**Міністерство освіти і науки України**
**Тернопільський національний технічний університет імені Івана Пулюя**

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету )
Кафедра програмної інженерії
(повна назва кафедри)

# КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

(назва освітнього ступеня)

на тему: «Розробка програмного забезпечення та тестування 2D ігрового застосунку з використанням мови програмування Python»

Виконала: студентка 4 курсу, групи СП-42
спеціальності 121 «Інженерія програмного забезпечення»
(шифр і назва спеціальності)

(підпис) Крупа М. В. (прізвище та ініціали)

Керівник (підпис) Цуприк Г. Б. (прізвище та ініціали)

Нормоконтроль (підпис) Стоянов Ю. М. (прізвище та ініціали)

Завідувач кафедри (підпис) Петрик М. Р (прізвище та ініціали)

Рецензент (підпис) (прізвище та ініціали)

Тернопіль
2026

Міністерство освіти і науки України
**Тернопільський національний технічний університет імені Івана Пулюя**

Факультет комп'ютерно-інформаційних технологій і програмної інженерії
(повна назва факультету)

Кафедра програмної інженерії
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Петрик М. Р.
(підпис) (прізвище та ініціали)
« 6 » квітня 2026 р.

# ЗАВДАННЯ
## НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня бакалавр
(назва освітнього ступеня)

за спеціальністю 121 «Інженерія програмного забезпечення»
(шифр і назва спеціальності)

студентці Крупі Марії Віталіївні
(прізвище, ім'я, по батькові)

1. Тема роботи «Розробка програмного забезпечення та тестування 2D ігрового застосунку з використанням мови програмування Python»

Керівник роботи Цуприк Г. Б., к.т.н., доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від « 06 » квітня 2026 року № 4/9-170
2. Термін подання студентом завершеної роботи 22.06.2026
3. Вихідні дані до роботи наукові літературні джерела

4. Зміст роботи (перелік питань, які потрібно розробити)
Вступ. 1 Аналіз предметної області та специфікація вимог
2. Проєктування та розробка програмного забезпечення
3. Тестування та аналіз якості програмного забезпечення
4. Безпека життєдіяльності та основи охорони праці
Висновки.
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
1. Тема роботи. 2. Актуальність, мета, задачі дослідження
3. Існуючі технології реалізації подібних систем.
4. Функціональні та нефункціональні вимоги .5. Загальна архітектура системи.
6. Варіанти використання. 7. Компоненти програми для налаштування параметрів системи.
8. Програмні засоби та технології. 9. Інтерфейси реалізації застосунку.
10. Тестування. 11. Висновки по роботі. 12. Слайди презентації.

6. Консультанти розділів роботи

| Розділ | Прізвище, ініціали та посада консультанта | Підпис, дата | |
|---|---|---|---|
| | | завдання видав | завдання прийняв |
| Безпека життєдіяльності та основи охорони праці | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

7. Дата видачі завдання 6 квітня 2026 р.

**КАЛЕНДАРНИЙ ПЛАН**

| № з/п | Назва етапів кваліфікаційної роботи | Термін виконання етапів роботи | Примітка |
|---|---|---|---|
| 1. | *Розробка технічного завдання* | *6.04 – 12.04* | Виконано |
| 2. | *Робота над першим розділом «Аналіз предметної області та специфікація вимог»* | *13.04 – 26.04* | Виконано |
| 3. | *Робота над другим розділом «Проєктування та розробка програмного забезпечення»* | *27.04 – 10.05* | Виконано |
| 4. | *Робота над третім розділом «Тестування та аналіз якості програмного забезпечення»* | *11.05 – 17.05* | Виконано |
| 5. | *Робота над четвертим розділом «Безпека життєдіяльності та основи охорони праці»* | *18.05 – 24.05* | Виконано |
| 6. | *Оформлення пояснювальної записки і графічного матеріалу* | *25.05 – 7.06* | Виконано |
| 7. | *Перевірка на академічний плагіат, перевірка керівником та консультантами* | *8.06 – 9.06* | Виконано |
| 8. | *Попередній захист кваліфікаційної роботи бакалавра* | *10.06* | Виконано |
| 9. | *Захист кваліфікаційної роботи бакалавра* | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

Студентка _____ Крупа М. В.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Цуприк Г. Б.
(підпис) (прізвище та ініціали)

# АНОТАЦІЯ

Розробка програмного забезпечення та тестування 2D ігрового застосунку з використанням мови програмування Python // Кваліфікаційна робота освітнього рівня «Бакалавр» // Марія Віталіївна Крупа // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра програмної інженерії, група СП-42 // Тернопіль, 2026 // С. 86, рис. – 26, табл. – 1, додат. – 7, бібліогр. – 27.

Ключові слова: 2D ігровий застосунок, об’єктно-орієнтоване програмування, мова Python, бібліотека Pygame, розробка ігор, тестування програмного забезпечення, архітектура програмного забезпечення, життєвий цикл розробки програмного забезпечення, Dungeon Crawler.

Кваліфікаційна робота присвячена розробці та тестуванню 2D ігрового застосунку засобами мови програмування Python із використанням бібліотеки Pygame.

У першому розділі виконано аналіз предметної області, розглянуто особливості жанру «Dungeon Crawler» та сучасні засоби розробки ігор. Сформовано специфікацію вимог до програмного забезпечення.

У другому розділі спроєктовано об’єктно-орієнтовану архітектуру застосунку: логіку переміщення персонажа, систему бойової взаємодії, механізми колізій, анімацію об'єктів та інтегровано користувацький інтерфейс.

У третьому розділі проведено функціональне та модульне тестування розробленого програмного забезпечення. Перевірено коректність роботи ігрових механік та стабільність системи. Оцінено загальну якість програмного продукту.

У четвертому розділі розглянуто питання охорони праці та безпеки життєдіяльності під час створення програмного забезпечення.

Об’єкт дослідження: 2D ігрові застосунки жанру «Dungeon Crawler».

Предмет дослідження: методи проєктування, реалізації та тестування 2D ігрового застосунку засобами мови програмування Python та бібліотеки Pygame.

# ABSTRACT

Software Development and Testing of a 2D Game Application Using the Python Programming Language // Bachelor's Degree Thesis // Mariia Vitaliivna Krupa // Ivan Pul'uj Ternopil National Technical University, Faculty of Computer and Information Systems and Software Engineering, Department of Software Engineering, Group SP-42 // Ternopil, 2026 // p. 86, figs. – 26, tables – 1, appendices – 7, bibliography – 27.

Keywords: 2D game application, object-oriented programming, Python, Pygame library, game development, software testing, software architecture, software development life cycle, Dungeon Crawler.

This thesis is devoted to the development and testing of a 2D game application using the Python programming language and the Pygame library.

The first chapter analyses the subject area, examines the characteristics of the 'Dungeon Crawler' genre and modern game development tools. It sets out the software requirements specification.

In the second chapter, an object-oriented application architecture is designed: character movement logic, a combat interaction system, collision mechanisms, object animation, and the user interface is integrated.

In the third chapter, functional and unit testing of the developed software was carried out. The correct operation of the game mechanics and the stability of the system were verified. The overall quality of the software product was assessed.

The fourth chapter examines issues of occupational health and safety during software development.

Research object: 2D 'Dungeon Crawler' games.

Research subject: methods of designing, implementing and testing a 2D game using the Python programming language and the Pygame library.

# ПЕРЕЛІК СКОРОЧЕНЬ І ТЕРМІНІВ

ПЗ – Програмне забезпечення.

ООП – Об’єктно-орієнтоване програмування.

Інді-гра (від англ. independent – незалежний) – Відеоігра, створена без фінансової підтримки чи контролю великих ігрових видавців.

Ігровий цикл – Основний цикл виконання гри, у межах якого обробляються події, оновлюється стан об’єктів і відображається графіка.

Ігрові механіки – Правила та способи взаємодії гравця з ігровим світом, персонажами, ворогами й об’єктами.

Колізія – Зіткнення або перетин ігрових об’єктів, яке обробляється програмною логікою гри.

Тайл (від англ. tile – плитка) – Елемент тайлової карти, що використовується для побудови ігрового середовища.

FPS (скорочено від англ. Frames Per Second – кадри за секунду) – кількість кадрів за секунду, що характеризує швидкість оновлення зображення у грі.

LoS (скорочено від англ. Line of Sight – пряма видимість) – Уявна пряма лінія, яка з'єднує персонажа з цільовим об'єктом.

**ЗМІСТ**

# ВСТУП

Сучасна індустрія комп'ютерних ігор є однією з найбільш динамічних галузей інформаційних технологій. Зростання популярності незалежної (інді) розробки сприяє поширенню інструментів та технологій, які дозволяють створювати повноцінні ігрові застосунки навіть невеликими командами або окремими розробниками. Одним із поширених підходів до створення 2D ігор серед розробників початківців є використання мови програмування Python у поєднанні з бібліотекою-рушієм Pygame, яка надає необхідні засоби для реалізації графічного інтерфейсу, обробки подій, анімації та взаємодії між ігровими об'єктами.

Розробка ігрових застосунків є комплексною задачею, що поєднує принципи об'єктно-орієнтованого програмування, проєктування програмних систем, реалізації алгоритмів взаємодії між об'єктами та тестування програмного забезпечення. Особливої уваги потребує створення архітектури гри, яка забезпечує зручність підтримки коду, можливість подальшого розширення функціональності та стабільну роботу застосунку в режимі реального часу.

Актуальність теми. Розробка 2D ігрових застосунків залишається актуальним напрямком програмної інженерії завдяки широкому використанню таких застосунків як у розважальній сфері, так і в освітніх цілях. Водночас процес створення гри дозволяє комплексно застосувати сучасні підходи до проєктування програмного забезпечення, реалізувати механізми керування персонажем, анімації, колізій, штучного інтелекту противників та взаємодії з ігровим світом. Використання бібліотеки Pygame робить можливим створення повноцінного ігрового застосунку із застосуванням об'єктно-орієнтованого підходу, що дозволяє дослідити особливості побудови архітектури ігор та оцінити ефективність реалізованих програмних рішень. Тому розробка та тестування 2D ігрового застосунку з використанням мови програмування Python є актуальним науково-технічним завданням.

Метою даної кваліфікаційної роботи є розробка та тестування 2D ігрового застосунку жанру «Dungeon Crawler» з використанням мови програмування Python та бібліотеки Pygame.

Для досягнення поставленої мети було вирішено такі задачі: провести аналіз існуючих засобів та технологій розробки 2D ігор; обґрунтувати вибір мови програмування Python та бібліотеки Pygame для реалізації програмного застосунку; спроєктувати архітектуру ігрового застосунку на основі об’єктно-орієнтованого підходу; реалізувати систему керування ігровим персонажем та механізми переміщення у двовимірному просторі; реалізувати систему бойової взаємодії, що включає використання зброї, стрільбу снарядами та нанесення шкоди супротивникам; реалізувати механізми взаємодії персонажа з ігровими об’єктами, зокрема збору монет та лікувальних предметів; реалізувати систему анімації персонажів та противників; провести тестування основних функціональних можливостей гри; проаналізувати результати тестування та оцінити працездатність розробленого програмного забезпечення.

Об’єкт дослідження – процес розробки та тестування двовимірного ігрового застосунку реального часу з використанням усіх етапів життєвого циклу програмного забезпечення та об’єктно-орієнтованого підходу.

Предмет дослідження – методи та засоби розробки 2D ігрових застосунків мовою програмування Python, архітектура програмного забезпечення на основі бібліотеки Pygame, алгоритми переміщення персонажів, обробки колізій, взаємодії ігрових об’єктів та організації ігрового циклу.

У роботі використано такі методи дослідження: аналіз науково-технічної літератури та сучасних програмних рішень (для дослідження сучасних підходів до розробки 2D ігор), системний аналіз (для проєктування структури програмного забезпечення), об’єктно-орієнтоване проєктування та програмування (для реалізації програмних компонентів гри), моделювання (для побудови UML-діаграм та опису архітектури системи), експериментальне тестування (для перевірки коректності роботи реалізованих механік та взаємодії між об’єктами), аналіз результатів тестування.

Наукова новизна одержаних результатів: розроблено програмний застосунок жанру «Dungeon Crawler» з використанням мови програмування Python та бібліотеки Pygame на основі об’єктно-орієнтованої архітектури; реалізовано систему взаємодії між ігровими об’єктами, що поєднує механізми керування

персонажем, бойової системи, збору предметів та поведінки противників у межах єдиного ігрового циклу; досліджено особливості використання бібліотеки Pygame для побудови ігрових систем реального часу та реалізації механізмів анімації, колізій і взаємодії об’єктів.

Практичне значення роботи полягає у створенні функціонального 2D ігрового застосунку, який може бути використаний як приклад реалізації ігрових механік засобами Python та Pygame. Розроблена архітектура програмного забезпечення може бути використана як основа для подальшого розширення функціоналу гри, додавання нових рівнів, типів ворогів, предметів та інших ігрових механік. Результати роботи також можуть бути використані в навчальному процесі під час вивчення дисциплін, пов’язаних із програмуванням, проєктуванням програмного забезпечення та розробкою комп’ютерних ігор.

Результати кваліфікаційної роботи пройшли апробацію шляхом практичної реалізації програмного застосунку та проведення його функціонального тестування.

Основні положення та результати кваліфікаційної роботи було апробовано та опубліковано у вигляді тез доповіді на IX Міжнародній студентській науково-технічній конференції «Природничі та гуманітарні науки. Актуальні питання» (див. Додаток А), організованій Тернопільським національним технічним університетом імені Івана Пулюя за участю українських та закордонних закладів вищої освіти.

# 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА СПЕЦИФІКАЦІЯ ВИМОГ

Ця робота присвячена дослідженню теми впливу структурованого підходу розробки на результат у сфері геймдеву. Предметна область достатньо популярна і має безліч продуктів, однак не кожна гра, особливо створена незалежними розробниками, може похвалитися якісною архітектурою програмного забезпечення, високою продуктивністю та можливістю подальшого масштабування.

Одним із ключових чинників успішної розробки є застосування сучасних принципів програмної інженерії. Використання об’єктно-орієнтованого програмування дозволяє структурувати програмний код у вигляді взаємопов’язаних класів та об’єктів, що підвищує його читабельність, спрощує супровід і повторне використання компонентів. Крім того, важливу роль відіграє дотримання принципів модульності, інкапсуляції, наслідування та поліморфізму, які забезпечують гнучкість програмної системи та полегшують внесення змін у майбутньому.

Застосування структурованого підходу до проєктування архітектури, розподілу функціональності між модулями та організації взаємодії між компонентами дає змогу створювати програмні продукти з вищим рівнем надійності та розширюваності. Особливо актуальним це є для ігрових застосунків, де необхідно реалізувати велику кількість взаємопов’язаних механік, забезпечуючи при цьому стабільну роботу програми в режимі реального часу.

## 1.1 Особливості жанру «Dungeon Crawler» та принципи побудови 2D ігор

Бродіння підземеллями (англ. «Dungeon Crawler») – це різновид рольової гри, ігровий процес якої зосереджений переважно на знищенні ворогів під час дослідження лабіринтоподібного або схожого на підземелля середовища, яке може генеруватися як вручну, так і випадковим чином [1]. У таких ігрових застосунках користувач (гравець) веде персонажа, тобто головного героя гри, крізь рівні, які поступово стають дедалі складнішими, а на шляху трапляються різного роду

перешкоди, наприклад, монстри. Окрім того, герой може підбирати предмети по-типу зброї, зілля, монет та інших ресурсів, передбачених сюжетом.

Однією із особливостей жанру є його вигляд, а саме 2D формат. Ця особливість робить розробку таких застосунків легшою і зручнішою у плані ігрових механік. Можна розділити піксельні об’єкти на квадрати однакового розміру, та з різними властивостями, щоб спростити роботу із різними модулями коду. Таким чином, 2D ігрові застосунки – це чудовий старт для розробника у геймдеві (розробка комп’ютерних ігор).

Перш за все, потрібно визначитись із концептом відеогри: жанр, мета, управління, скільки рівнів та сюжет. Наступним кроком побудови є вибір відповідного ігрового рушія. Це програмне забезпечення, яке надає набір інструментів та середовищ для створення відеоігор. Він забезпечує основну функціональність, необхідну для розробки ігор, включаючи графіку, фізику, звук, штучний інтелект, мережеву взаємодію та інтерфейс користувача. Основна мета ігрового рушія – спростити процес створення ігор, абстрагуючи складні технічні деталі та надаючи розробникам можливість зосередитися на творчих аспектах гри [2].

Базові механіки гри – ледь не найважливіший етап. Це про те, як користувач буде взаємодіяти із застосунком, якого типу перешкоди він долатиме, чи є якісь досягнення, збереження гри, як зберігатиметься прогрес і тому подібне.

Наступним чином потрібно розробити дизайн. Вигляд світу, персонажі, анімації – це все надважливі аспекти, адже це перше, із чим стикається гравець. Тут важливо, який посил хочеться донести до користувача, чи що хочеться довести самому собі. Наприклад, якщо це геніальна концептуальна задумка вашого розуму, то тут багато що залежатиме від того, як ви це представите перед людьми. Тут варто попрацювати над дизайном власноруч. Якщо ж головна мета проєкту засвоїти ігрові механіки, отримати досвід розробки чи просто розважитись – можна використати вже готові асети, щоб більше сфокусуватись на ігрових механіках.

На передостанньому етапі – це сама розробка продукту. Це створення нового проєкту, налаштування елементів керування, розробка базового геймплею, додавання перешкод і взаємодій. Тут відбувається найцікавіше: додавання

предметів, ворогів, виявлення колізій, механіка збільшення і зменшення здоров’я, збереження прогресу і отримання нагород. Також побудова рівнів, їхній дизайн та складність. Тут ще можна додати звуки та музику. Аналогічно з дизайном, можна розробити свої, а можна використати готові.

Завершальний етап – це тестування, оптимізація та розгортання проєкту на широкий загал. На цьому етапі перевіряється коректність роботи всіх ігрових механік, виявляються та усуваються помилки, які могли залишитися після розробки. Також виконується оптимізація продуктивності гри для забезпечення стабільної роботи на різних пристроях. Після завершення всіх перевірок проєкт готується до публікації та стає доступним для кінцевих користувачів [3].

### 1.2 Аналіз засобів та технологій розробки ігрових застосунків

Як було сказано у попередньому підрозділі, вибір ігрового рушія – важливий етап у розробці гри. Дякувати сучасним технологіям, розробник може обрати будь-яке програмне забезпечення на свій смак, адже їх є багато і на кожну мову програмування. Ігрові рушії є багаторазовими компонентами, які розробники використовують для побудови основи гри. Це дає їм більше часу зосередитись на унікальних елементах, таких як моделі персонажів, текстури, взаємодія об’єктів тощо [4]. Розібрано кілька найпопулярніших:

1. Unreal Engine від Epic Games. Оригінальна версія була випущена в 1998 році, а рушій досі широко використовується розробниками. Наприклад, такі популярні ігри як Hogwarts Legacy, серія Gears of War, The Witcher 3 (в окремих версіях і прототипах), Final Fantasy VII Remake, а також Red Dead Redemption 2 (частково у виробничих процесах та інструментах). Unreal Engine активно використовується як у великих AAA-проєктах, так і в інді-розробці завдяки високій якості графіки та широким можливостям для створення складних ігрових механік. З мінусів – складний для початківців.

Рисунок 1.1 – Ігри, розроблені за допомогою Unreal Engine [5]

2. Unity. Багатоплатформенний ігровий рушій, який дозволяє легко створювати інтерактивний 3D-контент. Дуже багато інді-розробників віддають перевагу Unity за його відмінні функціональні можливості, якісний контент та можливість його використання майже для будь-яких ігор [4]. Наприклад, такі відомі ігри як Hollow Knight, Cuphead, Among Us, Ori and the Blind Forest, Subnautica, Cities: Skylines та Genshin Impact створені з використанням Unity або його ключових компонентів.


Рисунок 1.2 – Ігри, розроблені за допомогою Unity [6]

3. Godot. Рушій чудово підходить для створення 2D та 3D ігор, пропонує величезний набір поширених інструментів, тому можна просто зосередитись на створенні своєї гри, не вигадуючи велосипед. Він безкоштовний у користуванні та з відкритим кодом через ліцензію MIT [4]. На відміну від попередніх представників, цей рушій підходить для початківців. Серед відомих ігор, створених на Godot, можна виділити такі проєкти як Dome Keeper, Cassette Beasts (версія на Godot 4), Brotato, Buckshot Roulette, Endoparasitic, а також низку інді-ігор та прототипів, які активно розробляються незалежними студіями. Godot особливо популярний серед інді-розробників завдяки зручній системі скриптів і швидкому прототипуванню ігрових механік.

4. Pygame. Створений як заміна для бібліотеки PySDL, яка перестала розвиватися, Pygame об’єднує та розширює можливості бібліотеки SDL (Simple DirectMedia Layer), яка забезпечує міжплатформовий доступ до основних мультимедійних апаратних компонентів системи, таких як звук, відео, миша, клавіатура та джойстик [7]. Як і попередник, це хороший рушій для старту у геймдеві. Використовується лише для створення простих 2D ігор. На відміну від повноцінних ігрових рушіїв, Pygame не має вбудованого редактора сцен або складних інструментів, тому розробник працює переважно через код. Серед відомих ігор, створених на Pygame, можна виділити такі проєкти як Frets on Fire, SolarWolf, Pygame Tower Defense та різні інді-ігри й навчальні демо. Pygame часто використовується для навчання основ геймдеву, оскільки добре пояснює базові принципи роботи ігрового циклу, подій та рендерингу.

## 1.3 Обґрунтування вибору мови програмування Python та бібліотеки Pygame

Для розробки ігрового застосунку було обрано мову програмування Python у поєднанні з бібліотекою Pygame. Такий вибір зумовлений поєднанням простоти синтаксису, високої читабельності коду та достатньої функціональності для створення 2D ігор.

Python є однією з найпопулярніших мов програмування завдяки своїй універсальності та низькому порогу входу. Вона дозволяє швидко реалізовувати ідеї, що особливо важливо в навчальних проєктах та прототипуванні ігор. Крім того, Python має велику кількість бібліотек, що спрощує процес розробки рішень для різних задач, таких як обробка подій, математичні обчислення та інше.

Бібліотека Pygame була обрана як основний інструмент для реалізації ігрової логіки та графіки. Вона надає базові можливості для створення 2D ігор, зокрема роботу з графікою, звуком, подіями клавіатури та миші, а також керування ігровим циклом. Pygame добре підходить для навчальних проєктів, оскільки дозволяє зрозуміти фундаментальні принципи роботи ігрових рушіїв без зайвої складності. Розробник самостійно будує структуру гри, реалізує фізику, колізії та ігрові механіки. Це сприяє глибшому розумінню процесів, що відбуваються всередині ігрових застосунків.

Таким чином, поєднання Python і Pygame є оптимальним рішенням для навчальної розробки 2D ігрового застосунку, оскільки забезпечує баланс між простотою реалізації та достатнім функціоналом для створення повноцінної гри.

### 1.4 Аналіз сучасних ігрових 2D проєктів жанру «Dungeon Crawler»

Традиційні ігри жанру зазвичай містять стандартні механіки: бродіння підземеллям, подолання перешкод у вигляді ворогів, збирання предметів. Однак завдяки тому, що сучасні ігри часто поєднують елементи різних жанрів, у нас з’явилося набагато більше варіантів серед сучасних представників жанру «Dungeon Crawler». Таким чином, ігор з цією позначкою вже значно більше і фактично кожен може знайти серед них щось, що більше лежить до серця.

Одним з таких класичних представників жанру є Enter the Gungeon (рисунок 1.3). Гра написана за допомогою Unity, мовою C#.


Рисунок 1.3 – Геймплей Enter the Gungeon [8]

Застосунок слідує усім традиціям жанру, де гравець бігає підземеллями та знешкоджує ворогів. Його урізноманітнює креативний аспект. У арсеналі героя може знаходитись як стандартна зброя, так і химерні зразки по-типу літери “R” та фотоапарату, що стріляє кулями. Таке різноманіття додає ігровому процесу пікантності та спонукає до експериментів [8].

Ще одним яскравим представником жанру є доволі популярна гра The Binding of Isaac: Rebirth (рисунок 1.4). Вона також написана за допомогою Unity та C#. Тут замість підземель гравець проходить різні кімнати, та суть та ж сама: він має пройти усі перешкоди та вийти із кімнати.


Рисунок 1.4 – Геймплей The Binding of Isaac: Rebirth

Популярна інді-гра Cult of the Lamb (рисунок 1.5), де підземелля є лише однією з елементів гри, тоді як повністю вона включає і інші механіки, такі як менеджмент ресурсів – це чудовий приклад того, як спільнота поєднує елементи різних жанрів та кожного разу додає щось своє. Проте саме механіка підемель тут беззаперечно є однією з найкращих як в плані бойовки, так і у візуальному плані.


Рисунок 1.5 – Геймплей Cult of the Lamb [9]

Не дивлячись на відмінність у обраних мовах та рушіях між цими прикладами та власним проєктом «DUNG», можна сформувати загальну картинку щодо майбутнього продукту. У всіх цих іграх є одна спільна річ – інформаційна панель. На ній зображено стан гравця: його рівень здоров’я, іноді вказано рівень, якісь речі з інвентарю та підрахунок очків, що можна взяти до уваги при розробці.

## 1.5 Специфікація вимог до програмного забезпечення. Варіанти використання

Основною метою специфікації вимог є якісне і детальне визначення функціональних можливостей системи, вимог до її продуктивності, зручності використання та подальшого розширення.

На основі аналізу предметної області було визначено перелік основних функціональних вимог. Для наочності наведено діаграму варіантів використання (див. рисунок 1.6), яка дозволяє уявити типи ролей та їх взаємодію із системою та зображує функціональні вимоги (те, що система може зробити) з точки зору користувача [10]. До функціональних вимог належать:

- створення та відображення елементів ігрового світу;
- керування персонажем за допомогою клавіатури;
- реалізація плавного переміщення персонажа у чотирьох напрямках;
- реалізація системи колізій між персонажем та об'єктами карти;
- підтримка анімації персонажа та противників;
- реалізація системи бойової взаємодії між персонажем та ворогами;
- можливість стрільби зі зброї та створення снарядів;
- реалізація логіки поведінки ворогів;
- розробка штучного інтелекту для ворогів.
- реалізація спеціальних атак босів;
- відображення рівня здоров’я персонажа;
- реалізація збору предметів та нарахування ігрових балів;
- відображення інформації про нанесену шкоду;
- підтримка прокручування карти під час переміщення персонажа;

- завершення гри при втраті всього запасу здоров'я персонажа.

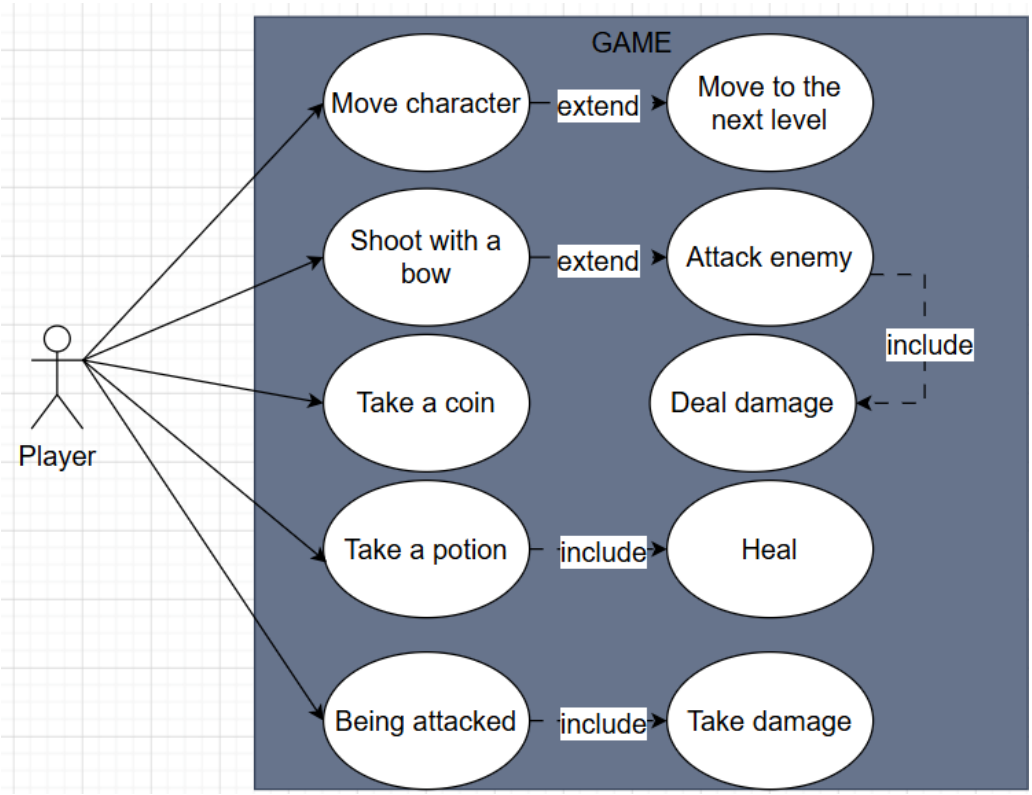


Рисунок 1.6 – Діаграма варіантів використання

Крім функціональних вимог було визначено перелік нефункціональних вимог. До них вимог належать:

- забезпечення стабільної роботи застосунку в режимі реального часу;
- підтримка частоти оновлення кадрів не менше 60 FPS;
- використання об’єктно-орієнтованого підходу до проєктування програмного забезпечення;
- забезпечення можливості подальшого розширення функціоналу гри;
- модульна структура програмного коду;
- простота супроводу та модифікації програмного забезпечення.

Для реалізації зазначених вимог було обрано мову програмування Python та бібліотеку Pygame. Python забезпечує високу швидкість розробки та читабельність коду, а Pygame надає необхідні інструменти для роботи з графікою, обробки подій користувача, анімації та керування ігровим циклом.

Результати проведеного аналізу вимог стали основою для подальшого проєктування архітектури програмного забезпечення, визначення структури класів та розробки основних ігрових механік.

### 1.6 Постановка задачі

Незважаючи на велику кількість комерційних ігор, значна частина інді-проєктів стикається з проблемами недостатньої оптимізації ігрової логіки, складністю масштабування архітектури та відсутністю чіткої структуризації програмного коду. Особливо актуальною є проблема створення 2D ігор із динамічним ігровим процесом, де одночасно взаємодіє велика кількість об’єктів. У таких умовах виникає необхідність у чіткій організації взаємодії між класами, ефективному управлінні станами об’єктів та обробці подій у реальному часі. Часто це проєкти без чіткої мети, з відсутністю грамотного підходу як розробника, який би слідував усім вимогам SOLID, ООП, дотримувався би повного життєвого циклу програмного забезпечення. Відсутність продуманої архітектури призводить до ускладнення підтримки коду та зниження продуктивності застосунку при збільшенні кількості ігрових сутностей.

Також важливим аспектом є синхронізація оновлення станів усіх об’єктів у межах ігрового циклу, що потребує використання єдиної структури керування світом гри.

Окремої уваги потребує реалізація системи бойової взаємодії, яка включає стрільбу з дистанційної зброї, генерацію снарядів, перевірку зіткнень із ворогами та нанесення шкоди. У цьому контексті важливо забезпечити коректну обробку колізій та своєчасне оновлення стану об’єктів, включаючи їх знищення або зміну стану (наприклад, отримання шкоди або оглушення).

Крім того, постає задача забезпечення гнучкої структури програмного коду, яка дозволяє легко додавати нові типи персонажів, ворогів, предметів та механік без суттєвих змін у вже реалізованих компонентах системи. Це досягається шляхом використання об’єктно-орієнтованого підходу, наслідування та розподілу відповідальності між класами.

На основі аналізу стану проблеми можна сформулювати науково-технічне завдання для кваліфікаційної роботи. Воно полягає у розробці 2D ігрового застосунку з видом зверху (top-down), реалізованого з використанням мови програмування Python та бібліотеки Pygame, який забезпечує:

- керування ігровим персонажем у реальному часі;
- взаємодію з ігровим світом (перешкоди, предмети, вороги);
- систему бойової взаємодії (стрільба, отримання та нанесення шкоди);
- реалізацію анімацій персонажів та ворогів;
- систему збору предметів та підрахунку ігрових ресурсів;
- коректну обробку колізій між об’єктами;
- оновлення ігрового стану в межах єдиного ігрового циклу.

Розробка також передбачає створення структурованої архітектури програмного забезпечення з використанням класів для різних типів ігрових сутностей, що забезпечує легкість супроводу та можливість подальшого розширення функціоналу.

Тестування застосунку включає перевірку коректності роботи ігрової механіки, обробки зіткнень, системи бою, а також стабільності роботи програми при тривалому виконанні ігрового циклу та великій кількості об’єктів на екрані.

Таким чином, реалізація даного проєкту спрямована на створення функціонального, структурованого та розширюваного 2D ігрового застосунку, який демонструє практичне застосування принципів об’єктно-орієнтованого програмування, обробки подій та побудови ігрових систем у Python, а також знань усіх етапів життєвого циклу ПЗ.

У цьому розділі досліджено предметну область розробки 2D ігор та вибрано технології і засоби розробки на основі сучасних рішень. Розглянуто кілька прикладів та сформовано базові вимоги до розроблюваного застосунку. На основі цих вимог було поставлено задачу для кваліфікаційної роботи.

# 2 ПРОЄКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

У цьому розділі представлено процес моделювання та проєктування архітектури, розробки програмного забезпечення гри. Розглянуто основні принципи побудови системи, визначено її структурні компоненти та описано їх взаємодію в межах загальної архітектури застосунку. Окрему увагу приділено реалізації ключових підсистем гри, зокрема механіці керування персонажем, бойовій системі, роботі з ігровими об’єктами та обробці взаємодії між елементами ігрового середовища.

## 2.1 Проєктування архітектури програмного забезпечення

Коректне і грамотне проєктування та моделювання архітектури системи є запорукою для успішної розробки кінцевого програмного продукту. На етапі збору та аналізу вимог програмного забезпечення визначено основні функціональні вимоги до гри. Розроблюваний застосунок повинен забезпечувати керування персонажем у реальному часі, взаємодію з ігровим світом, систему бою, анімацію об'єктів, збір предметів та обробку зіткнень між об'єктами.

Для реалізації поставлених вимог обрано об'єктно-орієнтований підхід, який дозволяє розділити функціональність програми на окремі незалежні компоненти та забезпечує можливість подальшого розширення функціоналу гри та підтримуваності коду. Основною перевагою такого підходу є інкапсуляція даних і поведінки об'єктів, що спрощує супровід та модифікацію програмного коду.

Діаграма варіантів використання дозволяє уявити типи ролей та їх взаємодію із системою та зображує функціональні вимоги (те, що система може зробити) з точки зору користувача. Може описуватись текстом (як це зроблено у пункті 1.5), або у вигляді діаграми (рисунок 2.1) [10].

Архітектура гри складається з декількох взаємопов'язаних підсистем:

- підсистема керування персонажем;
- підсистема ігрового світу;
- підсистема бойової взаємодії;

- підсистема предметів;
- підсистема відображення інформації;
- підсистема обробки подій користувача.

Центральним компонентом архітектури є клас World, який відповідає за формування ігрового рівня, зберігання інформації про об'єкти карти та створення всіх ігрових сутностей. Саме цей клас забезпечує взаємодію між елементами ігрового середовища.

Для представлення персонажів використовується клас Character. Він реалізує логіку переміщення, систему анімації, обробку станів персонажа, механізм отримання шкоди та взаємодію з іншими об'єктами гри. Клас використовується як для головного героя, так і для противників, що дозволяє уникнути дублювання коду.

Функціональність бойової системи реалізована за допомогою класів Weapon, Arrow та Fireball. Клас Weapon відповідає за керування зброєю персонажа та створення снарядів. Класи Arrow і Fireball реалізують логіку руху снарядів, перевірку зіткнень та нанесення шкоди цілі.

Для реалізації системи предметів використовується клас Item, який забезпечує можливість збору монет та лікувальних зіллів. Отримані предмети впливають на стан персонажа шляхом збільшення кількості набраних балів або відновлення здоров'я.

Структура взаємодії основних компонентів системи представлена на діаграмі класів (рисунок 2.2).

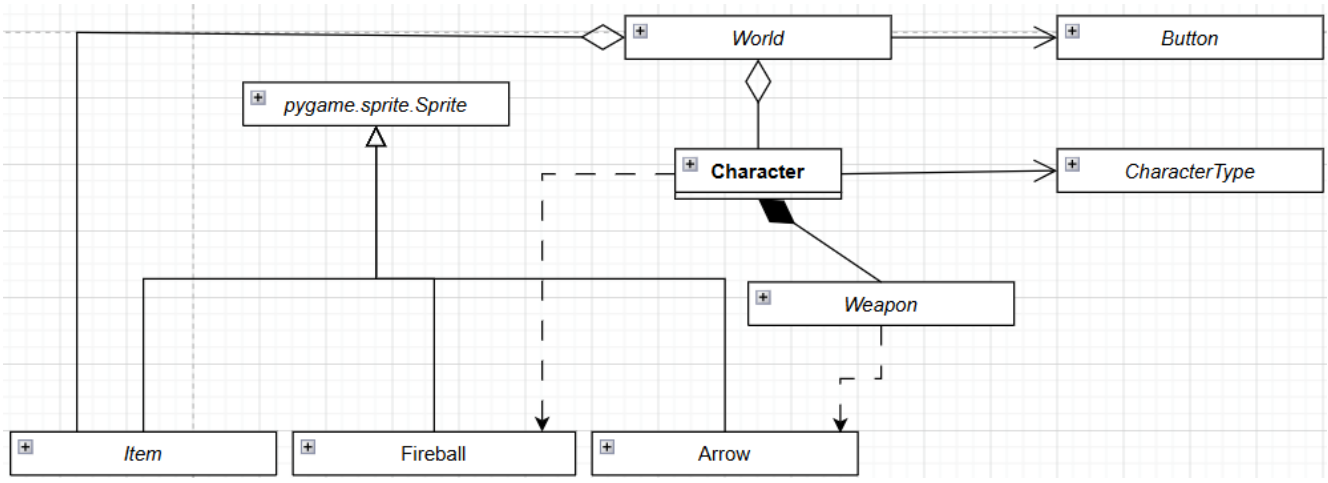


Рисунок 2.2 – Діаграма класів

На діаграмі показано усі зв’язки, передбачені для класів.

## 2.2 Реалізація об’єктно-орієнтованої структури застосунку

Однією з передумов коректної реалізації об’єктно-орієнтованого підходу є впорядкована структура файлів та модулів програмного забезпечення.

Оскільки у підрозділі 2.1 спроєктовано діаграму класів та визначено основні компоненти архітектури, то перш за все було створено окремі файли для реалізації кожного класу. Окрім основних файлів також створено головний (main) файл для запуску програми, файл з конфігураційними значеннями та папки з асетами для вигляду компонентів (про них детальніше у підрозділі 2.5). Усе це було залито на git (рисунок 2.3) за допомогою команди ініціалізації проєкту.


Рисунок 2.3 – Структура директорії проєкту

На діаграмі класів зображено класи Item, Fireball та Arrow як такі, що успадковують функціональність класу pygame.sprite.Sprite. Це клас із модуля pygame.sprite, призначений для використання як базовий клас для різних типів об’єктів у грі [11]. Наслідування відбувається у конструкторі дочірнього класу (див. лістинг 2.1).

Лістинг 2.1 – Наслідування класу Sprite

```
class Arrow(pygame.sprite.Sprite):
    def __init__(self, image, x, y, angle):
        super().__init__()
        self.original_image = image
        self.angle = angle
        self.image = pygame.transform.rotate(self.original_image,
self.angle - 90)
        self.rect = self.image.get_rect()
        self.rect.center = (x, y)
        #calculate speeds based on the angle:
        self.dx = math.cos(math.radians(self.angle)) *
config.ARROW_SPEED
        self.dy = -math.sin(math.radians(self.angle)) *
config.ARROW_SPEED
```

За допомогою методу super().__init__() клас Arrow успадковує функціональність базового класу Sprite. Це дозволяє автоматично інтегрувати об’єкт стріли до системи спрайтів, використовуючи стандартні механізми обробки, оновлення та відображення графічних об’єктів.

У програмі також залучений клас Group з того ж модуля для збереження та керування декількома об’єктами Sprite. Клас надає кілька корисних методів, наприклад add() та kill(), які використовуються у власному коді (див. лістинг 2.2). Метод add() додає об’єкт класу Sprite до групи, з якої можна керувати цими об’єктами.

Лістинг 2.2 – Використання класів Sprite та Group

```
groups = {
    «arrows»: pygame.sprite.Group(),
    «damage_texts»: pygame.sprite.Group(),
    «items»: pygame.sprite.Group(),
    ««Fireball»s»: pygame.sprite.Group()
}

for item in world.items:
    groups[«items»].add(item)
```

Тут це використовується для додавання предметів (монет та зілля) для того, щоб у майбутньому видаляти їх з гри повністю, якщо гравець їх підібрав. Метод kill() якраз відповідальний за видалення об’єкту з групи. У коді це використовується також для видалення стріл (лістинг 2.3) та вогняних куль.

Лістинг 2.3 – Видалення стріл

```
if (
     self.rect.right < 0 or
     self.rect.left > config.SCREEN_WIDTH or
     self.rect.bottom < 0 or
     self.rect.top > config.SCREEN_HEIGHT
):
     self.kill()
```

Це частина методу класу Arrow, де перевіряється, чи стріла досягла меж вікна гри, і якщо так, то цей об’єкт класу (стріла) видаляється, щоб не перевантажувати програму.

Клас Arrow використовуються класом Weapon, що зображено на діаграмі зв’язком залежності. Зокрема, зброя створює стріли під час атаки (див. лістинг 2.4), але не зберігає їх як власні складові частини, оскільки після створення вони належать до групи спрайтів (це прописується за допомогою методу add() аналогічно до предметів у лістингу 2.2).

Лістинг 2.4 – Створення об’єкта класу Arrow у класі Weapon

```
if (
     pygame.mouse.get_pressed()[0]
     and not self.fired
     and pygame.time.get_ticks() - self.last_shot_time >=
shot_cooldown
):
     arrow = Arrow(self.arrow_image, self.rect.centerx,
self.rect.centery, self.angle)
     self.fired = True
     self.last_shot_time = pygame.time.get_ticks()
```

Наведений фрагмент коду належить методу update() класу Weapon. Коли спрацьовує перевірка на лівий клік мишки, створюється стріла, а також записується час пускання стріли і прапорець fired стає True. Це зроблено для того, щоб контролювати частоту випускання стріл і гравець не міг пускати їх одна за одною. Він змінюється на False, коли користувач відпускає ліву клавішу мишки.

Подібно до цього також існує зв’язок залежності класу Fireball від класу Character. Клас Fireball – спеціальний тип зброї для ворога, а саме головного боса гри. У конструкторі класу Character (див. Додаток Б) ініціалізується булева змінна

is_boss, яка відповідає за те, чи ворог є босом. Тоді у методі mob() цього ж класу (лістинг 2.5) створюється вогняна куля, якщо це значення дорівнює True і якщо відстань між босом і гравцем менша за відведений діапазон попадання ворога. Вона також після створення додається до окремої групи спрайтів, як і стріла.

Лістинг 2.5 – Метод mob() класу Character

```
if self.is_boss and distance < config.BOSS_RANGE:
     «Fireball»_cooldown = 700
     if pygame.time.get_ticks() - self.last_fireball_attack >
fireball_cooldown:
          self.last_fireball_attack = pygame.time.get_ticks()
          angle = math.degrees(math.atan2(player.rect.centery -
self.rect.centery, player.rect.centerx - self.rect.centerx))
          fireball = Fireball(
               fireball_image,
               self.rect.centerx, self.rect.centery,
               player.rect.centerx, player.rect.centery
          )
          return fireball
```

Також клас Character пов’язаний зв’язком композиції з класом Weapon. Це означає, що об'єкт зброї є складовою частиною персонажа та належить йому. Зброя створюється для конкретного персонажа та використовується виключно ним під час гри. Створення зброї відбувається під час ініціалізації гравця (у конструкторі класу Character):

```
if character_type == CharacterType.PLAYER:
     self.bow = Weapon(bow_image, arrow_image)
```

Сам об’єкт класу Character створюється у класі World і пов’язаний зв’язком агрегації. Цей зв'язок показує, що ігровий світ містить персонажів, однак вони можуть розглядатися як окремі об'єкти. Клас World зберігає посилання на гравця та список ворогів і керує їх взаємодією в межах рівня. У методі spawn_units() (див. Додаток В) відбувається створення об’єктів класу Character (як для гравця, так і для ворогів) і збереження їх в класі World (див. лістинг 2.6).

Лістинг 2.6 – Створення об’єктів класу Character

```
elif tile == config.PLAYER_TILE:
```

```
      self.player = Character(
            image_x, image_y,
            config.PLAYER_HEALTH,
            character_images,
            CharacterType.PLAYER,
            bow_image,
            arrow_image)
            tile_data[0] = tile_images[0]
elif tile in (config.ENEMY_DATA.keys()):
    character_type, health, is_boss = config.ENEMY_DATA[tile]
    enemy = Character(
            image_x, image_y, health,
            character_images,
            character_type,
            size = 2 if is_boss else 1,
            boss=is_boss
     )
     self.enemies.append(enemy)
      tile_data[0] = tile_images[0]
```

Тут присвоюється важлива інформація про гравця, наприклад його початковий рівень здоров’я, прописаний у конфігураційному файлі, а також тип персонажа, який береться з окремого класу – CharacterType (лістинг 2.7). Цей клас реалізовано у вигляді переліку (Enum) та призначено для зберігання інформації про доступні типи персонажів у грі. Використання перерахувань дозволяє уникнути застосування рядкових констант у програмному коді та підвищує його надійність і читабельність.

Лістинг 2.7 – Клас CharacterType

```
class CharacterType(Enum):
    PLAYER = "elf"
    IMP = "imp"
    SKELETON = "skeleton"
    GOBLIN = "goblin"
    MUDDY = "muddy"
    TINY_ZOMBIE = "tiny_zombie"
    BIG_DEMON = "big_demon"
```

У цьому ж класі World за такою ж схемою створюються об’єкти класу Item зі всіма необхідними атрибутами (лістинг 2.8).

Лістинг 2.8 – Створення об’єктів класу Item

```
elif tile == config.COIN_TILE:
```

```
      coin = Item(image_x, image_y, "coin", item_images[0])
      self.items.append(coin)
      tile_data[0] = tile_images[0] #remove the tile image since it's
now an item
      elif tile == config.POTION_TILE:
        potion = Item(image_x, image_y, "potion", [item_images[1]])
        self.items.append(potion)
        tile_data[0] = tile_images[1]
```

У лістингу 2.9 наведено сам конструктор класу.

Лістинг 2.9 – Конструктор класу Item

```
def __init__(self, x: int, y: int, item_type: str, animations:
list):
      super().__init__()
      self.item_type = 0 if item_type == "coin" else 1  #0: coin, 1:
health potion
      self.animation_frames = animations
      self.frame_index = 0
      self.update_time = pygame.time.get_ticks()
      self.image = self.animation_frames[self.frame_index]
      self.rect = self.image.get_rect()
      self.rect.center = (x, y)
```

Як можна побачити, тут всього два типи: монети і зілля, тому для вибору потрібного предмету слугує тернарний оператор. При ініціалізації об’єкта тип прописується вручну, це корисно для читабельності коду, та у самому конструкторі залежно від типу предмета змінній item_type присвоюється значення 0 для монети або 1 для лікувального зілля, щоб потім код не виглядав громіздким. Тут не використовується Enum, як у CharacterType, тому що змінних всього дві.

### 2.3 Розробка системи управління станами гри

Для формалізації поведінки ігрових об’єктів та їх взаємодії найкраще підходить діаграма станів, адже показує життєвий цикл одного об'єкта, починаючи з моменту, коли об'єкт вперше з'являється на світ, і рухаючись через усі різні стани, в яких може перебувати об'єкт, доки він не буде викинутий і більше не буде використовуватися [12]. На рисунку 2.4 представлено діаграму станів для власної системи.

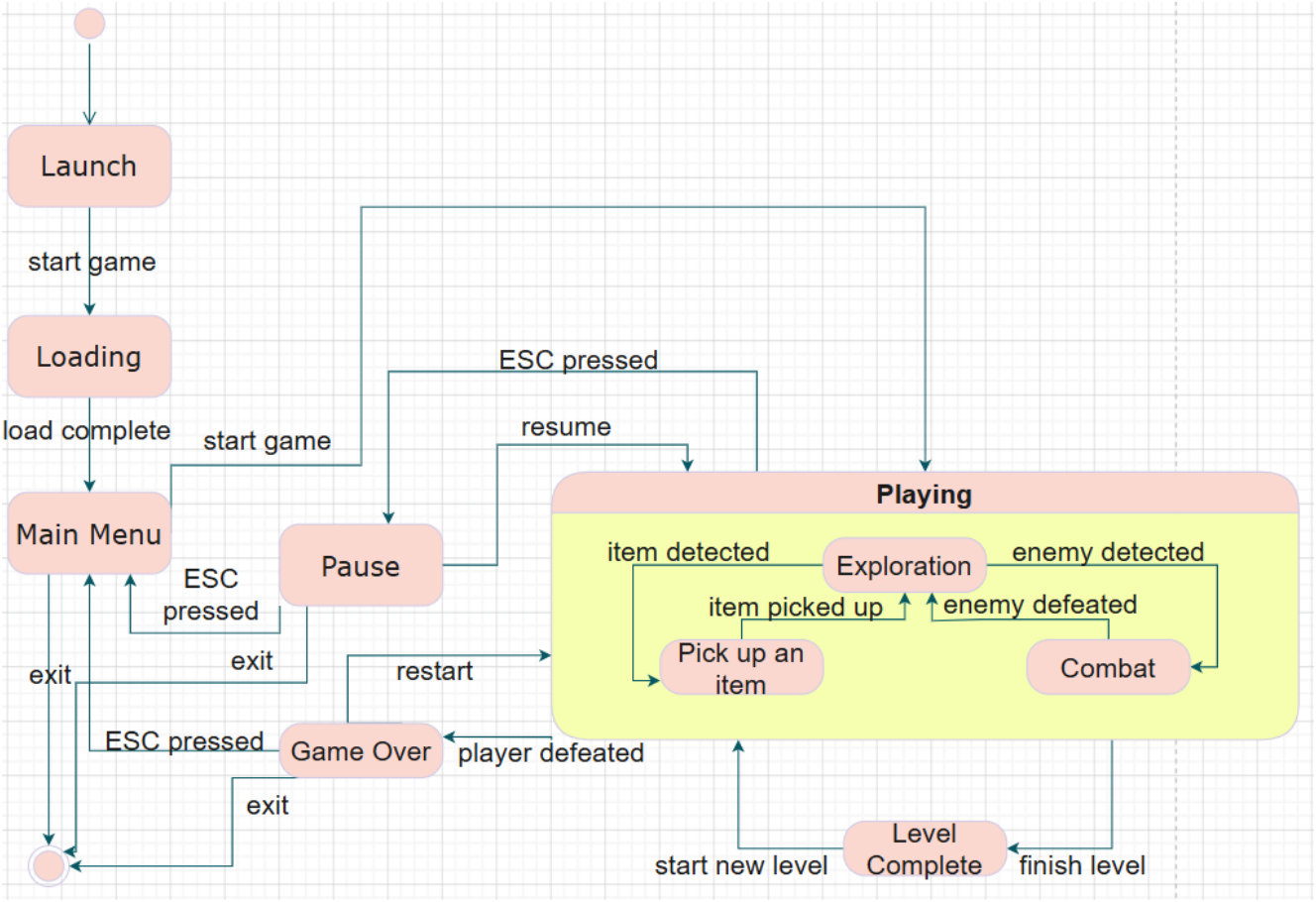


Рисунок 2.4 – Діаграма станів

Описана загальна логіка переходів між основними режимами роботи розробленого 2D ігрового застосунку. Вона відображає не окрему дію персонажа, а цілісний життєвий цикл гри: від моменту запуску програми до завершення ігрової сесії. Такий підхід є доцільним для проєкту жанру «Dungeon Crawler», оскільки в грі одночасно існує кілька рівнів логіки: завантаження ресурсів, робота головного меню, активний ігровий процес, обробка перемоги або поразки, а також повернення до попередніх режимів за діями користувача.

Початковим елементом діаграми є стартовий псевдостан, з якого система переходить у стан «Launch». Цей стан відповідає моменту запуску застосунку, коли ініціалізується середовище виконання, створюється вікно гри, підключаються необхідні модулі Pygame, завантажуються базові налаштування з конфігураційного файлу та готуються основні змінні. На цьому етапі гра ще не є доступною для взаємодії з користувачем як повноцінний ігровий процес, однак уже формується програмна основа для подальшої роботи.

Після запуску, за подією «start game», система переходить у стан «Loading». Цей стан призначений для підготовки ресурсів гри (див. лістинг 2.10): графічних зображень персонажа, ворогів, предметів, тайлів карти, зброї, снарядів, зілля, монети та інших об’єктів, які використовуються під час виконання програми. У контексті реалізованої архітектури цей етап пов’язаний із підготовкою класів World, Character, Item, Weapon, Arrow та Fireball, а також із формуванням груп спрайтів, які надалі забезпечують оновлення і відображення об’єктів на екрані. Після успішного завершення завантаження ресурсів виконується перехід «load complet» безпосередньо до стану «Playing». Це означає, що після підготовки всіх необхідних даних застосунок одразу запускає активний ігровий процес, не вимагаючи від користувача додаткового підтвердження через головне меню.

Лістинг 2.10 – Завантаження необхідних ресурсів

```
images = load_units_images()

world = load_level(images)
#create units:
player = world.player
enemies = world.enemies



groups = {
    "arrows": pygame.sprite.Group(),
    "damage_texts": pygame.sprite.Group(),
    "items": pygame.sprite.Group(),
    "«Fireball»s": pygame.sprite.Group()
}

for item in world.items:
    groups["items"].add(item)
```

Фрагмент коду належить файлу main, який наведений у Додатку Д. Функції load_units_images() та load_level() винесені у окремий файл utils (див. Додаток Е). Завантаження зображень відбувається з локальних файлів.

Стан «Main Menu» (рисунок 2.5) є центральним навігаційним станом поза межами активного ігрового процесу. У ньому користувач може почати гру або завершити роботу застосунку. Перехід «start game» зі стану головного меню переводить систему до стану «Playing», тобто безпосередньо до ігрового процесу.

Перехід «exit» веде до кінцевого стану, що означає завершення виконання програми. Наявність головного меню як окремого стану дозволяє чітко відокремити логіку інтерфейсу від логіки рівня, бо в меню не потрібно оновлювати ворогів, перевіряти бойові зіткнення або виконувати рух снарядів.

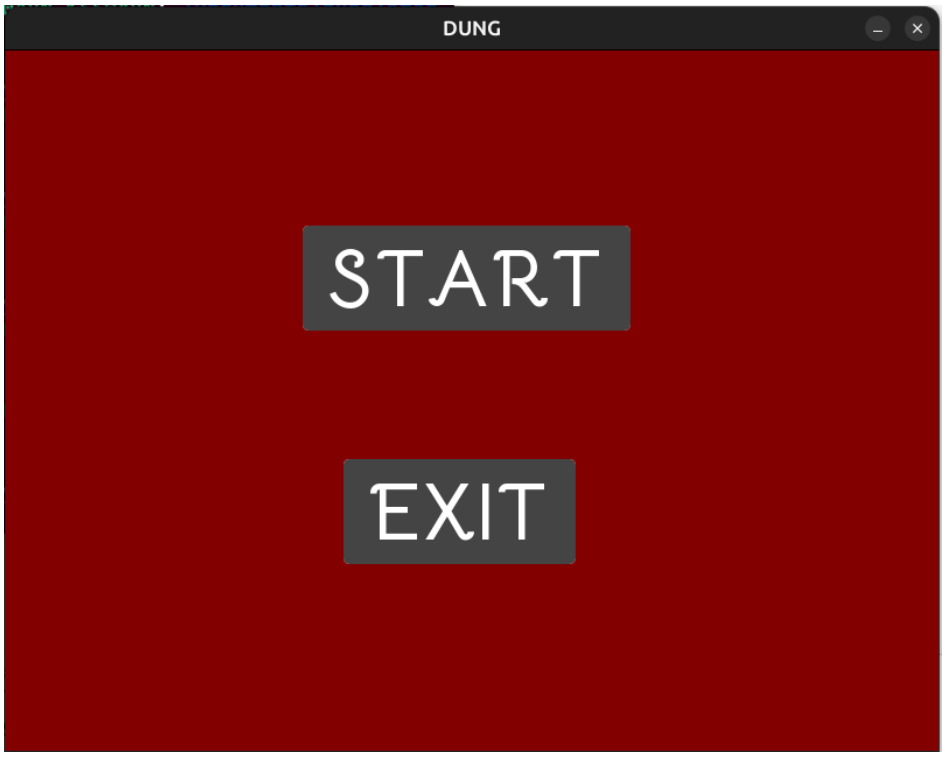


Рисунок 2.5 – Головне меню застосунку

Основним складеним станом діаграми є «Playing». Він охоплює всі дії, що відбуваються під час проходження рівня. У цьому стані працює головний ігровий цикл: обробляються події клавіатури та миші, оновлюється позиція персонажа, перевіряються колізії, рухаються вороги, створюються снаряди, відображаються анімації, оновлюється інтерфейс здоров’я та ігрових ресурсів. Саме в межах цього стану найбільш активно взаємодіють основні класи системи. Клас World відповідає за рівень і розміщення об’єктів, клас Character описує поведінку гравця та ворогів, клас Item забезпечує логіку предметів, а класи Weapon, Arrow і Fireball реалізують механіки бойової взаємодії. На рисунку 2.6 наведено скріншот ігрового процесу.


Рисунок 2.6 – Ігровий процес застосунку

У межах стану «Playing» виділено внутрішні підстани: «Exploration», «Pick up an item» та «Combat». Базовим підстаном є «Exploration», тобто дослідження ігрового світу. У цьому режимі гравець переміщується картою, оглядає рівень, уникає перешкод, наближається до предметів або ворогів. Для жанру така логіка є основною, адже більшість часу гравець перебуває саме в режимі пересування і пошуку наступної взаємодії. У програмній реалізації цей підстан відповідає звичайному оновленню координат персонажа, прокручуванню карти, перевірці меж рівня та відображенню об’єктів. Усе відбувається в головному файлі (main), у циклі while. Умова циклу: поки значення змінної run дорівнює True – виконувати програму. Значення змінюється в залежності від певних умов. Наприклад, у лістингу 2.11 наведено приклад зміни значення від того, які кнопки натиснув гравець.

Лістинг 2.11 – Фрагмент циклу while

```
if not start_game:
    screen.fill(config.MENU_COLOR)
```

```
        start_clicked = start_btn.draw(screen)
        if start_clicked:
            start_game = True
            start_intro = True
        exit_clicked = exit_btn.draw(screen)
        if exit_clicked:
            run = False
else:
        if pause_game:
            screen.fill(config.MENU_COLOR)
            resume_clicked = resume_btn.draw(screen)
            if resume_clicked:
                pause_game = False
            exit_clicked = exit_btn.draw(screen)
            if exit_clicked:
                run = False
```

Тут також згадується змінна start_game. Ця змінна ініціалізується на початку головного файлу і відрізняється від змінної run тим, що відповідає лише за стан «Playing», тобто навіть якщо start_game матиме значення False, програма все одно працюватиме, поки run буде True.

Якщо під час дослідження гравець виявляє предмет, виконується перехід «item detected» до підстану «Pick up an item». Цей стан описує ситуацію, коли персонаж наблизився до монети або лікувального зілля і може взаємодіяти з ним. У реалізації це відповідає перевірці зіткнення прямокутника персонажа з прямокутником предмета (про колізії у підрозділі 2.4). Після успішного підбору предмета відбувається подія «item picked up», і система повертається до стану «Exploration». При цьому предмет видаляється з групи спрайтів методом kill(), а стан персонажа або ігрові показники змінюються: для монети збільшується кількість балів, а для зілля відновлюється здоров’я.

Іншим важливим переходом зі стану «Exploration» є «enemy detected», який переводить систему до підстану «Combat». Цей підстан активується тоді, коли персонаж вступає у взаємодію з ворогом або коли ворог потрапляє в зону виявлення. У межах бойової логіки відбувається перевірка дистанції між персонажем і противником, оновлення поведінки ворога, створення снарядів, нанесення шкоди та відображення тексту отриманих пошкоджень. Для звичайних ворогів бойова взаємодія полягає в переслідуванні гравця або ближній атаці, а для боса додатково реалізовано створення об’єктів класу Fireball. Після знищення

противника виконується перехід «enemy defeated», і система повертається до підстану «Exploration».

Окремо на діаграмі показано можливість переривання активного ігрового процесу. Якщо під час стану «Playing» користувач натискає клавішу «ESC», виконується перехід «ESC pressed» до стану Pause (рисунок. 2.7). Це дозволяє гравцеві вийти з активного рівня без завершення всієї програми. Такий перехід є важливим з точки зору зручності користування, оскільки користувач має змогу повернутися до меню, не очікуючи поразки або завершення рівня.


Рисунок 2.7 – Меню при паузі

Якщо під час гри здоров’я персонажа зменшується до нуля, відбувається перехід «player defeated» до стану «Game Over». Цей стан фіксує завершення поточної ігрової спроби через поразку. У ньому відображається відповідне повідомлення і кнопки для подальших дій (див. рисунок 2.8). Із цього стану передбачено кілька варіантів переходу. Подія «restart» повертає систему до стану «Playing», що означає створення нової ігрової спроби. Подія «ESC pressed» повертає користувача до «Main Menu», а вже звідти за допомогою події «exit» можна завершити роботу застосунку через кінцевий стан.

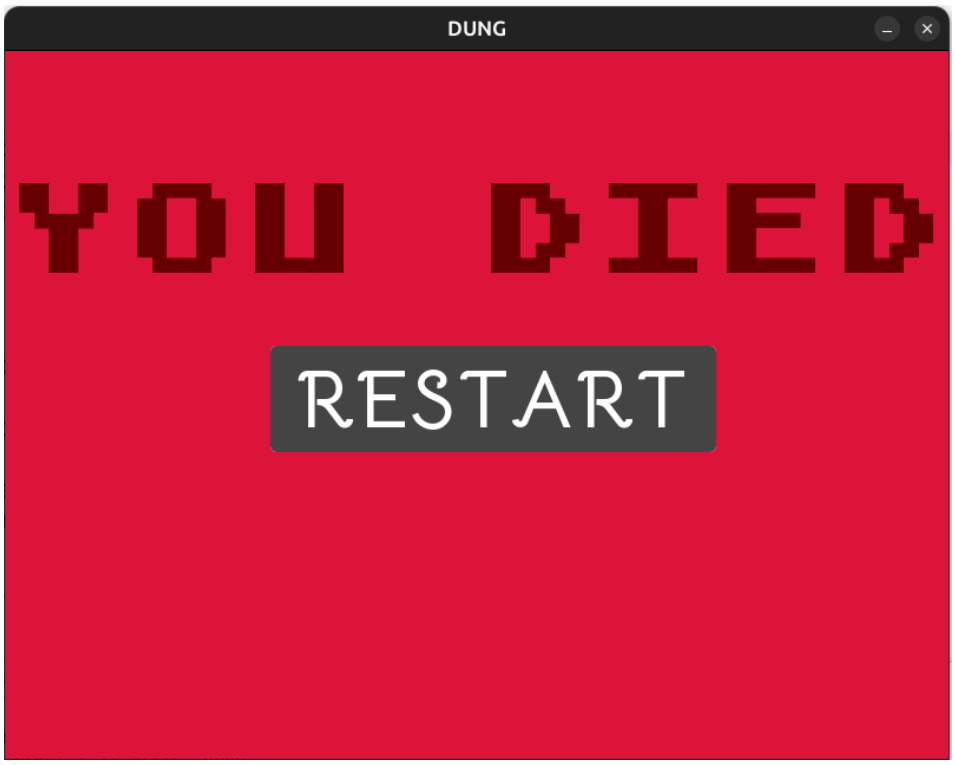


Рисунок 2.8 – Стан Game Over

Також діаграма містить стан «Level Complete», який відповідає успішному завершенню рівня. Перехід «finish level» виконується зі стану «Playing», коли гравець досягає умови проходження: перемагає необхідних ворогів, доходить до виходу або завершує поточну ігрову задачу. Після цього за подією «start new level» система знову переходить до стану «Playing», але вже з новими параметрами рівня. У програмній архітектурі це відбувається з повторним створенням об’єкта класу World, очищенням груп спрайтів, завантаженням нової карти та повторною ініціалізацією персонажів і предметів. У лістингу 2.12 наведено фрагмент коду з переходом на наступний рівень.

Лістинг 2.12 – Перехід на новий рівень

```
if level_complete:
     level += 1
     start_intro = True
     old_health = player.health
     old_score = player.collect_score

     world = load_level(images, groups, level)

     player = world.player
     enemies = world.enemies
```

```
player.health = old_health
player.collect_score = old_score

for item in world.items:
     groups["items"].add(item)
```

Тут видно, що значення рівня збільшується на один, в спеціальних змінних зберігаються старі значення характеристик гравця (здоров’я та кількість зібраних монет), створюється новий світ і значення характеристик переприсвоюються, оскільки світ створює гравця із конфігураційними початковими значеннями, а при переході на наступний рівень потрібно зберігати кількість очок та здоров’я із минулого. Також створюються нові вороги і предмети (монети та зілля).

### 2.4 Реалізація механік взаємодії ігрових об’єктів

Механіки взаємодії ігрових об’єктів у розробленому застосунку побудовані навколо єдиної тайлової системи координат. Тайл-мапи – це дуже популярна техніка у розробці 2D ігор, яка полягає у створенні ігрового світу або карти рівня з невеликих зображень правильної форми, що називаються тайлами. Це забезпечує підвищення продуктивності та економію пам'яті – великі файли зображень, що містять цілі карти рівнів, не потрібні, оскільки вони складаються з невеликих зображень або їх фрагментів, що повторюються [13]. У даному випадку кожен рівень зчитується з CSV-файлу (див. рисунок 2.9) як матриця числових ідентифікаторів, після чого клас World перетворює ці значення на графічні тайли, перешкоди, предмети, гравця та ворогів. Такий підхід дає змогу описувати простір рівня через сітку, а всі подальші перевірки руху, видимості та зіткнень виконувати через прямокутні області pygame.Rect.

Рисунок 2.9 – Приклад CSV-файлу для рівня

Основою цієї сітки є параметр TILE_SIZE, який у конфігураційному файлі обчислюється як $16 * SCALE$. Під час створення світу метод spawn_units() класу World проходить по рядках і стовпцях карти: індекс стовпця множиться на розмір тайла для визначення координати x, а індекс рядка – для координати y. Також тайлу надається відповідне зображення (якщо його значення не дорівнює –1, тоді цей прямокутник вважається пустим і нічого не відбувається), як у лістингах 2.6 та 2.8. Повний фрагмент коду наведений у Додатку В. На рисунку 2.10 показано налагоджувальне відображення сітки, яке візуально демонструє відповідність між клітинками карти та фактичним положенням об’єктів на екрані.

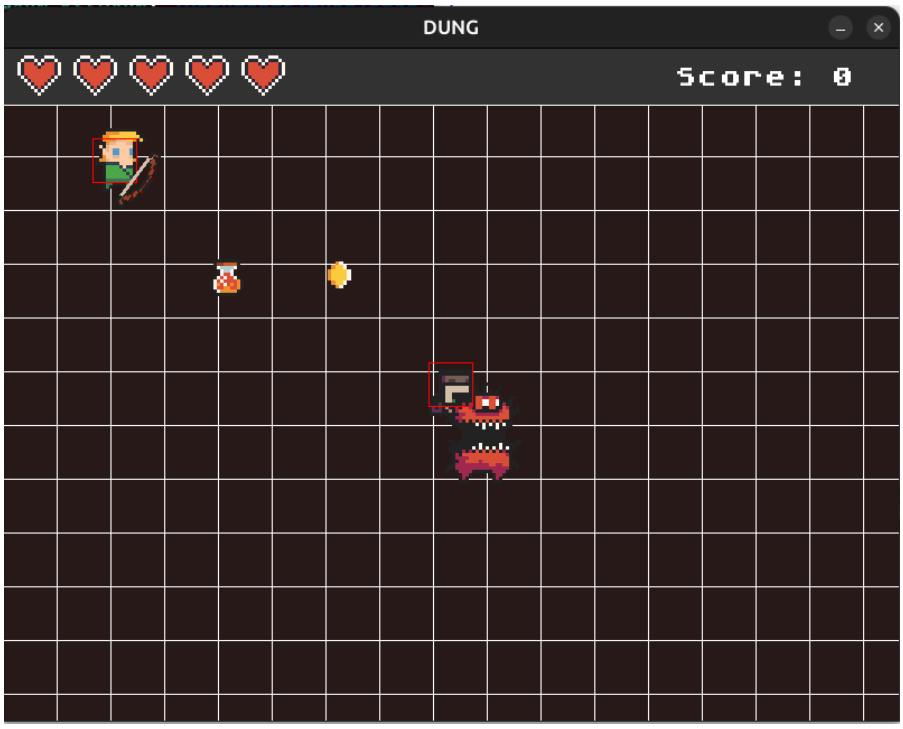


Рисунок 2.10 – Налагоджувальне відображення сітки та меж ігрових об’єктів

Червоні прямокутники на зображенні позначають межі колізій персонажів і об’єктів. Саме ці прямокутники, а не видима форма спрайта, використовуються для перевірки перетинів. Завдяки цьому взаємодія залишається передбачуваною: якщо координати прямокутників перетинаються, гра вважає, що між об’єктами відбувся контакт.

Після завантаження карти тайли стін, що мають ідентифікатор WALL_TILE, додаються до списку obstacle_tiles. Цей список є спільною основою для кількох механік: він обмежує переміщення персонажів, зупиняє стріли та вогняні кулі, а також використовується для перевірки видимості між ворогом і гравцем. На рисунку 2.11 видно, що стіни формують коридори рівня, предмети розміщуються в окремих клітинках, а персонажі залишаються прив’язаними до тієї ж тайлової системи.

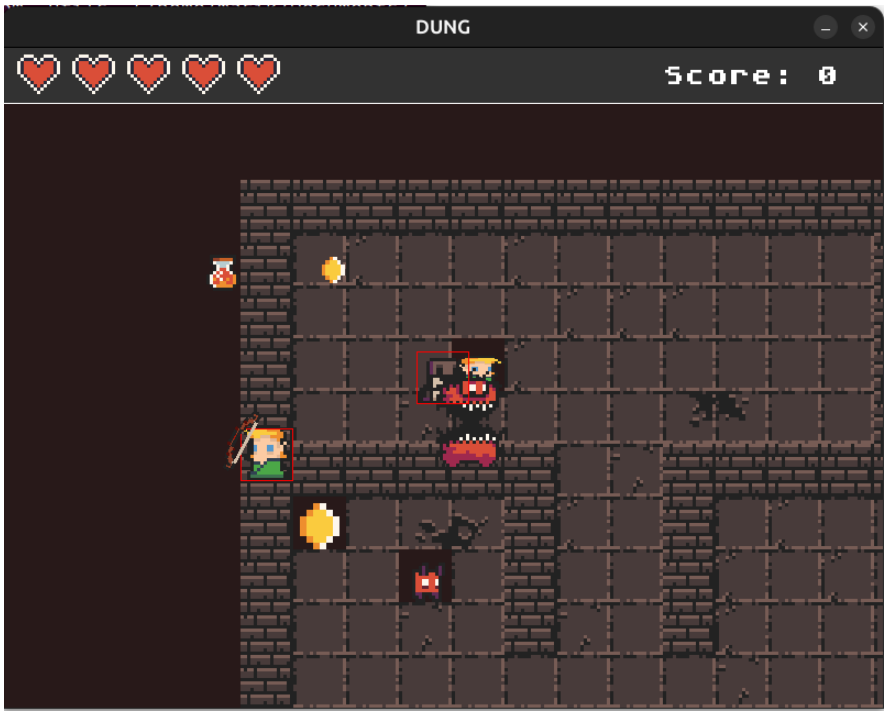


Рисунок 2.11 – Розміщення об’єктів рівня в тайловому середовищі

Колізії під час руху реалізовано у методі move() класу Character (див. лістинг 2.13). Спочатку об’єкт зміщується по осі x, після чого його прямокутник порівнюється з кожним прямокутником перешкоди. Якщо відбулося зіткнення, позиція персонажа коригується: під час руху праворуч його права межа вирівнюється за лівою межею стіни, а під час руху ліворуч – навпаки. Потім така сама перевірка виконується по осі y. Розділення руху на горизонтальну та вертикальну складові дозволяє коректно ковзати вздовж стін і не проходити крізь них.

Лістинг 2.13 – Вираховування колізій при зіткненні зі стіною

```
self.rect.x += dx
for tile in obstacle_tiles:
     if tile[1].colliderect(self.rect):
          if dx > 0: #moving right
               self.rect.right = tile[1].left
          elif dx < 0: #moving left
               self.rect.left = tile[1].right

self.rect.y += dy
for tile in obstacle_tiles:
     if tile[1].colliderect(self.rect):
          if dy > 0: #moving down
               self.rect.bottom = tile[1].top
```

```
        elif dy < 0: #moving up
            self.rect.top = tile[1].bottom
```

Механіка LoS використовується для поведінки ворогів. У методі mob() класу Character формується лінія між центром прямокутника ворога та центром прямокутника гравця (див. рисунок 2.12). Далі кожен прямокутник зі списку obstacle_tiles перевіряється методом clipline(): якщо лінія перетинає стіну, видимість вважається заблокованою. Якщо перешкод немає, ворог може реагувати на гравця та рухатися в його бік.

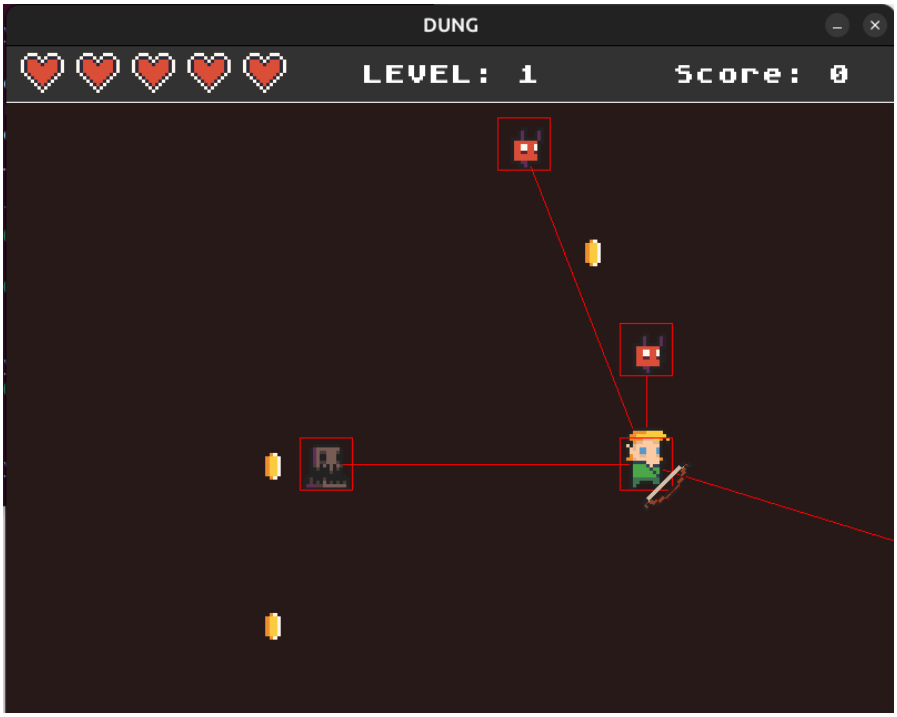


Рисунок 2.12 – Перевірка прямої видимості між ворогами та гравцем

Червоні лінії показують налагоджувальне представлення цієї перевірки. Коли лінія видимості не перетинає стіну, ворог визначає напрямок руху за різницею координат із гравцем. Якщо відстань до гравця менша за ATTACK_RANGE, він завдає шкоди й активує для гравця короткий стан hit, який тимчасово блокує повторне отримання шкоди. Для сильнішого ворога додатково передбачено дистанційну атаку: за умови наближення в межах BOSS_RANGE створюється об’єкт Fireball, що летить у напрямку поточного положення гравця.

Снаряди та предмети використовують ту саму систему прямокутних перетинів. Стріла створюється класом Weapon за напрямком до курсора миші, після чого клас Arrow обчислює її швидкість через кут пострілу. Під час оновлення стріла знищується, якщо виходить за межі екрана, влучає у стіну або перетинається з прямокутником живого ворога. У разі влучання зменшується здоров'я цілі, створюється текст із кількістю шкоди, а ворог тимчасово переходить у стан stun. Аналогічно предмети Item перевіряють перетин із гравцем: монета збільшує рахунок, а зілля відновлює здоров'я в межах максимального значення.

### 2.5 Інтеграція користувацького інтерфейсу та основних ігрових сценаріїв

Інтерфейс – один з важливих елементів розробки 2D застосунків, оскільки забезпечує зв'язок між станом гри та гравцем. Через нього користувач бачить здоров'я персонажа, номер рівня, кількість зібраних монет, а також може запускати гру, ставити її на паузу, відновлювати або завершувати.

Для того, щоб об'єкти не виглядали статично та одноманітно, підвантажуються не одне, а кілька зображень одного й того самого об'єкта, які послідовно відображаються через певний проміжок часу, в даному випадку зображення змінюється кожні 70 мілісекунд. Така зміна кадрів створює ефект анімації руху, що робить візуальне представлення персонажів, предметів та інших елементів гри більш реалістичним і привабливим для користувача. Наприклад, коли гравець рухається ліворуч, береться зображення героя, повернутого ліворуч, аналогічно і з правим боком. Також як для головного персонажа, так і для кожного ворога є по чотири зображення для неактивного та активного стану (див. рисунок 2.13). Зображення взяті з відкритого набору піксельної графіки «16x16 DungeonTileset II» (автор – 0x72) [14]. Усі графічні ресурси поширюються за ліцензією CC0, що дозволяє їх вільне використання у програмних продуктах без порушення авторських прав.

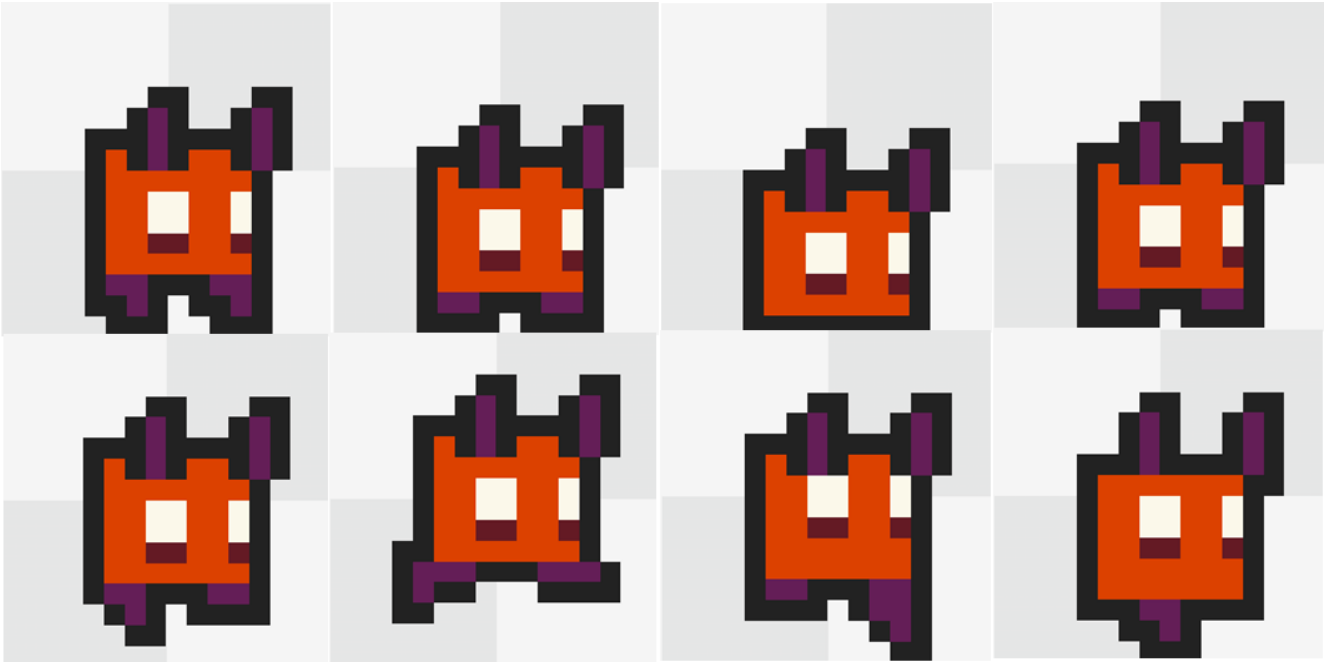

Рисунок 2.13 – Зображення для анімації ворога

У коді зміна анімації в залежності від стану контролюється в межах класу Character для персонажів за допомогою прапорця running:

```
self.update_action(1) if self.running else self.update_action(0) #1: run 0: idle
```

Для предметів також представлені різні зображення, та оскільки вони не можуть рухатись, для них не потрібно реалізовувати логіку зміни анімації.

Усі прямокутники об’єктів: персонаж, вороги, предмети, мапа – беруться із зображень. Для представлення кнопок, з якими може взаємодіяти гравець, також вибрано зображення.

У проєкті відведено окремий файл для класу Button (див. лістинг 2.14).

Лістинг 2.14 – Клас Button

```
class Button():
    def __init__(self, x, y, image):
        self.image = image
        self.rect = self.image.get_rect()
        self.rect.topleft = (x, y)

    def draw(self, surface):
        pressed = False
```

```
        #get mouse position
        position = pygame.mouse.get_pos()

        if self.rect.collidepoint(position) and
pygame.mouse.get_pressed()[0]:
            pressed = True

        surface.blit(self.image, self.rect)

        return pressed
```

Окрім конструктора, тут всього один метод – draw(). Він виводить зображення і перевіряє зіткнення курсора та прямокутника зображення кнопки. Якщо курсор в межах прямокутника та ліва клавіша миші натиснута – прапорець pressed стає True, і метод повертає це значення. Пізніше це значення використовується так у головному файлі:

```
exit_clicked = exit_btn.draw(screen)
     if exit_clicked:
          run = False
```

Також для інтуїтивності і зручності розроблено процес виведення завданої гравцем шкоди на екран. Для перевірки, чи завдано шкоду, використовується метод update() класу Arrow (див. Додаток Ж). У лістингу 2.15 показано, як саме відбувається перевірка на шкоду.

Лістинг 2.15 – Перевірка колізії між стрілою та ворогом

```
damage = 0
damage_pos = None
for enemy in enemies:
    if enemy.rect.colliderect(self.rect) and enemy.alive:
          damage = config.PLAYER_DAMAGE + random.randint(-5, 5)
         enemy.health -= damage
         damage_pos = enemy.rect
         enemy.hit = True
         self.kill()
         break
```

Після цього метод повертає значення damage та damage_pos у головному циклі, і якщо ці значення не пусті, тобто стріла зачепила ворога, відбувається створення об’єкта класу DamageText:

```
     damage_text = DamageText(damage_pos.centerx, damage_pos.y,
damage)
     groups["damage_texts"].add(damage_text)
```

Об’єкт додається в групу спрайтів і пізніше виводиться на екран за допомогою вже розглянутого методу draw().

Інша невід’ємна частина розробки ігор в жанрі «Dungeon Crawler» – це постійне пересування екрану в залежності від того, де знаходиться головний герой, типу камера зверху, яка слідує за гравцем. У коді така функціональність реалізована в методі move() класу Character. У лістингу 2.16 наведено конкретний фрагмент.

Лістинг 2.16 – Механіка пересування екрану

```
#left and right screen scroll:
if self.rect.right > config.SCREEN_WIDTH - config.SCROLL_THRESHOLD:
     screen_scroll[0] = config.SCREEN_WIDTH - config.SCROLL_THRESHOLD
- self.rect.right
     self.rect.right = config.SCREEN_WIDTH - config.SCROLL_THRESHOLD
if self.rect.left < config.SCROLL_THRESHOLD:
     screen_scroll[0] = config.SCROLL_THRESHOLD - self.rect.left
     self.rect.left = config.SCROLL_THRESHOLD

#up and down screen scroll:
if self.rect.bottom > config.SCREEN_HEIGHT -
config.SCROLL_THRESHOLD:
     screen_scroll[1] = config.SCREEN_HEIGHT -
config.SCROLL_THRESHOLD - self.rect.bottom
     self.rect.bottom = config.SCREEN_HEIGHT -
config.SCROLL_THRESHOLD
if self.rect.top < config.SCROLL_THRESHOLD:
     screen_scroll[1] = config.SCROLL_THRESHOLD - self.rect.top
     self.rect.top = config.SCROLL_THRESHOLD
```

Для визначення моменту початку переміщення камери використовується константа SCROLL_THRESHOLD, яка задає граничну відстань від країв екрана. Поки персонаж знаходиться в межах центральної області, екран залишається нерухомим. Коли ж герой наближається до одного з країв вікна гри на відстань, меншу за встановлений поріг, відбувається зміщення ігрового світу у протилежному напрямку.

Спочатку виконується перевірка горизонтального положення персонажа. Якщо права межа прямокутника персонажа (rect.right) перевищує допустиму позицію біля правого краю екрана, обчислюється величина зміщення

screen_scroll[0], після чого координата персонажа обмежується значенням порога. Аналогічно обробляється ситуація, коли персонаж наближається до лівого краю екрана.

Після цього виконується перевірка вертикального положення персонажа. Якщо герой досягає нижньої або верхньої межі області прокручування, розраховується відповідне вертикальне зміщення screen_scroll[1], а положення персонажа коригується таким чином, щоб він залишався в межах встановленої області видимості.

Для створення більш плавних візуальних переходів між різними ігровими станами та покращення загального користувацького досвіду було реалізовано клас ScreenFade (див. Додаток З). Він відповідає за анімацію появи та затемнення екрана під час завантаження рівня або поразки гравця.

У результаті проєктування була сформована модульна архітектура програмного забезпечення, яка забезпечує низьку зв’язаність між компонентами, спрощує підтримку програмного коду та створює основу для подальшого розширення функціональності гри.

Реалізація об’єктно-орієнтованої структури застосунку на основі базових класів бібліотеки Pygame (Sprite та Group) дозволила автоматизувати життєвий цикл ігрових сутностей та забезпечити ефективний контроль за використанням ресурсів (зокрема, через механізм очищення пам'яті від об'єктів, що вийшли за межі екрана). Чіткий поділ системи на логічні модулі, використання різнотипних зв'язків між класами та впровадження типізації за допомогою переліків (Enum) підвищили читабельність вихідного коду, спростили його відлагодження та заклали надійну основу для масштабування проєкту.

Діаграма станів демонструє, що розроблений застосунок має чітко структуровану логіку керування ігровим процесом. Вона відокремлює службові стани гри, такі як запуск, завантаження та меню, від активної логіки рівня. Усередині стану «Playing» додатково деталізовано ключові сценарії поведінки гравця: дослідження, підбір предметів і бій. Така модель спрощує реалізацію головного ігрового циклу, оскільки кожен стан має власну відповідальність і набір

допустимих переходів. Це також підвищує розширюваність застосунку: у майбутньому до діаграми можна додати стан паузи, налаштувань, збереження прогресу, вибору рівня або інвентарю без суттєвої зміни вже реалізованої логіки.

Сітка, видимість і колізії реалізовані не як окремі ізольовані елементи, а як частини однієї просторової моделі. Тайлова карта задає розташування об’єктів, pygame.Rect забезпечує універсальні межі взаємодії, а перевірки визначають фізичні контакти, перешкоди для руху та умови реакції ворогів на гравця. Завдяки цьому ігровий світ поводиться послідовно.

Інтеграція користувацького інтерфейсу, механіки плавного скролінгу камери та систем динамічного фідбеку дозволила об’єднати розрізнені підсистеми в цілісний інтерактивний комплекс. Створення універсального класу компонентів керування та реалізація алгоритму стеження за гравцем відносно встановленого порогу меж екрана сформували ергономічне та інтуїтивно зрозуміле середовище взаємодії користувача із застосунком, забезпечуючи динамічність геймплею відповідно до вимог жанру.

# 3 ТЕСТУВАННЯ ТА АНАЛІЗ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

У даному розділі розглянуто процес тестування та оцінювання якості розробленого ігрового застосунку. Описано організацію процесу тестування, наведено підходи до перевірки працездатності окремих компонентів системи та проаналізовано коректність реалізації основних ігрових механік. Особливу увагу приділено модульному тестуванню ключових елементів програмного забезпечення, перевірці роботи системи керування станами гри, а також аналізу отриманих результатів. На основі проведеного тестування виконано оцінку якості програмного продукту та визначено ступінь відповідності реалізованого застосунку поставленим функціональним вимогам.

## 3.1 Організація процесу тестування програмного забезпечення

Тестування ігор – це перевірка ігрових проєктів на помилки, баги та інші проблеми в роботі для подальшого їх усунення. Допомагає забезпечити стабільність, зручність і якість продукту. Зазвичай тести проводяться в процесі розробки та після її завершення перед релізом [15].

Більшість тестів застосунок «DUNG» пройшов протягом самої розробки, адже це та область, в якій можна побачити більшість багів ще на ранніх етапах життєвого циклу розробки програмного забезпечення.

Наприклад, той самий діагональний рух. Це те, що найбільше кидається в очі при розробці. Якщо гравець одночасно натискає дві клавіші переміщення, наприклад вгору та вправо або вниз та вліво, рух відбувається одночасно по двох осях координат. У такому випадку результуюча швидкість визначається як сума двох векторів руху. Згідно з теоремою Піфагора, горизонтальна та вертикальна складові утворюють катети прямокутного трикутника, а фактична швидкість діагонального руху відповідає довжині його гіпотенузи. Нехай швидкість руху вздовж кожної з координатних осей дорівнює $v$. Тоді при одночасному русі по горизонталі та вертикалі модуль результуючого вектора швидкості становитиме $v\sqrt{2}$. Звідси можна зробити висновок, що будь-яка діагональ завжди на $\sqrt{2}$ довша за

горизонтальну або вертикальну лінію. Це означає, що персонаж буде рухатися по діагоналі приблизно на 41% швидше, ніж по одній осі, що призведе до некоректної ігрової механіки. Для усунення цього ефекту виконується нормалізація вектора руху. Під час діагонального переміщення горизонтальна та вертикальна складові швидкості множаться на коефіцієнт $1/\sqrt{2}$, або $\sqrt{2}/2$. У результаті модуль результуючого вектора залишається рівним $v$, що забезпечує однакову швидкість пересування персонажа незалежно від обраного напрямку руху [16]. У методі move() класу Character це виглядає так:

```
if dx and dy:
     dx *= math.sqrt(2)/2
     dy *= math.sqrt(2)/2
```

Також під час створення кількох рівнів було помічено баг, пов'язаний з переходом через драбину на новий рівень. Якщо гравець підходить до сходів, занадто близько, він може випадково перейти на новий рівень, не завершивши все на попередньому, а світ розроблений так, що шляху назад немає. Тому окремо перевіряється не тільки перетин із тайлом сходів, а й відстань до його центра (див. лістинг 3.1). Це запобігає випадковому завершенню рівня при дотику до краю тайла.

Лістинг 3.1– Перевірка колізії між гравцем та сходами

```
if ladder_tile and ladder_tile[1].colliderect(self.rect):
     ladder_dist = math.hypot(
          (self.rect.centerx - ladder_tile[1].centerx),
          (self.rect.centery - ladder_tile[1].centery)
     )
     if ladder_dist < config.LADDER_RANGE:
          level_complete = True
```

Наведений фрагмент коду демонструє двоетапну перевірку взаємодії гравця з точкою виходу з рівня, що ефективно вирішує проблему випадкових переходів. Спочатку за допомогою методу colliderect() перевіряється базове перетинання прямокутника гравця (self.rect) з прямокутником тайла сходів. Якщо базове зіткнення зафіксовано, програма не змінює ігровий стан одразу, а переходить до

другого етапу перевірки просторового розташування. За допомогою функції math.hypot() з вбудованого модуля Python обчислюється Евклідова відстань між геометричними центрами обох об'єктів. Для цього розраховується різниця між координатами centerx та centery гравця і відповідними координатами тайла сходів.

Отримане значення відстані (ladder_dist) порівнюється із заданим у конфігураційному файлі порогом чутливості – константою LADDER_RANGE. Лише у тому випадку, коли обчислена відстань виявляється меншою за цей поріг, система розцінює дію як цілеспрямовану. Тоді змінна level_complete набуває значення True, що дає сигнал головному циклу згенерувати новий ігровий світ. Такий підхід залишає гравцеві простір для маневру, дозволяючи безпечно переміщуватися або вести бій у безпосередній близькості до сходів, не побоюючись передчасно завершити поточний етап гри через дотик до краю тайла.

### 3.2 Функціональне тестування

Функціональне тестування – один із видів тестування, спрямованого на перевірку відповідностей функціональних вимог ПЗ його реальним характеристикам. Основним завданням функціонального тестування є підтвердження того, що програмний продукт, який розробляється, володіє усім необхідним функціоналом [17].

Написання тест-планів як складової функціонального тестування забезпечує формалізований опис об’єкта тестування, визначає стратегію перевірки, порядок виконання тестових процедур, а також критерії початку і завершення тестування. Тест-плани формуються на основі варіантів використання, наведених у підрозділі 1.5, що дозволяє забезпечити повне покриття ключових сценаріїв роботи системи з точки зору кінцевого користувача. Нижче наведено кілька тест-планів.

1. Перехід на новий рівень через сходи.

Summary: Перевірити, що гравець може перейти на наступний рівень при досягненні виходу (сходів).

Preconditions: Користувач запустив гру та знаходиться на активному рівні (див. рисунок 3.1).

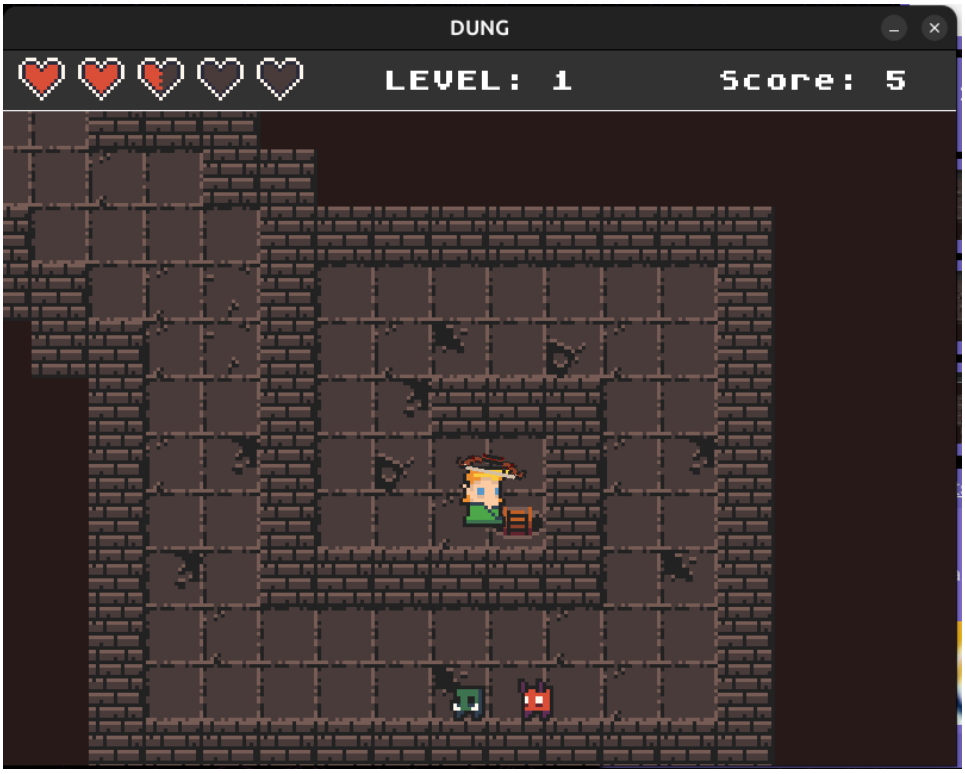


Рисунок 3.1 – Персонаж знаходиться на першому рівні

Steps to reproduce:

1) Керувати персонажем за допомогою клавіш руху (WASD).
2) Дістатися до області розташування сходів.
3) Перетнути центр тайла сходів.

Expected results: Система встановлює level_complete = True та ініціює завантаження нового рівня.

Postconditions: Гравець переноситься на наступний рівень, карта оновлюється (див. рисунок 3.2).

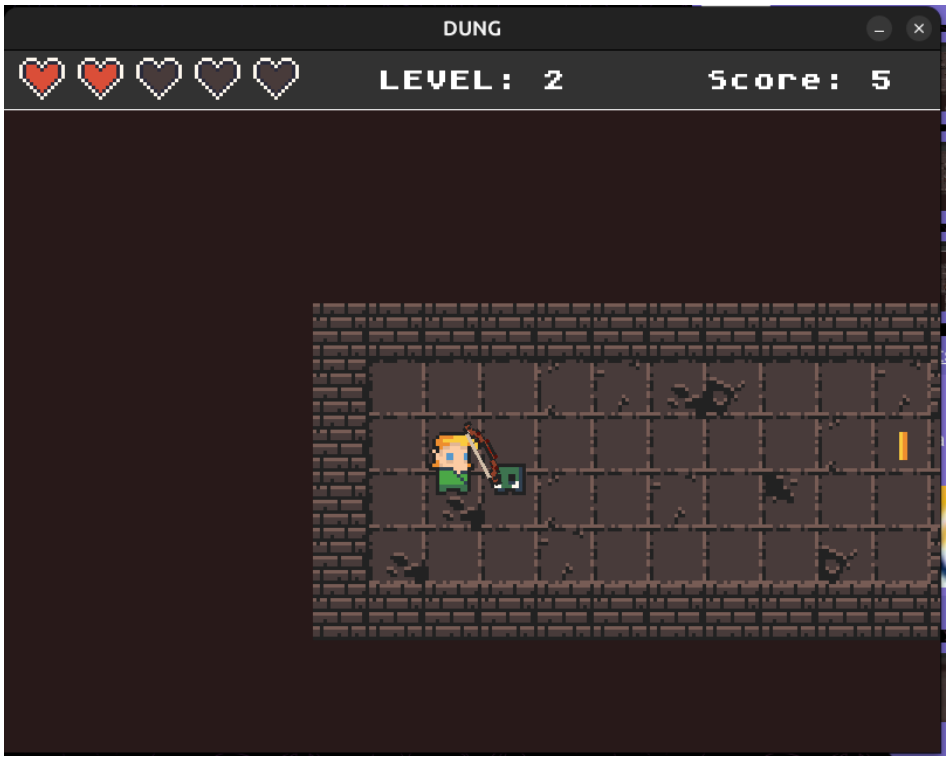


Рисунок 3.2 – Персонаж знаходиться на другому рівні

2. Збір ігрових предметів (монети).

Summary: Перевірити, що гравець отримує очки при зборі монет.

Preconditions: Гравець знаходиться на рівні, де присутні монети (див. рисунок 3.3).

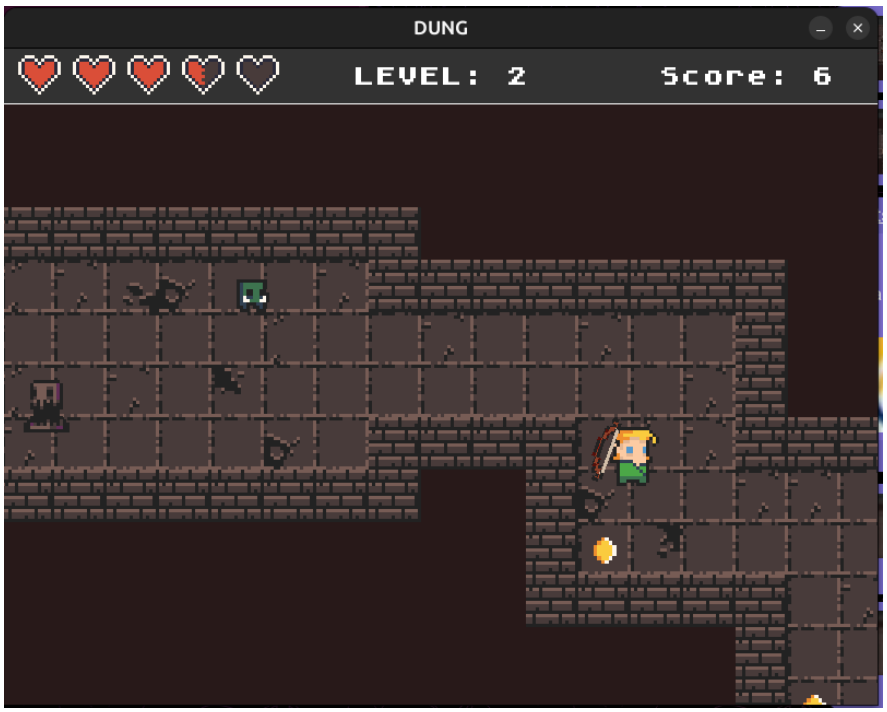


Рисунок 3.3 – Персонаж знаходиться біля монети

Steps to reproduce:

1) Перемістити персонажа в напрямку монети.
2) Перетнути межу колізії монети.
3) Перевірити оновлення рахунку.

Expected results: Монета зникає зі світу гри, значення collect_score збільшується на 1 (див. рисунок 3.4).


Рисунок 3.4 – Персонаж підібрав монету

Postconditions: Монета більше не відображається на мапі.

3. Стрілянина та знищення ворога

Summary: Перевірити коректність взаємодії стріли та ворога.

Preconditions: На рівні присутній ворог та активна зброя гравця.

Steps to reproduce:

1) Навести приціл миші на ворога.
2) Натиснути ліву кнопку миші для пострілу.
3) Дочекатися зіткнення стріли з ворогом.

Expected results: Стріла зникає після зіткнення, ворог отримує шкоду (health зменшується), відображається damage text.

Postconditions: При достатній шкоді ворог помирає і більше не може атакувати (рисунок 3.5).


Рисунок 3.5 – Ворог знешкоджений і не може атакувати

4. Отримання шкоди.

Summary: Перевірити коректність обробки ситуації, коли гравець отримує шкоду від ворога або снаряда.

Preconditions: Гравець знаходиться на рівні, на якому присутній ворог.

Steps to reproduce:

1) Розташувати персонажа в зоні атаки ворога.
2) Дочекатися моменту зіткнення.
3) Зафіксувати момент отримання шкоди.
4) Спостерігати за зміною стану здоров’я персонажа.

Expected results:

1) Значення здоров’я гравця зменшується на задану величину шкоди (ENEMY_DAMAGE).
2) Активується стан player.hit = True.
3) Запускається таймер невразливості (last_hit_time).
4) Візуальна індикація отримання шкоди (зменшення кількості сердець на інформаційній панелі) (див. рисунок 3.6).

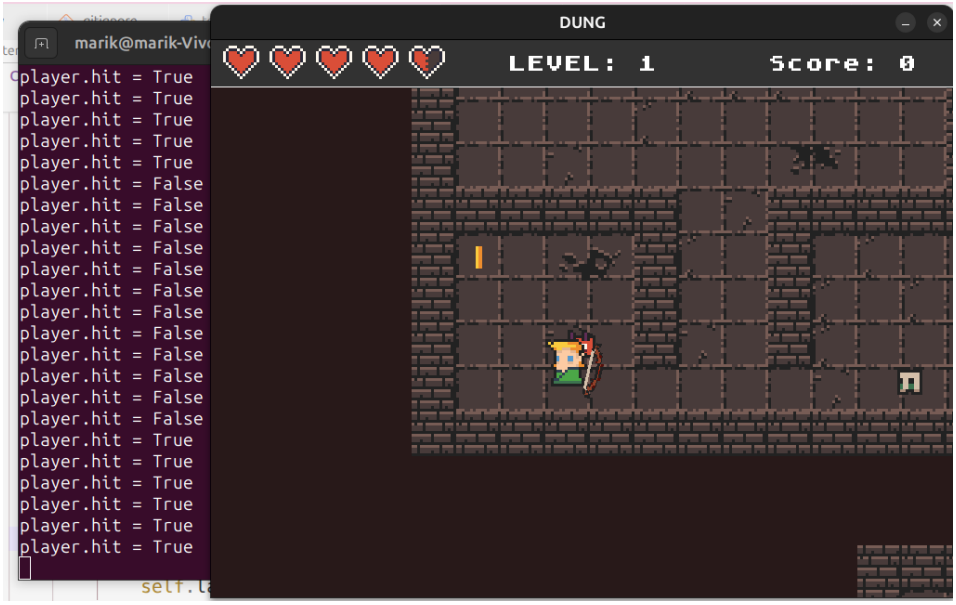


Рисунок 3.6 – Процес тестування отримання ушкоджень

Postconditions: Персонаж залишається живим, якщо health більше 0, або переходить у стан смерті (alive = False) при досягненні нуля.

### 3.3 Модульне тестування

Модульне тестування – тестування кожної атомарної функціональності додатку окремо, в штучно створеному середовищі. Саме потреба у створенні штучного робочого середовища для певного модуля вимагає знань в автоматизації тестування програмного забезпечення та деяких навичок програмування [18].

У межах цієї роботи модульне тестування було реалізовано для ключових компонентів ігрової системи «DUNG» із використанням стандартного модуля unittest мови Python. Основною метою тестування було перевірити коректність роботи базових механік гри, зокрема взаємодії ігрових об’єктів, системи завантаження рівнів, логіки руху персонажа та обробки ігрових подій у контрольованому середовищі.

Для забезпечення ізоляції тестованих модулів використовувалися штучно створені об’єкти (заглушки), зокрема спрощені графічні поверхні pygame.Surface, які замінювали реальні ігрові ресурси. Це дозволило виконувати тести без необхідності повного запуску ігрового рушія та графічного інтерфейсу.

У класі TestGameCore створено кілька методів тестування, один з яких тестує колізії зі стіною (лістинг 3.2).

Лістинг 3.2 – Фрагмент методу тестування на колізії зі стіною

```
def test_character_collision_with_wall(self):
     log.info("\n[TEST] Character collision with wall")
     fake_image = pygame.Surface((10, 10))
     char = Character(
          x=100, y=100,
          health=100,
          characters_animations={ CharacterType.PLAYER:
          [[fake_image]]},
          character_type=CharacterType.PLAYER,
          bow_image=fake_image,
          arrow_image=fake_image)

     wall_rect = pygame.Rect(120, 100, 50, 50)
     obstacle_tiles = [[None, wall_rect]]

     log.info(" - Moving character into wall...")
     char.move(10, 0, obstacle_tiles)
     log.info(f" - Character position: {char.rect}")
     log.info(f" - Wall position: {wall_rect}")
     self.assertFalse(char.rect.colliderect(wall_rect))
     log.info("[OK] No collision detected")
```

Цей тест моделює ситуацію взаємодії персонажа з об’єктом-бар’єром у контрольованому середовищі без запуску повного ігрового циклу. Для цього створюється спрощений графічний об’єкт pygame.Surface, який використовується як заглушка для анімацій персонажа, а також прямокутна область pygame.Rect, що імітує стіну на ігровій мапі.

Після ініціалізації об’єктів виконується переміщення персонажа у напрямку перешкоди методом move(). У процесі тестування перевіряється, що після виконання переміщення відсутнє перетинання (колізія) між прямокутником персонажа та прямокутником стіни. Це гарантує коректну роботу системи фізичних обмежень та запобігає проходженню гравця крізь тверді об’єкти рівня.

Додатково у тесті використовується логування ключових етапів виконання, що дозволяє відстежувати зміну координат персонажа та стану колізії під час тестування (див. рисунок 3.7).

```
test_arrow_out_of_bounds (test.TestGameCore.test_arrow_out_of_bounds) ...
[TEST] Arrow out of bounds
 - Arrow created at (900, 300)
 - Arrow alive: False
[OK] Arrow removed outside screen
ok
test_character_collision_with_wall (test.TestGameCore.test_character_collision_with_wall) ...
[TEST] Character collision with wall
 - Moving character into wall...
 - Character position: <rect(200, 200, 48, 48)>
 - Wall position: <rect(120, 100, 50, 50)>
[OK] No collision detected
ok
```

Рисунок 3.7 – Виведення результатів тестування у консоль

На таблиці 3.1 наведено результати усіх реалізованих модульних тестів основних компонентів гри.

Таблиця 3.1 – Результати модульного тестування

| № | Назва тесту | Функціональність | Очікуваний результат | Результат тестування |
|---|---|---|---|---|
| 1 | test_character_collision_with_wall | Колізія персонажа зі стінами | Персонаж не проходить перешкоди | Успішно |
| 2 | test_arrow_out_of_bounds | Видалення стріли за межами карти | Снаряд знищується при виході за екран | Успішно |
| 3 | test_world_spawn_units_basic_map | Завантаження карти та створення об’єктів | Коректне створення гравця та тайлів | Успішно |
| 4 | test_world_has_required_lists | Ініціалізація базових структур світу | Усі списки ініціалізуються пустими | Успішно |
| 5 | test_level_reload_after_death | Перезавантаження рівня після смерті | Створюється новий світ без старих об’єктів | Успішно |

### 3.4 Розгортання програмного забезпечення та системні вимоги

Оскільки програмна система реалізована графічний застосунок (віконна гра) на мові Python, вона не потребує складного апаратного забезпечення.

Операційна система: Windows 10 / 11, Linux або macOS.

Процесор: будь-який сучасний процесор (Intel/AMD від 1.5 GHz).

Оперативна пам’ять: від 2 GB.

Місце на диску: до 100 МВ.

Програмне забезпечення: Python 3.10 і новіше; бібліотеки, що використовуються у проєкті, встановлені через pip (файл requirements.txt).

Розгортання програмної системи здійснюється локально на комп’ютері користувача без використання серверної частини.

Для запуску гри необхідно:

1. Встановити Python з офіційного сайту [19].

2. Завантажити вихідний код програми.

3. Встановити залежності командою:

```
pip install -r requirements.txt
```

4. Запустити головний файл програми:

```
python main.py
```

Після запуску програми відкривається вікно гри і користувач взаємодіє з грою через клавіатуру та мишку.

### 3.5 Верифікація програмного забезпечення

Верифікація програмної системи ігрового застосунку «DUNG» виконувалася з метою підтвердження відповідності реалізованого функціоналу встановленим вимогам, а також перевірки коректності роботи основних ігрових механік. У межах верифікації було застосовано комплексний підхід, що включав функціональне та модульне тестування ключових компонентів системи.

Основним методом перевірки стало функціональне тестування, яке базувалося на тест-планах, сформованих відповідно до основних сценаріїв використання програмного продукту. У процесі тестування перевірялася коректність роботи ігрового застосунку в умовах реального використання, зокрема запуск графічного інтерфейсу, керування персонажем, взаємодія з об’єктами

ігрового світу, проходження рівнів, бойова система та отримання шкоди. Результати підтвердили, що основні сценарії виконуються коректно та відповідають вимогам.

Додатково було проведено модульне тестування окремих компонентів системи. Перевірці підлягали такі елементи, як система колізій, логіка руху персонажа, завантаження рівнів, обробка ігрових снарядів та ініціалізація ігрового світу. Окрему увагу приділено тестуванню крайових ситуацій та потенційно проблемних сценаріїв, зокрема взаємодії персонажа з перешкодами та переходу між рівнями.

Отримані результати верифікації підтверджують, що ігровий застосунок «DUNG» функціонує відповідно до поставлених вимог, забезпечує коректну роботу основних механік та демонструє стабільну поведінку під час експлуатації. Виявлені під час розробки недоліки були усунуті, що забезпечило загальну якість та надійність програмного продукту.

У третьому розділі виконано тестування, верифікацію та аналіз якості ігрового застосунку «DUNG», що включало організацію процесу тестування, перевірку основних функціональних сценаріїв, модульне тестування ключових компонентів, а також визначення системних вимог і способу розгортання програмної системи. Отримані результати підтвердили коректність реалізації ігрових механік, стабільність роботи застосунку та відповідність його функціоналу поставленим вимогам.

# 4 БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ ТА ОСНОВИ ОХОРОНИ ПРАЦІ

Безпека життєдіяльності – це навчальна дисципліна, сфера наукових знань, яка охоплює теорію та практику захисту людини від небезпечних і шкідливих факторів у всіх сферах людської діяльності, збереження безпеки та здоров’я людини в середовищі проживання. Її завданням є набуття знань про діяльність людини в усіх сферах суспільства як особистості, без чого не має мети суспільний розвиток [20].

Головне завдання науки про безпеку життєдіяльності полягає в убезпеченні людського життя та здоров'я в умовах сучасного техногенного середовища, а також у формуванні сприятливого та комфортного простору шляхом мінімізації загроз природного, антропогенного й технічного характеру [20].

Охорона праці – це система правових, соціально-економічних, організаційно-технічних, санітарно-гігієнічних та лікувально-профілактичних заходів і засобів, спрямованих на збереження здоров'я та працездатності людини в процесі праці. Вона відіграє важливу роль як соціальний чинник, оскільки, якими б вагомими не були трудові здобутки, вони не компенсують людині втраченого здоров'я, а тим паче життя [21].

## 4.1 Характеристика життєдіяльності людини у системі «людина – машина – середовище існування»

У сучасній науці про безпеку життєдіяльності використовується поняття системи «людина – середовище проживання», яке є основою наукового знання і описує процеси взаємодії людини (колективу чи населення) з навколишнім середовищем. Це базова система взаємодії з навколишнім світом. Однак у результаті активної техногенної діяльності людини був створений новий тип середовища перебування – техносфера. Створюючи її, людина мала на меті поліпшення свого середовища проживання. Однак створена руками й розумом людини техносфера багато в чому не виправдала очікувань, тому що те середовище,

яке б мало полегшити існування, стало небезпечним для людства. Саме тому останніми часом активно розвивається вчення про безпеку життєдіяльності саме в техносфері, головною метою якого є захист людини в техносфері від негативних впливів антропогенного і природного походження, забезпечення комфортних умов життєдіяльності [20].

Це питання особливо є актуальним у сфері інформаційних систем та технологій. Робітники цієї сфери – це люди, які постійно взаємодіють з машинами: комп'ютери, ноутбуки, телефони та інші технічні прилади, необхідні для розробки програмного забезпечення. Це, звісно, не єдина сфера, де працівники використовують технології, та одна з найбільш показових, де взаємодія в системі «людина – машина» є настільки тісною, безперервною та інформаційно насиченою.

У контексті професійної діяльності фахівців ІТ-сфери кожен з елементів системи «людина – машина – середовище існування» має свої специфічні особливості:

1. Людина – головний суб'єкт праці, інтелектуальна ланка системи. У цій сфері людина виконує складну розумову роботу, що супроводжується високим рівнем нервово-емоційного напруження. Від працівника вимагається постійна концентрація уваги, здатність обробляти великі масиви інформації, аналізувати код та швидко приймати рішення. Основне фізіологічне навантаження під час такої праці припадає на центральну нервову систему, зоровий аналізатор (через постійну фіксацію погляду на екрані) та опорно-руховий апарат (через тривалу статичну позу).

2. Машина – у галузі інформаційних технологій це поняття є комплексним і охоплює як апаратне забезпечення (персональні комп'ютери, сервери, монітори високої роздільної здатності, клавіатури, миші), так і програмне забезпечення (середовища розробки, операційні системи, бази даних). Специфіка «машини» тут полягає в тому, що взаємодія відбувається через людино-машинний інтерфейс, який повинен бути інтуїтивно зрозумілим, щоб мінімізувати кількість помилок і не створювати додаткового психологічного дискомфорту («інформаційного шуму»).

3. Середовище існування (виробниче середовище) – фізичні та санітарно-гігієнічні умови, в яких здійснюється робочий процес.

Дисбаланс або порушення взаємодії у цій системі призводить до негативних наслідків для здоров'я людини. Основні шкідливі виробничі фактори в ІТ-сфері включають: гіподинамію (малорухливий спосіб життя), статичне перенапруження м'язів шиї, плечового пояса та спини, синдром «сухого ока», порушення зору, а також психологічне вигорання. Саме тому важливо приділяти багато уваги організації безпечних і здорових умов праці, що є одним із головних завдань охорони праці.

### 4.2 Психофізичні чинники небезпеки та як їх уникнути

Успішна профілактика виробничого травматизму та професійної захворюваності можлива лише при умові ретельного вивчення причин їх виникнення, тож їх поділяють на наступні основні групи: організаційні, технічні, санітарно-гігієнічні, економічні, психофізіологічні [21].

До психофізичних чинників належать:

- помилкові дії внаслідок втоми працівника через надмірну важкість і напруженість роботи;
- монотонність праці;
- хворобливий стан працівника;
- необережність;
- невідповідність психофізіологічних чи антропометричних даних працівника використовуваній техніці чи виконуваній роботі;
- незадоволення роботою;
- несприятливий психологічний мікроклімат у колективі [21].

З усього вищеперечисленого можна зробити висновок про те, що причиною тут стає саме психічний стан, а погіршення психічного здоров’я людини часто є наслідком розумової діяльності, не завжди, але в більшості випадків.

Відомо, що при такій діяльності у роботі беруть участь центральна нервова система та органи чуття. При розумовій діяльності збільшується тиск, уповільнюється частота серцебиття, послаблюються обмінні процеси та зменшується кровопостачання кінцівок. Водночас, порівняно з фізичною

діяльністю при окремих видах розумової діяльності напруженість органів чуття зростає в 5-10 разів [21]. Це означає, що при довготривалій розумовій діяльності насамперед важливо дотримуватись строгих вимог щодо навколишнього середовища людини, яка працює.

Робота розробника програмного забезпечення – це виключно розумова праця, яка якщо і має причини небезпеки, то зазвичай психофізичні. Це людина, результат роботи якої безпосередньо залежить від продуктивності її мозку. Постійно робота за комп’ютером чи ноутбуком може стати причиною розвитку низки специфічних професійних захворювань та порушень здоров'я. Насамперед це впливає на фізичний стан:

1. Опорно-руховий апарат: тривале перебування у статичній позі призводить до остеохондрозу, болю в шиї та спині, порушення постави. Окремою проблемою є карпальний тунельний синдром (синдром зап'ястного каналу) – защемлення нерва через постійну роботу з мишкою та клавіатурою.

2. Зір: постійне фокусування на екрані викликає перенапруження очних м'язів, синдром «сухого ока», зниження гостроти зору та спазм акомодації.

3. Загальна гіподинамія: малорухливий спосіб життя сповільнює обмін речовин, погіршує кровообіг (зокрема й кровопостачання мозку) та підвищує ризик серцево-судинних захворювань.

Окрім фізичних чинників, величезне значення мають психологічні навантаження. Високий рівень відповідальності, необхідність постійно вивчати нові технології, концентрація на складному коді, жорсткі дедлайни та малорухливість призводять до хронічної втоми та порушень сну, емоційного та професійного вигорання, підвищення рівня тривожності або розвитку депресивних станів.

Щоб уникнути подібного розвитку подій, працедавцям та робітниками важливо дотримуватись гігієни праці. Це галузь практичної і наукової діяльності, що вивчає стан здоров'я працівників у його обумовленості умовами праці і на цій основі обґрунтовує заходи і засоби щодо збереження і зміцнення здоров'я працівників, профілактики несприятливого впливу умов праці [21]. Наприклад, для розробника у такому випадку способами уникнути перевтоми, вигорання чи

фізичних захворювань є облаштування ергономічного робочого місця, регулярні перерви, займатися спортом та дбати про якісний сон.

У цьому розділі розглянуто питання безпеки життєдіяльності та охорони праці в контексті професійної діяльності розробника програмного забезпечення. Проаналізовано особливості функціонування системи «людина – машина – середовище існування», визначено роль кожного її елемента та охарактеризовано основні шкідливі фактори, що виникають під час роботи з комп’ютерною технікою.

Також досліджено психофізичні чинники небезпеки, характерні для працівників ІТ-сфери, зокрема вплив тривалої розумової праці, статичних навантажень, гіподинамії та емоційного перенапруження. Встановлено, що дотримання принципів ергономіки, раціонального режиму праці та відпочинку, а також заходів гігієни праці сприяє збереженню здоров’я, підвищенню працездатності та зниженню ризику професійних захворювань.

# ВИСНОВКИ

У кваліфікаційній роботі бакалавра розв’язано актуальну науково-технічну задачу – проєктування, програмну реалізацію та комплексне тестування 2D ігрового застосунку «DUNG» на основі об’єктно-орієнтованої архітектури з використанням мови програмування Python та бібліотеки Pygame.

На основі системного аналізу предметної області та сучасних технологій ігрової індустрії обґрунтовано вибір стеку технологій. Використання Python у поєднанні з Pygame дозволило ефективно реалізувати ігрову логіку, систему обробки подій та рендерингу, сформувавши надійну основу для прототипування та повноцінної розробки.

Спроєктовано та програмно реалізовано модульну об’єктно-орієнтовану архітектуру застосунку. Теоретичне значення результату полягає у розробці та впровадженні гнучкої системи управління станами. Ця модель забезпечує строгий контроль над життєвим циклом гри, ізолює ігрову логіку від службових процесів та створює надійне підґрунтя для безпечного масштабування програмних систем реального часу.

Практично реалізовано ключові ігрові механіки в єдиному тайловому середовищі: систему переміщення з математичною нормалізацією швидкості діагонального руху, обробку колізій із перешкодами, бойову систему з відстеженням прямої видимості (LoS) для керування штучним інтелектом противників, а також алгоритми плавного скролінгу камери.

Досягнуто високих якісних та кількісних показників продукту. Розроблена система забезпечує стабільну частоту оновлення кадрів не менше 60 FPS навіть за умови активної генерації нових об'єктів. Інтегрована система покадрової анімації ігрових сутностей та графічних переходів (затемнення/поява) працює з точністю оновлення у 70 мс. Програмний код структуровано за принципами чистої архітектури з використанням ізольованих класів та переліків (Enum) для типізації.

Достовірність та надійність отриманих результатів повністю обґрунтовано проведенням багаторівневого тестування. Під час розробки програмного забезпечення було виявлено та усунено помилки, пов’язані з реалізацією окремих

ігрових механік, зокрема системи переміщення персонажа та механізму переходу між рівнями. Їх своєчасне виявлення та виправлення дозволило підвищити стабільність ігрового процесу та забезпечити коректну роботу застосунку. Вже на етапі тестування за допомогою вбудованої бібліотеки unittest успішно виконано серію ізольованих модульних тестів (перевірка розрахунків колізій, коректність ініціалізації світу, алгоритми видалення відпрацьованих спрайтів за межами екрана). Окрім цього, розроблені функціональні тест-плани підтвердили стовідсоткову стабільність роботи базових підсистем під час реальних ігрових сесій без виникнення критичних збоїв. Також визначено системні вимоги до програмного забезпечення та описано процес його розгортання, який передбачає локальний запуск застосунку після встановлення Python та необхідних бібліотек без потреби у серверній інфраструктурі.

Практичне використання та рекомендації: створений 2D ігровий застосунок є функціонально завершеним та працездатним програмним продуктом. Запропонована архітектурна модель рекомендується для подальшого комерційного або аматорського розширення шляхом додавання нових рівнів (через імпорт CSV-матриць), розширення набору ігрових персонажів та противників, впровадження нових предметів і механік взаємодії, а також імплементації системи інвентарю без необхідності рефакторингу ядра програми. Крім того, вихідний код проєкту та описані алгоритми доцільно використовувати у навчальному процесі під час вивчення дисциплін з об’єктно-орієнтованого програмування, проєктування програмного забезпечення та архітектури комп'ютерних ігор.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Merriam-Webster Dictionary. Dungeon crawler. URL: https://www.merriam-webster.com/dictionary/dungeon%20crawler (дата звернення: 5.06.2026).

2. Які є ігрові рушії без програмування? Just Smart. URL: https://justsmart.com.ua/uk/blog/yaki-ye-igrovi-rushiyi-bez-programuvannya/ (дата звернення: 5.06.2026).

3. Carol C. How to Create a 2D Game from Start to Finish (2025 Edition). Medium. URL: https://medium.com/@carolccc/how-to-create-a-2d-game-from-start-to-finish-2025-edition-34fe6f354c68 (дата звернення: 5.06.2026).

4. 10 найкращих ігрових рушіїв. ULab СумДУ. URL: https://ulab.sumdu.edu.ua/uk/10-najkrashhih-igrovih-rushiiv (дата звернення: 5.06.2026).

5. Over 80 Unreal Engine Games Highlighted During Recent Summer Events. Unreal Engine Blog. URL: https://www.unrealengine.com/blog/over-80-unreal-engine-games-highlighted-during-recent-summer-events (дата звернення: 5.06.2026).

6. Games Made with Unity. Razzem. URL: https://www.razzem.com/games-made-with-unity/ (дата звернення: 5.06.2026).

7. Top Python Game Engines. Real Python. URL: https://realpython.com/top-python-game-engines/ (дата звернення: 5.06.2026).

8. Best Dungeon Crawler Games. Eneba. URL: https://www.eneba.com/hub/games/best-dungeon-crawler-games/ (дата звернення: 5.06.2026).

9. Cult of the Lamb Is the One Cult You'll Never Want to Leave. Hey Poor Player. URL: https://www.heypoorplayer.com/2022/07/05/cult-of-the-lamb-is-the-one-cult-youll-never-want-to-leave/ (дата звернення: 5.06.2026).

10. Unity чи Unreal Engine: що обрати для розробки гри? DOU. URL: https://dou.ua/forums/topic/40575/ (дата звернення: 5.06.2026).

11. Pygame Sprite Module. Pygame Documentation. URL: https://www.pygame.org/docs/ref/sprite.html (дата звернення: 5.06.2026).

12. State Modelling. Max Zosim. URL: https://www.maxzosim.com/state-modelling/ (дата звернення: 5.06.2026).

13. Tilemaps. MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Games/Techniques/Tilemaps (дата звернення: 5.06.2026).

14. Dungeon Tileset II. Oryx Design Lab. URL: https://0x72.itch.io/dungeontileset-ii (дата звернення: 5.06.2026).

15. Тестування ігор: основні види та етапи. Arionis Games. URL: https://arionisgames.com/uk/services/full-cycle-gamedev/game-testing/ (дата звернення: 5.06.2026).

16. How to Fix Diagonal Movement in 2D Games. JS Legend Dev. URL: https://jslegenddev.substack.com/p/how-to-fix-diagonal-movement-in-2d (дата звернення: 5.06.2026).

17. Функціональне тестування. QALight. URL: https://qalight.ua/baza-znaniy/funktsionalne-testuvannya/ (дата звернення: 5.06.2026).

18. Модульне тестування. QALight. URL: https://qalight.ua/baza-znaniy/modulne-testuvannya/ (дата звернення: 5.06.2026).

19. Python Releases for Windows. Python Software Foundation. URL: https://www.python.org/downloads/ (дата звернення: 5.06.2026).

20. Безпека життєдіяльності та охорона праці : підручник / В. В. Сокуренко, О. М. Бандурка та ін. Харків : ХНУВС, 2021. 308 с.

21. Жидецький В. Ц. Охорона праці користувачів комп’ютерів : підручник. Львів : Афіша, 2020. 176 с.

22. Методичні вказівки до виконання кваліфікаційної роботи бакалавра для здобувачів спеціальності 121 – Інженерія програмного забезпечення, всіх форм навчання / укладачі: Михалик Д.М., Цуприк Г.Б., Бревус В.М. – Тернопіль: Тернопільський національний технічний університет імені Івана Пулюя, 2024. – 45 с. (https://elartu.tntu.edu.ua/handle/lib/50317)

23. Олянін, Д., Цуприк, Г. (2025) Transformer Neural Networks in Industry 4.0 / Д. Олянін, Г. Цуприк, Т. Говорущенко, О. Багрій-Заяць, І. Андрущак // Computer Information Technologies in Industry 4.0: proceedings of the 3rd International Workshop

(CITI-2025), Ternopil, Ukraine, 11–12 June 2025. – Ternopil : Ternopil Ivan Puluj National Technical University, 2025 (Scopus) https://ceur-ws.org/Vol-4057/\

24. Tsupryk, H., Olianin, D. (2025). Vydobuvannia danyh z tekstu vykorystovuiuchy transformerni neironni merezhi [Data extraction from text using Transformer Neural Networks]. Information Technology: Computer Science, Software Engineering and Cyber Security, 125–130, DOI: https://doi.org/10.32782/IT/2025-2-13

25. ОЛЯНІН D., & ЦУПРИК Н. (2025). Огляд ролі трансформерних нейроних мереж у видобувані інформації із неструктурованих даних. Measuring and computing devices in technological processes, 82(2), 360–364. https://doi.org/10.31891/2219-9365-2025-82-52

26. Tsupryk H. LLM-based Extraction from Resumes / D. Olianin, H. Tsupryk // Advanced Technologies in Scientific Research: collection of scientific papers with proceedings of the 1st International Scientific and Practical Conference, Rotterdam, Netherlands, 20–22 August 2025. – International Scientific Unity, 2025. – 72-76

27. Yaroslav Kotov, Evhenia Yavorska, Halyna Tsupryk, Róża Dzierżak 1 , Oleksandr Reshetnik, Viktoriia Bokovets (2025) Evaluating interoperability and data quality in FHIR-based AI assessment pipelines. Proc. SPIE 14009, Photonics Applications in Astronomy, Communications, Industry, and High Energy Physics Experiments 2025, 140091F (30 December 2025) https://doi.org/10.1117/12.3100561

# ДОДАТКИ

Додаток А

Апробація результатів

*IX Міжнародна студентська науково - технічна конференція*
*"ПРИРОДНИЧІ ТА ГУМАНІТАРНІ НАУКИ. АКТУАЛЬНІ ПИТАННЯ"*

УДК 004.4
Крупа М. – ст. гр. СП-42
*Тернопільський національний технічний університет імені Івана Пулюя*

**МОВА ПРОГРАМУВАННЯ PYTHON ЯК ЗАСІБ РОЗРОБКИ І ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ**

Науковий керівник: к.т.н., доцент Цуприк Г. Б.

Krupa M.
*Ternopil Ivan Puluj National Technical University*

**THE PYTHON PROGRAMMING LANGUAGE AS A TOOL FOR DEVELOPING AND TESTING SOFTWARE**

Supervisor: Tsupryk H. B.

Ключові слова: об'єктно-орієнтоване програмування, розробка та тестування, життєвий цикл програмного забезпечення.
Keywords: object-oriented programming, development and testing, software development life cycle.

Мова програмування Python є однією з найзручніших для розробки інді-проєктів у жанрі Dungeon Crawler, який вимагає чіткої організації програмної архітектури для забезпечення масштабованості та зручності підтримки коду. Переваги Python полягають в об'єктно-орієнтованому підході, широкому виборі бібліотек та зрозумілості синтаксису.

Змога підтримки повного життєвого циклу розробки програмного забезпечення є надважливим аспектом у побудові будь-якого програмного продукту, адже ґрунтується на оцінці якості кожного етапу.

Грамотне та ефективне використання принципів об'єктно-орієнтованого програмування сприяє забезпеченню масштабованості, зрозумілості та ізоляції коду – важливих для продукту характеристик.

Архітектура, побудована на основі абстрактних класів, дозволяє розділити відповідальність між ключовими компонентами системи. Базова модель передбачає використання незалежних модулів, зокрема для управління станом сутностей, генерації ігрового середовища та контролю глобального ігрового сеансу. Такий підхід спрямований на ізоляцію логіки обробки внутрішніх ігрових подій від представлення даних, що спрощує подальше розширення функціоналу.

Використання менеджера станів як основний механізм взаємодії, передбачає відслідковування переходів між ігровими фазами, обробкою команди користувача та ініціацією відповідних змін у параметрах об'єктів. Застосування патернів проєктування для управління елементами середовища забезпечує їх динамічне створення, взаємодію та видалення. Такий підхід орієнтований на оптимізацію життєвого циклу ігрових компонентів та підтримку стабільної швидкодії.

Особливу увагу приділено організації взаємодії між компонентами системи. Для зменшення зв'язаності між модулями передбачається використання принципів слабкої зв'язності та інверсії залежностей. Це дозволяє підвищити гнучкість архітектури та спростити модифікацію окремих компонентів без впливу на інші частини системи.

Окремим аспектом є питання масштабованості застосунку. Запропонований об'єктно-орієнтований підхід до побудови архітектури з використанням основних принципів ООП, зокрема інкапсуляції, наслідування та поліморфізму, а також розділення відповідальностей, дозволяє розширювати функціональні можливості гри шляхом додавання нових типів сутностей, механік взаємодії та сценаріїв без необхідності суттєвої зміни існуючого коду. Це є важливим для проєктів у жанрі Dungeon Crawler, які часто передбачають поступове ускладнення ігрового процесу.

Питання організації збереження та відновлення стану гри вирішується реалізацією механізмів серіалізації ігрових об'єктів, що дозволить зберігати поточний стан ігрового сеансу та відновлювати його у подальшому. Це підвищує зручність користування застосунком та створює основу для реалізації додаткових функцій, таких як контроль прогресу гравця.

Важливим етапом циклу розробки – тестування програмного забезпечення. Для перевірки надійності продукту, а саме ключових алгоритмів гри, добре підійде модульне тестування. Воно дозволить валідувати логіку зміни станів, перевіряти коректність розрахунків під час взаємодій між об'єктами та оцінювати стабільність роботи менеджера сеансів при різних ігрових сценаріях.

Запропонована архітектурна модель з використанням принципів об'єктно-орієнтованого програмування та підтримкою усіх етапів повного життєвого циклу програмного забезпечення орієнтована на якісні розробку та тестування, успішне розгортання програми початкового продукту, доведення її до рівня повноцінної гри та продовження підтримки та оновлення для подальших покращень та інтеграцій.

Література:

1. Олянін, Д., Цуприк, Г. (2025) Transformer Neural Networks in Industry 4.0 / Д. Олянін, Г. Цуприк, Т. Говорущенко, О. Багрій-Заяць, І. Андрущак // Computer Information Technologies in Industry 4.0: proceedings of the 3rd International Workshop (CITI-2025), Ternopil, Ukraine, 11–12 June 2025. – Ternopil : Ternopil Ivan Puluj National Technical University, 2025 (Scopus) https://ceur-ws.org/Vol-4057/

2. Tsupryk, H., Olianin, D. (2025). Vydobuvannia danyh z tekstu vykorystovuiuchy transformerni neironni merezhi [Data extraction from text using Transformer Neural Networks]. Information Technology: Computer Science, Software Engineering and Cyber Security, 125–130, DOI: https://doi.org/10.32782/IT/2025-2-13

3. ОЛЯНІН D., & ЦУПРИК Н. (2025). Огляд ролі трансформерних нейроних мереж у видобувані інформації із неструктурованих даних. Measuring and computing devices in technological processes, 82(2), 360–364. https://doi.org/10.31891/2219-9365-2025-82-52

4. Зосим М. (2023). Життєвий цикл розробки програмного забезпечення (Software Development Life Cycle - SDLC). https://www.maxzosim.com/software-development-life-cycle-sdlc/

5. Michał Piórkowski (2024). Python for Game Dev: Is It a Good or Bad Idea? https://sunscrapers.com/blog/python-for-game-development-is-it-a-good-or-bad-idea/

6. Sharon Hart (2022). Testing Python module-level code and why it smells. https://medium.com/@sharon.dev/testing-python-module-level-code-and-why-it-smells-3bf1cc5a79d5

Додаток Б

Конструктор класу Character

```
def __init__(
     self,
     x: int,
     y: int,
     health: int,
     characters_animations: list[list[list]],
     character_type: CharacterType,
     bow_image=None,
     arrow_image=None,
     size=1,
     boss=False
):
     if character_type == CharacterType.PLAYER:
          self.bow = Weapon(bow_image, arrow_image)
     self.flip = False
     self.frame_index = 0
     self.action = 0 #0: idle, 1: run
     self.running = False
     self.health = health
     self.alive = True
     self.is_boss = boss
     self.collect_score = 0
     self.character_type = character_type
     self.update_time = pygame.time.get_ticks()
     self.animation_frames =
characters_animations[self.character_type]

     self.image =
self.animation_frames[self.action][self.frame_index]
     self.rect = pygame.Rect(0, 0, config.TILE_SIZE * size,
config.TILE_SIZE * size)
     self.rect.center = (x, y)

     self.hit = False
     self.last_hit_time = pygame.time.get_ticks()
     self.last_fireball_attack = pygame.time.get_ticks()
     self.stunned = False
```

Додаток В

Метод spawn_units() класу World

```
def spawn_units(self, data, tile_images: list, item_images: list,
character_images: list, bow_image, arrow_image):
     self.level_length = len(data[0])
     for row_index, row in enumerate(data):
          for col_index, tile in enumerate(row):
               image = tile_images[tile]
               image_rect = image.get_rect()
               image_x = col_index * config.TILE_SIZE
               image_y = row_index * config.TILE_SIZE
               image_rect.center = (image_x, image_y)
               tile_data = [image, image_rect, image_x, image_y]
               #add special tiles to their respective lists:
               if tile == config.WALL_TILE:
                    self.obstacle_tiles.append(tile_data)
               elif tile == config.LADDER_TILE:
                    self.ladder_tile = tile_data
               elif tile == config.COIN_TILE:
                    coin = Item(image_x, image_y, "coin",
item_images[0])
                    self.items.append(coin)
                    tile_data[0] = tile_images[0] #remove the tile
image since it's now an item
               elif tile == config.POTION_TILE:
                    potion = Item(image_x, image_y, "potion",
[item_images[1]])
                    self.items.append(potion)
                    tile_data[0] = tile_images[1]
               elif tile == config.PLAYER_TILE:
                    self.player = Character(
                        image_x, image_y,
                        config.PLAYER_HEALTH,
                        character_images,
                        CharacterType.PLAYER,
                        bow_image, arrow_image)
                    tile_data[0] = tile_images[0]
               elif tile in (config.ENEMY_DATA.keys()):
                    character_type, health, is_boss =
config.ENEMY_DATA[tile]
                    enemy = Character(
                        image_x, image_y,
                        health,
                        character_images,
                        character_type,
                        size = 2 if is_boss else 1,
                        boss=is_boss)
                    self.enemies.append(enemy)
                    tile_data[0] = tile_images[0]
               if tile >= 0:
                    self.map_tiles.append(tile_data)
```

Додаток Д

Код файлу main.py

```
import pygame
import config
from button import Button
from effects.damage_text import DamageText
from effects.screen_fade import ScreenFade
from character_type import CharacterType
from utils import load_units_images, draw_info_panel, load_level

pygame.init()

screen = pygame.display.set_mode((config.SCREEN_WIDTH,
config.SCREEN_HEIGHT))
pygame.display.set_caption("DUNG")
font=pygame.font.Font("assets/fonts/AtariClassic.ttf", 20)
death_font = pygame.font.Font("assets/fonts/AtariClassic.ttf", 100)

clock = pygame.time.Clock()

level = config.START_LEVEL
start_intro = False
start_game = False
pause_game = False
level_complete = None

#movement variables:
move_left = False
move_right = False
move_up = False
move_down = False

images = load_units_images()

world = load_level(images)
#create units:
player = world.player
enemies = world.enemies

groups = {
    "arrows": pygame.sprite.Group(),
    "damage_texts": pygame.sprite.Group(),
    "items": pygame.sprite.Group(),
    ««Fireball»»s": pygame.sprite.Group()
}

for item in world.items:
    groups["items"].add(item)

intro_fade = ScreenFade("intro", config.BLACK, 4)
death_fade = ScreenFade("death", config.PINK, 4)
```

```
start_btn = Button(config.SCREEN_WIDTH // 2 - 145,
config.SCREEN_HEIGHT // 2 - 150, images["start_image"])
restart_btn = Button(config.SCREEN_WIDTH // 2 - 175,
config.SCREEN_HEIGHT // 2 - 50, images["restart_image"])
resume_btn = Button(config.SCREEN_WIDTH // 2 - 175,
config.SCREEN_HEIGHT // 2 - 150, images["resume_image"])
exit_btn = Button(config.SCREEN_WIDTH // 2 - 110,
config.SCREEN_HEIGHT // 2 + 50, images["exit_image"])

run = True
while run:

    clock.tick(config.FPS)

    if not start_game:
        screen.fill(config.MENU_COLOR)

        start_clicked = start_btn.draw(screen)
        if start_clicked:
            start_game = True
            start_intro = True
        exit_clicked = exit_btn.draw(screen)
        if exit_clicked:
            run = False

    else:
        if pause_game:
            screen.fill(config.MENU_COLOR)
            resume_clicked = resume_btn.draw(screen)
            if resume_clicked:
                pause_game = False
            exit_clicked = exit_btn.draw(screen)
            if exit_clicked:
                run = False
        else:
            screen.fill(config.BACGROUND_COLOR)
            if player.alive:
                dx = 0
                dy = 0
                if move_left:
                    dx -= config.MOVE_SPEED
                if move_right:
                    dx += config.MOVE_SPEED
                if move_up:
                    dy -= config.MOVE_SPEED
                if move_down:
                    dy += config.MOVE_SPEED

                #move units:
                screen_scroll, level_complete = player.move(dx, dy,
world.obstacle_tiles, world.ladder_tile)  #x and y screen roll for
scrolling the world map

                #update units:
                world.update(screen_scroll)
```

```
                for enemy in enemies:
                    ««Fireball»» = enemy.mob(player,
world.obstacle_tiles, screen_scroll, images["««Fireball»»_image"])
                    if ««Fireball»»:
                        groups["««Fireball»»s"].add(««Fireball»»)
                    if enemy.alive:
                        enemy.update()
                player.update()
                if player.character_type == CharacterType.PLAYER:
                    arrow = player.bow.update(player)
                    if arrow:
                        groups["arrows"].add(arrow)
                    for arrow in groups["arrows"]:
                        damage, damage_pos =
arrow.update(screen_scroll, enemies, world.obstacle_tiles)
                        if damage:
                            damage_text =
DamageText(damage_pos.centerx, damage_pos.y, damage)
                            groups["damage_texts"].add(damage_text)
                groups["fireballs"].update(screen_scroll, player,
world.obstacle_tiles)
                groups["damage_texts"].update(screen_scroll)
                groups["items"].update(screen_scroll, player)

            #draw units:
            world.draw(screen)
            for enemy in enemies:
                enemy.draw(screen)
            player.draw(screen)
            if player.character_type == CharacterType.PLAYER:
                player.bow.draw(screen)
                for arrow in groups["arrows"]:
                    arrow.draw(screen)
            for ««Fireball»» in groups["««Fireball»»s"]:
                ««Fireball»».draw(screen)
            groups["damage_texts"].draw(screen)
            groups["items"].draw(screen)
            draw_info_panel(screen, player, images["full_heart"],
images["half_heart"], images["empty_heart"], font, level)

            # show intro fade:
            if start_intro:
                if intro_fade.fade(screen):
                    start_intro = False
                    intro_fade.fade_counter = 0

            #show death fade:
            if not player.alive:

                if death_fade.fade(screen, death_font):
                    restart = restart_btn.draw(screen)
                    if restart:
                        death_fade.fade_counter = 0
                        start_intro = True
```

```
                        world = load_level(images)
                        player = world.player
                        enemies = world.enemies

                        for item in world.items:
                            groups["items"].add(item)
                        player.alive = True

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            run = False

        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_a:
                move_left = True
            if event.key == pygame.K_d:
                move_right = True
            if event.key == pygame.K_w:
                move_up = True
            if event.key == pygame.K_s:
                move_down = True

            if event.key == pygame.K_ESCAPE:
                pause_game = True

            if level_complete:
                level += 1
                start_intro = True
                old_health = player.health
                old_score = player.collect_score

                world = load_level(images, groups, level)

                player = world.player
                enemies = world.enemies
                player.health = old_health
                player.collect_score = old_score

                for item in world.items:
                    groups["items"].add(item)

        if event.type == pygame.KEYUP:
            if event.key == pygame.K_a:
                move_left = False
            if event.key == pygame.K_d:
                move_right = False
            if event.key == pygame.K_w:
                move_up = False
            if event.key == pygame.K_s:
                move_down = False


    pygame.display.update()
pygame.quit()
```

Додаток Е

Допоміжний файл utils.py

```
import csv
import config
import pygame
from character_type import CharacterType
from world import World


def scale_image(image, scale):
    return pygame.transform.scale(
        image,
        (
            image.get_width() * scale,
            image.get_height() * scale
        )
    )

def draw_text(screen, text, font, color, x, y):
    img = font.render(text, True, color)
    screen.blit(img, (x, y))

def draw_info_panel(screen, player, full_heart, half_heart,
empty_heart, font, level=1):
    pygame.draw.rect(screen, config.PANEL_COLOR, (0, 0,
config.SCREEN_WIDTH, config.PANEL_HEIGHT))
    pygame.draw.line(screen, config.WHITE, (0, config.PANEL_HEIGHT),
(config.SCREEN_WIDTH, config.PANEL_HEIGHT))
    half_heart_drawn = False

    for i in range(0, config.PLAYER_HEALTH//20):
        #show full heart
        if player.health >= (i+1)*20:
            screen.blit(full_heart, (10 + i*50, 0))
        #show empty heart
        elif player.health % 20 != 0 and not half_heart_drawn:
            screen.blit(half_heart, (10 + i*50, 0))
            half_heart_drawn = True
        else:
            screen.blit(empty_heart, (10 + i*50, 0))

    draw_text(
        screen,
        f"LEVEL: {level}",
        font,
        config.WHITE,
        config.SCREEN_WIDTH//2.5,
        15
    )

    draw_text(
        screen,
        f"Score: {player.collect_score}",
```

```
        font,
        config.WHITE, config.SCREEN_WIDTH - 200,
        15
    )

def load_units_images() -> dict:
    start_image =
scale_image(pygame.image.load("assets/images/buttons/button_start.pn
g").convert_alpha(), config.BUTTON_SCALE)
    restart_image =
scale_image(pygame.image.load("assets/images/buttons/button_restart.
png").convert_alpha(), config.BUTTON_SCALE)
    resume_image =
scale_image(pygame.image.load("assets/images/buttons/button_resume.p
ng").convert_alpha(), config.BUTTON_SCALE)
    exit_image =
scale_image(pygame.image.load("assets/images/buttons/button_exit.png
").convert_alpha(), config.BUTTON_SCALE)

    empty_heart =
scale_image(pygame.image.load("assets/images/items/heart_empty.png")
.convert_alpha(), config.SCALE)
    full_heart =
scale_image(pygame.image.load("assets/images/items/heart_full.png").
convert_alpha(), config.SCALE)
    half_heart =
scale_image(pygame.image.load("assets/images/items/heart_half.png").
convert_alpha(), config.SCALE)

    bow_image =
scale_image(pygame.image.load("assets/images/weapons/bow.png").conve
rt_alpha(), config.WEAPON_SCALE)
    arrow_image =
scale_image(pygame.image.load("assets/images/weapons/arrow.png").con
vert_alpha(), config.WEAPON_SCALE)
    ««Fireball»»_image =
scale_image(pygame.image.load("assets/images/weapons/»«Fireball»».pn
g").convert_alpha(), config.»«FIREBALL»»_SCALE)

    coin_images = [

scale_image(pygame.image.load(f"assets/images/items/coin_f{i}.png").
convert_alpha(), config.ITEM_SCALE)
        for i in range(4)
    ]

    potion_image =
scale_image(pygame.image.load("assets/images/items/potion_red.png").
convert_alpha(), config.SCALE)

    tile_images = []
    for i in range(config.TILE_TYPES):
        img =
pygame.image.load(f"assets/images/tiles/{i}.png").convert_alpha()
```

```
        scaled_img = pygame.transform.scale(img, (config.TILE_SIZE,
config.TILE_SIZE))
        tile_images.append(scaled_img)

    characters_animations = {}

    for character in CharacterType:
        animation_frames = []

        for animation in config.ANIMATION_TYPES:
            animation_list = []

            for i in range(4):
                img =
pygame.image.load(f"assets/images/characters/{character.value}/{anim
ation}/{i}.png").convert_alpha()
                scaled_img = scale_image(img, config.SCALE)
                animation_list.append(scaled_img)

            animation_frames.append(animation_list)

        characters_animations[character] = animation_frames

    return {
        "start_image": start_image,
        "restart_image": restart_image,
        "resume_image": resume_image,
        "exit_image": exit_image,
        "characters_animations": characters_animations,
        "bow_image": bow_image,
        "arrow_image": arrow_image,
        "««Fireball»»_image": ««Fireball»»_image,
        "coin_images": coin_images,
        "potion_image": potion_image,
        "full_heart": full_heart,
        "half_heart": half_heart,
        "empty_heart": empty_heart,
        "tile_images": tile_images
    }

def load_level(images: dict, groups: dict = None,
level=config.START_LEVEL) -> list[list[int]]:

    if groups:
        groups["arrows"].empty()
        groups["damage_texts"].empty()
        groups["items"].empty()
        groups["««Fireball»»s"].empty()

    world_map = [[-1] * config.WORLD_SIZE for _ in
range(config.WORLD_SIZE)]
    #load the world map from a file:
    with open(f"levels/level{level}_data.csv", newline="") as
csvfile:
        reader = csv.reader(csvfile, delimiter=",")
```

```
    for row_index, row in enumerate(reader):
        for tile_index, tile in enumerate(row):
            world_map[row_index][tile_index] = int(tile)

world = World()
world.spawn_units(
    world_map,
    images["tile_images"],
    [images["coin_images"], images["potion_image"]],
    images["characters_animations"],
    images["bow_image"],
    images["arrow_image"]
)
return world
```

Додаток Ж

Метод update() класу Arrow

```
def update(self, screen_scroll: list, enemies: list, obstacle_tiles:
list) -> tuple[int, pygame.Rect | None]:
     self.rect.x += screen_scroll[0] + self.dx
     self.rect.y += screen_scroll[1] + self.dy

     #check collision with obstacle tiles:
     for tile in obstacle_tiles:
          if self.rect.colliderect(tile[1]):
               self.kill()

        #check if arrow is off screen
     if (
          self.rect.right < 0 or
          self.rect.left > config.SCREEN_WIDTH or
          self.rect.bottom < 0 or
          self.rect.top > config.SCREEN_HEIGHT
     ):
          self.kill()

     #check collision with enemies:
     damage = 0
     damage_pos = None
     for enemy in enemies:
          if enemy.rect.colliderect(self.rect) and enemy.alive:
               damage = config.PLAYER_DAMAGE + random.randint(-5, 5)
               enemy.health -= damage
               damage_pos = enemy.rect
               enemy.hit = True
               self.kill()
               break
     return damage, damage_pos
```

Додаток З

Файл класу ScreenFade

```
import pygame
import config

class ScreenFade():
    def __init__(self, type, color, speed):
        self.type = type
        self.color = color
        self.speed = speed
        self.fade_counter = 0

    def fade(self, screen, death_font = None):
        fade_complete = False

        if self.type == "intro":
            self.fade_counter += self.speed
            pygame.draw.rect(
                screen,
                self.color,
                (0 - self.fade_counter, 0, config.SCREEN_WIDTH // 2,
config.SCREEN_HEIGHT)
            )
            pygame.draw.rect(
                screen,
                self.color,
                (config.SCREEN_WIDTH // 2 + self.fade_counter, 0,
config.SCREEN_WIDTH, config.SCREEN_HEIGHT)
            )
            pygame.draw.rect(
                screen,
                self.color,
                (0, 0 - self.fade_counter, config.SCREEN_WIDTH,
config.SCREEN_HEIGHT // 2)
            )
            pygame.draw.rect(
                screen,
                self.color,
                (0, config.SCREEN_HEIGHT // 2 + self.fade_counter,
config.SCREEN_WIDTH, config.SCREEN_HEIGHT)
            )

        if self.type == "death":
            self.fade_counter += self.speed
            pygame.draw.rect(
                screen,
                self.color,
                (0, 0, config.SCREEN_WIDTH, self.fade_counter)
            )
            alpha = min(self.fade_counter * 2, 255)

            text = death_font.render("YOU DIED", True, config.D_RED)
            text_rect = text.get_rect(
```

```
            center=(
                config.SCREEN_WIDTH // 2,
                config.SCREEN_HEIGHT // 2 - 150
            )
        )
        text.set_alpha(alpha)

        screen.blit(text, text_rect)

    if self.fade_counter >= config.SCREEN_WIDTH:
        fade_complete = True

    return fade_complete
```