990-тих роках зацікавленість до 2D знизилась. Тому що вперше було реалізовано повністю тривимірне середовище , де гравець міг вільно переміщатися у просторі в усіх напрямках. Но зацікавленість до 2D-платформерів вернулась у 21 столітті. Жанр пережив новий розквіт завдяки незалежним розробникам. Приклади таких ігор: Celeste, Ori and The Blind Forest, Hollow Knight. Ці ігри доказали , що 2D-платформери можуть бути не лише технологічно розвиненими, а й художньо цінними.

Хоча платформери виросли в геометричній прогресії за десятиліття, вони, як і раніше, є одними з найдоступніших розваг для багатьох геймерів.

1.3 Цільова аудиторія

Цільову аудиторію 2D-платформерів приваблює простота геймплею, цікаві дизайни рівнів та невисокі системні характеристики до пристроїв. Гра такого жанру орієнтована на таких гравців:

- початківців у сфері відеоігор;
- користувачі віком 12-30 років;
- фанати класичних ігор;
- гравці які цінують свій час.

Такий аналіз допоможе охопити потенційну аудиторію гри при розміщенні продукту на різних платформах (ПК, мобільних пристроях).

1.4 Дослідження існуючих аналогів

Ігри жанру 2D-платформер, які були використані як приклади для розробки власного продукту:

- Super Mario Bros;
- Spelunky Classic;
- VVVVVV.

1.4.1 Super Mario Bros

Super Mario Bros є класичним 2D-платформером, розробленим компанією Nintendo для консолі NES. Суть цієї гри у тому, що гравець керує персонажом Маріо, який має врятувати принцесу, проходячи рівні, убиваючи ворогів і збираючи бонуси. У більшій частині всіх рівнів розкидані монети, які Маріо може збирати, та особливі скриньки зі знаком питання на них. Вдаряючи їх кулаком з нижніх позицій, Маріо здобуває або монети, або певну корисну річ. Також, у цій грі, щоб перемогти ворога, потрібно на нього пригнути. У іншому випадку, якщо він тебе торкнеться то ти програєш.

Розберемо основні механіки гри Super Mario Bros. Персонаж може рухатись вправо та вліво, стрибати для подолання ворогів та перешкод, збирання різних бонусів, взаємодія з об’єктами оточення.

Переваги цієї гри:

– рівні з поступовим зростанням складності;
– просте керування персонажем;
– наявність різних видів ворогів.

Недоліки:

– відсутність підказок для гравця;
– камера слідкує за горизонтальним рухом та фіксована;
– відсутність індикаторів здоров’я ворогів;
– фіксована траєкторія стрибків.

Гра Super Mario Bros є фундаментальним прикладом 2D-платформера з класичними механіками, такими як рух, стрибки, збирання бонусів та боротьба з ворогами. Її переваги полягають у зрозумілому управлінні, поступовому ускладненні рівнів та різноманітності противників, що сприяє геймплейному інтересу. Водночас, гра має низку обмежень, зокрема фіксовану камеру, відсутність системи підказок та індикаторів здоров’я ворогів, що знижує зручність ігрового процесу для сучасного користувача. Незважаючи на ці недоліки, Super Mario Bros залишається еталоном жанру та основою для аналізу і вдосконалення сучасних 2D-платформерів.

Приклад рівня у грі Super Mario Bros можна побачити на рис. 1.1

Рис. 1.1 Super Mario Bros (рівень у грі)

1.4.2 Spelunky Classic

Spelunky Classic – це 2D платформер з елементами roguelike (піджанр рольових відеоігор), де гравець досліджує підземелля, уникаючи пасток і ворогів. Рівні генеруються випадково, що створює унікальні ігрові ситуації в кожному проходженні. Ціль гри у тому щоб захватити як можна більше скарбів.

У цій грі основні механіки:

– рух персонажа вліво, вправо , вгору та вниз;
– вміння персонажа стрибати;
– використання інструментів для взаємодії з рівнем;
– смерть персонажа відбувається коли отримає достатньо шкоди

Перевагами цієї гри буде:

– випадкова генерація рівнів;
– велика кількість ворогів та пасток

Недоліки гри:

– стрибки різкі, некеровані в польоті;
– відсутність індикаторів здоров'я для ворогів;
– графіка дуже проста;
– вороги мають просту поведінку (рух за фіксованими маршрутами)

Гра вважається одним із найвпливовіших інді-проєктів, що показав, як класичний жанр платформера може бути поєднаний з сучасними геймдизайнерськими рішеннями. У подальшому були випущені ігри Spelunky HD та Spelunky 2.

Приклад рівня у грі Spelunky Classic можна побачити на рис. 1.2

Рис. 1.2 Spelunky Classic (рівень у грі)

1.4.3 VVVVVV

VVVVVV — інді-платформер, що акцентує увагу на зміні гравітації замість традиційних стрибків. Гравець керує капітаном Вердіком, намагаючись врятувати свою команду у мінімалістичному світі. Основними механіками цієї гри є рух персонажа вліво та вправо, зміна гравітації, подолання перешкод через зміну гравітації.

Перевагами цієї гри стане:

– керування гравітацією замість стрибків;

– лаконічний дизайн рівнів.

Недоліки гри:

– відсутність бойових механік;

– камера фіксована;

– примітивна графіка;

– відсутність підказок.

Гра здобула популярність завдяки оригінальній ігровій механіці та ретро-стилю.

Приклад рівня у грі VVVVVV можна побачити на рисунку 1.3

Рис. 1.3 VVVVVV (рівень у грі)

1.4.4 Таблиця аналізу ігор у жанрі 2D-платформер

Зведені результати аналізу існуючих ігор жанру 2D-платформери та їх порівняння у таблиці 1.1.

Таблиця 1.1

Зведена таблиця аналізу існуючих ігор жанру 2D-платформери

Показник	**Super Mario Bros**	**Spelunky Classic**	**VVVVVV**	**Night Hero**
Рух персонажа (вліво, вправо, стрибає, атакує)	+	+	-	+
Наявність ворогів	+	+	-	+
Реакція ворогів на гравця	-	-	-	+
Індикатор здоров'я ворогів	-	-	-	+
Стиль графіки	8-бітна піксельна	піксель-арт	стиль ретро	піксель-арт
Підказки для гравця	-	-	-	+
Можливість ставити гру на паузу	+	+	+	+
Камера слідкує за гравцем	лише по горизонталі у право	ривкове переключення екранів	статичні екрани	по горизонталі та по вертикалі
Пастки	+	+	+	+
Рух ворогів по маршруту	+	+	-	+
Регенерація здоров’я	-	+	-	+
Наявність екрана “Game Over”	+	+	-	+

1.5 Визначення функціональних та нефункціональних вимог

Після проведених аналізів можна сформулювати функціональні та нефункціональні вимоги до своєї гри жанру 2D-платформер. Це забезпечить розробку конкурентного продукту.

1.5.1 Функціональні вимоги

Функціональні вимоги це детальні специфікації, які визначають дії, поведінку та функціональні можливості, які повинна виконувати система, щоб відповідати призначеній меті [3].

Функціональні вимоги до програмного продукту:

– Взаємодія з грою відбувається за рахунок клавіатури та комп’ютерної мишки;

– головний персонаж рухається вліво, вправо, стрибає та атакує;

– ворожі персонажі рухаються за заданим маршрутом або реагують на наближення гравця;

– після втрати всіх очок здоров’я гравця на екрані має з’явитися повідомлення “Game Over”;

– реалізовані пастки (шипи), які убивають гравця;

– гравець має можливість перезапустити гру після смерті або повернутися до головного меню;

– над головами ворожих персонажів повинен показуватися індикатор здоров’я (health bar);

– реалізований об'єкт серце для регенерації здоров'я головного персонажа;

– гра містить підказки для гравця;

– можливість ставити гру на паузу;

– реалізація системи стеження за персонажем (камера сцени рівня переслідує гравця);

– анімації дій персонажів.

1.5.2 Нефункціональні вимоги

Нефункціональні вимоги визначають якість роботи програмного забезпечення, технічні обмеження, надійність, продуктивність та зручність використання [3].

Нефункціональні вимоги програмного продукту:

- всі аудіосигнали мають відтворюватися без затримок, з гучністю, що не перевищує комфортний рівень (до 70 дБ);
- повне завантаження гри має тривати не більше 20 секунд;
- завантаження рівнів не повинно перевищувати 5 секунд;
- кількість кадрів під час гри не менше 30.

Наведені нефункціональні вимоги є важливими критеріями якості програмного продукту, що безпосередньо впливають на зручність його використання, технічну стабільність та загальне враження від ігрового процесу.

По-перше, вимога до відтворення аудіосигналів без затримок та з обмеженням гучності (до 70 дБ) забезпечує комфортне звукове супроводження гри. Вчасне й коректне відтворення звукових ефектів (наприклад, стрибків, атак, збирання бонусів) підсилює ігрову атмосферу, а контрольована гучність запобігає перевантаженню слуху та робить гру доступною для ширшої аудиторії, зокрема для дітей.

По-друге, встановлення обмеження на повне завантаження гри до 20 секунд сприяє позитивному першому враженню користувача. Дослідження показують, що сучасні гравці очікують швидкого доступу до контенту, і довге очікування може призвести до зниження інтересу або навіть відмови від гри.

Третє, обмеження часу завантаження рівнів до 5 секунд гарантує безперервність ігрового процесу, що особливо важливо для 2D платформерів із частими переходами між локаціями. Це дозволяє уникнути розривів у геймплеї, зберігає динаміку та утримує увагу гравця.

Нарешті, вимога забезпечити не менше 30 кадрів на секунду (FPS) є мінімальним стандартом продуктивності для ігор цього жанру. Такий рівень продуктивності дозволяє забезпечити плавність анімацій, точне реагування на дії гравця та стабільне візуальне сприйняття. У сукупності це гарантує комфортну взаємодію з грою навіть на пристроях із середніми технічними характеристиками.

2 ЗАСОБИ РЕАЛІЗАЦІЇ

2.1 Інтегроване середовище розробки

Не менш важливу роль у розробці програмного забезпечення грає вибір інтегрованого середовища розробки(IDE), яке надає розробникам усі необхідні інструменти для написання, тестування та налагодження коду в єдиній системі [4]. IDE об’єднує в собі редактор коду, компілятор, засоби візуалізації, налагоджувач та інші необхідні інструменти для розробки. Мій вибір впав на інтегроване середовище розробки Visual Studio.

Visual Studio — це потужне інтегроване середовище розробки, створене компанією Microsoft, яке широко використовується для створення програмного забезпечення різного типу: ігор, вебзастосунків, десктопних і мобільних додатків. Його головною перевагою є підтримка широкого спектра мов програмування, таких як C#, C++, JavaScript, TypeScript, Python, а також HTML і CSS для верстки сайтів. Visual Studio має зручний інтерфейс, інтелектуальне автозаповнення коду (IntelliSense), розширену підтримку .NET, Unity та інтеграцію з платформами хмарних обчислень, такими як Azure. Середовище дозволяє ефективно працювати з проєктами під Windows, Android, iOS, macOS та Linux. Visual Studio також підтримує командну роботу через інтеграцію з Git, GitHub та Azure DevOps, забезпечує можливості для налагодження, тестування, створення інтерфейсів за допомогою візуальних редакторів і має доступ до великої кількості розширень через власний Marketplace. Завдяки цим функціям Visual Studio вважається одним із найпотужніших і найзручніших інструментів для професійної розробки програмного забезпечення.

На рисунку 2.1 наведено приклад інтерфейсу Visual Studio.

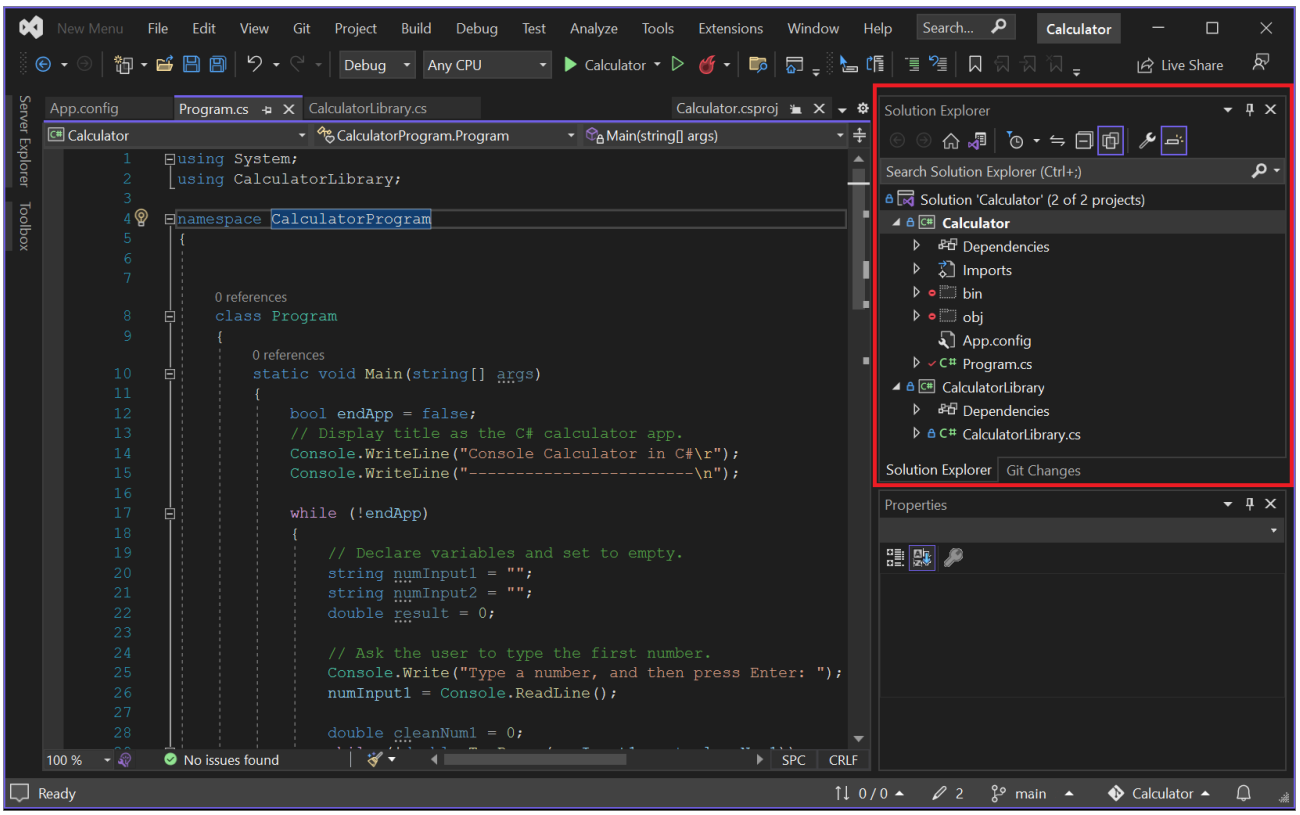


Рис. 2.1 Visual Studio (інтерфейс IDE)

2.2 Визначення мови програмування

Мова програмування – це формалізована мова, яка дозволяє програмісту формулювати алгоритми, структури даних та інструкції для виконання комп'ютером. Якщо простіше то це спосіб спілкування між людиною та машиною. Завдяки цьому створюються програми. Також мови програмування мають правила синтаксису як і мови якою спілкуються люди. Без знань як правильно написати код створити програму не вийде. Існують мови високого рівня (ближчі до людської мови) та низького рівня (ближчі до машинного коду).

У розробці ігор мови програмування використовуються для створення логіки персонажів, керування, взаємодії з користувачем, фізики, для роботи з графікою, звуком тощо.

2.3 Мова програмування C#

C# - це мова програмування, яка є однією з популярних мов для розробки програмного забезпечення. І вона не просто так займає визначне місце у світі розробки. Вона зацікавлює розробників своєю простотою, універсальністю, сучасним синтаксисом [6].

C# підтримує об’єктно-орієнтоване програмування, велику кількість бібліотек, автоматичне керування пам’яттю, що дозволяє створювати складні програми. Особливої популярності C# набув серед розробників ігор завдяки глибокій інтеграції з рушієм Unity. Що робить мову програмування ідеальним для розробки ігор жанру 2D-платформер.

Приклади використання C# у іграх цього жанру:

– керування персонажем (рух, атака);
– створення UI інтерфейсу;
– реалізація бойових механік, зіткнень з об’єктами;
– реалізація подій (перемога, смерть, перехід між рівнями тощо).

Отже, C# є ідеальною мовою програмування для створення ігор завдяки потужним інструментам та інтеграції з рушієм Unity завдяки Unity Scripting API, що у свою чергу надає доступ до всього функціоналу рушія.

2.4 Огляд на рушій Unity

Unity – це популярний багатоплатформовий рушій для розробки ігор, який широко використовується для створення 2D та 3D ігор. Надає можливість для створення програмних продуктів для різних платформ (ПК, ігрові консолі тощо). Unity відомий завдяки своїй доступності, гнучкості, великим набором інструментів та величезній спільноті розробників [7].

2.4.1 Особливості рушія Unity для розробки 2D ігор

Ігровий рушій має ряд особливостей для розробки 2D ігор. Чим і заохочує розробників різних рівнів програмування. Розберемо основні особливості Unity.

Unity має окремий 2D-режим, що забезпечує зручне використання сцени та розміщення спрайтів (двовимірне зображення, що використовується в комп'ютерній графіці). Рушій підтримує роботу зі спрайтами, які використовуються для анімацій (через Animation Editor) або розбивати їх на окремі кадри. Компонент Sorting Layers надає можливість визначити порядок накладання об'єктів на сцені , а Sprite Render відповідає за їх відображення на сцені.

Немало важливою особливістю є робота з фізикою. Для цього у Unity існують компоненти Rigidbody2D та Collider2D. Rigidbody2D – додає фізику об'єкту таку як масу, швидкість, гравітацію. А Collider2D виявляє зіткнення об'єктів.

Також існує система побудови рівнів (Tilemap). Ця система дозволяє створювати рівні за допомогою тайлів, які є невеличкими спрайтами. Побудова рівнів відбувається за допомогою плиток. Що дає можливість заощадити час на створенні рівнів, а також зекономити використання пам'яті та ЦП.

Приклад використання Tilemap наведено на рисунку 2.2

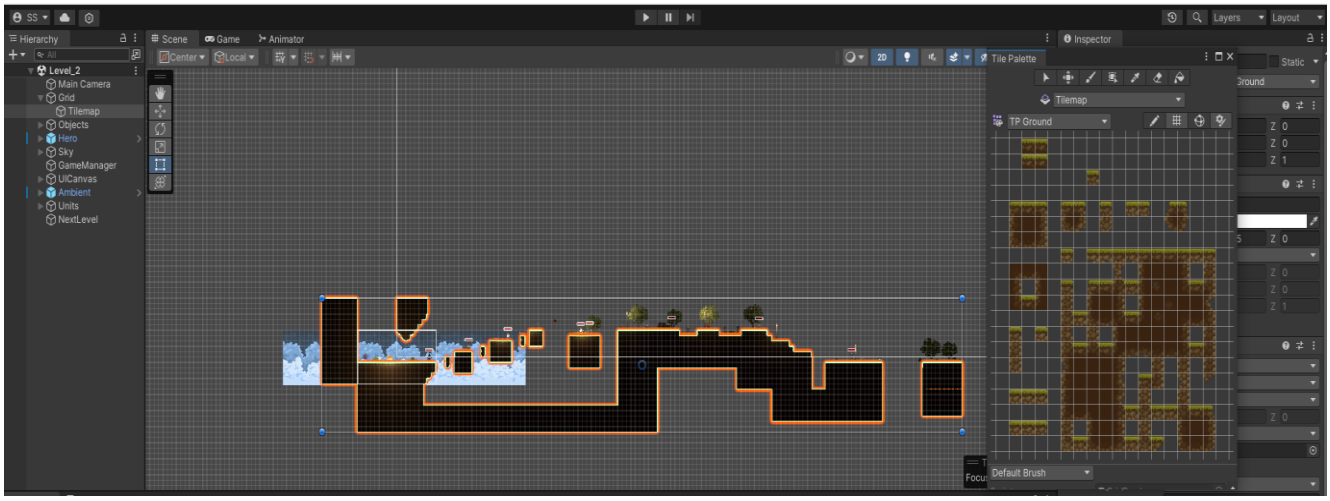


Рис. 2.2 Tilemap(використання системи в Unity)

У Unity є вбудована система UI це означає, що можна легко додавати такі речі як індикатор здоров'я, кнопки меню, паузи, налаштувань. UI елементи

розміщуються в об’єкт Canvas, який масштабується разом з екраном, що дає адаптивність на різних пристроях.

Ще одна прекрасна особливість Unity у тому, що для того щоб протестувати гри потрібно лише натиснути на кнопку Play. А коли програмний продукт готовий то Unity дозволяє експортувати гру на різні платформи: Windows, Linux. Android тощо.

Отже, Unity надає необхідні інструменти для розробки ігор жанру 2D-платформер.

2.4.2 Переваги та недоліки ігрового рушія Unity

Як і будь-яка технологія, рушій Unity має свої переваги та недоліки. Спочатку розберемо його переваги. Unity має зручний інтерфейс та легкодоступну документацію, що прекрасно підходить для новачків. В додаток до сказаного, рушій надає безкоштовну версію для некомерційного використання, що робить його ідеальним для студентів, інді-розробників та викладачів.

Як уже говорилось раніше, Unity це в першу чергу про кросплатформеність, що значить про випуск проєктів на різні платформи (Windows, macOS, Linux тощо). Це дасть можливість зацікавити більше користувачів та розкрити весь потенціал гри.

Unity Asset Store має великий набір готових моделей, спрайтів, анімацій, звукових ефектів, рівнів та навіть готових фрагментів коду, які можна легко інтегрувати у свій проєкт. Також у цей магазин можна й самому публікувати свої роботи, щоб інші користувачі економили свій час на оформлення візуального аспекту гри або написані коду.

Ще одною перевагою рушія є можливість створення як 2D, так і 3D ігор. Що немало важливо, Unity дозволяє поєднувати 2D та 3D елементи в одному проєкті. Ця універсальність дає велику можливість для фантазій розробників (наприклад створювати гібридні ігри).

Також у Unity зрозумілий інтерфейс редактора сцен, що дозволяє зручно розміщувати об’єкти, налаштовувати компоненти, додавати анімації та взаємодію без необхідності писати код. Інтерфейс рушія буде зрозумілий навіть початківцям.

Команда Unity постійно займається покращенням свого рушія, додаючи новий функціонал, оптимізують продуктивність.

Тепер розберемо недоліки ігрового рушія. Адже, ні одна технологія не може бути без своїх недоліків. Проєкти Unity страдають від великих розмірів збірки. Навіть найпростіший проєкт може займати значну кількість пам’яті, що не завжди підходить, якщо розробляти для мобільних пристроїв застосунок.

Також Unity має проблеми з переходом програмного продукту на іншу версію, що у свою чергу викликає зміну API, несумісність плагінів та помилки при відкритті старих проєктів.

Високі вимоги до оптимізації. Unity має проблеми з використанням ресурсів за замовчуванням. Це дуже помітно у проєктах, які орієнтовані на мобільні пристрої, де обмежується графічна продуктивність або оперативна пам'ять.

2.4.3 Використання мови програмування C# в Unity

Мова програмування C# використовується в Unity як основна. Саме вона використовується для створення всього функціоналу гри. Тобто від базових взаємодії з об’єктами до побудови складних ігрових механік. Завдяки підтримці об’єктно-орієнтованого програмуванню та широким можливостям інтеграції, мова програмування C# дозволяє створювати як прості, так і масштабні проєкти.

Unity повністю інтегрує C# у свою архітектуру, надаючи розробникам доступ до численних класів, методів та властивостей для взаємодії з ігровим середовищем. Створення керування персонажем, поведінки об’єктів, обробки подій, створення бойових механік, взаємодії з анімаціями, фізикою та звуками – усе це можна реалізувати завдяки засобам C# у межах рушія.

У процесі розробки створюються скрипти, які підключаються до об’єктів у редакторі Unity. Наприклад, скрипт атаки персонажа прикріплюється до ігрового

героя і надає можливість йому проводити атаки. Такі скрипти надають можливість у редакторі змінювати необхідні компоненти.

Крім того, Unity забезпечує інтеграцію з середовищем розробки Visual Studio, яке автоматично відкриває скрипти, підсвічує помилки, надає підказки тощо. Використання мови програмування C# у Unity сприяє кросплатформеній розробці. Один і той самий код може використовуватися для гри під різні платформи. Що дає велику економію часу, і не потрібно перезаписувати проєкт.

Отже, C# є ключовим інструментом управління всією логікою гри: від механіки до користувацького інтерфейсу, від анімацій до фізичних взаємодій. Це означає реалізацію повноцінного продукту, який можна легко підтримувати в майбутньому.

2.4.4 Обмеження мови програмування C# в Unity

Враховуючи переваги використання мови програмування C# в Unity все ж рушій має обмеження, які варто враховувати при розробці гри жанру 2D-платформер. Обмеження зумовлені як самою мовою, так і технічною реалізацією її інтеграції з рушієм Unity.

При створенні довготривалого проєкту, виникає проблема залежності від версії Unity, яка визначає доступні можливості мови C#. Наприклад, нові можливості доступні лише у нових версіях рушій, що обмежує розробників, які працюють зі старішими, але стабільними версіями Unity.

Під час гри, де кожен кадр генерує нові об’єкти, автоматичне керування пам’яттю в C# може викликати падіння продуктивності гри. Це особливо помітно при переході на інший рівень або при масових боях.

Щоб досягти плавної гри та високої продуктивності, необхідно враховувати ці проблеми ще на ранніх стадіях розробки гри.

2.5 Система контролю версій

GitHub – це хмарна платформа, де можна зберігати, ділитися та працювати разом з іншими над написанням коду [17]. Він підтримує як приватні, так і публічні репозиторії. Репозиторій – це сховище, у якому зберігається весь код проєкту. При розробці програмного забезпечення можна контролювати версії проєкту завдяки Git – система контролю версій. У GitHub усі зміни зберігаються у вигляді комітів, що надає можливість легко відстежувати правки. Також можна створювати окремі гілки, не змінюючи основну версію гри. Якщо нова версія продукту стала нестабільною то можна вернутись на версію, де все працювало. Ще GitHub надає можливість спільно працювати над проєктом.

Unity підтримує роботу з системою контролю версій Git, що дає можливість не вносити непотрібні службові файли до репозиторію. Використання GitHub під час розробки гри дозволяє уникнути проблем з нестабільними версіями програмного продукту.

2.6 Редактор зображень Aseprite

Aseprite – це редактор зображень, що дозволяє малювати та робити 2D-анімації спрайтів у стилі піксель-арт. Програма розроблена для потреб геймдеву та забезпечує всі необхідні функції для створення візуального стилю гри у 2D середовищі.

Рис. 2.3 Aseprite (приклад інтерфейсу)

3 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ГРИ

3.1 Концепція гри

Розроблена гра являється 2D-платформером з елементами бойової системи, перешкод та пасток, що поєднує в собі динамічний геймплей, піксельну графіку та атмосферу напруженого проходження рівнів. Гравець керує головним героєм, який повинен дістатися до кінця рівня долаючи ворогів та уникаючи смерті. Але кожен раз коли гравець буде добиратись до кінця рівня, де є таємничий туман, його буде переносити гра на новий рівень з новими викликами для гравця. Гравцю потрібно буде продумувати свої дії аби не програти.

Таблиця 3.1

Концептуальна таблиця гри

Назва гри	Night Hero
Жанр	Platformer
Простір	2D
Мова програмування	C#
Візуальний стиль	Піксель-арт
Платформи	Windows
Сюжет	Нічний герой прокидається без пам’яті біля костру під підніжжям обриву. Але він помічає, що за ним гналися через наявність стріл на обриві. Помічає попереду нього ворогів, які бажають його вбити. Він розуміє, що його завдання йти далі вперед.

Таким чином, концепція гри Night Hero передбачає динамічний геймплей, красивий дизайн рівнів, атмосферні звукові ефекти та скритий сюжетний посил.

3.2 Діаграма use case

UML - уніфікована мова моделювання, що використовується розробниками програмного забезпечення для візуалізації процесів та роботи систем [19]. Це набір правил для створення діаграм. UML не являється мовою програмування. За допомогою створення діаграм можна швидко зрозуміти структуру програмного продукту або ідею.

Use case diagram (діаграма варіантів використання) – це тип діаграми UML, який моделює взаємодію користувача з функціональними можливостями цієї системи. Проте така діаграма не показує порядок виконання кроків.

Use case diagram складається з таких елементів:

Actor: особа або система, що взаємодіє із застосунком. Позначається фігурою людини з підписом.

Use case: конкретний сценарій, який може виконати актор. Позначаються у вигляді еліпсу.

Association: лінії між акторами та use case, що показує дії які може виконувати актор.

System: межі системи, всередині яких реалізовані функції. Позначаються прямокутником, що охоплює всі варіанти використання. Того й діаграма має таку назву варіанти використання.

Кожен елемент діаграми use case має своє функціональне та логічне призначення. Задача таких діаграм показати, що може робити користувач у системі. Це дозволяє створити повну модель взаємодії користувача з програмним продуктом.

Для показу взаємодії гравця з системою створено діаграму варіантів використання.

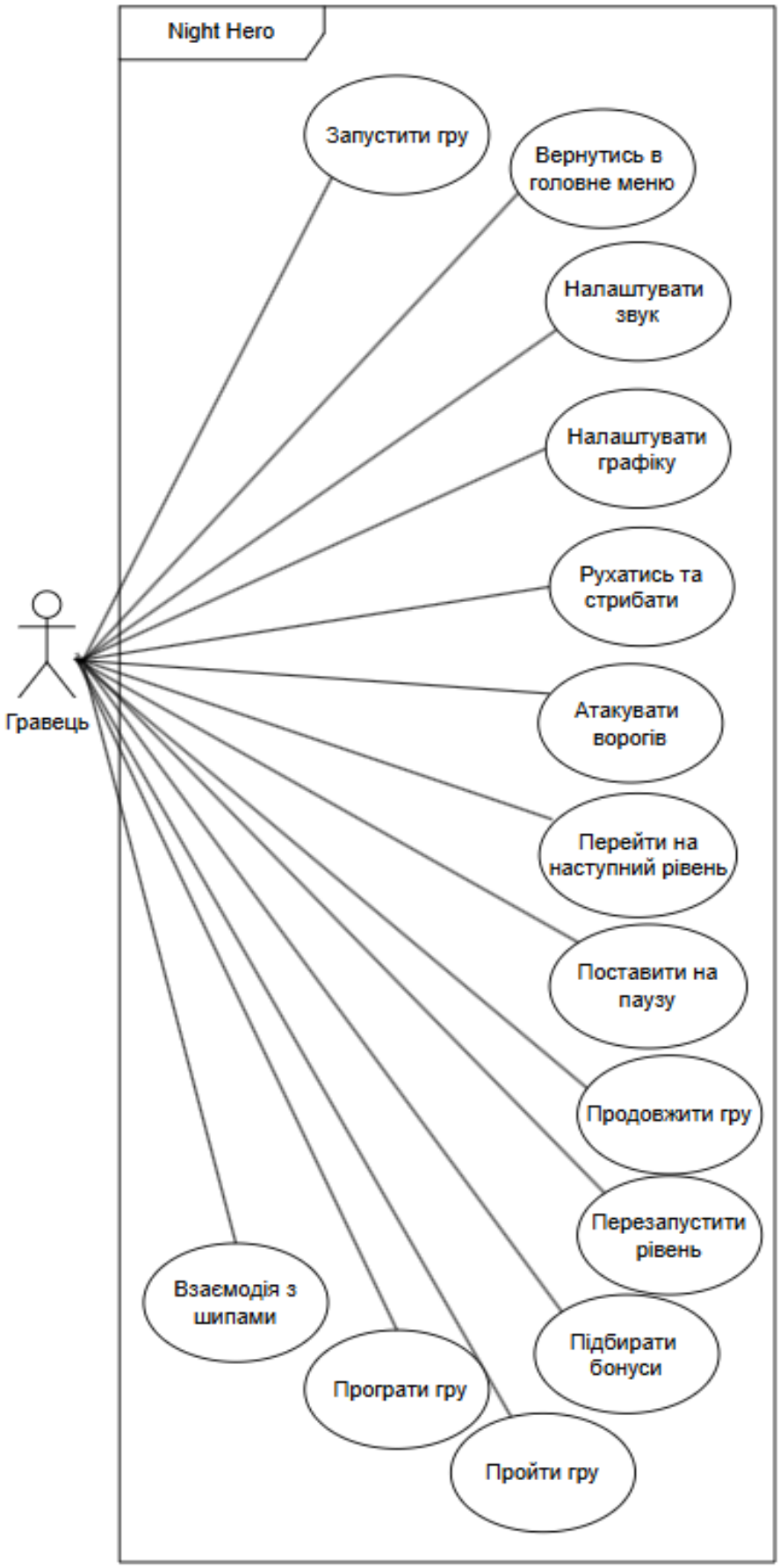


Рис. 3.1 Діаграма варіантів використання

3.3 Діаграма активності

Діаграма активності – це один із типів діаграм UML, який використовується для моделювання послідовності дій у системі. Якщо розглядати діаграму у контексті розробки гри, то вона демонструє, як гравець буде поводити себе на певних етапах у грі.

Основні елементи діаграми активності наведені у таблиці 3.2

Таблиця 3.2

Таблиця основних елементів діаграми активності

Елемент	**Позначення**	**Пояснення**
Start (початок)	Коло	Початкова точка
Action (дія)	Прямокутник із закругленими кутами	Операція або крок, який виконується
Decision Node (умова)	Ромб	Розгалуження
Flow (потік)	Стрілка	Показує послідовність між умовами
End (кінець)	Подвійне коло	Кінцева точка, що означає завершення процесу

Завдяки чіткій візуалізації дій та умов, діаграма активності допомагає розробникам проаналізувати і структурувати логіку геймплею, що спрощує реалізацію відповідного функціоналу у коді гри.

Додатково, діаграми активності є корисними при командній розробці, оскільки дозволяють легко передавати задум ігрової логіки іншим учасникам команди – програмістам, дизайнерам, тестувальникам. Вони можуть слугувати основою для створення тестових сценаріїв, адже наочно показують можливі шляхи, які гравець може пройти в рамках одного процесу.

Розроблена діаграма активності, якщо гравець при активній фазі гри захоче поставити гру на паузу, що зображена на рисунку 3.2

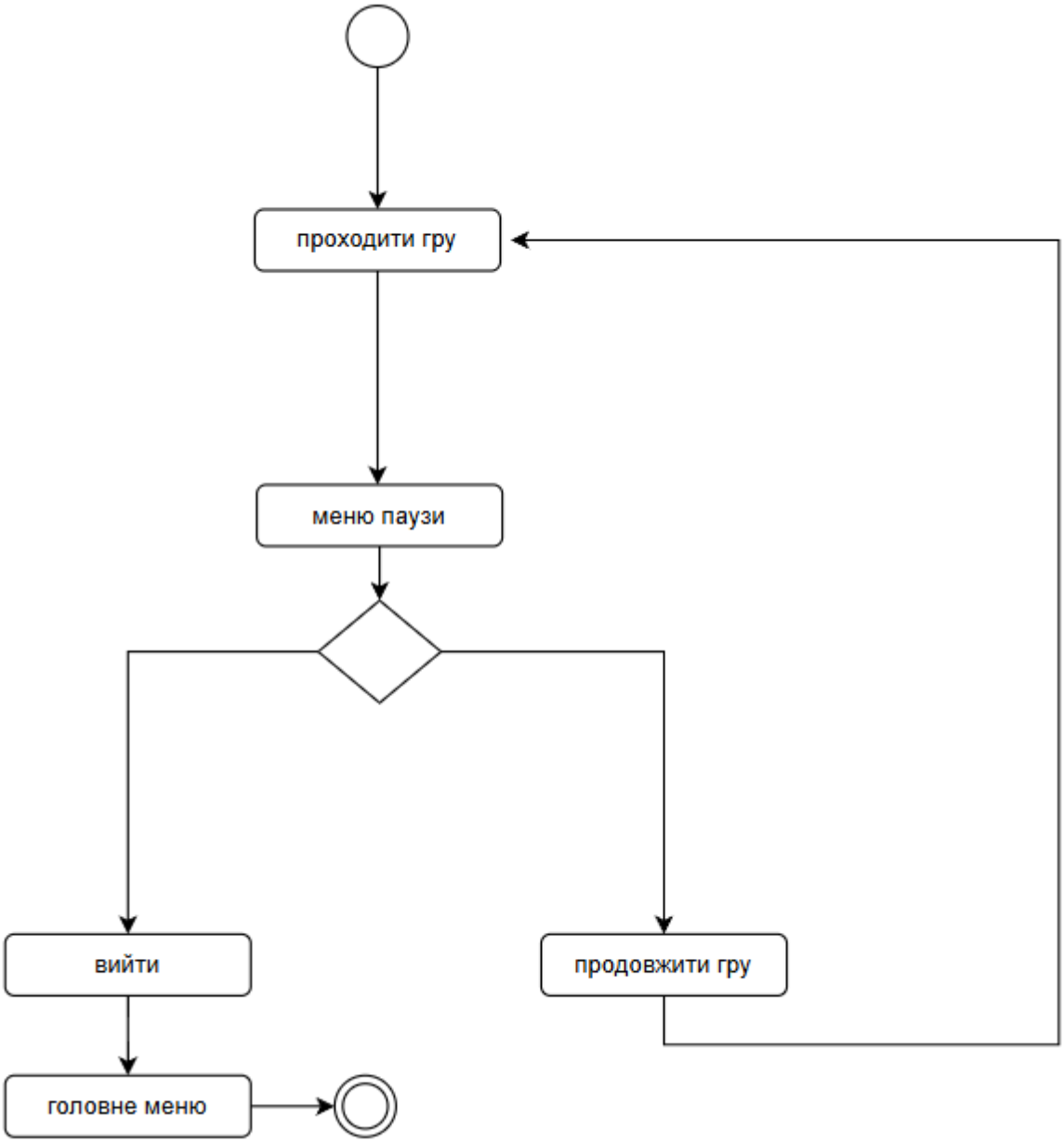


Рис. 3.2 Діаграма активності

3.4 Діаграма класів

У процесі проєктування гри є визначення структури класів, що реалізують функціональну логіку проєкту. Діаграма класів дозволяє представити архітектуру об’єктно-орієнтованої системи. Якщо конкретніше то це основні класи, які

використовуються у грі, зв’язки між ними, структура ієрархії, яка забезпечує логіку гри, інтерфейс тощо.

Для початку розпочнемо з класів, які відповідають за функціонал у головному меню.

MainMenu – цей клас відповідає за роботу кнопок в головному меню, а саме “Play”, “Options” та “Quit”. Має методи запуску гри та виходу із гри.

OptionsMenu – цей клас відповідає за меню налаштувань. Тут є методи які дають змогу налаштувати звук, графіку, роздільність екрану та режим відображення вікна.

Далі розберемо інші класи які є у проєкті.

EndGame – цей клас відповідає за обробку переходу до меню привітання з перемогою. Має метод який реагує на вхід гравця в тригер.

NextLevel – цей клас відповідає за обробку переходу гравця на наступний рівень при вході у тригер. Має метод який перевіряє тег об’єкта, якщо це гравець, то завантажує рівень.

VolumePow – цей клас керує гучність аудіо. Має методи, які ініціалізують, присвоюють та оновлюють гучність.

Pause – цей клас реалізовує функціональність паузи, відновлення гри та повернення в меню.

GameManager – цей клас відповідає за керування ігровим станом: програш, перезапуск, вихід у меню.

ClueManager – цей клас відповідає за показ текстових підказок у грі. Має методи які ініціалізують анімацію, вмикають та вимикають події, показують, закривають підказку, завдяки лічильнику.

ClueTriger – цей клас активує або вимикає підказку при вході, або виході у тригер. Має методи виклику та хования підказки.

Trap – цей клас завдає шкоди гравцю або ворогу при зіткненні з пасткою.

SkyParallax – це клас, який створює ефект паралаксного прокручування фону в залежності від руху камери. Має методи, які визначають початкову позицію та

визначають довжину спрайту, також обчислюють нову позицію спрайту, щоб створити ефект паралаксу.

Enemy – цей клас відповідає за ворогів у грі. Має метод Start, який ініціалізує здоров'я, підключає AI, метод TakeDamage, який відповідає за зменшення здоров'я, запускає анімації, метод Die відповідає за смерть противника.

AIChase – це клас, який реалізовує поведінку штучного інтелекту ворога, що переслідує, стрибає та змінює напрямок в залежності від гравця.

Attack – це клас, який реалізовує атаки ворога по гравцю за допомогою BoxCast (перевірка чи знаходиться гравець у зоні атаки) та cooldown (запуск атаки).

EnemyHealthBar – це клас, який відповідає за візуалізацію здоров'я ворога через slider та canvas – це UI компоненти Unity.

EnemyPatrol – це клас, який реалізовує патрулювання ворога за допомогою двох крайніх точок.

Hero – це основний клас гравця, який реалізовує керування персонажем: рух, стрибки, фізику, анімації.

PlayerCombar – це клас, який реалізовує бойову систему головного персонажа у грі. У тому числі й звуків.

SmoothCameraFollow – це клас, що забезпечує плавне переслідування камери за гравцем. Має метод, який плавно переміщує камеру ближче до позиції гравця.

Health – це клас, що відповідає за здоров'я гравця. Тобто отримання шкоди, смерть, відновлення здоров'я.

HealthBar – клас, що відповідає за візуальне оформлення здоров'я гравця. Іншими словами, UI-індикатор здоров'я гравця.

HealthCollectible – клас, який відповідає за збирання бонусу серце, яке додає здоров'я гравцю.

Розроблена діаграма класів, щоб продемонструвати структуру програмного продукту. У ній розписані класи, їхні властивості, методи, а також взаємозв'язки між ними, що дозволяє краще зрозуміти логіку побудови коду та організацію внутрішніх зв'язків у грі.

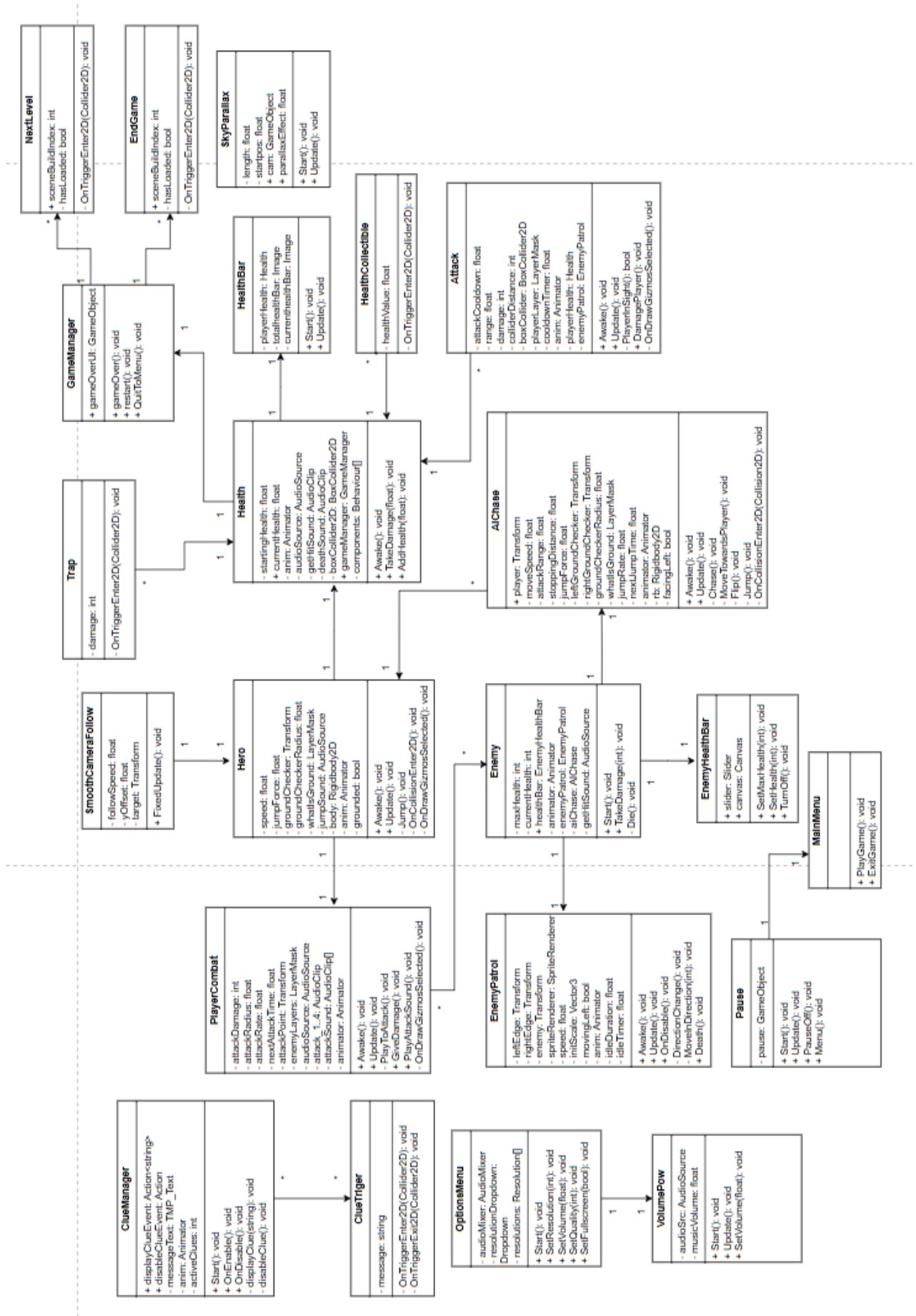


Рис. 3.3 Діаграма класів гри

На основі діаграми можна зробити висновок, що проєкт реалізовано з дотриманням принципів об’єктно-орієнтованого програмування, таких як інкапсуляція, модульність і повторне використання коду.

Таким чином, побудована діаграма класів допомогла систематизувати структуру гри, забезпечила цілісне бачення архітектури.

3.5 Розробка графічного інтерфейсу

Графічний інтерфейс є важливим елементом будь-якої комп’ютерної гри, оскільки забезпечує взаємодію між користувачем та ігровим середовищем.

У проєкті Night Hero інтерфейс реалізовано за допомогою системи UI, яка надається рушієм Unity.

Вона дозволяє створювати інтерактивні компоненти інтерфейсу, такі як кнопки, слайдери, текстові поля, індикатори тощо, використовуючи Canvas як основу для їх розміщення.

Розберемо основні елементи графічного інтерфейсу.

Головне меню – дозволяє гравцю розпочати гру, перейти до налаштувань або вийти з гри. Створено на основі елементів Button, із реалізацією переходу між сценами за допомогою SceneManager.LoadScene().

Також присутнє відображення назви гри за допомогою компоненту Text (або TextMeshPro), що підсилює стилізацію загального оформлення меню.

Приклад головного меню можна побачити на рисунку 3.4

Рис. 3.4 Головне меню гри

Меню налаштувань – реалізовує можливість регулювання гучності, перемикання якості графіки, зміни роздільної здатності екрана та перемикач екранного режиму.

Приклад меню налаштувань на рис. 3.5

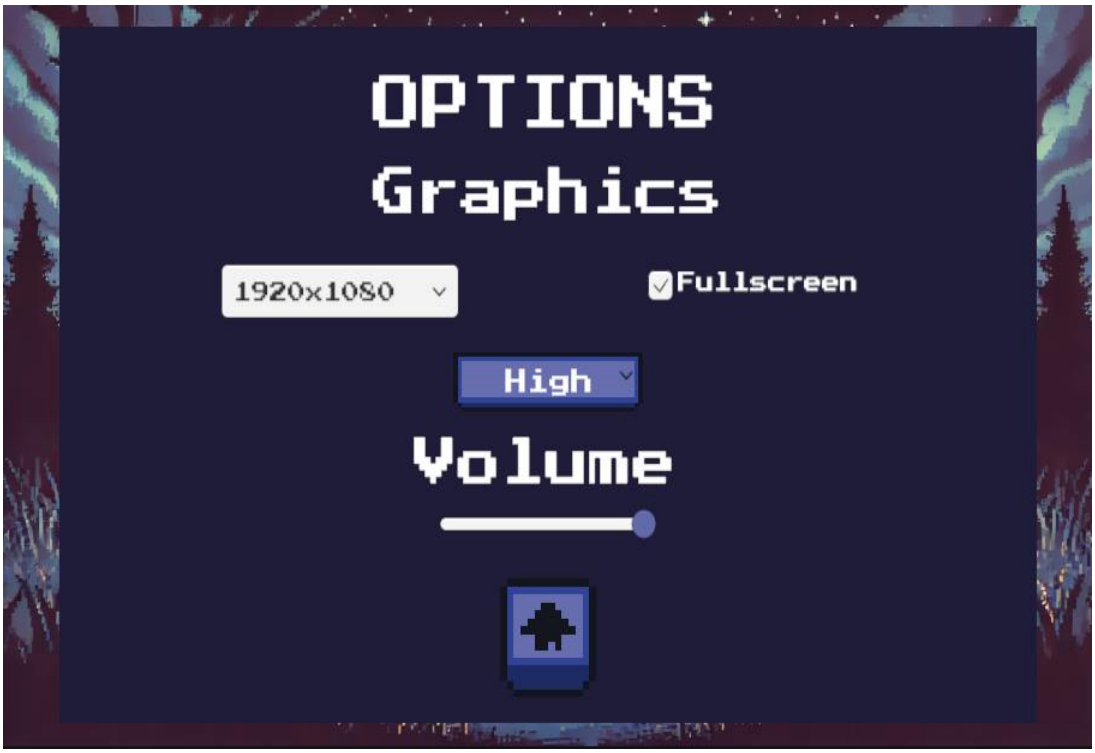


Рис. 3.5 Меню налаштувань

Меню паузи – активується натисканням клавіші Escape під час проходження рівня. Зупиняє час у грі (через встановлення значення Time.timeScale = 0) та дозволяє повернутися до головного меню або продовжити гру. При активації меню паузи на екрані відображається окрема UI-панель із кнопками, які забезпечують взаємодію користувача з відповідними функціями. Для зручності гравця меню також може містити додаткові опції, такі як налаштування звуку або перегляд підказок щодо керування. Після натискання кнопки "Продовжити" значення Time.timeScale повертається до 1, і гравець продовжує гру з того самого місця без втрати прогресу.

Такий підхід дозволяє забезпечити безперервність геймплею та зручне керування перебігом гри (див. рисунок 3.6).

Рис 3.6 Меню паузи

Індикатор здоров’я – візуально показує кількість життів гравця у вигляді заповненого Image. Значення оновлюються відповідно до поточного стану класу Health. Приклад індикатора здоров’я на рис. 3.6

Підказки — це текстові повідомлення, що з’являються на екрані під час активації певних тригерів у грі, наприклад, коли гравець наближається до

важливого об’єкта або входить у нову зону. Вони слугують інструментом комунікації між грою та користувачем, допомагаючи зорієнтуватися в ігровому процесі, пояснюють механіки чи надають інструкції. Реалізовані за допомогою бібліотеки TextMeshPro, яка забезпечує високоякісне відображення тексту, розширені можливості стилізації та підтримку ефектів, зокрема — анімації появи, зникнення, прозорості або масштабування. Відображення підказок контролюється за допомогою скриптів, які реагують на події зіткнення або входу в зону дії тригера (OnTriggerEnter2D). При цьому текст автоматично прив’язується до Canvas, розміщується в певній області екрана та може бути прихованим або показаним залежно від умов.

Приклад реалізованих підказок наведено на рисунку 3.7.

Рис. 3.7 Підказка у грі

Меню завершення гри – це меню з’являється після проходження гри, де вітають гравця з перемогою. При нажаті кнопки MENU, гравець вертається у головне меню гри.

Рис. 3.8 Меню завершення гри

Екран смерті – це меню з’являється після смерті гравця. У ньому можна перезапустити рівень або вернутись у головне меню гри.

Рис. 3.9 Екран смерті

Усі UI елементи розміщені в Canvas та використовують компоненти Unity UI. Створений графічний інтерфейс є інтуїтивно зрозумілим і мінімалістичним, що відповідає стилю піксель-арт.

3.6 Розробка рівнів у грі

Рівні у проєкті виступають основними сценами, в яких відбувається геймплей. Їх проєктування має на меті створення цікавого, динамічного й збалансованого ігрового процесу. Кожен рівень включає у себе платформні структури, ворогів, пастки, різних об’єктів та підказки.

Розробка рівнів починається з планування структури. Тобто, на цьому етапі визначаються основні елементи дизайну: стартова точка гравця, розміщення платформ, ворогів та об'єктів взаємодії. Для зручного розміщення платформ та декорацій було використано систему Tilemap, яка дозволяє створювати рівні у вигляді сітки. Це забезпечило швидку побудову і редагування ігрових сцен.

Рівні включають у себе ворогів з різною поведінкою, пастки, підказки, зони тригерів для переходу на наступний рівень, елементи бонусів з якими можна взаємодіяти. Список інструментів для реалізації:

- Unity Editor
- 2D Tile Palette
- Prefab елементи
- Canvas та UI

Також для покращення візуального ефекту були використані вбудовані інструменти Unity для роботи з освітленням рівнів.

Приклад редактору розробленого рівня можна побачити на рисунку 3.10.

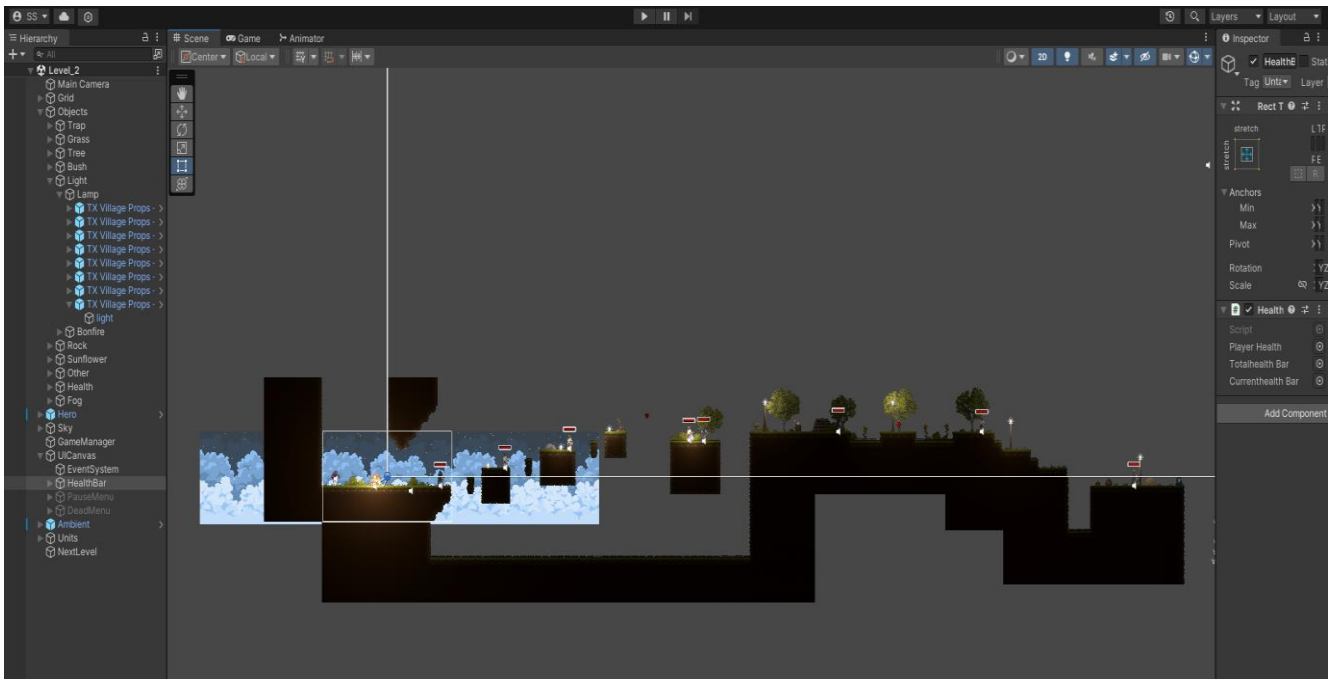


Рис. 3.10 Редактор рівня у рушії Unity

Отже, рівні у грі – це справжній виклик для гравця тому, що мають елементи бою, продуманих пасток, перешкод та різних класів ворогів. Також, гра надає візуальне враження від гри, завдяки різним анімаціям та освітленням.

3.7 Розробка ігрової механіки

Ігрова механіка – це сукупність правил, алгоритмів та взаємодій, що визначають поведінку персонажа, ворогів і об’єктів у грі. У проєкті Night Hero основний акцент було зроблено на динамічний платформерний геймплей з бойовими елементами та взаємодією з оточенням.

Переміщення персонажа: головний герой (Hero) здатен рухатись вліво і вправо, стрибати та змінювати напрямок погляду. Для цього використано компонент Rigidbody2D та обробку введення з клавіатури. Стан анімації перемикається за допомогою Animator.

Механіка стрибка: реалізована перевірка на наявність під ногами (OverlapCircle), що дозволяє стрибати тільки тоді, коли персонаж перебуває на землі. Стрибок запускається натисканням клавіші “W”.

Бойова система: гравець може атакувати ворогів (Enemy) за допомогою ближнього бою. Атака виконується по натисканню пробілу (Space) з активним радіусом ураження (attackRadius). Коли ворог опиняється в зоні атаки, йому наноситься шкода через метод TakeDamage.

Отримання шкоди та смерть: у класі Health зберігається поточне та максимальне здоров'я гравця. Якщо гравець отримує шкоду (від пасток або ворогів), здоров'я зменшується. У разі досягнення нуля – відбувається анімація смерті та викликається gameOver() з класу GameManager.

Шкала здоров'я: HealthBar синхронізується з Health та відображає поточний стан у вигляді графічного індикатора.

Патрулювання та переслідування ворогів: клас EnemyPatrol забезпечує рух ворога між контрольними точками. Коли гравець потрапляє в зону видимості, AIChase переключає ворога в режим переслідування.

Пастки: об'єкти Trap завдають шкоди гравцю та ворогам при зіткненні.

Бонуси: об'єкти HealthCollectible при дотику до гравця додають певне значення до здоров'я і зникають зі сцени. На рис. 3.11 зображено об'єкт HealthCollectible.

Рис. 3.11 Об'єкт HealthCollectible (бонус у грі)

3.8 Інтеграція аудіо

Ніяка гра не буде цікава гравцям без наявності звукової системи. Така система підсилює атмосферу та зацікавленість до гри.

У грі реалізовані звуки навколишнього середовища: костру, вітру, печери та звуків ночі (звуки бабок). Також звуки при отриманні шкоди, атаки, стрибка та смерті.

Приклади наявних аудіо файлів у грі представлено на рисунку 3.12.

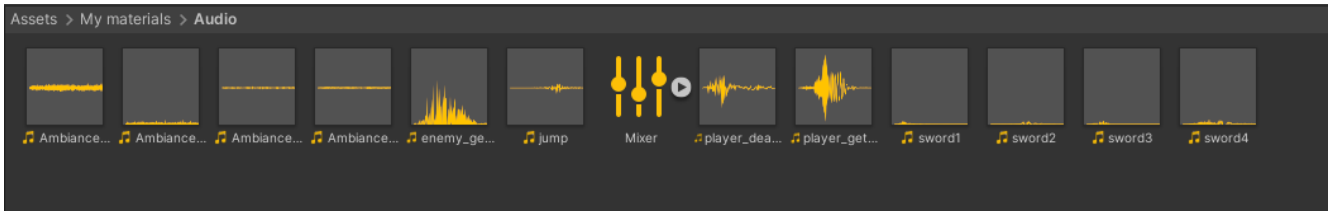


Рис 3.12 Аудіо файли

Також у грі можна налаштовувати гучність звуку. У меню налаштувань (OptionsMenu) реалізовано зміну гучності за допомогою слайдера (Slider), який дозволяє користувачу інтуїтивно регулювати бажаний рівень звучання. Значення слайдера передається у AudioMixer — вбудований інструмент Unity для роботи зі звуком, який забезпечує гнучке керування аудіосигналами. AudioMixer дозволяє змінювати гучність окремих аудіоканалів (музики, ефектів) незалежно один від одного, що покращує контроль над звуковим балансом у грі.

Крім того, слайдер може бути пов’язаний із текстовим індикатором відсоткового значення або піктограмами для візуального зворотного зв’язку. Такий підхід до реалізації звукових налаштувань сприяє персоналізації досвіду гри та робить інтерфейс більш дружнім і зручним для гравця.

3.9 Завантаження проєкту на GitHub

Для контролю версій програмного продукту було використано платформу GitHub, на яку завантажено повну структуру проєкту.

Застосування системи контролю версій дозволяє ефективно відстежувати всі зміни, що вносяться до коду, зберігати історію розробки та, у разі необхідності, швидко повертатися до стабільних попередніх версій.

GitHub також забезпечує спільну роботу над проєктом, що є особливо корисним при колаборації з іншими розробниками або під час подальшого розширення функціоналу. Кожне оновлення, коміт або виправлення зберігається з відповідним описом, що дозволяє чітко розуміти мету змін та уникнути помилок, пов’язаних з перезаписом файлів чи втратами коду.

Таким чином, завантаження проєкту на GitHub не лише дозволяє зберігати безпечні резервні копії файлів, але й створює зручне середовище для управління проєктом.

4 ТЕСТУВАННЯ ГРИ

4.1 Призначення тестування програмного продукту

Тестування гри є важливим етапом розробки, який дозволяє виявити та усунути помилки, покращити якість ігрового процесу та забезпечити стабільну роботу застосунку. Чим більше функціоналу буде протестовано, тим краще буде продукт на релізі.

Основними цілями для тестування є:

– перевірка відповідності вимогам: функціонал продукту має працювати згідно вимогам у технічному завданні. Тестування дозволяє перевірити, чи відповідає певна функціональність цим вимогам;

– виявлення помилок: тестування дозволяє виявити баги у роботі класів, функцій або інтерфейсу ще до того, як гру отримають кінцеві користувачі. Наприклад, помилки в бойовій системі або несправність меню можуть суттєво знизити якість гри;

– підвищення стабільності гри: після кожного додавання або зміни функціоналу тестування допомагає впевнитися, що новий код не зламав вже реалізовану логіку;

– покращення взаємодії з користувачем: тестування дозволяє переконатися, що усі елементи працюють очікувано, і користувач не стикається з труднощами при навігації або керуванні персонажем;

– регресійне тестування: тестування після змін, щоб переконатися, що нові функції не зламали старі;

– тестування продуктивності: тестування спрямоване на виявлення просідань FPS, затримок або зависань.

Отже, тестування є невід’ємною частиною розробки програмного продукту, що забезпечує його надійність, зручність для користувача та відповідність технічним вимогам.

4.2 Тестові сценарії

У проєкті Night Hero було застосовано переважно ручне функціональне тестування з акцентом на ключові механіки, взаємодію з UI та стабільність проходження рівнів.

Були сформульовані та проведені тестові сценарії. З ціллю виявлення відмінностей з вимогами до програмного продукту.

Переміщення гравця. Очікуваний результат: персонаж рухається вліво, вправо згідно натискань клавіш. Тестування показало, що дійсно персонаж рухається відповідно до очікуваного результату. Клавіша “A” рухає персонажа у ліво, а клавіша “D” рухає у право. Тестування цього сценарію було проведено успішно.

Стрибок гравця. Очікуваний результат: персонаж стрибає тільки при знаходженні на землі. Тестування показало, що гравець дійсно стрибає, коли знаходиться на платформі зі шаром ground. Тестування відповідає очікуваному результату.

Рис. 4.1 Стрибок у грі

Атака. Очікуваний результат: після натискання кнопки “Space” активується анімація атаки у гравця. Тестування показало, що при натисканні кнопки активується анімація атаки. Тестування проведено успішно.

Рис. 4.2 Атака гравця

Отримання шкоди. Очікуваний результат: після атаки ворогом по гравцю, зменшується здоров’я на індикаторі. При потраплянні під атаку ворога у гравця віднімається певна кількість одиниць здоров’я, що одразу відображається на інтерфейсі у вигляді зникнення відповідної кількості сердець на індикаторі здоров’я. Якщо здоров’я зменшується до нуля, активується стан поразки (Game Over). Тестування показало, що логіка втрати здоров’я працює коректно: після атаки ворога індикатор здоров’я миттєво оновлюється, і серця зникають згідно з отриманою шкодою. Крім того, було перевірено, що отримання шкоди викликає анімацію ушкодження гравця та короткочасне мерехтіння персонажа для візуального зворотного зв’язку. Тестування пройшло успішно, помилок у роботі механіки виявлено не було.

Рис. 4.3 Отримання шкоди

Підбір бонуса здоров’я. Очікуваний результат: при взаємодії з об’єктом HealthCollectible, добавляється серце до індикатора здоров’я. Тестування показало, що при взаємодії з об’єктом HealthCollectible, добавляється здоров’я у гравця. Тестування відповідає очікуваному результату.

Смерть ворога. Очікуваний результат: ворог переходить у стан смерті та лежить на поверхні платформ. Анімація смерті активується одразу після зниження його здоров’я до нуля. Після цього вимикається можливість руху та атаки ворога, зупиняється фізична взаємодія з гравцем, але колайдер ворога залишається активним, щоб забезпечити правдоподібну фізичну присутність на сцені. Тестування показало, що ворог після смерті коректно переходить у відповідний стан, відтворюється анімація смерті, і об’єкт не зникає з рівня, а лишається на поверхні платформи у нерухомому положенні. Така реалізація сприяє підвищенню реалістичності ігрового процесу та дозволяє зберігати візуальний слід битви на рівні.

Рис. 4.4 Стан смерті ворога

Відображення підказки. Очікуваний результат: при вході у зону ClueTrigger з’являється повідомлення, яке інформує гравця про важливу дію, об’єкт або ситуацію в грі. Підказка реалізована за допомогою системи UI з використанням компонента TextMeshPro, що забезпечує чітке й стильне відображення тексту. Активація повідомлення відбувається через подію OnTriggerEnter2D, яка викликає появу текстового блоку з анімацією появи для привернення уваги гравця. Повідомлення автоматично зникає після виходу з тригер-зони або через певний проміжок часу. Тестування показало, що при вході гравця у зону ClueTrigger підказка коректно з’являється, відображається у правильному місці на екрані, та не заважає ігровому процесу. Поведінка відповідає очікуванню, тестування завершено успішно, збоїв або помилок не виявлено.

Рис 4.5 Поява підкази

Поведінка ворога з механікою переслідування. Очікуваний результат: ворог переслідує гравця на визначеній відстані. Тестування показало, що механіка переслідування у ворога працює. Тест успішний.

Поведінка ворога з механікою патрулювання. Очікуваний результат: ворог патрулює між двома точками, змінюючи напрямок руху після досягнення кожної з них. Така поведінка реалізована за допомогою системи waypoints – двох фіксованих координат, між якими пересувається ворог. Після досягнення крайньої точки ворог розвертається та продовжує рух у зворотному напрямку. Під час патрулювання ворог постійно перебуває в активному стані та може реагувати на гравця, якщо той потрапить у зону виявлення. Тестування показало, що механіка патрулювання працює стабільно: ворог плавно переміщується між заданими точками, своєчасно змінює напрямок руху, не виходить за межі патрульного маршруту й не застрягає. Поведінка відповідає очікуваному результату, тестування пройшло успішно.

Смерть гравця. Очікуваний результат: стан гравця переходить в анімацію смерті та висвічується екран смерті з можливістю вернутись в меню або продовжити гру. Тестування показало, що після смерті гравця, стан персонажа

переходить в анімацію смерті та висвічується екран смерті. Тест відповідає очікуваному результату.

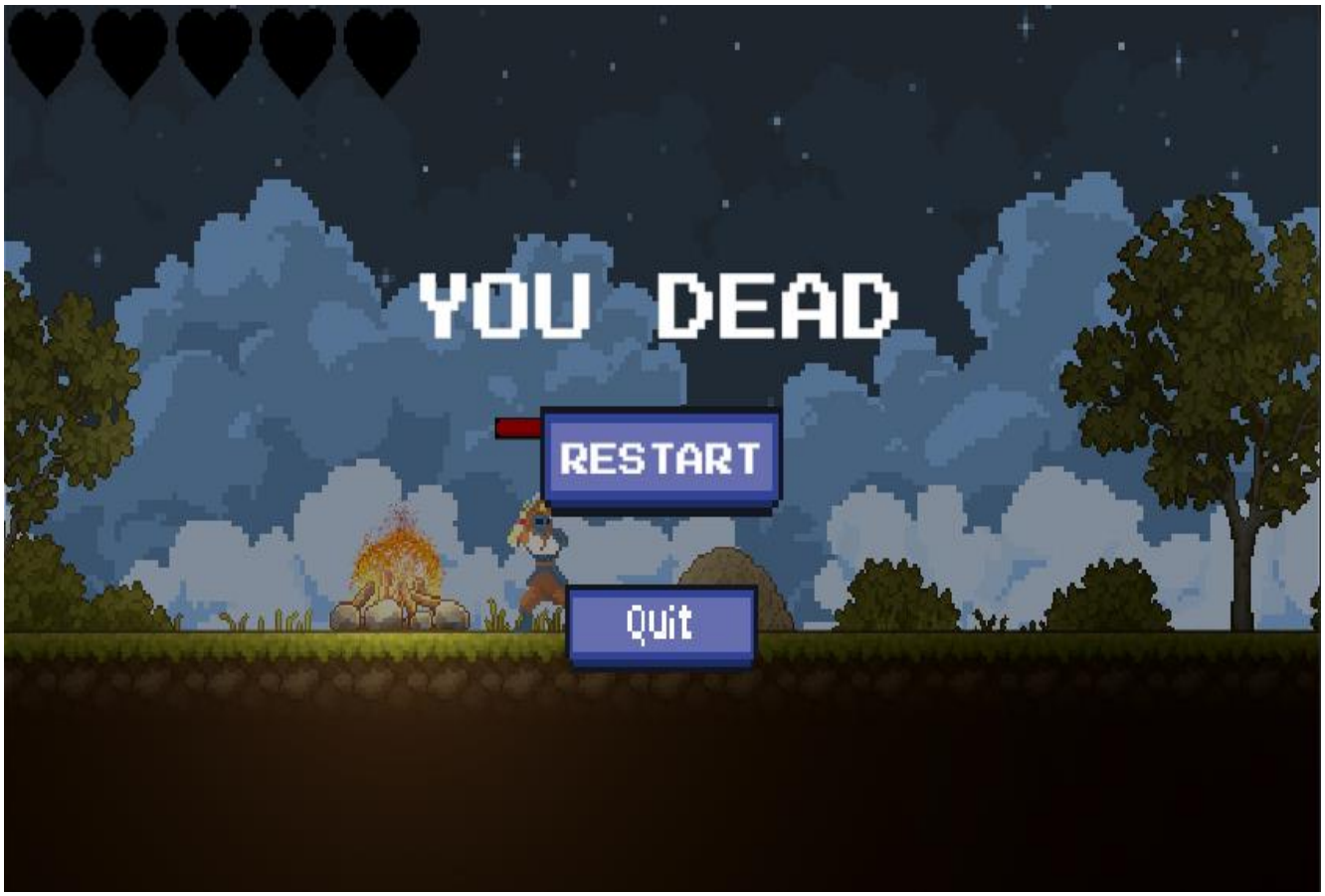


Рис. 4.6 Смерть гравця

Робота кнопок в екрані смерті. Очікуваний результат: кнопка “RESTART” перезавантажує рівень та кнопка “Quit” виходить у головне меню. Тестування показало, що кнопки відповідають очікуваному результату.

Пастки убивають. Очікуваний результат: при взаємодії гравця та ворогів з пастками, то вони помирають. Тестування показало, що пастки працюють згідно вимог.

Індикатор здоров’я ворога. Очікуваний результат: при отриманні шкоди від гравця, індикатор здоров’я гравця зменшується на відповідну кількість втрачених очок здоров’я. Тестування показало, що логіка роботи індикатора здоров’я відповідає очікуваному результату.

Рис. 4.7 Індикатор здоров’я ворога

Кількість шкоди. Очікуваний результат: противники LightBandit та HeavyBandit повинні наносити шкоди гравцю, рівної 1 та 2 серця відповідно. Тестування показало, що гравець від ударів ворогів виду LightBandit, втрачає 1 серце, а від виду HeavyBandit, втрачає 2 серця. Тестування відповідає очікуваному результату.

Перехід на наступний рівень. Очікуваний результат: при досягненні зони тригеру NextLevel, завантажується новий рівень. Результати тестування показали, що гравець при досягненні зони тригеру NextLevel, дійсно, переходить на новий рівень.

Пауза під час гри. Очікуваний результат: коли натиснуто кнопку “Escape” на клавіатурі, час зупиняється, і показується меню паузи. Тестування показало, що при натисканні кнопки “Escape”, час у грі зупиняється, і показується меню паузи. Тестування відповідає очікуваним результатам.

Рис. 4.8 Пауза під час гри

Кнопки у меню паузи. Очікуваний результат: наявні кнопки у меню паузи (Continue та Quit) мають продовжити гру або вийти в головне меню. Результати тестування показали, що кнопки відповідають очікуванням.

Завершення гри. Очікуваний результат: відкривається меню завершення гри, де можна вийти в головне меню. Тестування показало, що після проходження останнього рівня, відкривається меню завершення гри, де вітають гравця з перемогою, та можливістю вийти у головне меню. Результат тестування є успішним.

Налаштування звуку. Очікуваний результат: при зміні положення slider у грі через налаштування, змінюються дБ відповідно до нового значення. Результат тестування показав, що при зміні положення slider, звук робиться голосніше або тихіше, що відповідає очікуваному результату.

Зміна роздільної здатності. Очікуваний результат: при виборі нової роздільної здатності (дивитись рис. 4.9), гра переходить до вибраної роздільної здатності. Тестування показало, що у грі реалізована зміна роздільної здатності відповідно до вибраного користувачем розширення.

Це означає, що тестування пройшло успішно.

Рис. 4.9 Список розширень екрану

Зміна якості графіки. Очікуваний результат: параметри графіки оновлюються відповідно до вибраного користувачем значення. Тестування показало, що параметри графіки оновлюються відповідно до очікуваного результату.

Змінна екранного режиму. Очікуваний результат: змінюється екранний режим. Тестування показало, що користувач без проблем змінює екранний режим у грі.

Запуск гри. Очікуваний результат: гра запускається без помилок та вилетів. Тестування показало, що запуск гри працює без помилок, і гравця зустрічає меню гри. Результат тестування відповідає очікуваному.

Звук костра на рівні. Очікуваний результат: об’єкт Bonfire має мати звуковий елемент та звукову доріжку зі звуками костру. Тестування показало, що результат тестування відповідає очікуваному результату.

Рис. 4.10 Елемент звуку на об’єкті Bonfire

Камера сцени переслідує гравця. Очікуваний результат: камера слідкує за позицією гравця під час руху, також має можливість рухатись по горизонталі та вертикалі. Тестування показало, що камера переслідує гравця по вертикалі та горизонталі.

У результаті тестування гри Night Hero було підтверджено правильну роботу всіх ключових компонентів ігрового проєкту. Тестування охопило різні аспекти: від базових дій гравця (руху, стрибків, атаки) до перевірки складніших механік, таких як бойова система, обробка шкоди, взаємодія з об’єктами середовища, підказки та переходи між рівнями. Особливу увагу було приділено правильності роботи користувацького інтерфейсу, коректності дій меню, та можливості налаштування параметрів гри.

У підсумку, тестування гри Night Hero стало важливим етапом контролю якості, підтвердило технічну готовність продукту та засвідчило досягнення поставлених цілей розробки.

ВИСНОВКИ

У результаті виконаної кваліфікаційної роботи було розроблено гру жанру 2D-платформер, який має динамічний ігровий процес, бойову систему та цікаво реалізовані рівні. Гра розроблялась з використанням рушія Unity та мови програмування C#.

Було досягнуто таких результатів:

– Проаналізовано предметну область: визначено сутність жанру 2D-платформер, історію розвитку предметної області, цільову аудиторію програмного продукту, досліджено існуючі аналоги (їх переваги та недоліки), визначено функціональні та нефункціональні вимоги до проєкту.

– Засоби реалізації: інтегроване середовище розробки Visual Studio, мова програмування C#, ігровий рушій Unity, GitHub для контролю версій та Aseprite для створення та анімацій спрайтів.

– Розроблена гра має ігрові механіки: переміщення персонажа, бойову систему, систему здоров’я, штучний інтелект ворогів, збір бонусів, пастки, підказки.

– Проєкт було завантажено на GitHub, що забезпечує можливість контролю версій та подальшого розширення.

– Розроблено архітектуру продукту за допомогою UML діаграм (діаграма класів, діаграма активності, діаграма варіантів використання). Це допомогло спростити планування реалізації функціоналу та сприяли кращому розумінню структури гри.

Розроблена гра відповідає всім поставленим вимогам, що дозволяє задовольнити потреби гравців та розвити їх навики реакції, просторового мислення та пам’яті.

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

1. Скиба С.С., Ярошевський О.В. Інтеграція IoT-технологій через гру 2D платформер. Міжнародна науково-технічна конференція «Сучасний стан та перспективи розвитку IoT». 15 квітня 2025 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2025. (*подано до друку*)
2. Скиба С.С., Ярошевський О.В. Визначення вимог до гри жанру 2D платформер. Всеукраїнська науково-технічна конференція «*Застосування програмного забезпечення в інформаційно-комунікаційних технологіях*», 24 квітня 2025 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2025. С. 233-234.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Red Bull. The evolution of platform games in 9 steps. URL: https://www.redbull.com/in-en/evolution-of-platformers (дата звернення: 27.04.2025).
2. Kelleher Bros. Defining the Platformer genre — Kelleher Bros. URL: https://www.kelleherbros.com/blog/2024/3/4/defining-the-platformer-genre#:~:text=Your%20stereotypical%20platformer%20is%20a,just%20to%20name%20a%20few (дата звернення: 27.04.2025).
3. Visure Solutions. Функціональні та нефункціональні вимоги (з прикладами). URL: https://surli.cc/vchagj (дата звернення: 28.04.2025).
4. Kharkov Trend. Що таке інтегроване середовище розробки. URL: https://kharkov-trend.in.ua/uk/articles-3316-shho-take-integrovane-seredovyshhe-rozrobky (дата звернення: 29.04.2025).
5. Microsoft Learn: Build skills that open doors in your career. What is the Visual Studio IDE? URL: https://learn.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide?view=vs-2022 (дата звернення: 29.04.2025).
6. FoxmindEd. Середовище програмування сі шарп та його переваги. URL: https://foxminded.ua/seredovyshche-prohramuvannia-si-sharp/ (дата звернення: 01.05.2025).
7. Lemon School. Що таке Unity. URL: https://lemon.school/blog/shho-take-unity (дата звернення: 02.05.2025).
8. ElAr. Порівняння ігрових рушіїв для створення 2d платформеру. URL: https://openarchive.nure.ua/entities/publication/2ea47884-ac69-4b2f-a1c7-c7aea5feb3fa. (дата звернення: 02.05.2025).
9. О. Іваха, Д. Гритчин, Л. Остапенко. Особливості створення 2D-ігор засобами Unity. Всеукраїнської (з міжнародною участю) науково-практичної конференції молодих учених, 11-12 травня 2022 р., Харків, Харківський національний педагогічний університет імені Г. С. Сковороди. С. 120.

10. Бондаренко С. М. Розробка 2D-ігор з використанням Unity: посібник для початківців. Київ: Вид. дім «Освіта», 2021. 210 с.
11. Hocking J. Unity in Action, Third Edition: Multiplatform game development in C#. Manning Publications Co. LLC, 2022. 416 p.
12. Patrick Felicia. The Ultimate Guide to 2D games with Unity: Build your favorite 2D Games easily with Unity. Independently published, 2020. 487 p.
13. Hocking J. Unity in Action, Second Edition: Multiplatform Game Development in C#. Manning Publications Co. LLC, 2018. 400 p.
14. Lanzinger F. 2D Game Development with Unity. CRC Press, 2020. 336 p.
15. Tykoski S. Mastering Game Design with Unity 2021: Immersive Workflows, Visual Scripting, Physics Engine, GameObjects, Player Progression, Publishing, and a Lot More (English Edition). BPB Publications, 2022. 540 p.
16. Wells R. Unity 2020 by Example: A Project-Based Guide to Building 2D, 3D, Augmented Reality, and Virtual Reality Games from Scratch, 3rd Edition. Packt Publishing, Limited, 2020. 676 p.
17. GitHub Docs. About GitHub and Git - GitHub Docs. URL: https://docs.github.com/en/get-started/start-your-journey/about-github-and-git (дата звернення: 06.05.2025).
18. Aseprite - Animated sprite editor & pixel art tool. Aseprite. URL: https://www.aseprite.org/docs/ (дата звернення: 07.05.2025).
19. DOU. URL: https://dou.ua/forums/topic/40575/ (дата звернення: 08.05.2025).
20. Державний університет інформаційно-комунікаційних технологій. Застосування UML (частина 3). Діаграма класів - Class Diagram. URL: https://duikt.edu.ua/ua/news-1-626-8002-zastosuvannya-uml-chastina-3-diagrama-klasiv----class-diagram_kafedra-kompyuternih-nauk-ta-informaciynih-tehnologiy (дата звернення: 09.05.2025).

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка гри в жанрі 2D-платформер

Виконав студент 4 курсу

групи ПД-44
Скиба Сергій Сергійович
Керівник роботи

асистент кафедри ІПЗ Ярошевський Олександр Вікторович

Київ – 2025

МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: покращення реалізації ігрового процесу шляхом впровадження механік стрибків, атак і взаємодії з об’єктами, реалізації фізично коректного руху персонажа, налаштування адаптивної камери, а також забезпечення навігації за допомогою системи візуальних підказок.
Об’єкт дослідження: процес створення комп’ютерних ігор жанру платформер для розважальних цілей користувачів.

Предмет дослідження: програмне забезпечення, що дозволяє керувати персонажем, проходити рівні з різноманітними перешкодами та ворогами у грі жанру 2D-платформер.

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Огляд та аналіз предметної області.
2. Проєктування архітектури гри.
3. Програмна реалізація та опис функціонування гри в жанрі 2D-платформер.
4. Тестування гри.

3

АНАЛІЗ АНАЛОГІВ

Показник	Super Mario Bros	Spelunky Classic	VVVVVV	Night Hero
Наявність ворогів	+	+	-	+
Реакція ворогів на гравця	-	-	-	+
Індикатор здоров'я ворогів	-	-	-	+
Стиль графіки	8-бітна піксельна	Піксель-арт	Ретро	Піксель-арт
Підказки для гравця	-	-	-	+
Можливість ставити гру на паузу	+	+	+	+
Камера слідкує за гравцем	Лише по горизонталі вправо	Ривкове переключення екранів	Статичні екрани	По горизонталі та по вертикалі
Пастки	+	+	+	+
Рух ворогів по маршруту	+	+	-	+
Регенерація здоров’я	-	+	-	+
Наявність екрана “Game Over”	+	+	-	+

4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги:

- Взаємодія з грою здійснюється за допомогою клавіатури та комп'ютерної мишки.
- Головний персонаж рухається вліво, вправо, стрибає та атакує.
- Ворожі персонажі, які рухаються за заданими маршрутами або реагують на гравця.
- Після втрати всіх сердець має з'являтися екран "Game Over".
- Реалізовані пастки (шипи) проти головного героя.
- Гравець має можливість перезапустити гру після смерті або повернутися до головного меню.
- Над головами ворожих персонажів повинно показуватись їх здоров'я.
- Реалізований об'єкт серце для регенерації здоров'я головного персонажа.
- Гра містить підказки для гравця.
- Можливість ставити гру на паузу.
- Камера сцени рівня переслідує гравця.
- Анімації дій персонажів

Нефункціональні вимоги:

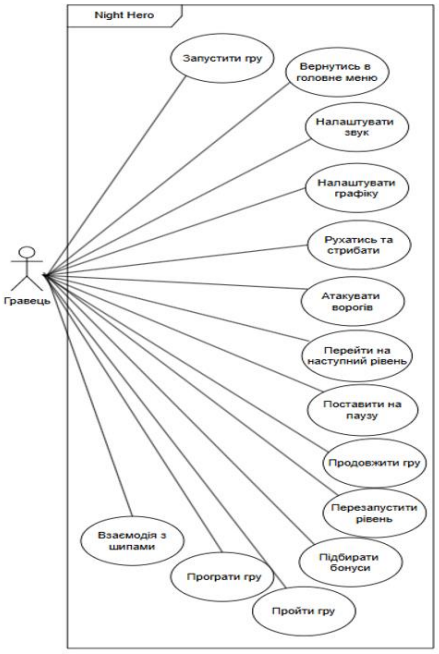
- Всі аудіосигнали мають відтворюватися без затримок, з гучністю, що не перевищує комфортний рівень (до 70 дБ)
- Повне завантаження гри має тривати не більше 20 секунд
- Завантаження рівнів не повинно перевищувати 5 секунд.
- Кількість кадрів під час гри є не меншою 30.

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

Visual Studio

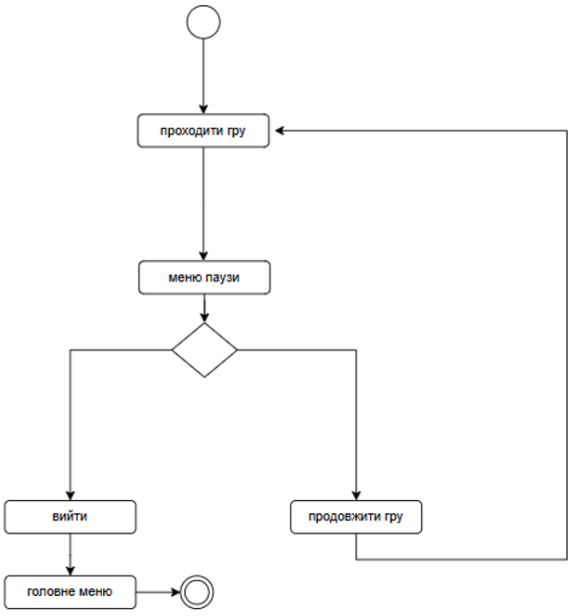
Aseprite

ДІАГРАМА ВАРІАНТІВ ВИКОРИСТАННЯ



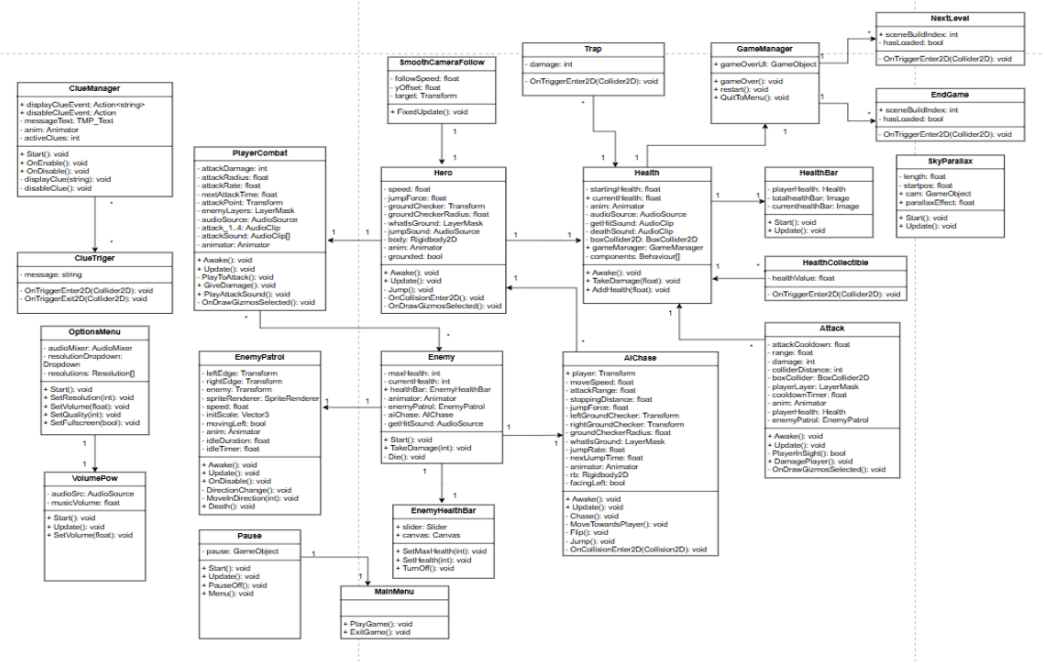
7

ДІАГРАМА АКТИВНОСТІ



8

ДІАГРАМА КЛАСІВ



ЕКРАННІ ФОРМИ

Головне меню

Меню налаштування

ЕКРАННІ ФОРМИ

Рівень у грі

Екран смерті

11

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

Тези доповідей:

1. Скиба С.С., Ярошевський О.В. Інтеграція IoT-технологій через гру 2D платформер. Матеріали Міжнародної науково-технічної конференції «Сучасний стан та перспективи розвитку IoT». Збірник тез. 15 квітня 2025 року, ДУІКТ, м. Київ (*подано до друку*)
2. Скиба С.С., Ярошевський О.В. Визначення вимог до гри жанру 2D платформер. Матеріали Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». Збірник тез. 24 квітня 2025 року, ДУІКТ, м. Київ, с. 233-234.

12

ВИСНОВКИ

1. Проаналізовано предметну область (сутність жанру, історія розвитку, цільова аудиторія, аналоги), що дозволило створити концепцію гри.
2. Визначені функціональні та нефункціональні вимоги до Night Hero за допомогою дослідження аналогів, які у подальшому були реалізовані у грі.
3. Спроєктовано архітектуру гри за допомогою UML діаграм: діаграма активності, діаграма класів, діаграма варіантів використання. Це допомогло спростити планування реалізації функціоналу та сприяли кращому розумінню структури гри.
4. Визначені засоби реалізації та задіяні для розробки гри. Зокрема, Unity використано як основне середовище для побудови логіки та візуалізації ігрового процесу, мова програмування C# застосована для створення скриптів керування об'єктами та подіями. Aseprite використано для створення та анімації спрайтів. GitHub забезпечив контроль версій проєкту.
5. Проведено ручне тестування, що допомогло знайти та виправити помилки до релізу гри. Наприклад, провалювання головного героя скрізь текстури.

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

```
using System;
using System.Collections;
using System.Collections.Generic;
using TMPro;
using UnityEngine;

public class ClueManager : MonoBehaviour
{
    public static Action<string> displayClueEvent;
    public static Action disableClueEvent;
    [SerializeField] private TMP_Text messageText;
    private Animator anim;
    private int activeClues;
    public void Start()
    {
        anim = GetComponent<Animator>();
    }
    private void OnEnable()
    {
        displayClueEvent += displayClue;
        disableClueEvent += disableClue;
    }
    private void OnDisable()
    {
        displayClueEvent -= displayClue;
        disableClueEvent -= disableClue;
    }
    private void displayClue(string message)
    {
        messageText.text = message;
        anim.SetInteger("state", ++activeClues);
    }
    private void disableClue()
    {
        anim.SetInteger("state", --activeClues);
    }
}
```

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

public class ClueTriger : MonoBehaviour
{
    [Header("Clue Text")]
    [TextArea(3, 10)]
    [SerializeField] private string message;
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("Player"))
        {
            ClueManager.displayClueEvent?.Invoke(message);
        }
    }
    private void OnTriggerExit2D(Collider2D collision)
    {
        if (collision.CompareTag("Player"))
        {
            ClueManager.disableClueEvent?.Invoke();
        }
    }
}
```

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

public class AIChase : MonoBehaviour
{
    public Transform player;
    [Header("Moving")]
    [SerializeField] private float moveSpeed ;
    [Header("Attack")]
    [SerializeField] private float attackRange ;
    [SerializeField] private float stoppingDistance;
    [Header("Jumping")]
    [SerializeField] private float jumpForce;
    [SerializeField] private Transform leftGroundChecker;
    [SerializeField] private Transform rightGroundChecker;
    [SerializeField] private float groundCheckerRadius;
    [SerializeField] private LayerMask whatIsGround;
    [SerializeField] private float jumpRate = 2f;
    private float nextJumpTime = 0f;
    [Header("Animator")]
    [SerializeField] private Animator animator;
    private Rigidbody2D rb;
    private bool facingLeft = true;
    private void Awake()
    {
        animator = GetComponent<Animator>();
        rb = GetComponent<Rigidbody2D>();
    }
    void Update()
    {
        Chase();
    }
    private void Chase()
    {
        float distanceToPlayer =
Vector2.Distance(transform.position, player.position);
        if (distanceToPlayer < attackRange)
        {
            if (distanceToPlayer > stoppingDistance)
            {
                MoveTowardsPlayer();
                animator.SetBool("moving", true);
            }
            else
            {
                animator.SetBool("moving", false);
            }
        }
        else
        {

            animator.SetBool("moving", false);
        }
        bool wannaJumpLeft =
Physics2D.OverlapCircle(leftGroundChecker.position,
groundCheckerRadius, whatIsGround);
        bool wannaJumpRight =
Physics2D.OverlapCircle(rightGroundChecker.position,
groundCheckerRadius, whatIsGround);
        if (Time.time >= nextJumpTime)
        {
            if (wannaJumpLeft || wannaJumpRight)
```

```
            {
                Jump();
                nextJumpTime = Time.time + 1f / jumpRate;
            }
        }
    }
    private void MoveTowardsPlayer()
    {
        Vector2 direction = player.position - transform.position;
        rb.velocity = new Vector2(direction.normalized.x * moveSpeed, rb.velocity.y);
        if (direction.x < 0 && !facingLeft || direction.x > 0 && facingLeft)
        {
            Flip();
        }
    }
    void Flip()
    {
        facingLeft = !facingLeft;
        Vector3 scale = transform.localScale;
        scale.x *= -1;
        transform.localScale = scale;
    }
    private void Jump()
    {
        rb.velocity = new Vector2(rb.velocity.x, jumpForce);
        animator.SetBool("jumping", true);
    }
    private void OnCollisionEnter2D(Collision2D collision)
    {
        if (collision.gameObject.CompareTag("Ground"))
        {
            animator.SetBool("jumping", false);
        }
    }
}

using System.Collections;
using System.Collections.Generic;
using Unity.Burst.CompilerServices;
using UnityEngine;

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

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

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

    private Animator anim;
    private Health playerHealth;

    private EnemyPatrol enemyPatrol;

    private void Awake()
    {
        anim = GetComponent<Animator>();
        enemyPatrol = GetComponentInParent<EnemyPatrol>();
    }
    private void Update()
    {
        cooldownTimer += Time.deltaTime;

        if (PlayerInSight())
        {
            if (cooldownTimer >= attackCooldown)
            {
                cooldownTimer = 0;
                anim.SetTrigger("attack");
            }
        }
        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.collider.GetComponent<Health>();
        }

        return hit.collider != null;
    }
    private void OnDrawGizmosSelected()
    {
        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));
    }

    public void DamagePlayer()
    {
        if (PlayerInSight())
        {
            playerHealth.TakeDamage(damage);
        }
    }
}

using System.Collections;
using System.Collections.Generic;
using Unity.VisualScripting;
using UnityEngine;

public class Enemy : MonoBehaviour
{

    [SerializeField] private int maxHealth;
    private int currentHealth;
    public EnemyHealthBar healthBar;

    [SerializeField] private Animator animator;

    [SerializeField] private EnemyPatrol enemyPatrol;

    [SerializeField] private AIChase aiChase;

    [SerializeField] private AudioSource getHitSound;
```

```
    private void Start()
    {
        currentHealth = maxHealth;
        healthBar.SetMaxHealth(maxHealth);

        aiChase = GetComponent<AIChase>();
    }

    public void TakeDamage(int damage)
    {
        animator.SetTrigger("hurt");
        getHitSound.Play();
        currentHealth -= damage;
        healthBar.SetHealth(currentHealth);

        if (currentHealth <= 0)
        {
            Die();
            healthBar.TurnOff();
        }
    }

    private void Die()
    {
        animator.SetBool("isDead", true);

        if (aiChase != null)
        {
            aiChase.enabled = false;
        }
        else if (enemyPatrol != null)
        {
            enemyPatrol.Death();
        }

        this.GameObject().layer =
LayerMask.NameToLayer("DeadEnemy");
    }

}

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

public class EnemyHealthBar : MonoBehaviour
{
    public Slider slider;
    public Canvas canvas;
    public void SetMaxHealth(int health)
    {
        slider.maxValue = health;
        slider.value = health;
    }

    public void SetHealth(int health)
    {
        slider.value = health;
    }

    public void TurnOff()
    {
        canvas.enabled = false;
    }
}

using System.Collections;
using System.Collections.Generic;
using Unity.VisualScripting;
using UnityEngine;

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

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

    [Header("Movement parameters")]
    [SerializeField] private float speed = 4;


    private Vector3 initScale;
    private bool movingLeft;

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

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

    private void Awake()
    {
        initScale = enemy.localScale;
    }

    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 OnDisable()
    {
        anim.SetBool("moving", false);
    }

    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);
        // Face direction

            spriteRenderer.flipX = Mathf.Abs(initScale.x) *
initScale.x * _direction > 0;


        //Move direction
        enemy.position = new Vector3(enemy.position.x +
Time.deltaTime * _direction * speed,
            enemy.position.y, enemy.position.z);
    }

    public void Death()
    {
        speed = 0;
    }
    private void OnAnimatorMove()
    {

    }
}

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

public class SkyParallax : MonoBehaviour
{
    private float length, startpos;
    public GameObject cam;
    public float parallaxEffect;
    void Start()
    {
        startpos = cam.transform.position.x;
        length = GetComponent<SpriteRenderer>().bounds.size.x;
    }

    void Update()
    {
        float temp = (cam.transform.position.x * (1 -
parallaxEffect));
        float dist = (cam.transform.position.x * parallaxEffect);

        transform.position = new Vector3(startpos + dist,
cam.transform.position.y, transform.position.z);

        if (temp > startpos + length) startpos += length;
        else if (temp < startpos - length) startpos -= length;
    }
}

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

public class Health : MonoBehaviour
{
    [SerializeField] private float startingHealth;
    public float currentHealth;
    private Animator anim;
    private bool  dead;
    [SerializeField] AudioSource audioSource;
    [SerializeField] private AudioClip getHitSound;
    [SerializeField] private AudioClip deathSound;

    private BoxCollider2D boxCollider2D;

    public GameManager gameManager;

    [Header("Components")]
    [SerializeField]private Behaviour[] components;
    private void Awake()
    {
        currentHealth = startingHealth;
        anim = GetComponent<Animator>();

        boxCollider2D = GetComponent<BoxCollider2D>();
    }
    public void TakeDamage(float _damage)
    {
        currentHealth = Mathf.Clamp(currentHealth - _damage, 0,
startingHealth);

        if (currentHealth > 0)
        {
            //player hurt
            anim.SetTrigger("hurt");
            //iframes
            audioSource.clip = getHitSound;
            audioSource.Play();
        }
        else
        {
            //player dead
            if (!dead)
            {
                anim.SetTrigger("die");
                audioSource.Stop();
                audioSource.clip = deathSound;
                audioSource.Play();

                foreach (Behaviour component in components)
                {
                    component.enabled = false;
                }

                boxCollider2D.sharedMaterial = null;

                dead = true;

                gameManager.gameOver();
            }

        }
    }

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


}

using System.Collections;
using System.Collections.Generic;
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 / 10;
   }
   private void Update()
   {
      currenthealthBar.fillAmount = playerHealth.currentHealth / 10;
   }
}
```

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

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

   private void OnTriggerEnter2D(Collider2D collision)
   {
      if(collision.tag == "Player")
      {

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

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;


public class MainMenu : MonoBehaviour
{
   public void PlayGame()
   {

SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);
   }
   public void ExitGame()
   {
      Application.Quit();
   }

}
```

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Audio;
using UnityEngine.UI;

public class OptionsMenu : MonoBehaviour
{
   public AudioMixer audioMixer;

   public Dropdown resolutionDropdown;

   Resolution[] resolutions;

   void Start()
   {
      resolutions = Screen.resolutions;

      resolutionDropdown.ClearOptions();

      List<string> options = new List<string>();

      int currentResolutionIndex = 0;

      for (int i = 0; i < resolutions.Length; i++)
      {
         string option = resolutions[i].width + "x" + resolutions[i].height;
         options.Add(option);

         if (resolutions[i].width == Screen.currentResolution.width &&
            resolutions[i].height == Screen.currentResolution.height)
         {
            currentResolutionIndex = i;
         }
      }
      resolutionDropdown.AddOptions(options);
      resolutionDropdown.value = currentResolutionIndex;
      resolutionDropdown.RefreshShownValue();
   }

   public void SetResolution(int resolutionIndex)
   {
      Resolution resolution = resolutions[resolutionIndex];
      Screen.SetResolution(resolution.width, resolution.height, Screen.fullScreen);
   }

   public void SetVolume(float volume)
   {
      audioMixer.SetFloat("master", volume);
   }

   public void SetQuality(int qualityIndex)
   {
      QualitySettings.SetQualityLevel(qualityIndex);
   }

   public void SetFullscreen(bool isFullscreen)
   {
      Screen.fullScreen = isFullscreen;
   }
}
```

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;


public class Pause : MonoBehaviour
{
   [SerializeField]
   GameObject pause;
   void Start()
   {
      pause.SetActive(false);
   }

   void Update()
```

```
    {
        if (Input.GetKeyUp(KeyCode.Escape))
        {
            pause.SetActive(true);
            Time.timeScale = 0;
        }
    }

    public void PauseOff()
    {
        pause.SetActive(false);
        Time.timeScale = 1;
    }

    public void Menu()
    {
        SceneManager.LoadScene(0);
        Time.timeScale = 1;
    }
}

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

public class Hero : MonoBehaviour
{
    [Header("Horizontal Movement")]
    [SerializeField] private float speed;

    [Header("Jumping")]
    [SerializeField] private float jumpForce;
    [SerializeField] private Transform groundChecker;
    [SerializeField] private float groundCheckerRadius;
    [SerializeField] private LayerMask whatIsGround;

    [Header("Sounds")]
    [SerializeField] private AudioSource jumpSound;

    private Rigidbody2D body;
    private Animator anim;
    private bool grounded;

    public void Awake()
    {
        body = GetComponent<Rigidbody2D>();
        anim = GetComponent<Animator>();
    }
    private void Update()
    {
        float horizontalInput = Input.GetAxis("Horizontal");
        body.velocity = new Vector2(Input.GetAxis("Horizontal")
* speed, body.velocity.y);

        //Flip player when moving left-right
        if (horizontalInput >  0.01)
            transform.localScale = new Vector3(4,4,1);
        else if (horizontalInput < -0.01f)
            transform.localScale = new Vector3(-4,4,1);

        bool                    isGrounded                    =
Physics2D.OverlapCircle(groundChecker.position,
groundCheckerRadius, whatIsGround);

        if (Input.GetKeyDown(KeyCode.W) && isGrounded)
        {
            Jump();
        }

        //Set animator
        anim.SetBool("run", horizontalInput != 0);
        anim.SetBool("grounded", grounded);
    }

    private void Jump()
    {
        body.velocity = new Vector2(body.velocity.x, jumpForce);
        anim.SetTrigger("jump");
        jumpSound.Play();
        grounded = false;
    }
    private void OnCollisionEnter2D(Collision2D collision)
    {
        if(collision.gameObject.CompareTag("Ground"))
            grounded = true;
    }

    private void OnDrawGizmosSelected()
    {
        Gizmos.DrawWireSphere(groundChecker.position,
groundCheckerRadius);
    }
}
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Random = UnityEngine.Random;

public class PlayerCombat : MonoBehaviour
{

    [Header("Combat")]
    [SerializeField] private int attackDamage;
    [SerializeField] private float attackRadius;
    [SerializeField] private float attackRate = 2f;
    private float nextAttackTime = 0f;
    [SerializeField] Transform attackPoint;
    [SerializeField] LayerMask enemyLayers;
    private AudioSource audioSource;
    [SerializeField] private AudioClip attack_1;
    [SerializeField] private AudioClip attack_2;
    [SerializeField] private AudioClip attack_3;
    [SerializeField] private AudioClip attack_4;

    private AudioClip[] attackSound;

    private Animator animator;

    private void Awake()
    {
        animator = GetComponent<Animator>();

        audioSource = GetComponent<AudioSource>();
        attackSound = new AudioClip[]
        {
            attack_1,
            attack_2,
            attack_3,
            attack_4
        };
    }

    private void Update()
    {
        if (Time.time >= nextAttackTime)
        {
            if (Input.GetKeyDown(KeyCode.Space))
            {
                PlayToAttack();
```

```
            nextAttackTime = Time.time + 1f / attackRate;
        }
    }
}

private void PlayToAttack()
{
    animator.SetTrigger("attack");
    PlayAttackSound();
}
public void GiveDamage()
{
    Collider2D[] hitEnemies = Physics2D.OverlapCircleAll(attackPoint.position, attackRadius, enemyLayers);

    foreach (Collider2D enemy in hitEnemies)
    {

enemy.GetComponent<Enemy>().TakeDamage(attackDamage);
    }
}

private void OnDrawGizmosSelected()
{
    Gizmos.DrawWireSphere(attackPoint.position, attackRadius);
}
public void PlayAttackSound()
{
    if (attackSound != null && attackSound.Length > 0)
    {
        int randomIndex = Random.Range(0, attackSound.Length);

        audioSource.clip = attackSound[randomIndex];
        audioSource.Play();
    }
}
}

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

public class SmoothCameraFollow : MonoBehaviour
{
    [SerializeField] private float followSpeed;
    [SerializeField] private float yOffset;
    [SerializeField] private Transform target;

    void FixedUpdate()
    {
        Vector3 newPos = new Vector3(target.position.x, target.position.y + yOffset, -10f);
        transform.position = Vector3.Slerp(transform.position, newPos, followSpeed * Time.deltaTime);
    }
}

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

public class EndGame : MonoBehaviour
{
    public int sceneBuildIndex;

    private bool hasLoaded = false;

    private void OnTriggerEnter2D(Collider2D collision)
    {
        print("Trigger Entered");

        if (!hasLoaded && collision.CompareTag("Player"))
        {
            hasLoaded = true;

            SceneManager.LoadScene("WinMenu");
        }
    }

}

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

public class GameManager : MonoBehaviour
{
    public GameObject gameOverUI;

    public void gameOver()
    {
        gameOverUI.SetActive(true);
    }

    public void restart()
    {
        string currentSceneName = SceneManager.GetActiveScene().name;
        SceneManager.LoadScene(currentSceneName);
    }

    public void QuitToMenu()
    {
        SceneManager.LoadScene("MENU");
    }
}

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

public class NextLevel : MonoBehaviour
{
    public int sceneBuildIndex;

    private bool hasLoaded = false;

    private void OnTriggerEnter2D(Collider2D collision)
    {
        print("Trigger Entered");

        if (!hasLoaded && collision.CompareTag("Player"))
        {
            hasLoaded = true;

            int nextSceneIndex = SceneManager.GetActiveScene().buildIndex + 1;

            if (nextSceneIndex < SceneManager.sceneCountInBuildSettings)
            {
                SceneManager.LoadScene(nextSceneIndex, LoadSceneMode.Single);
```

```
            }
            else
            {
                Debug.Log("No more scenes to load!");
            }
        }
    }

}

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

public class Trap : MonoBehaviour
{
    [SerializeField] private int damage;

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.tag == "Player" )
        {

collision.GetComponent<Health>().TakeDamage(damage);

        } else if (collision.tag == "Enemy"){

collision.GetComponent<Enemy>().TakeDamage(damage);
        }
    }
}

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

public class VolumePow : MonoBehaviour
{
    private AudioSource audioSrc;
    private float musicVolume = 1f;

    void Start()
    {
        audioSrc = GetComponent<AudioSource>();
    }
    void Update()
    {
        audioSrc.volume = musicVolume;
    }
    public void SetVolume(float vol)
    {
        musicVolume = vol;
    }
}
```