МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему: **«РОЗРОБКА 2D ПЛАТФОРМЕР ГРИ З ВИКОРИСТАННЯМ ІГРОВОГО РУШІЯ UNITY»**

Виконала: студентка 4 курсу, групи 6.1212-пі-1
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми програмна інженерія
(назва освітньої програми)
В.Д. Романова
(ініціали та прізвище)
Керівник доцент кафедри програмної інженерії, доцент, к.т.н. Лимаренко Ю.О.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)
Рецензент Директор ТОВ Голден-Тім Калін М.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2026

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти бакалавр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма програмна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2025 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТЦІ

Романовій Владі Дмитрівні
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка 2D платформер гри з використанням ігрового рушія Unity

керівник роботи Лимаренко Юлія Олексіївна, к.т.н., доцент
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 02 » грудня 2025 року № 1741-с

2. Строк подання студентом роботи 01.06.2026 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Визначення вимог до застосунку.
2. Проєктування проєкту.
3. Програмна реалізація проєкту.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою доповіді

6. Консультанти розділів роботи

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

7. Дата видачі завдання 09.12.2025 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	19.12.2025	
2.	Збір вихідних даних.	14.01.2026	
3.	Обробка методичних та теоретичних	16.02.2026	
	джерел.		
4.	Розробка першого та другого розділу.	01.04.2026	
5.	Розробка третього розділу.	18.05.2026	
6.	Оформлення та нормоконтроль	25.05.2026	
	кваліфікаційної роботи бакалавра.		
7.	Захист кваліфікаційної роботи.	16.06.2026	

Студент _____ В.Д. Романова
(підпис) (ініціали та прізвище)

Керівник роботи _____ Ю.О. Лимаренко
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота бакалавра «Розробка 2D платформер гри з використанням ігрового рушія Unity»: 47 с., 45 рис., 2 табл., 5 джерел.

ВІДЕОГРА, ГРА, КОМП'ЮТЕРНА ГРА, ПАТЕРНИ ПРОГРАМУВАННЯ, ПЛАТФОРМЕР, UNITY, VISUAL CODE.

Об'єкт дослідження – процес створення відеогри з використанням ігрового рушія Unity.

Предмет дослідження – Unity, Visual Code.

Мета роботи – розробка повноцінної гри з інтерактивним інтерфейсом користувача за допомогою Unity.

Метод дослідження – аналіз підходів сучасного ігрового дизайну, моделювання ігрової логіки з використанням рушія Unity.

У результаті розроблено ігровий застосунок, продемонстровано процес розробки програмного забезпечення, наведено діаграму прецедентів, сутностей та зв'язків.

SUMMARY

Bachelor’s qualifying paper “Development of a 2D Platformer Game using the Unity Game Engine”: 47 pages, 45 figures, 2 tables, 5 references.

COMPUTER GAME, GAME, OOP, PLATFORMER, PROGRAMMING PATTERNS, UNITY, VIDEO GAME, VISUAL STUDIO CODE.

The object of the study is the process of creating a video game using the Unity game engine.

The subject of research is Unity, Visual Code.

The goal of the work is to develop a full-fledged game with an interactive user interface using Unity.

The method of research is an analysis of modern game design approaches, modeling game logic using the Unity engine.

As a result, a game application is developed, the software development process is demonstrated, and a diagram of precedents, entities, and relationships is provided.

ЗМІСТ

Завдання на кваліфікаційну роботу ……..…….2
Реферат 4
Summary 5
Вступ..... 7
1 Визначення вимог до застосунку..... 9
1.1 Аналіз предметної області 9
1.2 Аналіз існуючих ігрових рушіїв..... 10
1.3 Формування функціональних вимог..... 12
1.4 Формування нефункціональних вимог..... 13
2 Проєктування проєкту 15
2.1 Проєктування архітектури 15
2.2 Використання патернів..... 17
3 Програмна реалізація застосунку 19
3.1 Програмна реалізація персонажа..... 19
3.2 Програмна реалізація інвентарю, предметів та карти..... 26
3.3 Вороги та пастки 34
3.4 Система кат-сцен..... 35
3.5 Функціональне тестування..... 37
3.6 Збірка застосунку 39
Висновки 46
Перелік посилань..... 47

ВСТУП

На сьогодні індустрія відеоігор є однією з найприбутковіших сфер в ІТ. Відеоігри – це комбінація технологій та креативності, заснована на принципах комп'ютерних наук [3]. Незважаючи на те що галузь з'явилася відносно нещодавно, вона вже займає одну з головних позицій. Це має просте пояснення: розвиток технологій, який стався в результаті появи інтернету. Зазвичай комп'ютерні ігри класифікуються як один із видів розваг, але варто зазначити, що це не єдине їхнє застосування. Сучасні навчальні ресурси використовують ігри для створення симуляцій для навчання спеціалістів у різних сферах.

Метою кваліфікаційної роботи є розробка повноцінної гри. Досягнення поставленої мети передбачає реалізацію таких завдань:

– проаналізувати предметну область;
– визначення архітектури;
– сформулювати вимоги до системи;
– розробити систему;
– провести тестування створеного проєкту.

Предмет дослідження – Unity.

Об'єкт дослідження – процес створення відеогри з використанням ігрового рушія Unity.

Методи дослідження – аналіз підходів сучасного ігрового дизайну, моделювання ігрової логіки з використанням рушія Unity.

Робота складається зі вступу, трьох розділів, висновку та переліку використаних джерел.

Перший розділ присвячений аналізу предметної області та використаного програмного забезпечення, наведенню вимог, які потрібно виконати для успішної реалізації проєкту. У другому розділі – детальний опис проєктування. Наведено діаграми та використано патерни. У третьому

розділі було описано процес реалізації, тестування та збірки додатка. Також наведено труднощі, які виникали на шляху, та те, як їх було усунуто.

1 ВИЗНАЧЕННЯ ВИМОГ ДО ЗАСТОСУНКУ

1.1 Аналіз предметної області

Платформер – це жанр ігор, у якому основна дія обертається навколо бігу та стрибків. Найкращим прикладом є Super Mario Bros, оскільки саме ця гра класифікувала «платформер» як жанр відеоігор. SMB мала такий монументальний успіх, що 2D-платформер став домінантним стилем відеоігор на роки і залишається актуальним сьогодні. Фактично, жанр став настільки розповсюдженим, що уявлення багатьох людей про те, що таке відеогра, по суті, схоже на платформер [1].

У грі такого жанру гравець рухається по рухомих або нерухомих платформах. У 2D-стилі персонаж зазвичай рухається зліва направо. На шляху гравець може зустріти різноманітні предмети, такі як павер-апи, зброя або предмети для відновлення здоров’я. Також присутні перешкоди у вигляді ворогів, пасток або головоломок.

Перші відеоігри такого жанру почали випускатися на початку 1980-х. Яскравими прикладами є Sonic the Hedgehog, Pitfall, Alex Kidd, Bonk, Donkey Kong Country та Kirby, і це лише деякі з них. Їхній успіх і дав поштовх до створення ігор такої категорії [2]. Через таку розповсюдженість поява піджанрів була лише питанням часу. Ці піджанри можна вважати суб’єктивними, оскільки кожна гра має свої унікальні механіки та може комбінувати в собі багато жанрів. Найбільш впізнавані піджанри наведено в таблиці 1.1.

Таблиця 1.1 – Піджанри платформерів

№	Назва	Піджанр	Опис	Представники
1	Метроїдванія	Екшн-платформер	Зосередженість на дослідженні світу та здобутті нових навичок	Hollow Knight, Nine Sols, Rain World

Продовження таблиці 1.1

№	Назва	Піджанр	Опис	Представники
2	Платформер «Бігай і стріляй»	Екшн-платформер	Головною дією є постійна стрілянина	Mega Man, Metroid.
3	Платформер Beat'Em Up	Екшн-платформер	Виділяється ближнім боєм	X-Men: Mutant Apocalypse, Astro Boy: Omega Factor
4	Мазокор	Влучний платформер	Несправедливі механіки, розіграш у формі гри	Kazio, Mario Maker
5	Комічна екшн-гра	Головоломка-платформер	Головоломки, відсутність бійок	Donkey Kong, Spanky's Quest, Penguin Land, The Lost Vikings, Lemmings
6	Ізометричний платформер	3D-платформер	Тривимірний світ	Super Mario 64, Spyro the Dragon, Prince of Persia

Таким чином, платформери займають важливе місце в індустрії. Вони забезпечують розваги для багатьох геймерів. Неможливо сказати, куди в майбутньому буде рухатися жанр, але можна сказати, що він залишиться в індустрії надовго.

1.2 Аналіз існуючих ігрових рушіїв

Зі зростанням популярності сфери відеоігор зросла і кількість програмного забезпечення для їх розробки. Розглянемо найбільш використовувані варіанти.

Unreal Engine – це універсальний ігровий рушій. На початку свого існування він був призначений для розробки ігор тільки від першої особи, але наступні оновлення дозволили розширити категорії створення. Він підтримує розробку не тільки для ПК, а й для телефонів, тому є багатофункціональним. Проте найбільш унікальною якістю цієї програми є те, що будь-який

задуманий проєкт можна створити без використання коду за допомогою візуального програмування [5]. Велика кількість документації дозволяє легко ознайомитися з програмою початківцям. Також найбільшою перевагою є вільна ліцензія, яка дозволяє безкоштовне використання. Взагалі, рушій можна використовувати не тільки для створення ігор, а й для візуального контенту, дизайну та графіки. Звичайно, як і всі рушії, він має свої недоліки. Для комфортної роботи з редактором комп'ютер повинен відповідати системним вимогам, таким як: не менше ніж 8 ГБ оперативної пам'яті, найновіша версія ОС, наявність однієї з найновіших відеокарт і процесорів тощо. Через велику кількість налаштувань Unreal Engine підходить для створення великих і комплексних проєктів у стилі 3D.

Godot – це рушій з відкритим кодом та доступом. Він не займає багато місця, на відміну від таких громіздких рушіїв, як Unreal Engine. Godot надає можливість створення ігор як у 2D-форматі, так і у 3D [2]. Проте 3D-можливості не такі просунуті, як у інших представлених рушіїв. Рушій має свою мову програмування, схожу за принципами на Python.

Unity – це найпопулярніший ігровий та кросплатформовий рушій. Його популярність відкриває доступ до дуже багатьох матеріалів для вивчення початківцям. Програма підтримує майже всі загальнодоступні ОС та платформи, на яких можна запустити ігри, такі як: Windows, Linux, MacOS, Android, iOS, PlayStation, Xbox, Nintendo Switch і т. д. [4]. Також важливо зауважити, що Unity має одну з найкращих оптимізацій серед інших рушіїв, що позитивно позначається на швидкості та якості роботи над проєктами. Ще однією з переваг можна назвати магазин матеріалів, який надає доступ до вже готових ресурсів, таких як скрипти, спрайти та інше. Unity має 3 ліцензії: дві з яких для комерційного використання створеного проєкту, а третя – для персонального або некомерційного використання.

Отже, зважаючи на всі плюси й мінуси, можна сказати, що з представлених та проаналізованих ігрових рушіїв вибір був зроблений правильно.

1.3 Формування функціональних вимог

Формування вимог – найважливіша стадія розробки програмного забезпечення. Вона допомагає створити бачення функціонала, який надалі буде реалізовано.

На рисунку 1.1 наведено, що входить до функціональних вимог.

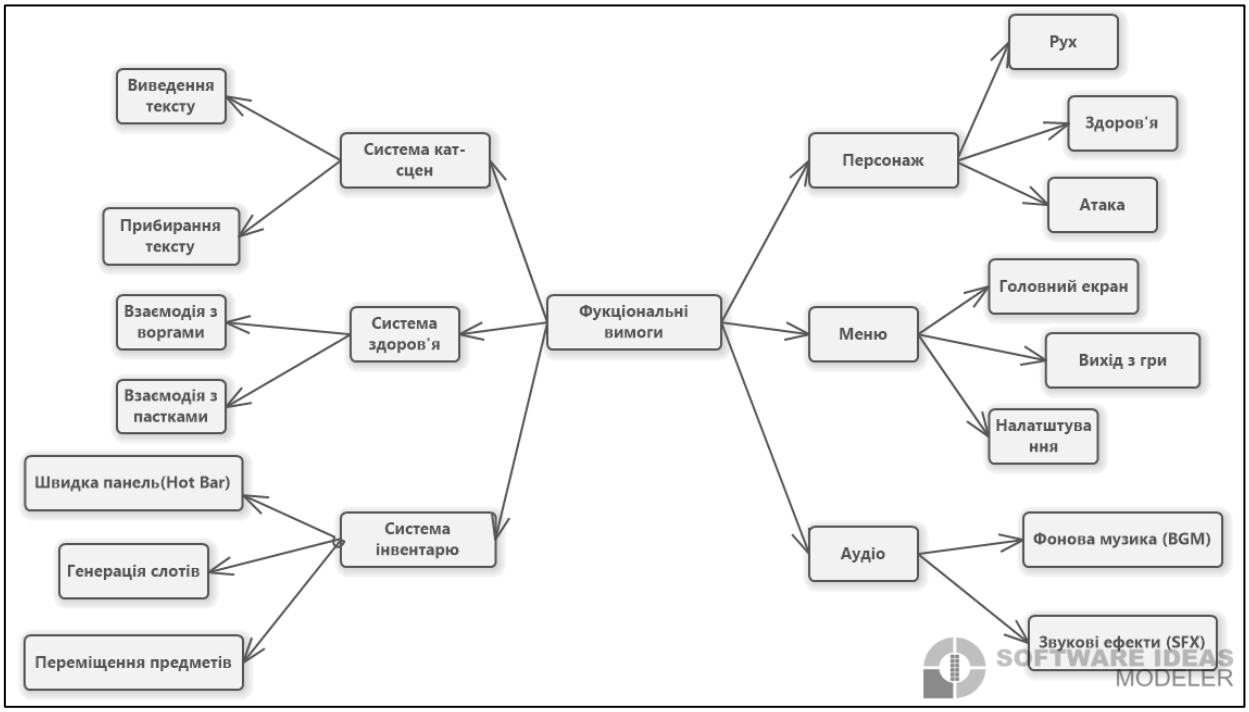


Рисунок 1.1 – Функціональні вимоги до застосунку

Система кат-сцен:

– система забезпечує відображення тексту;
– користувач має можливість прибрати текст з екрана.

Система здоров'я:

– користувач має можливість взаємодіяти з пастками, які завдають шкоди або посилюють стрибок.

Система інвентарю:

– система при запуску гри динамічно створює слоти, в яких будуть зберігатися предмети;
– система надає можливість переміщувати предмети по слотах;

– система дає можливість швидко використовувати предмети за допомогою швидкої панелі.

Персонаж:

– система дозволяє користувачу пересувати персонажа в правий або лівий бік, дозволяє стрибати та атакувати.

Меню:

– система містить головне меню, з якого користувач може запустити гру або вийти з неї;
– система надає можливість користувачу налаштовувати гучність фонової музики або ефектів.

Аудіо:

– система повинна мати фонову музику для створення атмосфери та занурення у гру;
– система повинна мати ефекти при отриманні персонажем шкоди та отриманні посилення.

Функціональні вимоги є основними, тому мають найвищий пріоритет для виконання та завершення проєкту.

1.4 Формування нефункціональних вимог

Функціональні вимоги визначають, що повинна робити система. На відміну від них, нефункціональні вимоги визначають, якою повинна бути система для забезпечення високої якості кінцевого продукту.

Детальний опис нефункціональних вимог наведено нижче на рисунку 1.2.

Зручність використання: застосунок повинен мати інтуїтивно зрозумілий та легкий інтерфейс. Інтерфейс не повинен бути перевантажений додатковими і непотрібними елементами.

Сумісність: програма повинна безперебійно працювати на ОС Windows 10 та 11.

Швидкість: застосунок має швидко запускатися.

Продуктивність: швидкі переходи між кімнатами без затримок.

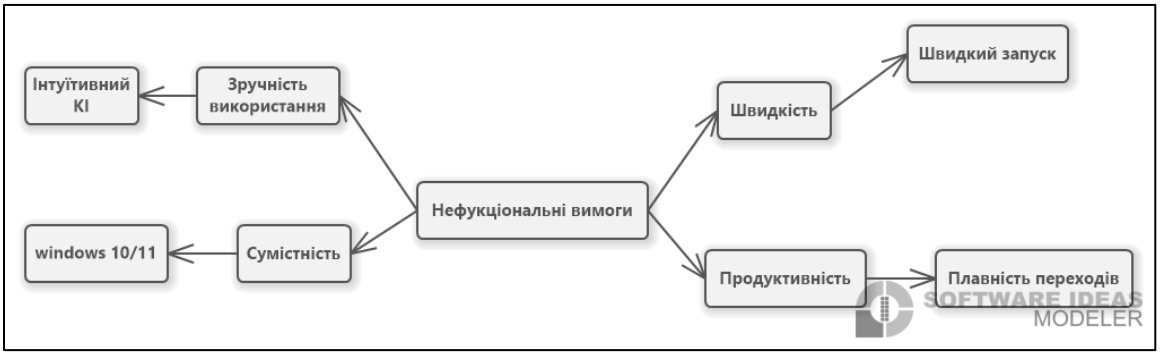


Рисунок 1.2 – Нефункціональні вимоги до застосунку

Таким чином було проаналізовано програмне забезпечення та сформовано вимоги. Отже, можемо переходити до реалізації проєкту.

2 ПРОЄКТУВАННЯ ПРОЄКТУ

2.1 Проєктування архітектури

Структура розробленого додатка базується на компонентно-орієнтованій архітектурі (CBA). Для ігрового рушія Unity така архітектура є природною. Структуру гри замість монолітного коду було розбито на окремі логічні рівні.

Завдяки такому рішенню було спрощено процес налагодження та виправлення помилок. Структурно додаток можна розділити на такі рівні:

– фізичний рівень;
– рівень користувацького інтерфейсу;
– рівень глобальних сервісів;
– рівень сценарної логіки.

Фізичний рівень відповідає за базову ігрову логіку. До базової логіки входять: контроль гравця (рух, атака, здоров’я), інтелект ворога, ієрархія об’єктів.

Рівень користувацького інтерфейсу відповідає за зворотний зв’язок із гравцем. До шару входять: підсистема інвентарю, динамічна карта та система візуальних сповіщень.

Рівень глобальних сервісів відповідає за системи, які існують в єдиному екземплярі, але мають глобальний доступ із будь-якоЇ точки програми. До таких систем входять аудіосистема та система переходів між кімнатами.

Останній рівень сценарної логіки побудований на успадкуванні та абстракції. Цей шар дозволяє будувати складні постановки або виводити текст для допомоги чи пояснення механік гравцеві.

На рисунку 2.1 зображено UML-діаграму прецедентів, яка показує можливості користувача в розробленому додатку. Головним і єдиним

актором у системі є Користувач (Гравець), який є ініціатором усіх процесів. Тобто майже все, що може робити гравець, те і повинен робити персонаж.

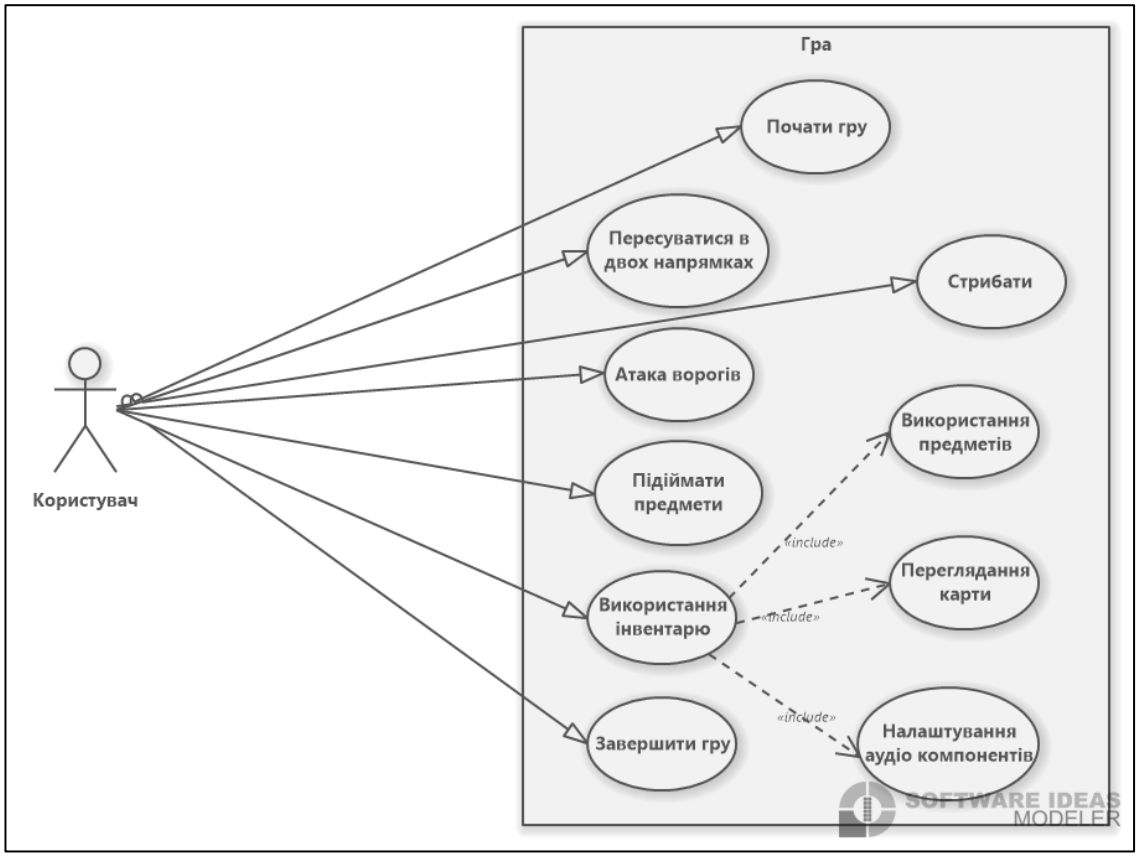


Рисунок 2.1 – Діаграма варіантів використання

Після запуску гри актор «Користувач» може взаємодіяти із системою. Перше, що побачить користувач, здійснивши вхід у систему, – це головний екран, на якому зможе перейти до першого рівня або вийти з гри.

Якщо користувач обере «Почати гру», то перейде на наступний екран та отримає доступ до керування персонажем, тобто пересування, стрибків і т. ін.

Після цього актор «Користувач» має можливість взаємодіяти з інвентарем. Це передбачає використання вже підібраних предметів, перегляд карти (тобто свого місцезнаходження) та налаштування аудіокомпонентів, що означає можливість коригування гучності аудіо.

Отже, після завершення концептуального рівня можна переходити до програмної реалізації.

2.2 Використання патернів

У межах розробленого програмного продукту було використано такі архітектурні патерни:

– одинак;
– спостерігач;
– шаблонний метод та поліморфізм;
– команда.

Патерн «Одинак» використано для створення глобальних сервісів, таких як менеджери звукового супроводу (SoundManager, MusicManager), контролер сповіщень (ItemPickUpUi) та менеджер динамічної карти (MapController_Dynamic). Патерн забезпечує те, що ці критично важливі підсистеми існують у строго єдиному екземплярі протягом усього ігрового сеансу. Це надає іншим модулям безпечну глобальну точку доступу без необхідності виконувати ресурсомісткий пошук об’єктів на сцені.

Шаблон «Спостерігач» застосовано для інтеграції користувацького інтерфейсу з ігровою логікою. Наприклад, взаємодія з налаштуванням гучності реалізована через делегати та підписки на події (AddListener). Це дозволяє системі реагувати на дії користувача виключно в момент їх виникнення, повністю уникаючи неефективної перевірки станів.

Шаблонний метод та поліморфізм виступають фундаментом підсистеми катсцен та ієрархії ігрових предметів. Базові абстрактні класи CutsceneBase та Item визначають загальну основу через віртуальні методи Execute та UseItem, дозволяючи нащадкам (наприклад, Cutscene_Text) перевизначати конкретну поведінку. Такий підхід гарантує дотримання принципу відкритості/закритості (Open/Closed Principle) – система відкрита для додавання нових предметів чи сцен, але закрита для модифікації базового ядра.

Компонентний патерн дозволив уникнути проблеми успадкування. Ігрові сутності (головний персонаж, вороги) формуються шляхом композиції

невеликих незалежних скриптів-компонентів. Це забезпечує максимальну ізоляцію логіки (відокремлення фізики пересування від управління здоров'ям або інвентарем) та високий рівень повторного використання коду.

Патерн «Команда» було використано для реалізації руху гравця. Замість хардкоду, система генерує виклик методу. Таке розмежування дозволяє реалізувати підтримку різних пристроїв введення. Детальну структуру патерну можна переглянути на рисунку 2.2.

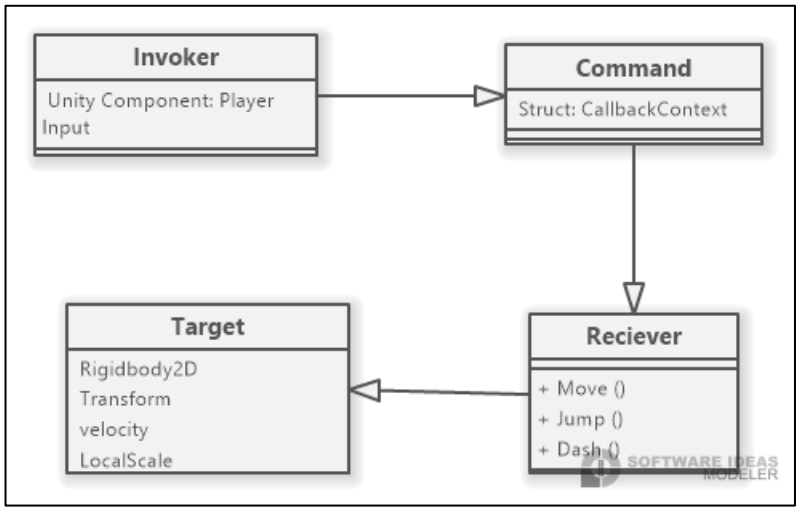


Рисунок 2.2 – UML-діаграма використаного патерну Команда

Використання зазначених патернів проєктування дозволило створити чисту архітектуру додатка. Такий підхід мінімізує ризик виникнення помилок та створює можливість для розширення функціоналу гри у майбутньому.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ЗАСТОСУНКУ

3.1 Програмна реалізація персонажа

Першим етапом реалізації буде додавання Rigidbody2D. Цей компонент приєднується до будь-якого ігрового об’єкта, додає фізику і дозволяє рухатися тільки по осях XY [4]. З багатьох налаштувань цього елемента в Unity одним із головних є тип тіла Rigidbody2D, який визначає поведінку руху та взаємодію з колайдером.

Динамічний тип є найбільш інтерактивним серед інших. Кінематичний тип передбачає виключно чітке керування з боку користувача. На відміну від динамічного тіла, кінематичне не піддається звичайній фізиці, такій як тяжіння. Воно все ще рухається зі своєю швидкістю, але на це не впливають інші сили. Статичний тип не рухається під час симуляції. Загалом він використовується тільки для нерухомих об’єктів.

Проаналізувавши всі типи, вибір було зроблено на користь динамічного, оскільки у вимогах було зазначено можливість взаємодії між об’єктами. Зазвичай компонент Rigidbody2D поєднують з Collider2D, який визначає форму для зіткнень об’єктів і їхньої взаємодії між собою.

Наступним етапом є додавання компонента Player Input. За допомогою цього компонента було визначено налаштування того, як дії відповідають викликам у коді. Після додавання компонента в скрипті тепер є можливість використання вбудованої функції InputAction.CallbackContext, яка дозволяє стежити за станом дії та зворотним викликом.

Після додавання всіх необхідних елементів можна почати написання коду. Для реалізації пересування персонажа було використано поведінковий патерн «Команда» для оптимізації та підвищення продуктивності коду. Такий вибір дозволяє перетворити запити на об’єкти, що зручно для складання команд в чергу. Ініціатором є Player Input, який стежить за

натисканням кнопок і після цього створює об’єкт. Об’єктом виступає функція InputAction.CallbackContext. На рисунку 3.1 наведено приклад використання методу.

```
public void Move(InputAction.CallbackContext context)
{
   if (PauseController.IsGamePaused)
   {
      horizontalMovement = 0;
      return;
   }

   horizontalMovement = context.ReadValue<Vector2>().x;
}
```

Рисунок 3.1 – Приклад використання функції InputAction.CallbackContext

У коді метод був застосований для перевірки натискання кнопки користувачем. Потік виконання буде наведено нижче у таблиці 3.1.

Таблиця 3.1 – Потік виконання функції Move

№	**Опис**
1	Гравець тисне на кнопку на клавіатуру
2	Player Input фіксує та створює об’єкт
3	Створюється команда Callback context
4	Команда передається у метод Move
5	Скрипт PlayerMovement виконує пересування в потрібний бік

Такий метод використовувався для створення інших функцій стосовних до можливостей користувача. Наприклад, функція Dash, яка дозволяє користувачу робити ривок, що переміщує гравця вперед. Функцію Dash можна подивитися на рисунку 3.2.

```
public void Dash(InputAction.CallbackContext context)
    if (context.performed && canDash)
    {
        StartCoroutine(DashCoroutine());
    }}
```

Рисунок 3.2 – Функція Dash

Основний функціонал ривку реалізовано у функції DashCoroutine (див. рис. 3.3).

```
public IEnumerator DashCoroutine()
{
   Physics2D.IgnoreLayerCollision(7, 8, true);
    canDash = false;
    isDashing = true;
    float dashDirection = isFacingRight ? 1f : -1f;
    rb.linearVelocity = new Vector2(dashDirection * dashSpeed, rb.linearVelocity.y);
    yield return new WaitForSeconds(dashDuration);
    rb.linearVelocity = new Vector2(0f, rb.linearVelocity.y);
    isDashing = false;
    Physics2D.IgnoreLayerCollision(7, 8, false);
    yield return new WaitForSeconds(dashCooldown);
    canDash = true;
}
```

Рисунок 3.3 – Функція DashCouroutine

У реалізації було використано метод Coroutine, який може призупиняти та відновлювати виконання, коли потрібно. У випадку рушія Unity, такий метод може починати виконуватися в одному кадрі, а потім відновлюватися в іншому, що дозволяє розподіляти завдання на кільки кадрів [4]. У межах застосунку використання Coroutine є оптимальним архітектурним рішенням, оскільки у функціоналі потрібні таймери для використання можливостей без блокування основного виконання гри.

У функції DashCouroutine є виклик методу IgnoreLayerCollision, який тимчасово вимикає фізичні зіткнення між гравцем та ворогом. Це надає персонажу стан невразливості і дозволяє проходити крізь ворогів, не

втрачаючи здоров’я під час ривка. Також під час активації ривка функція вмикає візуальний ефект trailRenderer, що дає гравцю знати, що ривок було виконано успішно.

Наступним кроком є додавання системи здоров’я. Саме система здоров’я – один із головних ключів для взаємодії з ворогами. У програмі користувач. Ключовим компонентом у реалізації був PlayerHealth, що відповідає за управління всією системою. До основних завдань системи входять отримання шкоди після зіткнення з ворогом та відновлення здоров’я після смерті.

Для реалізації отримання шкоди було використано функцію OnTriggerEnter2D (див. рис. 3.4).

```
private void OnTriggerEnter2D(Collider2D collision)
   {
      Enemy enemy = collision.GetComponent<Enemy>();
      if(enemy)
      {
         TakeDamage(enemy.damage);
      }
      Trap trap = collision.GetComponent<Trap>();
      if(trap && trap.damage > 0)
      {
         TakeDamage(trap.damage);
      }}
```

Рисунок 3.4 – Функція OnTriggerEnter2D

Ця функція базується на подійно-орієнтованій моделі рушія Unity. Коли межі колайдерів гравця та ворога перетинаються, рушій генерує програмну подію [4]. У програмі відбувається перевірка на зіткнення з ворогом або з паскою, і після цього викликається функція TakeDamage, яка віднімає здоров’я.

Аналогічно до ривка було зроблено візуальний індикатор, який інформує користувача про надходження шкоди. Метод не був дуже складним у реалізації. В ньому відбувається зміна стану: змінюється колір спрайта на червоний (див. рис. 3.5).

```
private IEnumerator FlashRed()
{
    spriteRenderer.color = Color.red;
    yield return new WaitForSeconds(0.2f);
    spriteRenderer.color = Color.white;
}
```

Рисунок 3.5 – Функція FlashRed

Потім метод робить затримку, яка триває 0,2 секунди ігрового часу. І тільки після цього гравець відновлює початковий стан, колір спрайту повертається до оригінального .

Очевидно, наступним етапом після логіки про отримання шкоди буде створення логіки для смерті. У реалізації було використано патерн Спостерігач. Патерн було використано для створення логіки смерті персонажа. Роль підписника виконувала функція Start(), яка підписувалася на події GameController.OnReset та HealthItem.OnHealthCollect (див. рис. 3.6).

```
void Start()
{
    ResetHealth();

    spriteRenderer = GetComponent<SpriteRenderer>();
    GameController.OnReset += ResetHealth;
    HealthItem.OnHealthCollect += Heal;
}
```

Рисунок 3.6 – Приклад застосування патерну «Спостерігач»

Метод ResetHealth автоматично відновлює здоров’я користувачу після смерті. У цьому випадку було використано синхронізацію з UI. Скрипт гравця взаємодіє з інтерфейсом об’єкта HealthUI. Таким чином, коли користувач помирає і хоче спробувати знову, викликається функція SetMaxHeart, яка повертає здоров’я до максимального встановленого значення (див. рис. 3.7).

```
void ResetHealth()
{
    currentHealth = maxHealth;
    healthUI.SetMaxHearts(maxHealth);
}
```

Рисунок 3.7 – Функція ResetHealth

Завершальним етапом розробки гравця було створення механіки атаки. Для забезпечення чистоти коду архітектуру було розділено на два скрипти: PlayerAttack та AttackArea. PlayerAttack є контролером атаки, що визначає її тривалість та відповідає за управління анімацією (див. рис. 3.8 – 3.9). На відміну від попередньої подійно-орієнтованої системи, тут застосовується синхронізація з клавіатурою. Такий вибір був зумовлений мінімальною затримкою та швидкістю відповіді програми на натискання клавіші.

```
void Start()
  {
    attackArea = transform.GetChild(0).gameObject;
  }
```

Рисунок 3.8 – Фрагмент коду з скрипту PlayerAttack

```
void Update()
  {
    if(Input.GetKeyDown(KeyCode.X))
    {
      Attack();
    }

    if (attacking)
    {
      timer += Time.deltaTime;

      if (timer > timeAttack)
      {
        timer = 0f;
        attacking = false;
        attackArea.SetActive(attacking);
      }
    }
  }
```

Рисунок 3.9 – Фрагмент коду з скрипту PlayerAttack

Клас AttackArea використовується для візуальної відповіді в момент удару. Якщо ворога знайдено, то клас викликає метод TakeDamage для ворога (див. рис. 3.10).

```
using UnityEngine;

public class AtackArea : MonoBehaviour
{
    private int damage = 1;

    private void OnTriggerEnter2D(Collider2D collider)
    {
        Enemy enemy = collider.GetComponent<Enemy>();
        if (enemy)
        {
            enemy.TakeDamage(damage);
        }
    }
}
```

Рисунок 3.10 – Клас AttackArea

У результаті розробки було отримано ігрового персонажа, який відповідає сучасним стандартам та принципам об’єктно-орієнтованого програмування (ООП). Замість написання всіх механік коду в одному файлі, що є антипатерном, їх було розділено. Така архітектура створена на основі компонентного підходу. Повний список усіх доданих елементів до об’єкта гравця наведений нижче на рисунку 3.11.

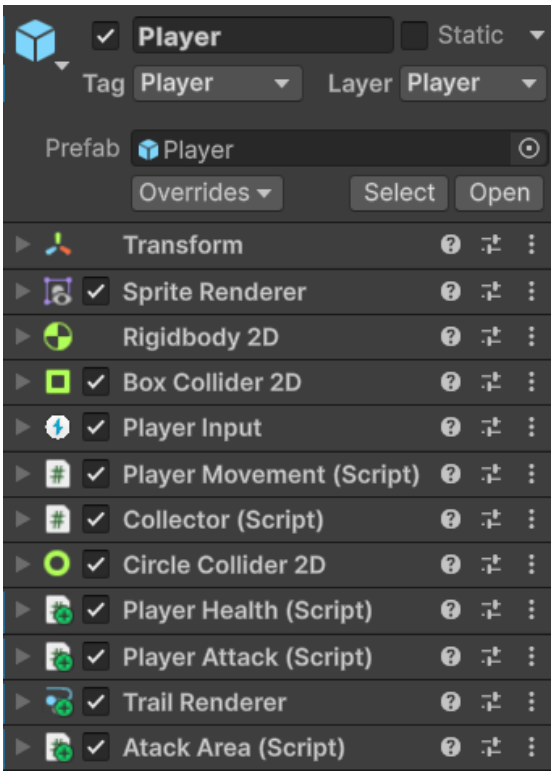


Рисунок 3.11 – Список елементів головного героя

Підсумовуючи, можна сказати, що персонаж відповідає визначеним вимогам, відзначається чутливістю до взаємодії з користувачем та стабільною роботою.

3.2 Програмна реалізація інвентарю, предметів та карти

Основним засобом взаємодії користувача з програмою є система інвентарю. В ньому передбачено багато функцій, таких як генерація слотів, створення динамічної карти та налаштування аудіокомпонентів. Ієрархію об’єктів інтерфейсу наведено нижче (див. рис. 3.12).

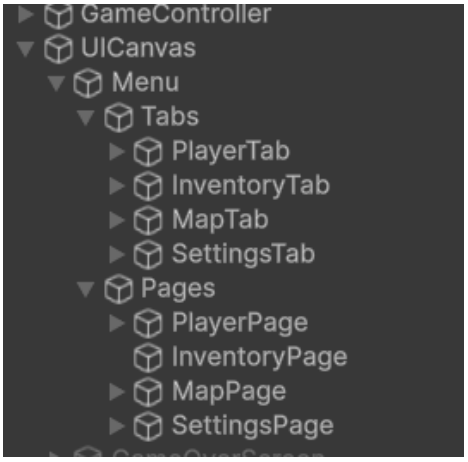


Рисунок 3.12 – Ієрархія системи інвентарю

Для виконання визначених вимог було створено 4 сторінки, які виконують відповідні завдання. PlayerPage сфокусована на наданні користувачу додаткову інформацію про механіки (див. рис. 3.13).

InventoryPage відповідає за генерацію слотів, де будуть зберігатися предмети. Цей об’єкт є одним із найбільш вживаних у проєкті, тому від його роботи буде залежати приємність ігрового процесу. На цей елемент призначено компонент компонування Grid Layout Group, який розподіляє слоти та вирівнює комірки. Створення слотів відбувається динамічно і тільки під час запуску гри. Фрагмент коду наведено на рисунку 3.14.



Рисунок 3.13 – Вигляд допоміжної сторінки користувача

```
void Start()
    {
        for (int i = 0; i < slotCount; i++)
        {
            Slot slot = Instantiate(slotPrefab, inventoryPanel.transform).GetComponent<Slot>();
            if (i < itemPrefab.Length && itemPrefab[i] != null)
            {
                GameObject item = Instantiate(itemPrefab[i], slot.transform);
                item.GetComponent<RectTransform>().anchoredPosition = Vector2.zero;
                slot.currentItem = item;

                if (item.GetComponent<ItemHandler>() == null)
item.AddComponent<ItemHandler>();
                if (item.GetComponent<CanvasGroup>() == null)
item.AddComponent<CanvasGroup>();
            }
        }
    }
```

Рисунок 3.14 – Фрагмент коду з скрипту InventoryController

Як тільки користувач підбирає предмет, система автоматично починає пошук вільного місця для розміщення об’єкта. Наступним кроком системи є створення клону підібраного елемента, який перетворюється з фізичного на інтерфейсний. Така механіка потрібна для використання предмета як в інвентарі, так і поза його межами.

Оскільки закриваючи інвентар, користувач не бачить своїх предметів і не може їх швидко використовувати, було прийнято рішення створити панель швидкого доступу. Користувач має змогу перетягнути предмет із слота інвентарю в комірку панелі. Після цього гравець при натисканні кнопки, що відповідає номеру слота, зможе використовувати предмет. Така можливість запрограмована у функції UseItemInSlot (див. рис. 3.15).

```
void UseItemInSlot(int index)
{
    Slot slot = hotBarPanel.transform.GetChild(index).GetComponent<Slot>();
    if(slot != null && slot.currentItem != null)
    {
        Item item = slot.currentItem.GetComponent<Item>();

        if (item != null)
        {
            bool isConsumed = item.UseItem();

            if (isConsumed)
            {
                Destroy(slot.currentItem);
                slot.currentItem = null;
            }
        }
    }
}
```

Рисунок 3.15 – Функція UseItemInSlot

Першим ділом метод отримає доступ до комірки в системі. І тільки після підтвердження наявності предмету у тому місці скрипт отримує доступ до компонента Item і його функції UseItem, яка дозволяє використати об’єкт.

У грі передбачено використання двох предметів або павер-апів: HealthItem, який відновлює здоров’я, та SpeedItem, що надає користувачу тимчасове збільшення швидкості. Діаграма зв’язків між головним та залежними класами наведена нижче (див. рис. 3.16).

У діаграмі можна побачити, що обидва об’єкти здоров’я та швидкості пов’язані з Item зв’язком наслідування. Метод UseItem в об’єктах неведено нижче (див. рис. 3.17 – 3.18).

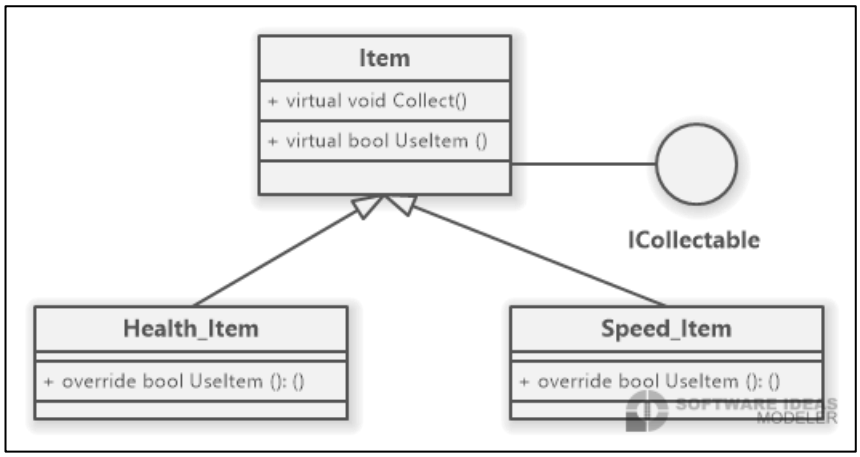


Рисунок 3.16 – Зв’язок між класами предметів

```
using UnityEngine;
using System;

public class HealthItem : Item
{
    public int healAmount = 1;
    public static event Action<int> OnHealthCollect;

    public override bool UseItem()
    {
        base.UseItem();
        OnHealthCollect?.Invoke(healAmount);
        return true;
    }
}
```

Рисунок 3.17 – Використання функції у HealthItem

```
public class SpeedItem : Item
{
    public static event Action<float> OnSpeedCollected;
    public float speedMultiplier = 1.5f;

    public override bool UseItem()
    {
        base.UseItem();
        OnSpeedCollected?.Invoke(speedMultiplier);
        return true;
    }
}
```

Рисунок 3.18 – Використання функції у SpeedItem

Предмет HealthItem додає 1 одиницю здоров’я, максимальна кількість – 5. Предмет SpeedItem пришвидшує користувача на 1.5.

Наступна сторінка – MapPage, яка фокусується на візуалізації карти. Карта – важливий елемент навігації у грі з відкритим світом. Замість статичного зображення, як і у комірках, реалізовано генерацію динамічної мапи під час запуску гри, залежно від розташування та розмірів кімнат у самому застосунку.

При написанні скрипту було використано патерн «Одинак», який надає іншим механікам доступ до оновлення карти (див. рис. 3.19).

```
private void CreateAreaUI(PolygonCollider2D area, bool isCurrent)
   {
      GameObject areaImage = Instantiate(areaPrefab, mapParent);
      RectTransform rectTransform = areaImage.GetComponent<RectTransform>();
      Bounds bounds = area.bounds;
      rectTransform.sizeDelta = new Vector2(bounds.size.x * mapScale, bounds.size.y *
mapScale);
      rectTransform.anchoredPosition = bounds.center * mapScale;
      areaImage.GetComponent<Image>().color = isCurrent ? currentAreaColor :
defaultColour;
      uiAreas[area.name] = rectTransform;
   }
```

Рисунок 3.19 – Фрагмент коду з скрипту MapController

Спершу скрипт знаходить і аналізує об’єкти, з яких буде створювати мапу, тобто кімнати. Головною проблемою у реалізації були перенесення межі кімнат та їх розмір. Застосовуючи масштабування (mapScale), метод перетворює ігрові метрики на пікселі у мапу. Також у програмі було задумано переходи між кімнатами, які зупиняють рух гравця на деякий час задля уникнення багів. У іграх жанру платформер перехід із кімнати в кімнату – звичайна практика. Як тільки колайдер гравця і один з колайдерів кімнат стикаються, скрипт звертається до Instance у MapController через патерн Одинак і переносить гравця. Користувач втрачає можливість руху на декілька секунд, поки персонаж переноситься на вказану відстань, на якій користувач зможе піти далі (див. рис. 3.20).

```
private void UpdatePlayerPosition(GameObject player)
{
    Vector3 newPos = player.transform.position;

    switch(direction)
    {
        case Direction.Up:
            newPos.y += additivePos;
            break;
        case Direction.Down:
            newPos.y -= additivePos;
            break;
        case Direction.Right:
            newPos.x += additivePos;
            break;
        case Direction.Left:
            newPos.x -= additivePos;
            break;
    }
    player.transform.position = newPos;
}
```

Рисунок 3.20 – Функція UpdatePlayerPosition

Користувача можна зміщувати як по осі X, так і по осі Y. Такий метод дозволяє створювати переходи зліва направо та зверху вниз. Саме це дозволяє створення цікавих і різноманітних рівнів.

Після успішного переходу між кімнатами, якщо користувач відкриє карту ще раз, він побачить, що герой перемістився. Колір попередньої кімнати зміниться на сірий, а поточної – на зелений, і іконка персонажа перенесеться автоматично. UpdateCurrentArea: функція, що реалізує оновлення поточної області (див. рис. 3.21).

```
public void UpdateCurrentArea(string newCurrentArea)
{
    foreach(KeyValuePair<string, RectTransform> area in uiAreas)
    {
        area.Value.GetComponent<Image>().color = area.Key == newCurrentArea ? currentAreaColor : defaultColour;
    }

    MovePlayerIcon(newCurrentArea);
}
```

Рисунок 3.21 – Реалізація функцій UpdateCurrentArea

Цикл foreach перебирає словник за парами ключ-значення. Потім замість використання конструкції if-else код був написаний в один рядок.

Система порівнює ключ із назвою поточної кімнати: якщо відбувся збіг, то застосовується колір активної зони. Метод MovePlayerIcon переміщує іконку гравця на мапі. Замість звернення до словника ще раз, для скорочення операцій було використано метод TryGetValue. У разі передавання неіснуючої кімнати метод поверне false, і подальший код перевірки не викликатиметься.

Фінальним кроком для розробки системи інвентарю стало додавання аудіосупроводу та можливість його налаштування. Музика дуже сильно впливає на занурення гравця в ігровий процес. Для безперебійної та швидкої роботи системи архітектуру було розділено на три компоненти: бібліотека, в якій зберігаються ефекти SoundLibrary, менеджер програвання ефектів SoundManager та менеджер фонової музики MusicManager.

Soundlibrary за видачу та зберігання усіх використаних аудіо-ефектів в масивах. У бібліотеці було реалізовано метод GetRandom, який замість відтворення одного й того самого ефекту вибирає з бібліотеки випадковий аудіокліп (див. рис. 3.22).

```
public AudioClip GetRandom(string name)
{
    if(soundDictionary.ContainsKey(name))
    {
        List<AudioClip> audioClips = soundDictionary[name];
        if(audioClips.Count > 0)
        {
            return audioClips[UnityEngine.Random.Range(0,audioClips.Count)];
        }
    }
    return null;
}
```

Рисунок 3.22 – Реалізація методу GetRandom

SoundManager відповідає за відтворення ефектів, беручи їх з масиву. У методі Play використовується функція PlayOneShot, яка дозволяє накладати звуки один на одного (див. рис. 3.23).

Вигляд створених ефектів наведено нижче на рисунку 3.24.

```
public static void Play(string soundName)
{
    AudioClip audioClip = soundLibrary.GetRandom(soundName);
    if(audioClip != null)
    {
        audioSource.PlayOneShot(audioClip);
    }
}
```

Рисунок 3.23 – Реалізація методу Play

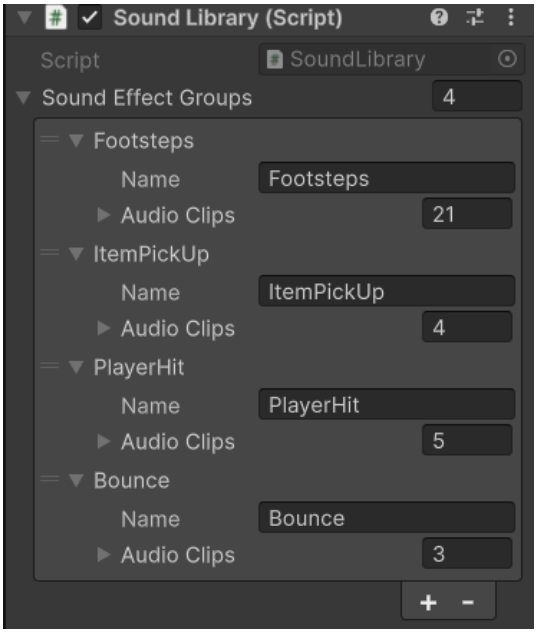


Рисунок 3.24 – Перелік ефектів

На відміну від ефектів, фонова музика довша та не використовується з такою великою кількістю. Під час ігрового процесу буде програватися тільки одна музична доріжка, яка буде програватися нескінченно. Таку механіку забезпечує метод PlayBackgroundMusic (див. рис. 3.25).

```
public static void PlayBackgroundMusic(bool resetSong, AudioClip audioClip = null)
  {
    if(audioClip != null){
      Instance.audioSource.clip = audioClip;
    }
    if(Instance.audioSource.clip != null)
    {
      if(resetSong)
      {
        Instance.audioSource.Stop();}
      Instance.audioSource.Play();}}
```

Рисунок 3.25 – Метод PlayBackgroundMusic

Якщо при запуску метод отримує команду перезапуску, то музика продовжує програватися.

Користувачу у свою чергу надається можливість налаштовувати гучність до власних бажань. Гравець може налаштовувати гучність за допомогою повзунків.

Підбиваючи підсумки, можна сказати, що створена система успішно виконує визначені вимоги. Розроблена система не використовує антипатерни та має стійку архітектуру. Така архітектура дозволяє розширити функціонал при потребі.

3.3 Вороги та пастки

Ворог не використовує складних алгоритмів для пошуку шляху до гравця. Такий вибір був зроблений, оскільки для платформера більш ефективним та оптимізованим рішенням буде використання логіки патрулювання. Ворог переслідує гравця, долає перешкоди та після смерті залишає предмет.

Ворог самостійно може приймати рішення, у методі Update щосекундно відбувається перевірка на зіткнення. Перевірка відбувається на знаходження на землі, першкод перед ним таких як стіна або прірва. Рух противника спрямований у бік гравця. Для обчислення шляху використовується функція Mathf.Sign, яка повертає 1 або -1 від різниці координат між гравцем і ворогом (див. рис. 3.26).

```
void Update()
   {
      isGrounded = Physics2D.Raycast(transform.position, Vector2.down, 1f, groundLayer);

      float direction = Mathf.Sign(player.position.x - transform.position.x);

        bool isPlayerAbove = Physics2D.Raycast(transform.position, Vector2.up, 5f, 1 << player.gameObject.layer);
```

```
    if(isGrounded)
    {
       rb.linearVelocity = new Vector2 (direction * chaseSpeed, rb.linearVelocity.y);

            RaycastHit2D groundInFront = Physics2D.Raycast(transform.position, new
Vector2(direction, 0), 2f, groundLayer);

             RaycastHit2D gapAhead = Physics2D.Raycast(transform.position + new
Vector3(direction, 0 , 0), Vector2.down, 2f, groundLayer);

       RaycastHit2D platformAbove = Physics2D.Raycast(transform.position, Vector2.up, 5f,
groundLayer);

       if(!groundInFront.collider && !gapAhead.collider)
       {
          shouldJump = true;
       }
       else if (isPlayerAbove && platformAbove.collider)
       {
          shouldJump = true;
       }
    }
  }
```

Рисунок 3.26 – Реалізація переслідування користувача

Система смерті була реалізована схожим чином на гравця. Різниця полягає в тому, що після своєї смерті об’єкт ворога видаляється та ворог залишає після себе предмет.

3.4 Система кат-сцен

У розробленому проєкті кат-сцени слугують помічником для користувача, щоб спростити навігацію та надати інформацію про можливості. Центром в цій системі буде об’єкт CutsceneBase, який можна буде дублювати при необхідності ще однієї кат-сцени. Він має ключові елементи, необхідні для його роботи. Детальну діаграму зв’язків між компонентами можна розглянути на малюнку 3.27.

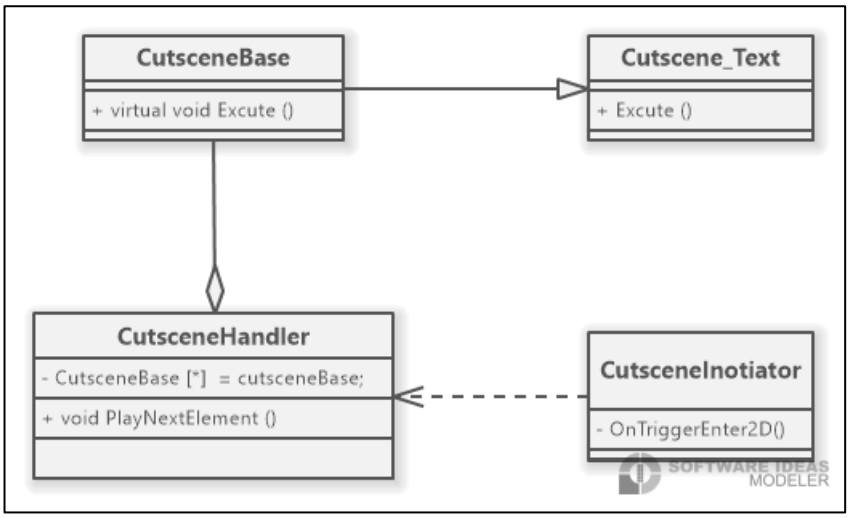


Рисунок 3.27 – UML Діаграма зв’язків між компонентами кат-сцени

Базовий клас CutsceneBase містить віртуальний метод Excute, який буде змінюватися залежно від потреб сцени. Від цього класу через зв’язок успадкування проходить реалізація у класі Cutscene_Text. Cutscene_Text перевизначає метод для виведення тексту.

За управління сценою відповідає клас CutsceneHandler, який пов’язаний з базовим класом агрегацією, так як зберігає всередині масиви об’єктів базового класу та за допомогою функції PlayNextElement перебирає їх. Цей клас обробки є незалежним від реалізації тільки одного типу кат-сцен, що дозволяє додавати нові типи подій за потреби. Запуск виконання виконується компонентом CutsceneInitiator, який слідкує за зіткненням гравця з тригером, маючи зв’язок залежності до обробника.

Кожна кат-сцена має такі параметри:

– довжину (вказується в рушії у секундах);

– текст;

– позиція тексту (посередині, зверху, знизу);

– анімацію.

Вигляд кат-сцени наведено на рисунку 3.28.

Таким чином було створено гнучку та стійку до помилок систему кат-сцен. Ця система дозволяє конструювати складні рівні та за потреби надавати користувачу підказки.

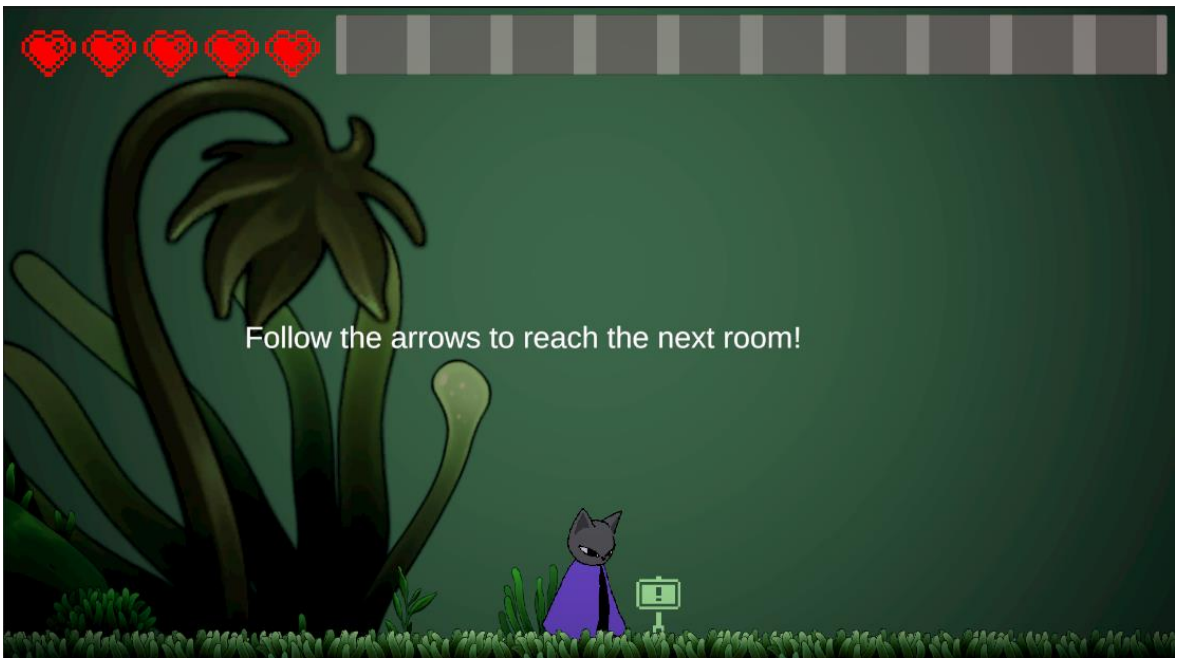


Рисунок 3.28 – Вигляд кат-сцени

3.5 Функціональне тестування

Тестування є одним з найважливіших елементів будь-якого застосунку для виявлення дефектів та їх усунення. А в випадку відеогри це останній етап підготовки гри перед випуском. Користувач має дуже багато можливостей, і всі прорахувати неможливо. Тому тестування здійснювалося на кожному етапі розробки. Процес тестування відеогри проводився вручну без застосування сторонніх продуктів. Середовищем для проведення тестів був Play Mode, доступний у рушії Unity. Такий метод тестування дозволяв відстежувати усі зміни, аналізувати їх швидше та вирішувати помилки якнайшвидше.

Основними об’єктами тестування стали функціональні компоненти гри: механіки пересування та бою персонажа, система інвентарю, аудіосупровід, система кат-сцен та загальний інтерфейс. Під час виконання тестів було виявлено ряд помилок, які було успішно усунено.

Першим тестуванням були можливості персонажа. Помилок з основним функціоналом виявлено не було. Пересування було плавним,

стрибки відновлювалися тільки при приземленні, ривок був зрозумілим та надавав можливість гравцю проходити крізь ворогів. Помилка виникла під час використання атаки. Під час удару по ворогу шкода завдавалася не тільки ворогам, а й самому гравцю. Після аналізу було виявлено, що рушій передавав подію зіткнення з колайдеру до основного компоненту гравця Rigidbody2D. Для вирішення проблеми фізику самої атаки було відділено від тіла гравця. Вигляд вирішення наведено на рисунку 3.29.

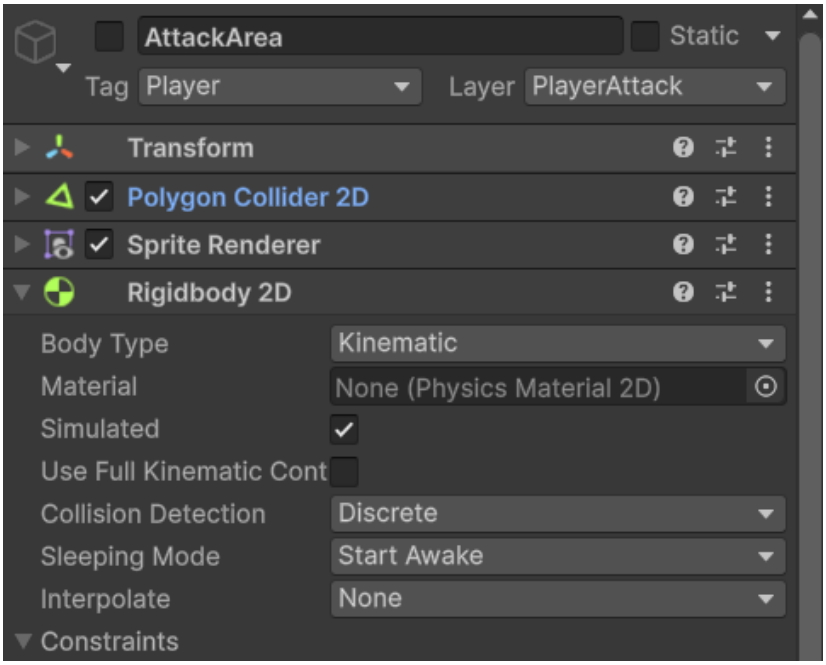


Рисунок 3.29 – Основні компоненти об’єкту AttackArea

Наступною проблемою було виявлено неможливість взаємодії з підбираними предметами у слотах в інвентарі. Після появи предметів у слотах користувач не мав змоги переміщати їх за допомогою миші, як було задумано. Висновок: механіка Drag & Drop не працювала коректно. Для вирішення проблеми предметам, які підбиралися зі сцени, було додано обробку інтерфейсних подій. До коду, який відповідав за інвентар, було додано логіку, яка в момент клонування предмету додавала йому компонент ItemHandler для обробки кліків.

Помилка також була виявлена у роботі аудіосистеми. Звуки кроків продовжували відтворюватися навіть під час перебування гравця у повітрі

напкриклад, під час стрибка. З метою усунення помилки було додано умови, так як виклик методу PlayFootstep було тісно пов’язано з перевіркою функції GroundCheck(), а в разі зупинки часу або виклику меню паузи запускався метод StopFootsteps(), який скасовував усі заплановані аудіо-події через CancelInvoke().

При початковому запуску текстової кат-сцени користувач не мав можливості взаємодіяти з інвентарем, натискати на кнопки. З метою вирішення в інспекторі було вимкнено параметр Raycast Target, що відновило роботу інтерфейсу і взаємодію з ним.

Після проведення ручного тестування було виявлено низку проблем, які впливали на основні функції користувача і не тільки. Після завершення тестування та фінальних виправлень, можна переходити до останнього етапу проєкту.

3.6 Збірка застосунку

Збірка проєкту в Unity передбачає компіляцію коду, стиснення всіх використаних ресурсів, а також пакування залежностей в один файл. Цей процес потребує ретельного зважування, так як налаштування будуть впливати на продуктивність та розмір кінцевого застосунку. Вибір платформи був зроблений на Windows, так як це передбачалося у вимогах.

Ігровий рушій Unity має дві варіації збірки: development та release. Різниця між ними полягає в готовності продукту. Якщо обрати збірку для розробки, то буде можливість перевірки споживання пам’яті та інших параметрів. Продуктивність такої збірки є дуже низькою та розбір значно більшим ніж у звичайної. Релізна збірка це фінальний варіант продукту. Рушій автоматично видаляє невикористані скрипти або код та збирає застосунок без можливості аналізу. Така збірка гарантує високу продуктивність та мінімальний розмір файлу.

У налаштуваннях центрального процесора (CPU) було обрано Intel 64, так як це сучасний стандарт для ПК на базі Windows. На відміну від 32-бітної архітектури, 64-бітна дозволяє створеному додатку використовувати більше 4 ГБ оперативної пам’яті, що важливо для плавної роботи застосунку. Вибір налаштувань детальніше можна переглянути на рисунку 3.30.

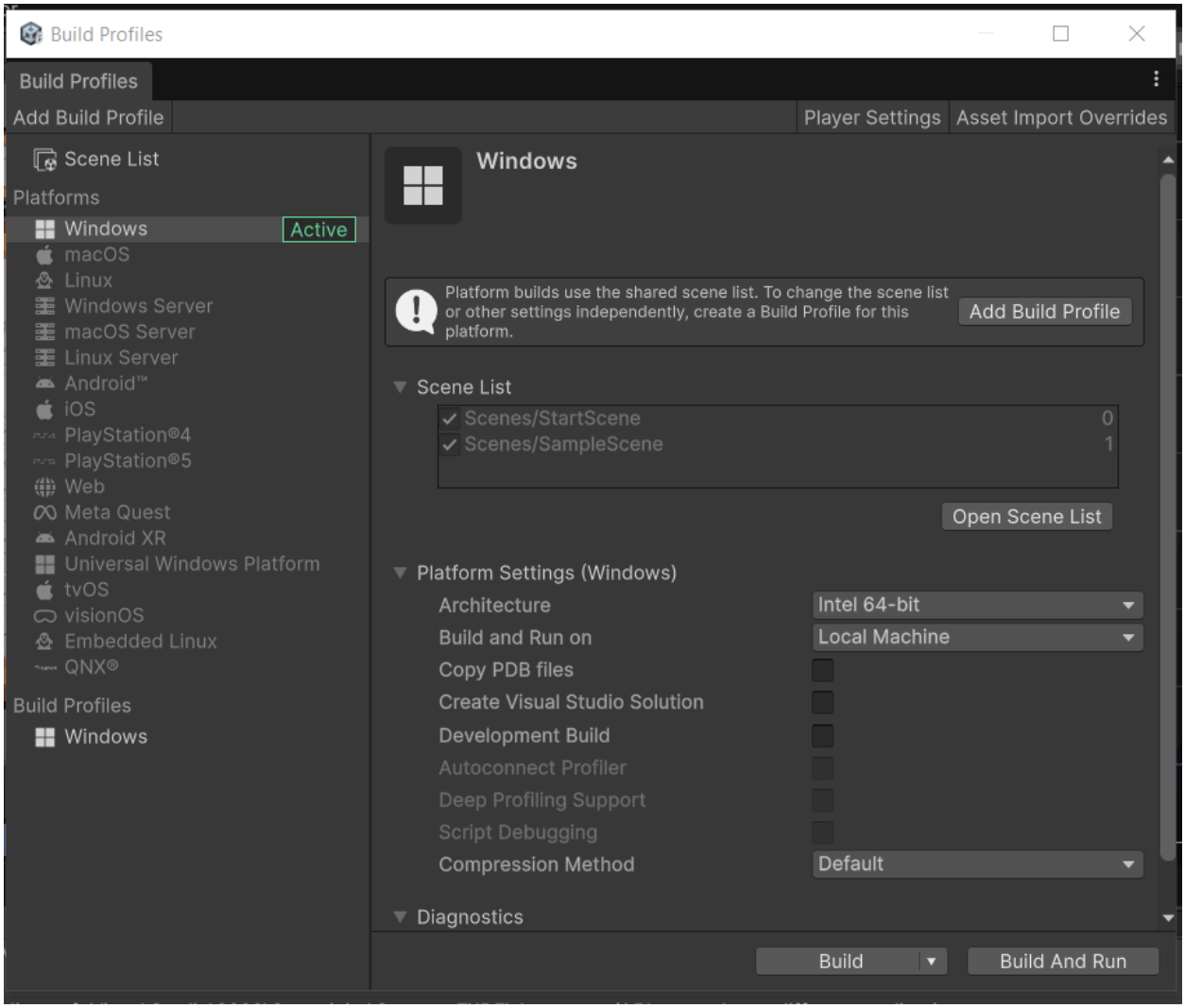


Рисунок 3.30 – Обрані налаштування для збірки проєкту

В результаті виконання практичного завдання було розроблено повноцінний ігровий додаток, візуальний інтерфейс, якого спроєктовано з урахуванням стандартів.

Головний екран гри є мінімалістичним та не перенавантажує гравця непотрібною інформацією (див. рис. 3.31).



Рисунок 3.31 – Головний екран

Після натиску на кнопку початку гри користувач зможе керувати персонажем та безпосередньо взаємодіяти з системою. Зверху екрана користувач побачить своє здоров'я та панель швидкого доступу (рис. 3.32).

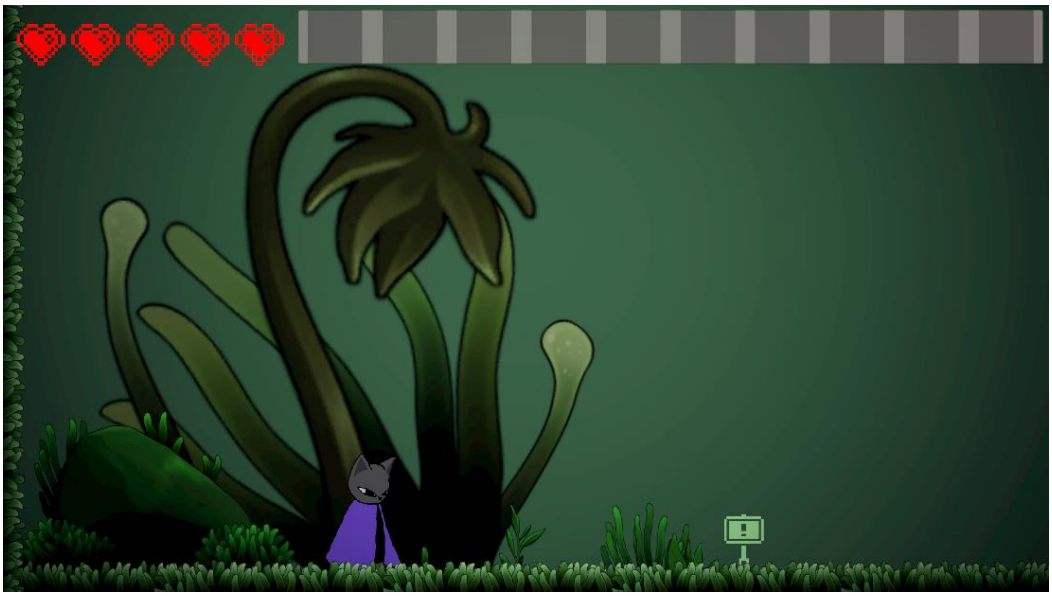


Рисунок 3.32 – Вигляд після головного екрану

Після входу в гру гравець безпосередньо зможе взаємодіяти з системою. Однією з найважливіших взаємодій є атака. Вона відображається у вигляді півкола для розуміння гравця, як далеко можливо вдарити ворога (рис. 3.33).

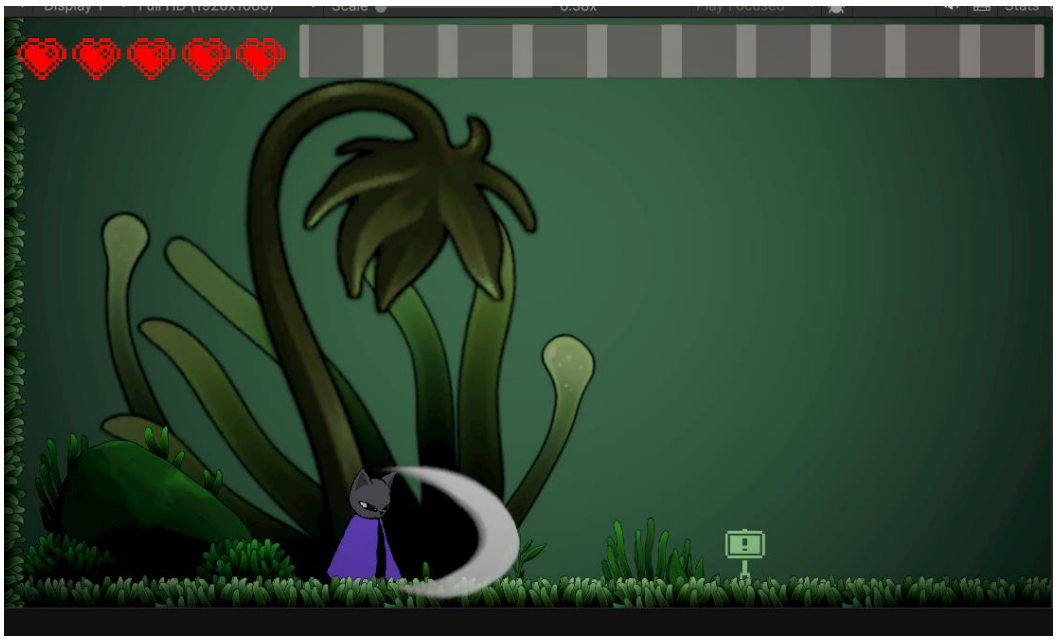


Рисунок 3.33 – Вигляд атаки гравця

Наступною взаємодією є ривок. Ривок відображається у вигляді лінії переміщення з початкової точки до точки завершення ривка (рис. 3.34).

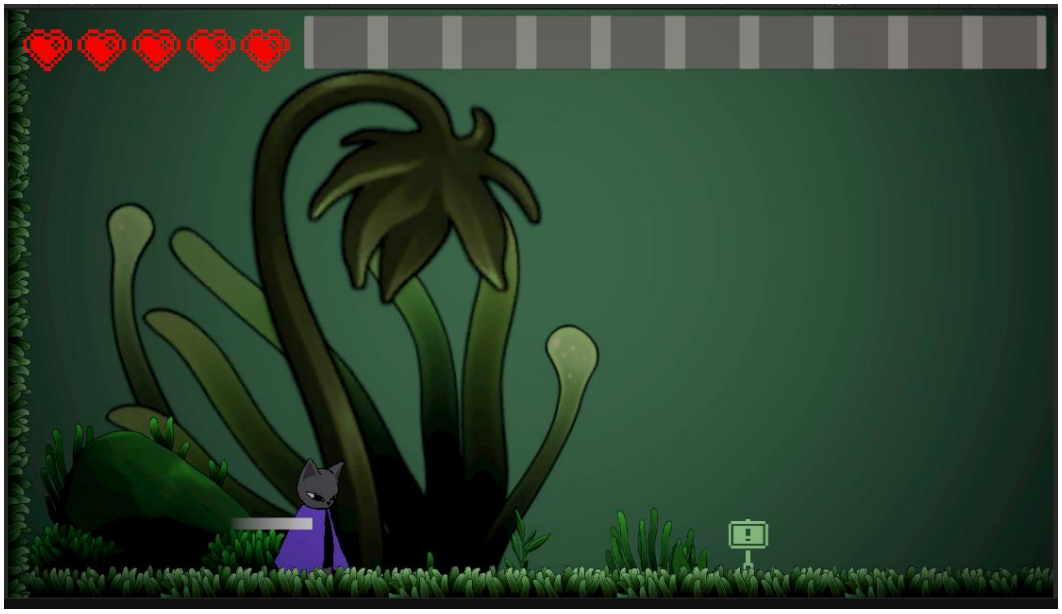


Рисунок 3.34 – Вигляд ривку персонажа

Звичайно, за описаними умовами користувач має можливість підбирання предметів. Для повідомлення гравцю про підняття видається ефект та з нижнього лівого боку виводиться назва та картинка предмету, кого було піднято (рис. 3.35).

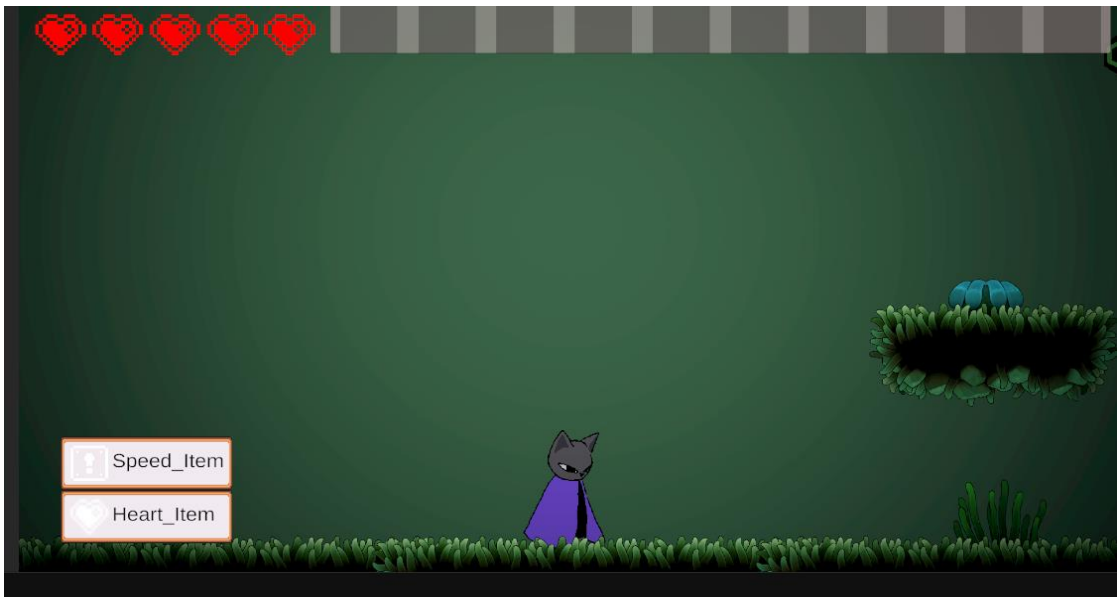


Рисунок 3.35 – Вигляд підняття предметів

Звичайно після підняття предметів користувач може використовувати їх. Для віртуалізації використаного предмету було використано TrailRenderer (див. рис. 3.36).

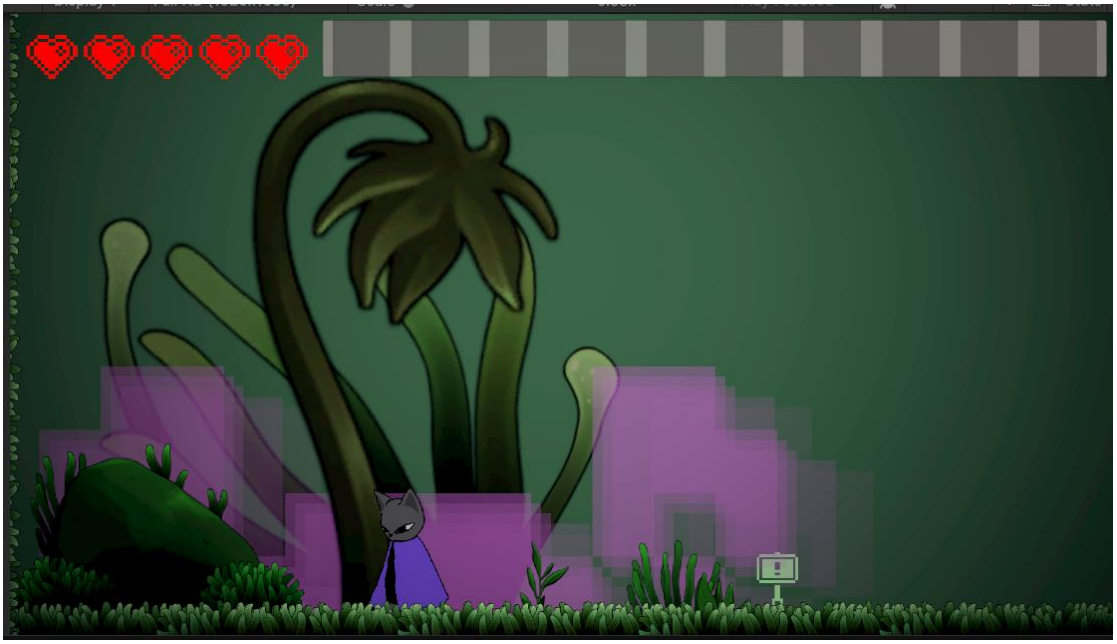


Рисунок 3.36 – Вигляд після використання предмету швидкості

Одним із варіантів взаємодії є інвентар, у якому користувач може пересуватися вкладками. Вигляд вкладки інвентарю наведено нижче на рисунках 3.37 – 3.38.

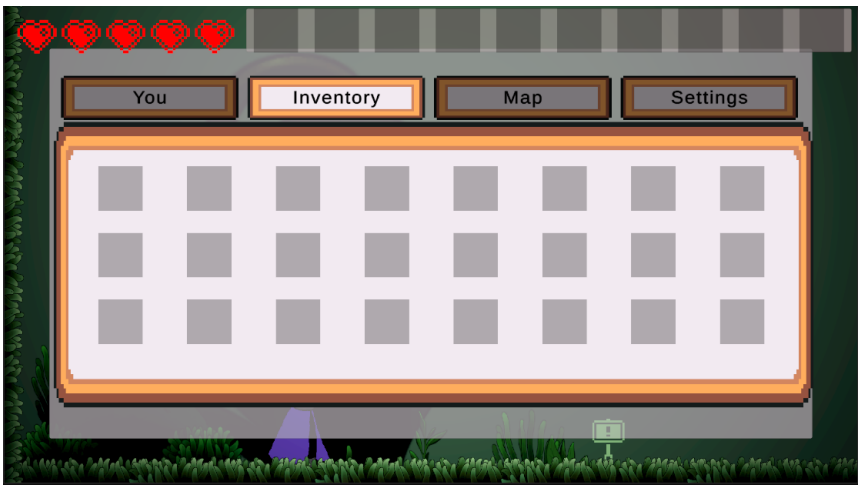


Рисунок 3.37 – Вигляд пустого інвентарю

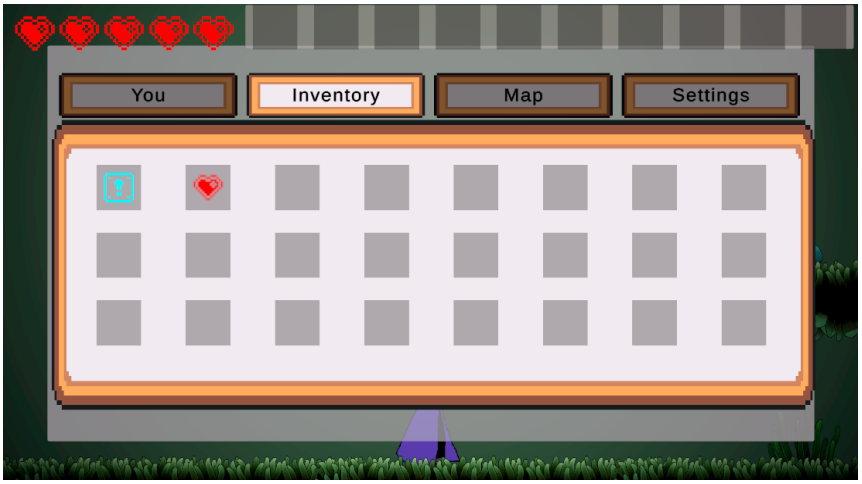


Рисунок 3.38 – Вигляд предметів в інвентарі

Наступною вкладкою є карта, яка допомагає гравцю в навігації та змінюється залежно від переміщення гравця (див. рис. 3.39 – 3.40).

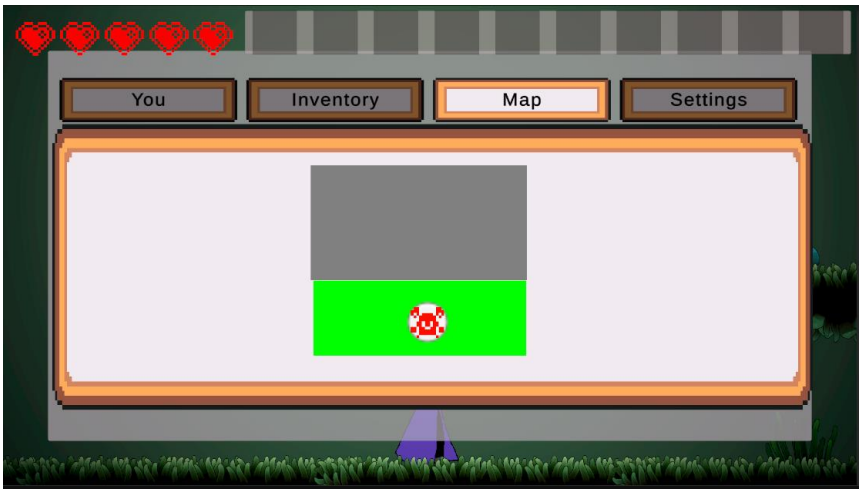


Рисунок 3.39 – Вигляд карти на початку гри



Рисунок 3.40 – Вигляд карти після переміщення користувача

Наступна вкладка – це налаштування, де користувач має змогу взаємодіяти з системою аудіокомпонентів. Вкладка має два повзунки для налаштування фонової музики та ефектів відповідно та кнопку виходу (див. рис. 3.41).



Рисунок 3.41 – Вигляд налаштувань

Таким чином, було завершено практичну частину виконання кваліфікаційної роботи.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було проведено аналіз предметної області та найпопулярніших ігрових рушіїв. У результаті створено повноцінну відеогру в жанрі платформер. Головною метою розробки було глибоке застосування принципів об’єктно-орієнтованого програмування (ООП) та архітектурних патернів у середовищі ігрового рушія Unity.

Під час виконання роботи було вирішено багато завдань та досягнуто вагомих практичних результатів. Розроблена програмна архітектура базується на принципах SOLID, що забезпечило чітке розділення логіки та слабку зв’язність підсистем. Використання класичних патернів проєктування (таких як «Одинак», «Спостерігач», «Команда» та «Шаблонний метод») створило надійний фундамент для подальшого масштабування проєкту. Ця архітектурна база дозволила успішно реалізувати складні механіки: систему здоров’я, інвентар з механікою перетягування предметів (Drag & Drop), динамічну карту, систему кат-сцен, а також штучний інтелект ворогів.

Особливу увагу було приділено користувацькому інтерфейсу. Створений користувацький інтерфейс є інтуїтивно зрозумілим. Система взаємодіє з гравцем за допомогою аудіосупроводу та візуальних складових – від спливаючих сповіщень про підняття предметів до ефектів при отриманні шкоди. Використання сучасних методів програмування забезпечило плавні переходи між локаціями без блокування основного потоку виконання гри.

Кінцевий результат – повністю готовий ігровий застосунок, що відповідає всім сформованим функціональним та нефункціональним вимогам. Завдяки правильній конфігурації релізної збірки з використанням компілятора IL2CPP, продукт працює плавно, високопродуктивно та безперебійно на ОС Windows, водночас забезпечуючи надійний захист вихідного коду від декомпіляції.

ПЕРЕЛІК ПОСИЛАНЬ

1. Kelleher L. Defining the Platformer genre. *Kelleher Bros*. URL: https://www.kelleherbros.com/blog/2024/3/4/defining-the-platformer-genre (дата звернення: 04.03.2026).

2. Godot Docs – 4.6 branch. URL: https://docs.godotengine.org/en/stable/ (дата звернення: 14.01.2026).

3. Bycer J. The Art of the Platformer. *Game Developer*. URL: https://www.gamedeveloper.com/design/the-art-of-the-platformer (дата звернення: 04.03.2026).

4. Unity Documentation. URL: https://docs.unity.com/en-us (дата звернення: 16.01.2026).

5. Unreal engine 5.7 documentation. URL: https://dev.epicgames.com/documentation/unreal-engine/unreal-engine-5-7-documentation (дата звернення: 17.01.2026).