Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти перший (бакалаврський)

Ігровий веб-застосунок з використанням рушія Godot
(тема)

Виконав:
здобувач 4 року навчання,
групи КІУКІ-21-5
Костянтин МОРОЗОВ
(власне ім'я, прізвище)

Спеціальність
123 «Комп'ютерна інженерія»
(код і повна назва спеціальності)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма
Комп'ютерна інженерія
(повна назва освітньої програми)

Керівник: Юрій КОЛТУН
(посада, власне ім'я, прізвище)

Допускається до захисту

Завідувач кафедри ЕОМ ________ Андрій КОВАЛЕНКО
(підпис) (власне ім'я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти перший (бакалаврський)

Спеціальність 123 «Комп’ютерна інженерія»
(код і повна назва)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма Комп’ютерна інженерія
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _______________
(підпис)
“____”_______________ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Морозову Костянтину Івановичу
(прізвище, ім’я, по батькові)

1. Тема роботи Ігровий веб-застосунок з використанням рушія Godot

затверджена наказом по університету від “ 26 ” травня 2025 р. № 424 Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 17 червня 2025 р.

3. Вхідні дані до роботи 1) Ігровий рушій Godot Engine;
2) Скриптова мова GDScript.

4. Перелік питань, що потрібно опрацювати у роботі
1) аналіз предметної області
2) аналіз технологій, що використовуються при розробці
3) опис програмної реалізації застосунку
4) інструкція користувача
5) висновки

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій

Слайд-презентація – 11 слайдів

6. Консультанти розділів роботи (заповнюється за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Аналіз проблеми та огляд існуючих рішень	27.05.25	
2	Вибір технології розробки	31.05.25	
3	Розробка основних сцен гри	01.06.25 – 07.06.25	
4	Розробка візуальної складової застосунку	08.06.25 – 09.06.25	
5	Оформлення матеріалів кваліфікаційної роботи	10.06.25 – 12.06.25	
6	Подання кваліфікаційної роботи керівникові та	13.06.25 – 14.06.25	
	її попередній захист		
7	Подання кваліфікаційної роботи на рецензу-	16.06.25	
	вання		

Дата видачі завдання “ 26 ” травня 2025 р.

Здобувач ____________________
(підпис)

Керівник роботи ____________________ доц. Юрій КОЛТУН
(підпис) (посада, власне ім'я, прізвище)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 64 с., 29 рис., 1 дод., 14 джерел.

РОЗРОБКА ЗАСТОСУНКУ, ІГРОВИЙ РУШІЙ, СКРИПТ, КОРИСТУВАЦЬКИЙ ІНТЕРФЕЙС, АНАЛІЗ, ТЕСТУВАННЯ, GDSCRIPT, GODOT ENGINE.

Метою кваліфікаційної роботи є розробка гри в жанрі roguelike з акцентом на динамічний геймплей та систему прогресії персонажа.

Метод вирішення задачі – використання ігрового рушія Godot Engine, мови програмування GDScript та компонентного підходу до архітектури програмного продукту.

Розроблено працездатний прототип гри, в якому реалізовано автоматичну систему атаки, механізм збору досвіду, підвищення рівня персонажа та вибору покращень. Гравець взаємодіє з хвилями ворогів на статичній арені.

Кодова база проєкту побудована з використанням модульних компонентів та сигнальної системи Godot Engine, що забезпечує гнучкість, перевикористання коду та можливість подальшого розширення функціональності гри.

Програму складено мовою GDScript з використанням рушія Godot Engine у вбудованому середовищі розробки. Графічні асети створено за допомогою редактора Aseprite.

ABSTRACT

Bachelor’s thesis: 64 pages, 29 figures, 1 appendices, 14 sources.

APPLICATION DEVELOPMENT, GAME ENGINE, SCRIPT, USER INTERFACE, ANALYSIS, TESTING, GDSCRIPT, GODOT ENGINE.

The purpose of the qualification work is to develop a roguelike game with an emphasis on dynamic gameplay and a character progression system.

The method for solving the task is the use of the Godot Engine game engine, the GDScript programming language, and a component-based approach to the software product's architecture.

A functional game prototype has been developed, in which an automatic attack system, an experience collection mechanism, character leveling, and an upgrade selection system are implemented. The player interacts with waves of enemies in a static arena.

The project's codebase is built using modular components and Godot Engine's signal system, which ensures flexibility, code reusability, and the potential for further expansion of the game's functionality.

The program was written in the GDScript language using the Godot Engine within its integrated development environment. Graphical assets were created using the Aseprite editor.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ 8
ВСТУП 9
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ 11
1.1 Історія виникнення ігрових застосунків 11
1.2 Top-down ігри 13
1.3 Rogelike-ігри 15
1.4 Аналіз існуючих застосунків 16
1.4.1 Гра "Hades" 17
1.4.2 Гра "Enter the Gungeon" 19
1.4.3 Гра "Dead Cells" 20
1.4.4 Гра "Vampire Survivors" 21
1.5 Постановка задачі 23
2 АНАЛІЗ ТЕХНОЛОГІЙ, ЩО ВИКОРИСТОВУЮТЬСЯ ПРИ РОЗРОБЦІ 24
2.1 Ігровий рушій Godot 24
2.2 Мова програмування GDScript 26
2.3 Графічний редактор Aseprite 28
2.4 Патерн проєктування Singleton 30
2.5 Патерн проєктування Observer 31
2.6 Патерн проєктування Factory Method 31
2.7 Патерн проєктування Component 33
3 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ ЗАСТОСУНКУ 34
3.1 Огляд структури проєкту 34
3.2 Каталог scene 35
3.3. Структура та функціональність сцени Level 36
3.3.1. Ієрархія вузлів сцени Level 36
3.3.2. Скриптова логіка сцени Level 38

3.4 Структура сцени Player 40
3.4.1 Скрипт сцени Player 41
3.5 Архітектура компонентів 43
3.6 Система генерації ворогів 46
4 ІНСТРУКЦІЯ КОРИСТУВАЧА 49
ВИСНОВКИ 55
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 57
ДОДАТОК А Графічний матеріал кваліфікаційної роботи 58

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

ООП – об'єктно-орієнтоване програмування

2D – двовимірний простір

API – програмний інтерфейс застосунку (англ., Application Programming Interface)

Area2D – вузол рушія Godot Engine, що використовується для визначення двовимірних областей та виявлення зіткнень

CharacterBody2D – вузол рушія Godot Engine, призначений для створення двовимірних персонажів, що керуються гравцем або скриптом

Exp – досвід (experience), ігрова одиниця, що використовується для підвищення рівня персонажа

FPS – кількість кадрів на секунду, одиниця виміру продуктивності гри (англ., Frames Per Second)

GDScript – вбудована об'єктно-орієнтована мова програмування рушія Godot Engine

GPU – графічний процесор (англ., Graphics Processing Unit)

GUI – графічний інтерфейс користувача (англ., Graphical User Interface)

HP – одиниці здоров'я (англ., hit points / health points)

Node2D – базовий вузол для всіх двовимірних об'єктів у рушії Godot Engine

TAU – математична константа, що дорівнює 2 * PI

UI – інтерфейс користувача (англ., User Interface)

ВСТУП

Сучасний світ відеоігор є динамічною та багатогранною галуззю, яка постійно еволюціонує та розширює свої межі. Від перших простих ігор, створених для наукових та дослідницьких цілей, до сучасних високотехнологічних проєктів, ігрові застосунки пройшли тривалий шлях розвитку. Історія відеоігор бере свій початок ще з середини 20 століття, коли перші інтерактивні ігри з'явилися на великих обчислювальних машинах. З розвитком технологій ігри ставали все складнішими та різноманітнішими, завойовуючи популярність серед широкої аудиторії.

У 1970-х роках з'явилися перші комерційні ігри, такі як "Pong" від Atari, що стали початком "золотого віку" аркадних ігор. Такі ігри, як "Pac-Man" та "Space Invaders", стали культовими та заклали основи для подальшого розвитку індустрії. З появою персональних комп'ютерів та розвитком графічних технологій у 1990-х роках, ігрові застосунки стали ще більш захопливими та реалістичними. Складніша графіка, сюжетні лінії та геймплей дозволили створювати імерсивні світи, які приваблювали мільйони гравців.

Розвиток комп'ютерних технологій та зростання обчислювальної потужності відіграли ключову роль в еволюції відеоігор. З появою персональних комп'ютерів у 1980-х роках, ігри стали доступнішими для широкої аудиторії. Комп'ютери, такі як Commodore 64 та Apple II, стали популярними платформами, дозволяючи розробникам створювати все складніші та графічно насичені проєкти. З розвитком графічних процесорів та зростанням обчислювальної потужності, ігри ставали все реалістичнішими та імерсивнішими.

Інтернет також зіграв важливу роль у розвитку індустрії. З появою онлайн-ігор та мережних технологій, гравці отримали можливість взаємодіяти один з одним у реальному часі, що відкрило нові горизонти для

багатокористувацьких ігор і віртуальних світів, таких як "World of Warcraft".

Окрім цього, розвиток мобільних технологій та смартфонів у 2000-х роках відкрив нові можливості для ігрової індустрії. Мобільні ігри, такі як "Angry Birds", "Candy Crush" та "Pokémon GO", стали надзвичайно популярними та доступними для мільйонів користувачів по всьому світу. Ці ігри не лише розважають, але й часто інтегруються в повсякденне життя гравців, створюючи унікальні та захопливі досвіди.

Ще одним важливим жанром є Roguelike-ігри, які виникли у 1980-х роках та беруть свій початок від гри "Rogue". Цей жанр характеризується процедурною генерацією рівнів, складним геймплеєм та елементами постійної смерті. Ідея гри "Rogue" полягала у тому, що гравець проходив підземелля, яке кожного разу генерувалося по-новому, що робило гру цікавою та непередбачуваною. Сучасні Roguelike-ігри продовжують традиції жанру, додаючи інноваційні елементи та приваблюючи гравців своєю непередбачуваністю та високою складністю.

Розробка сучасних ігрових застосунків вимагає використання потужних інструментів та технологій. Одним з таких інструментів є ігровий рушій Godot, який є сучасним відкритим та безкоштовним рушієм для розробки ігор. Його популярність пояснюється поєднанням доступності, гнучкості та простоти використання, що робить його привабливим для інді-розробників та професійних студій. Godot також підтримує кросплатформність, дозволяючи створювати ігри для різноманітних платформ.

Таким чином, сучасна ігрова індустрія пропонує широкий спектр інструментів та технологій для розробки ігрових застосунків. Від історичних коренів до сучасних інновацій, ігрові застосунки продовжують розвиватися, пропонуючи гравцям все нові та захопливі досвіди. Ця робота присвячена аналізу існуючих ігрових застосунків, розробці концепції нового ігрового проєкту та використанню сучасних технологій для його реалізації.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Історія виникнення ігрових застосунків

Ігрові застосунки мають багату історію, яка бере початок ще з середини 20 століття. Перші інтерактивні ігри були розроблені для дослідницьких цілей та призначались для великих обчислювальних машин, наприклад, комп’ютерів університетів чи військових установ. Однією з перших ігор стала "Tennis for Two" 1958 року, яка була створена фізиком Вільямом Хігінботамом. Ця гра мала просту графіку та дозволяла двом гравцям змагатись на осцилографі (рисунок 1.1).

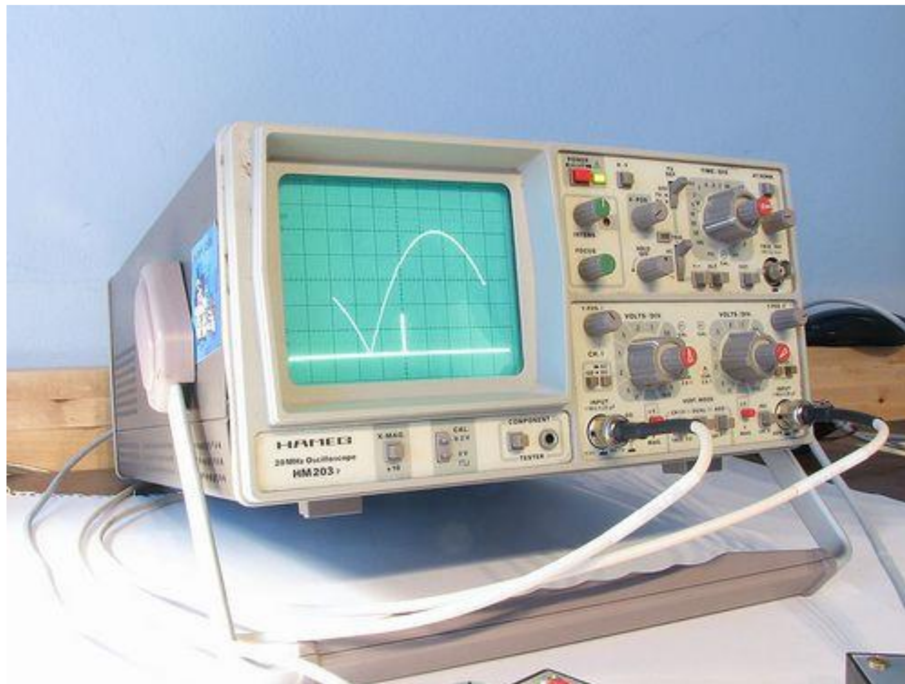
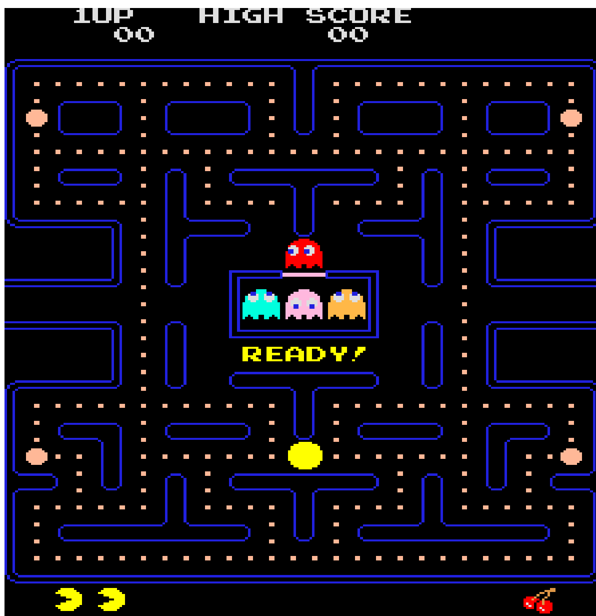


Рисунок 1.1 – Перша електронна гра "Tennis for Two" з графічним дисплеєм

Основною метою створення гри "Tennis for Two" було продемонструвати можливості нового обладнання та осцилографа в лабораторії, де працював Хіггінботам. Осцилограф використовувався для вимірювання електричних сигналів, але Хіггінботам вирішив використовувати його як візуальний пристрій для відображення графіки в

реальному часі. Його задум полягав у тому, щоб створити просту гру для розваги відвідувачів лабораторії, в основному, для демонстрації наукових можливостей та технологій.

У 1970-х роках, з розвитком технологій, з'явилися перші комерційні ігри, такі як "Pong" від Atari. Цей успіх став початком "золотого віку" аркадних ігор [1], коли такі ігри, як "Pac-Man" (рисунок 1.2 а) та "Space Invaders" (рисунок 1.2 б), стали популярними в усьому світі. Водночас ігри стали доступними для домашніх консолей, таких як Atari 2600.



а) б)

Рисунок 1.2 – Інтерфейс ігор: а) "Pac-Man"; б)"Space Invaders"

З 1990-х років поява персональних комп'ютерів та розвиток графічних технологій відкрили нові можливості для ігрових застосунків. Складніші графіка, сюжетні лінії та геймплей дозволили створювати більш захопливі та реалістичні ігри. Це десятиліття також відзначилося появою консолей нового покоління, таких як PlayStation і Nintendo 64, які стали масово популярними.

У 2000-х роках інтернет-підключення значно розширило можливості ігор. Мережні та багатокористувацькі ігри стали нормою, а мобільні пристрої перетворили ігри в доступний формат на кожен день. Революція смартфонів

у 2010-х роках зробила ігри розповсюдженими, завдяки популярним застосункам на кшталт "Angry Birds", "Candy Crush" та "Pokémon GO". Сьогодні ігрові застосунки є невід’ємною частиною життя мільйонів користувачів, які можуть насолоджуватися іграми будь-де й будь-коли.

1.2 Top-down ігри

Top-down ігри – це ігри, у яких камера знаходиться зверху, надаючи гравцеві вид на об'єкти та персонажів з висоти пташиного польоту. Такі ігри часто акцентують на стратегії, дослідженні або екшен-елементах, дозволяючи гравцю контролювати персонажа або групу персонажів для вивчення та взаємодії з ігровим світом. Камера в таких іграх зазвичай розташована перпендикулярно до поверхні ігрового світу, що забезпечує чітке розуміння навколишнього простору. Історично топ-даун перспективи стали популярними ще в епоху аркадних ігор та класичних консолей. Одним з перших великих прикладів є серія "The Legend of Zelda" (рисунок 1.3), яка використовувала топ-даун камеру для створення відкритих ігрових світів, у яких гравець міг вільно рухатися та вирішувати головоломки. Подібні ігри, як "Gauntlet" або "Robotron: 2084", також прославились своїм "зверху-вниз" підходом, пропонуючи гравцям динамічний екшен з великою кількістю ворогів на екрані одночасно.

З розвитком технологій та зміни уподобань гравців, топ-даун ігри стали різноманітнішими. Спочатку це були переважно екшн-ігри або шутери, але згодом до жанру приєдналися стратегічні симулятори, рольові ігри, а також нові інтерпретації класичних ідей. Наприклад, ігри, такі як "Bastion" та "Hyper Light Drifter", що використовують топ-даун перспективу, змішують класичний дизайн з сучасною графікою та сюжетом, надаючи новий погляд на стару механіку. Популярність топ-даун ігор часто залежала від того, як ця перспектива дозволяла полегшити управління, зберігаючи при цьому зрозумілу навігацію та взаємодію з навколишнім світом.

Рисунок 1.3 – Гра "The Legend of Zelda" 1986-го року

У стратегічних іграх, таких як "StarCraft" чи "Civilization", топ-даун погляд дозволяє зручно планувати ходи, контролювати великі території та здійснювати ефективне управління. У пригодницьких та ролевих іграх ця перспектива дозволяє детально розглянути навколишні об'єкти, а також зручно керувати персонажем, не відволікаючись на обмеження камери.

Інтерес до топ-даун ігор залишався високим навіть в умовах домінування 3D-графіки в сучасному геймінгу. Гравці знову повертаються до цих ігор завдяки їхній простоті та привабливому геймплею. Порівняно з більш складними тривимірними іграми, топ-даун ігри часто пропонують більш інтуїтивно зрозумілий досвід, що зберігає при цьому насиченість контенту та захопливу гру [2]. Водночас незалежні студії активно створюють нові ігри в цьому жанрі, що підтверджує його актуальність та попит серед геймерів.

Таким чином, top-down ігри мають довгу історію та багатий розвиток, і попри технологічний прогрес, вони продовжують залишатися популярними завдяки своїй універсальності, простоті управління та можливості зосередитись на стратегічних та тактичних елементах гри.

1.3 Rogelike-ігри

Roguelike-ігри – це жанр відеоігор, що виник у 1980-х роках і бере свою назву від гри "Rogue", яка була випущена у 1980 році. Головні особливості цього жанру – процедурна генерація рівнів [3], складний геймплей та елементи постійної смерті (permadeath), коли після поразки гравець починає гру з самого початку.

Ідея гри "Rogue" полягала у тому, що гравець проходить підземелля, яке кожного разу генерується по-новому, тому кожна нова гра була унікальною. Цей аспект процедурної генерації зробив гру цікавою та непередбачуваною. Постійна смерть теж стала особливістю жанру: якщо гравець втрачає всі життя, він не може продовжити збережену гру, а починає новий сеанс із новими умовами.

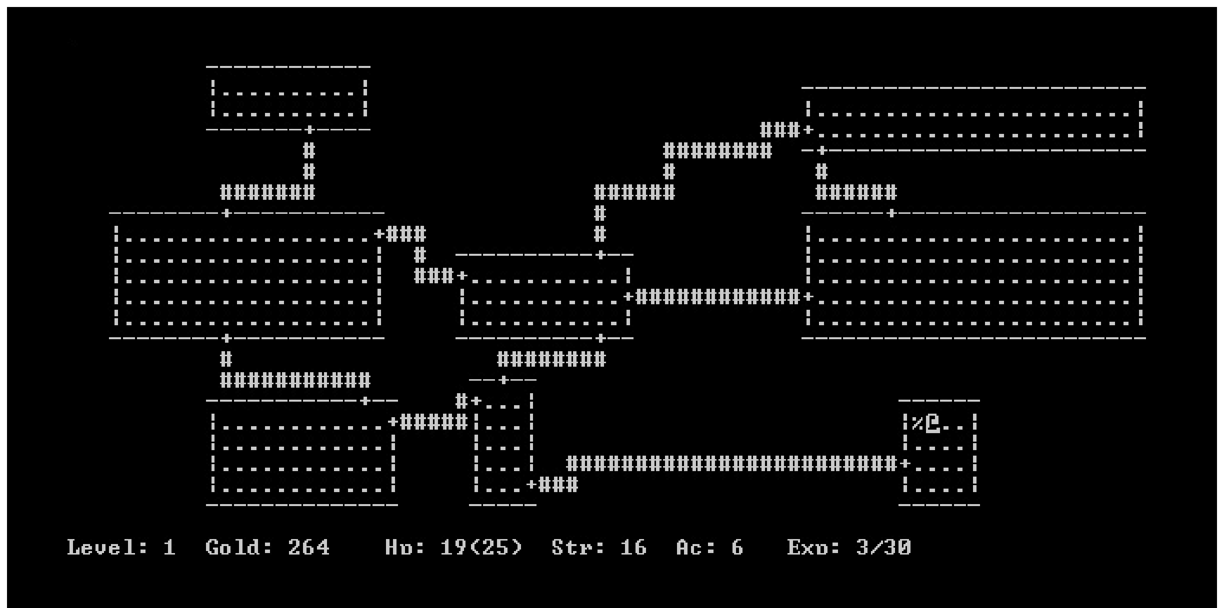


Рисунок 1.4 – Гра "Rogue" 1980-го року

У 1990-х роках Roguelike-ігри не були надто популярними, але з розвитком інді-ігор у 2000-х цей жанр почав відроджуватися. Ігри як "NetHack" та "ADOM" (Ancient Domains of Mystery) продовжували традиції класичного Roguelike, а пізніше з’явилися більш інноваційні та комерційно

успішні представники жанру, такі як "The Binding of Isaac", "Spelunky", "Dead Cells" та "Hades". Ці ігри іноді відносять до піджанру Roguelite, оскільки вони часто додають можливість певного прогресу, навіть якщо гравець програє. Сьогодні Roguelike-ігри є важливою частиною ігрової індустрії. Вони приваблюють гравців своєю непередбачуваністю, високою складністю та можливістю кожного разу відкривати нові механіки й стратегії.

1.4 Аналіз існуючих застосунків

Сучасні проєкти у жанрі Roguelike демонструють широкий спектр підходів до реалізації ключових механік жанру. Їх успіх полягає в умінні поєднувати традиційні складові, такі як процедурна генерація та перманентна смерть, із сучасними технологічними та дизайнерськими розробками. Це робить ігри жанру не лише привабливими для досвідчених гравців, які цінують класичний підхід, але й цікавими, а головне зрозумілими, для новачків, які прагнуть більш інтуїтивного ігроладу та візуальної привабливості [4].

Однією з важливих особливостей сучасних Roguelike-ігор є їхня адаптація під різні платформи, включно з консолями, комп’ютерами та мобільними пристроями, шо дозволяє охопити широку аудиторію та зробити жанр універсальним. Гравці можуть насолоджуватися складним, але водночас захопливим досвідом навіть на пристроях з обмеженими можливостями, завдяки оптимізації графіки та спрощенню управління без втрати основних ігрових механік.

Сучасні ігри жанру Roguelike часто використовують елементи інтерактивності, які виходять за рамки традиційного геймплею. Це можуть бути сюжетні елементи, які розвиваються з кожним новим проходженням, або індивідуалізовані системи розвитку персонажа, які зберігають частковий прогрес між сесіями. Такий підхід дозволяє підтримувати мотивацію гравця до повторного проходження гри, відкриваючи нові можливості або аспекти

сюжету.

Технологічний прогрес також вплинув на аудіовізуальну складову жанру. Якщо ранні Roguelike-ігри використовували текстову графіку або примітивну 2D-анімацію, то сучасні проєкти можуть похвалитися деталізованими візуальними ефектами, які підсилюють атмосферу гри. Саундтреки в таких іграх відіграють важливу роль у створенні емоційного зв’язку з гравцем, забезпечуючи глибше занурення у гру.

Ще один важливий аспект – це акцент на експериментах з механіками. Сучасні проєкти активно впроваджують інновації, що комбінують Roguelike із іншими жанрами, такими як стратегії, шутери чи RPG. Таким чином, сучасні Roguelike-проекти слугують чудовим прикладом того, як традиційні ідеї можуть розвиватися й адаптуватися до змінних потреб аудиторії. Вони демонструють, що жанр залишається актуальним, залишаючи простір для новаторства, зберігаючи при цьому свою ідентичність та основні цінності. Тож розглянемо деяких представників жанру.

1.4.1 Гра "Hades"

Гра "Hades" – це екшн-Roguelike гра від студії Supergiant Games, яка була випущена у 2020 році. Гра швидко здобула визнання завдяки своїй глибокій історії, динамічному геймплею та незвичайній комбінації елементів Roguelike з наративним розвитком.

Сюжет гри розгортається у грецькій міфології та слідує за сином Аїда Загреєм, який прагне втекти з підземного світу та дістатися до Олімпу, щоб розкрити правду про своє походження та знайти свою матір. Аїд, володар підземелля, постійно заважає йому самими різноманітними способами.

Гра вирізняється унікальним поєднанням процедурної генерації рівнів із поступовим відкриттям сюжету через численні діалоги та взаємодії з персонажами, кожен з яких має власну мотивацію та глибоко пророблений характер. Взаємодія гравця з героями, такими як Афіна, Артеміда чи Нікс, не

лише збагачує ігровий процес, а й сприяє поступовому розкриттю міфологічного світу, що створює відчуття живого, цілісного всесвіту, який реагує на дії та прогрес гравця.

Рисунок 1.5 – Геймплей гри "Hades"

Геймплей у "Hades" (рисунок 1.4) – це комбінація динамічного бою та глибокої персоналізації. Гравець проходить різні рівні підземного світу, а також на своєму шляху бореться із ворогами та босами, і використовує унікальну зброю, спеціальні здібності та дари від богів Олімпу, таких як Зевс, Афіна, Посейдон та Афродіта. Кожен божий дар змінює спосіб гри, додаючи унікальні здібності й посилення, що дозволяє комбінувати різні стратегії й стилі бою.

Однак "Hades" має і кілька недоліків. Насамперед це складність, що може відштовхнути гравців, які не мають досвіду у подібних іграх, що вимагають від людини доволі високого рівня реакції. У грі немає звичного для багатьох проєктів вибору складності, тому гравців, які хочуть просто поринути в сюжет, може чекати непрохідний бар’єр складності у вигляді різноманітних механік гри.

1.4.2 Гра "Enter the Gungeon"

"Enter the Gungeon" – це популярна Roguelike-гра, розроблена компанією Dodge Roll та випущена у 2016 році. Її жанр можна описати як шутер із видом зверху та елементами Roguelike. Гра поєднує динамічний геймплей із процедурною генерацією рівнів, різноманітною зброєю та складними ворогами. Сюжет гри розгортається навколо групи героїв, які вирушають до підземелля під назвою "Gungeon", щоб знайти легендарну зброю, що може знищувати минуле. Кожен із персонажів має свою унікальну мотивацію стерти певний спогад із життя, що додає особливостей до їхнього розвитку в грі. Геймплей (рисунок 1.5) побудований на швидких боях, ухиленні від куль та ретельному виборі зброї. У грі є безліч видів зброї – від звичайних пістолетів та автоматів до кумедних і фантастичних екземплярів, таких як риба-меч чи «пістолет-радуга».

Рисунок 1.6 – Геймплей гри "Enter the Gungeon"

Як плюсом, так і мінусом гри можна відзначити процедурну генерацію кожного рівня. Гравцю майже ніколи не попадеться один і той самий рівень з

однаковими ворогами. Але в той же час, це і мінус – так звана рандомна генерація не завжди буде на боці гравця. Якщо йому не пощастить з предметами, то спроба пройти рівень перестане залежати від навичок певного гравця і він з високим шансом програє.

1.4.3 Гра "Dead Cells"

"Dead Cells" (рисунок 1.6) – це популярна Roguelike-гра з елементами Metroidvania, яка була розроблена французькою студією Motion Twin та випущена у 2018 році. Гра поєднує елементи платформера, екшену та дослідження з процедурною генерацією рівнів, швидким бойовим процесом і системою постійної смерті, де кожна поразка означає початок гри з нуля.

Рисунок 1.7 – Геймплей гри "Dead Cells"

Водночас гравець має змогу поступово відкривати нову зброю, здібності та покращення, що робить кожне нове проходження унікальним. Стильна піксельна графіка та енергійний саундтрек доповнюють атмосферу гри, створюючи глибоке й затягуюче ігрове враження.

Сюжет гри розповідає про безіменного персонажа, відомого як "В’язень" або "Голова", який є реінкарнацією в’язня в тілі без голови. Він досліджує зруйнований острів та його небезпечні підземелля, намагаючись розкрити причини зараження, яке спустошило це місце. Гра відома мінімалістичним підходом до сюжету: гравець отримує лише натяки та уривки історії через діалоги, написи та оточення, що дає простір для власних інтерпретацій.

Геймплей зосереджений на динамічних боях, швидкому ухиленні від атак та прокачуванні персонажа. Гравець починає кожну гру з базовою зброєю, але під час проходження рівнів знаходить нові типи зброї, бонуси та здібності, які можуть змінити стиль бою. Гра надає велику кількість різної зброї: від мечів та луків до бомб та пасток, а також спеціальні мутації, які додають додаткові пасивні здібності. Різноманітність комбінацій дозволяє експериментувати з різними тактиками та адаптуватися до стилю гри кожного гравця. Крім того, складність гри зростає поступово, що підтримує відчуття виклику та мотивацію до подальших спроб.

Недоліком даної гри можна відзначити те, що гра має так звану бічну прокрутку або сайд-скролер. Тобто гра, у якої точка зору розташована збоку, а екранні персонажі зазвичай можуть рухатися лише ліворуч або праворуч. Не всім гравцям подобається дана графічна особливість проєкту.

1.4.4 Гра "Vampire Survivors"

"Vampire Survivors" – це гра в жанрі roguelike з елементами bullet hell, розроблена та видана Poncle (Лука Галанте), яка здобула значну популярність на початку 2022 року завдяки своєму мінімалістичному геймплею та високій реіграбельності. Гра характеризується простою піксельною графікою та зосереджена на поступовому нарощуванні сили персонажа шляхом збору досвіду та покращень зброї та пасивних предметів. Сюжет гри є досить умовним і слугує радше тлом для ігрового процесу. Гравець обирає одного з

доступних персонажів, кожен з яких має унікальну стартову зброю та пасивні бонуси, і потрапляє на арену, де йому доводиться безперервно відбиватися від хвиль все сильніших ворогів. Мета кожної ігрової сесії – протриматися якомога довше, ідеально – до досягнення встановленого ліміту часу (зазвичай 30 хвилин), після чого з'являється фінальний бос або подія.

Геймплей "Vampire Survivors" (рисунок 1.8) вирізняється своєю простотою управління: гравець контролює лише рух персонажа, тоді як атаки відбуваються автоматично. Ключовим елементом є збір дорогоцінних каменів досвіду, що випадають з переможених ворогів. Накопичивши достатньо досвіду, гравець отримує новий рівень та можливість обрати одне з кількох випадкових покращень: нову зброю, покращення існуючої зброї, пасивний предмет або покращення пасивного предмета. Зброя та предмети можуть синергувати між собою, створюючи потужні комбінації. Особливістю гри є "еволюції" зброї, які відбуваються при максимальному рівні певної зброї та наявності відповідного пасивного предмета, перетворюючи зброю на значно потужніший варіант.

Рисунок 1.8 – Геймплей гри "Vampire Survivors"

Однією з сильних сторін "Vampire Survivors" є відчуття прогресії. Навіть після програшу, гравець заробляє золото, яке можна витратити на постійні покращення для всіх персонажів або на розблокування нових героїв та предметів, що робить кожну наступну спробу трохи легшою та заохочує до подальшої гри.

Однак, попри свою захопливість, гра може здатися дещо одноманітною для деяких гравців через обмежену кількість карт на старті та повторюваність основного геймплейного циклу. Візуальний стиль, хоч і є навмисно простим, може не сподобатися тим, хто віддає перевагу більш деталізованій графіці. Тим не менш, успіх "Vampire Survivors" породив цілий піджанр ігор, що свідчить про ефективність її основних механік та привабливість для широкої аудиторії.

1.5 Постановка задачі

Протягом виконання кваліфікаційної роботи слід вирішити такі задачі:

- провести аналіз вже існуючих застосунків на дану тематику;
- розробити концепцію ігрового застосунку;
- здійснити пошук ігрових асетів (текстури, звуки, шрифти), які б вписувались у розроблену концепцію;
- створити дизайн проєкту;
- написати скрипти та розробити анімацію;
- провести тестування проєкту.

2 АНАЛІЗ ТЕХНОЛОГІЙ, ЩО ВИКОРИСТОВУЮТЬСЯ ПРИ РОЗРОБЦІ

2.1 Ігровий рушій Godot

Godot – це сучасний відкритий та безкоштовний рушій для розробки ігор, який забезпечує потужний набір інструментів для створення ігрових проєктів різного рівня складності [5]. Його популярність пояснюється поєднанням доступності, гнучкості та простоти використання, що робить його привабливим для інді-розробників та професійних студій. Рушій підтримує 2D та 3D графіку, дозволяючи створювати як прості казуальні ігри, так і складні імерсивні світи. Однією з ключових переваг Godot є його орієнтованість на кросплатформність. Розробники можуть створювати ігри для різноманітних платформ, включаючи Windows, macOS, Linux, Android, iOS та веббраузери [6]. Це досягається завдяки інтегрованій підтримці експорту проєктів у потрібний формат без необхідності використання сторонніх інструментів.

Архітектура Godot базується на сценах і вузлах, що дозволяє організувати проєкти логічно та інтуїтивно. Кожен вузол являє собою базовий елемент, який може мати властивості, функції та залежності [7]. Цей підхід спрощує управління великими проєктами та робить процес розробки більш прозорим. Інтерфейс Godot (рисунок 2.1) відзначається інтуїтивністю та орієнтованістю на потреби розробників. Він організований таким чином, щоб полегшити доступ до ключових інструментів та функцій, необхідних для створення ігор. Основний робочий простір поділений на кілька вкладок: 2D, 3D, Script та AssetLib, кожна з яких відповідає за певний аспект розробки. Це дозволяє розробникам швидко перемикатися між різними задачами без необхідності відкривати додаткові застосунки чи вікна.

Однією з найсильніших сторін інтерфейсу є візуальний редактор сцен. Завдяки системі вузлів розробники можуть створювати складні структури, просто додаючи та налаштовуючи елементи через редактор.

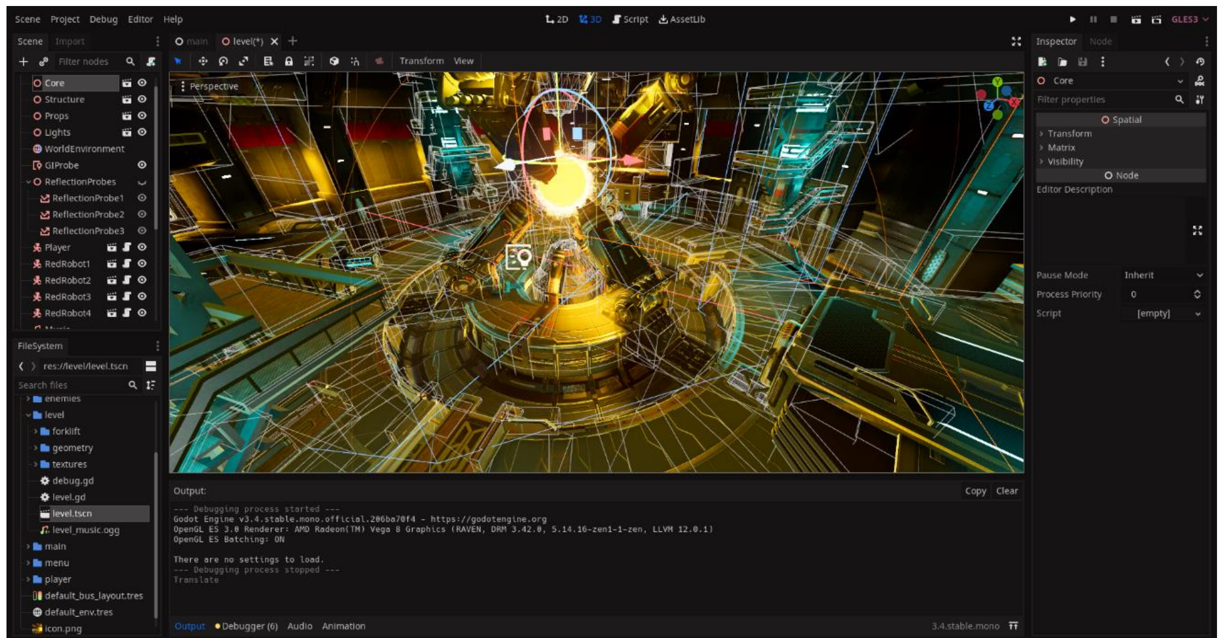


Рисунок 2.1 – Інтерфейс рушія

Інструменти для роботи з 2D та 3D об'єктами інтегровані в середовище, що дозволяє безпосередньо змінювати об'єкти, їхні властивості та позиціювання в реальному часі (рисунок 2.2).

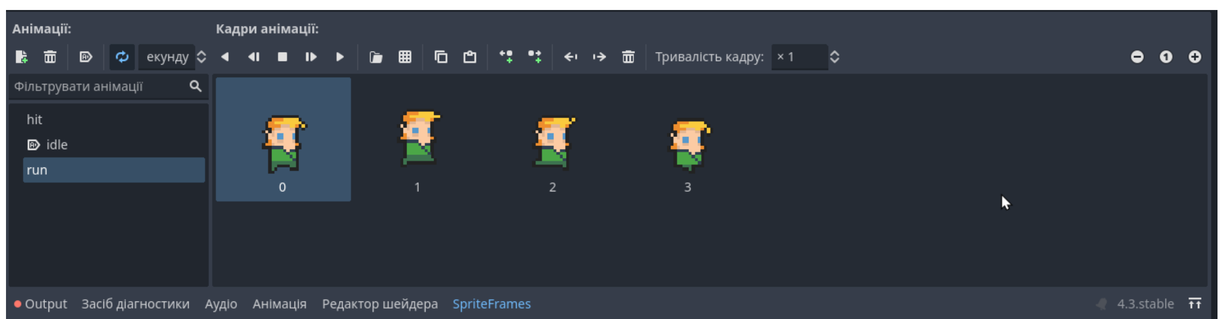


Рисунок 2.2 – Інтерфейс роботи з анімаціями

Панель інспектора надає доступ до властивостей обраного вузла, що спрощує налаштування поведінки та зовнішнього вигляду об'єктів. Крім того, вбудований редактор коду дозволяє писати скрипти безпосередньо в середовищі, що значно прискорює процес розробки. Інтерфейс підтримує підсвічування синтаксису, автозавершення коду та інтеграцію з системою відлагодження, роблячи роботу програмістів комфортнішою.

Ще однією важливою особливістю є можливість кастомізації

інтерфейсу. Розробники можуть налаштовувати розташування панелей, створювати власні шорткати та організовувати робочий простір відповідно до своїх потреб. Це особливо корисно для великих проєктів, де оптимізація часу та зручність роботи мають велике значення.

Крім того, Godot має власну мову програмування GDScript, яка нагадує Python за синтаксисом і створена спеціально для роботи з рушієм. Вона дозволяє швидко реалізовувати ігрову логіку без необхідності глибоких знань програмування. Рушій також підтримує інші мови, такі як C#, C++ та VisualScript, що розширює можливості інтеграції та адаптації.

Завдяки активній спільноті та відкритому коду, Godot постійно вдосконалюється, отримуючи нові функції та виправлення, що робить його не лише технічно досконалим, але й привабливим з погляду довгострокового використання для будь-яких типів проєктів.

2.2 Мова програмування GDScript

GDScript – це спеціалізована мова програмування, яка була створена командою Godot для розробки ігрової логіки. Вона інтегрована безпосередньо в рушій, що забезпечує тісний зв'язок з усіма його функціями та інструментами. Основною метою GDScript є спрощення роботи з ігровими механіками та швидка реалізація задумів розробників без надмірного ускладнення процесу.

Мова має синтаксис, схожий на Python, що робить її легкою для вивчення, особливо для початківців. Завдяки простій структурі коду та мінімалістичному дизайну, програмісти можуть уникати зайвих технічних деталей, зосереджуючись на створенні ігрових функцій. Наприклад, відсутність необхідності в оголошенні змінних типів дозволяє писати код швидше, водночас GDScript підтримує статичну типізацію для підвищення надійності та оптимізації [7]. В лістингу 2.1 наведено код, який ілюструє послідовність Фібоначчі.

Лістинг 2.1 – Послідовність Фібоначчі

```
func _ready():
     var nterms = 5
     print("Fibonacci sequence:")
     for i in range(nterms):
          print(fibonacci(i))
func fibonacci(n):
     if n <= 1:
          return n
     else:
          return fibonacci(n - 1) + fibonacci(n - 2)
```

Однією з найпотужніших сторін GDScript є його інтеграція з системою вузлів Godot. Наприклад, через мову легко взаємодіяти з будь-яким елементом сцени, змінювати його властивості або викликати функції. Така інтеграція дозволяє скоротити кількість необхідного коду та спрощує взаємодію між компонентами гри.

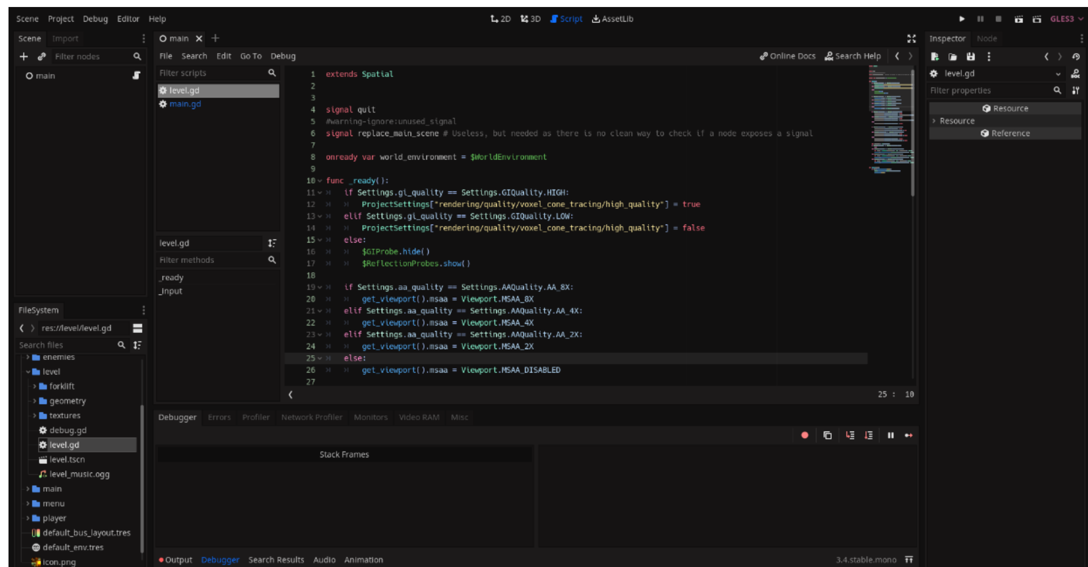


Рисунок 2.3 – Редагування файлу GDScript за допомогою вбудованого редактора

GDScript також підтримує об’єктно-орієнтований підхід, так званий принцип ООП, включаючи класи, наслідування та перевизначення методів, що дозволяє організовувати складні ігрові системи в більш структурований спосіб. Крім того, у мові реалізовані сигнали (аналог подій), які полегшують

комунікацію між вузлами, забезпечуючи гнучкість у створенні взаємодії.

Ще однією важливою перевагою GDScript є його продуктивність у контексті Godot. Оскільки мова створена саме для цього рушія, її виконання оптимізоване для максимальної ефективності. Хоча вона може поступатися за швидкістю виконання таким мовам, як C++, у більшості ігрових сценаріїв це непомітно завдяки високій продуктивності рушія [8].

Активна підтримка спільноти й документація роблять GDScript зручним інструментом як для новачків, так і для досвідчених розробників. Вона ідеально підходить для швидкого прототипування та розробки повноцінних ігор у межах Godot, забезпечуючи баланс між простотою та функціональністю.

2.3 Графічний редактор Aseprite

Aseprite – це графічний редактор, призначений для створення піксельної графіки та анімації [9]. Він широко використовується інді-розробниками, художниками та дизайнерами для створення 2D-спрайтів, іконок, а також анімацій для відеоігор [10]. Однією з головних особливостей Aseprite є його інтерфейс (рисунок 2.4), який оптимізований для роботи з піксельною графікою. Він дозволяє ефективно створювати та редагувати пікселі, забезпечуючи максимальну точність та контроль. Програма має компактний розмір і невисокі системні вимоги, що робить її доступною для широкого кола користувачів, навіть на менш потужних пристроях. Крім того, Aseprite підтримує роботу з шарами, що дозволяє розділяти різні елементи анімації або композиції, а також працювати з кількома зображеннями одночасно. Це значно підвищує продуктивність під час.

Інструменти для анімації в Aseprite є ще однією важливою перевагою цього редактора. Програма дозволяє створювати покадрові анімації, використовуючи так званий таймлайн для налаштування послідовності кадрів та швидкості анімації. Це дає можливість художникам точніше

налаштовувати рухи, візуальні ефекти та зміну станів персонажів або об'єктів. Aseprite також підтримує використання "петлі" для створення безперервних анімацій, що особливо корисно для ігор, де потрібні циклічні рухи. Завдяки функції попереднього перегляду, користувач може одразу оцінити результат анімації та внести необхідні корективи до кожного кадру.

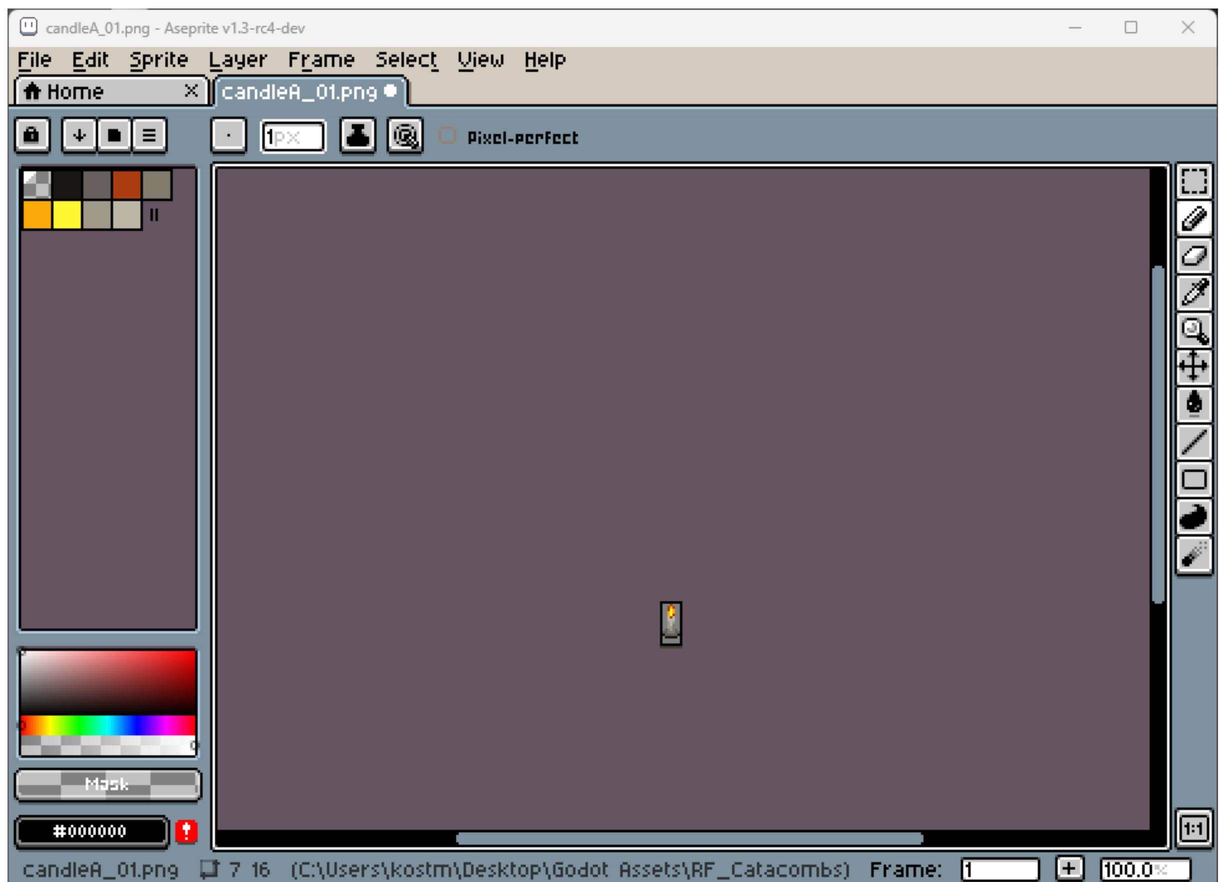


Рисунок 2.4 – Інтерфейс застосунку Aseprite

Ще однією важливою рисою Aseprite є його зручна підтримка різних форматів файлів, таких як PNG, GIF, BMP, а також спеціалізовані формати для анімацій, які можна безпосередньо імпортувати в ігрові рушії, що забезпечує легкість інтеграції готових графічних ресурсів у розробку ігор чи додатків [11]. Інтерфейс програми надає художнику широкі можливості для налаштування палітри кольорів та інструментів, що дозволяє досягти потрібного стилю графіки для конкретного проєкту.

Aseprite також підтримує автоматизацію деяких процесів за допомогою

скриптів, що дає можливість прискорити рутинні операції, наприклад, для генерації анімацій або роботи з великими наборами спрайтів, що робить його корисним інструментом не тільки для одиничних художників, але й для команд, які працюють над великими проєктами. Завдяки своїй простоті, потужним інструментам для анімації та широким можливостям для інтеграції в ігрові рушії, Aseprite є важливим інструментом для розробників 2D ігор, особливо для тих, хто створює ретростилеві або піксельні проєкти.

2.4 Патерн проєктування Singleton

Singleton (рисунок 2.5) – це патерн, який забезпечує наявність лише одного екземпляру класу та надає глобальний доступ до цього екземпляра [12].

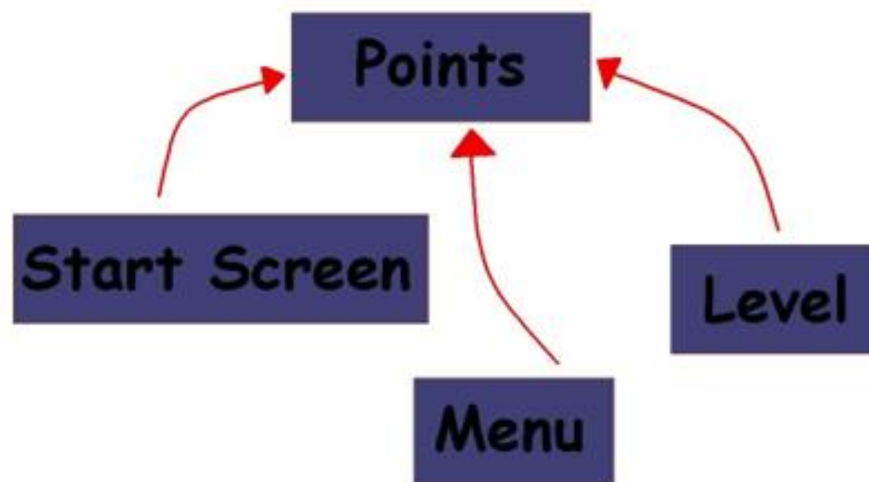


Рисунок 2.5 – Приклад патерну Singleton

У Godot цей патерн часто використовується для управління глобальними налаштуваннями, менеджерами ресурсів або системами, такими як аудіо-менеджер або менеджер рівнів. Він дозволяє легко отримати доступ до централізованих функцій або даних з будь-якої частини гри. Це особливо корисно для управління станом гри, наприклад, зберігання інформації про стан гравця, налаштувань гри або управління ігровими ресурсами.

2.5 Патерн проєктування Observer

Observer – це патерн, який дозволяє одному об'єкту (суб'єкту) повідомляти інші об'єкти (спостерігачів) про зміни свого стану (рисунок 2.6). У Godot цей патерн можна реалізувати за допомогою сигналів. Він забезпечує зв'язок між різними частинами гри, дозволяючи одному об'єкту реагувати на зміни в іншому, що особливо корисно для створення динамічних ігрових систем. Наприклад, це може бути використано для оновлення інтерфейсу користувача при зміні стану персонажа або для синхронізації стану між різними компонентами гри.

Завдяки сигналам, які реалізують принцип слабкого зв’язування, об’єкти не потребують прямого доступу один до одного, що спрощує підтримку та масштабування коду. Це особливо важливо в розробці великих проєктів, де численні об’єкти повинні залишатися незалежними, але водночас координовано взаємодіяти. Такий підхід сприяє кращій модульності, полегшує тестування окремих компонентів та дозволяє розробникам швидше вносити зміни без ризику зробити критичну помилку.

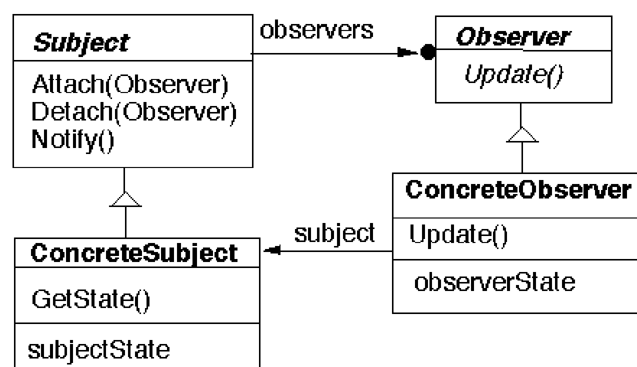


Рисунок 2.6 – Приклад патерну Observer

2.6 Патерн проєктування Factory Method

Factory Method – це патерн, який визначає інтерфейс для створення об'єктів у суперкласі, але дозволяє підкласам змінювати тип створюваних

об'єктів (рисунок 2.7). У Godot цей патерн можна використовувати для створення різних типів ворогів або предметів [13]. Він дозволяє абстрагувати процес створення об'єктів, роблячи код більш гнучким та легким для розширення. Це особливо корисно для ігор з великою кількістю різних об'єктів, де потрібно легко додавати нові типи ворогів або предметів без значних змін в коді, що вже існує.

У практичному застосуванні це може виглядати як базовий клас EnemySpawner, який визначає метод для створення ворогів, а його підкласи реалізують створення конкретних типів ворогів із різними параметрами, анімаціями чи поведінкою. Такий підхід не лише спрощує масштабування ігрових систем, а й дозволяє уникати дублювання коду, зберігаючи архітектуру проєкту чистою та керованою.

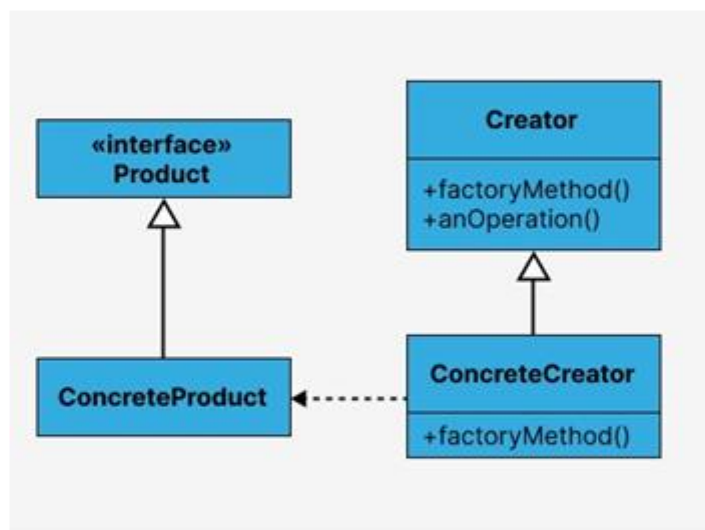


Рисунок 2.7 – Концепція патерну Factory Method

Завдяки використанню Factory Method у Godot можна централізовано керувати логікою створення об’єктів, що спрощує відлагодження та тестування. Наприклад, у грі зі зростаючим рівнем складності патерн дозволяє динамічно створювати ворогів відповідно до поточного етапу або умов гри, без необхідності змінювати основну логіку. Крім того, цей підхід полегшує повторне використання коду в різних сценах, оскільки механізм

створення об’єктів винесений в окремий, незалежний компонент. У результаті Factory Method сприяє підвищенню гнучкості та стійкості ігрової архітектури до змін.

2.7 Патерн проєктування Component

Component – це патерн, що дозволяє інкапсулювати окремі функції або поведінку в самостійні об'єкти (компоненти), які можна комбінувати для створення складніших об'єктів (рисунок 2.8). У Godot цей підхід реалізовано через систему вузлів (nodes), де кожен вузол є окремим компонентом із власними властивостями та методами. Кожен з них відповідає за певний аспект поведінки об’єкта, що допомагає уникнути дублювання коду та забезпечує структуровану архітектуру. Компоненти легко додавати, видаляти чи змінювати без впливу на загальну логіку.

Наприклад, ігровий персонаж може складатися з вузлів для руху, анімації, обробки зіткнень та системи здоров’я. Замість монолітного класу така структура забезпечує гнучкість, повторне використання компонентів в інших об’єктах і полегшує командну роботу, дозволяючи розробникам працювати над різними частинами незалежно [14].

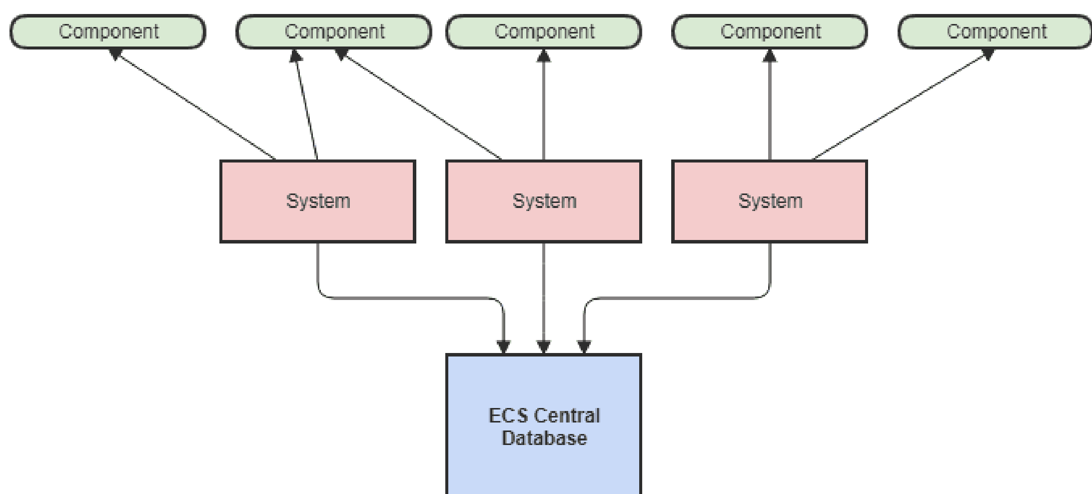


Рисунок 2.8 – Концепція патерну Component

3 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ ЗАСТОСУНКУ

3.1 Огляд структури проєкту

Програмна реалізація ігрового застосунку базується на ігровому рушії Godot Engine, який надає гнучку систему організації проєктів. Структура файлів проєкту (рисунок 3.1) є логічною та відображає модульний підхід до розробки. Коренева директорія проєкту, позначена як res://, містить усі ресурси та скрипти, необхідні для функціонування гри.

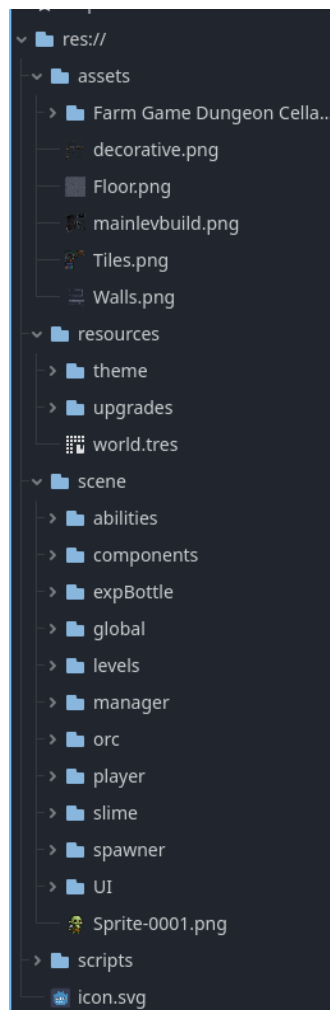


Рисунок 3.1 – Структурне дерево проєкту

Така структуризація дозволяє ефективно керувати проєктом, забезпечує модульність та полегшує навігацію між різними компонентами

гри. Кожна директорія має чітко визначене призначення, що сприяє підтримці та розширенню функціонала застосунку.

3.2 Каталог scene

Детальніше розглянемо каталог scene, так як він є ключовим каталогом основного функціоналу гру. Директорія містить файли сцен (.tscn або .scn), які є основними будівельними блоками в Godot. Сцени представляють собою ієрархію вузлів (Nodes) і можуть описувати окремих персонажів, елементи інтерфейсу, рівні гри або будь-які інші ігрові об'єкти. Ця директорія має розгалужену структуру для кращої організації:

- abilities: містить сцени, що реалізують різноманітні здібності персонажів;
- components: містить вузли або сцени, які багаторазово використовуються. Вони додають певну функціональність іншим об'єктам (наприклад, компонент здоров'я, руху, атаки);
- expBottle: сцени, пов'язані з об'єктами, що надають досвід гравцю (пляшки з досвідом);
- global: використовується для розміщення глобальних скриптів або сцен, доступних з будь-якої частини гри, наприклад, обробник подій, який сигналізує про підвищення рівня гравця;
- levels: призначена для зберігання сцени, що описує ігровий рівень;
- manager: містить сцени, що керують певними аспектами гри, наприклад, менеджер досвіду, менеджер часу, менеджер апргрейдів;
- orc: вказує на сцени, пов'язані з ігровими персонажами-орками (вороги);
- player: містить сцени, що описують гравця, його анімації, логіку управління;
- slime: аналогічно до orc, призначена для сцен, пов'язаних з ворогами-слаймами;

- spawner: містить сцену, відповідальну за створення (спавн) ворогів;
- UI: містить сцени, що формують інтерфейс користувача (меню, таймер, інтерфейс прокачування тощо).

3.3. Структура та функціональність сцени Level

Центральним елементом ігрового процесу є сцена Level (рисунок 3.2), яка представляє собою основний ігровий рівень та находиться у файлі level.tscn. Вона побудована на основі вузла типу Node і містить у собі різноманітні дочірні вузли, що відповідають за різні аспекти ігрової логіки, візуалізації та взаємодії. Сцена Level також має власний скрипт, що визначає її поведінку та керує взаємодією між її компонентами.

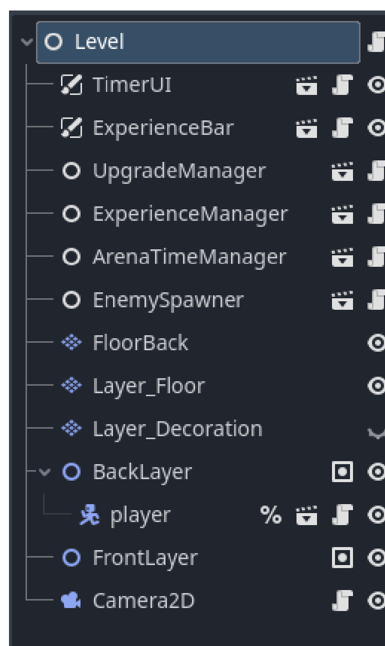


Рисунок 3.2 – Ієрархія сцени Level

3.3.1. Ієрархія вузлів сцени Level

Ієрархія вузлів сцени Level ретельно структурована для забезпечення належного функціонування та візуального представлення ігрового світу (рисунок 3.3):

- TimerUI: вузол, який відповідає за відображення ігрового таймера;
- ExperienceBar: вузол, що візуалізує прогрес набору досвіду гравцем. Це стандартна смуга прогресу, яка заповнюється в міру отримання досвіду;
- UpgradeManager: вузол, який керує системою покращень для гравця. Він відповідає за відображення доступних покращень, обробку та застосування відповідних змін до характеристик або здібностей персонажа;
- ExperienceManager: менеджер, що відповідає за логіку нарахування та обробки досвіду;
- ArenaTimeManager: вузол, що керує часом на арені, а точніше збільшує складність ворогів протягом певного відрізку часу;
- EnemySpawner: відповідає за створення (спавн) ворогів на ігровому рівні;
- FloorBack: графічний елемент, що представляє задній фон підлоги ігрового середовища;
- Layer_Floor: шар, що відповідає за візуалізацію основної поверхні підлоги, по якій переміщуються персонажі;
- Layer_Decoration: шар для розміщення декоративних елементів на рівні, які не впливають на ігрову механіку, але покращують візуальне сприйняття;
- BackLayer: вузол, що слугує контейнером для елементів, які повинні рендеритися позаду основного ігрового персонажа;
- player: ключовий вузол, що представляє ігрового персонажа, яким керує гравець;
- FrontLayer: вузол, що слугує контейнером для елементів, які повинні рендеритися попереду основного ігрового персонажа, наприклад, ефекти шкоди;
- Camera2D: вузол камери, що визначає видиму область ігрового світу та слідує за гравцем.

Така ієрархія забезпечує чітке розділення відповідальності між компонентами сцени, полегшує керування порядком відображення шарів та

взаємодію між ігровими об'єктами. Дана ієрархія не заважає при необхідності масштабуванню або модернізації сцени, що особливо актуально, коли ігровий застосунок знаходиться на етапі розробки.

Рисунок 3.3 – Візуальне представлення сцени Level

3.3.2. Скриптова логіка сцени Level

Сцена Level має власний скрипт, який розширює базовий клас Node та реалізує ключову логіку рівня. Розглянемо його основні аспекти (лістинг 3.1).

Лістинг 3.1 – Скрипт сцени Level

```
extends Node
@export var end_screen_scene:PackedScene
@onready var player: CharacterBody2D = %player
# Called when the node enters the scene tree for the first time.
func _ready() -> void:
	player.health_component.died.connect(on_died)
# Called every frame. 'delta' is the elapsed time since the
previous frame.
func _process(delta: float) -> void:
	pass
func on_died():
	var end_screen_instance = end_screen_scene.instantiate() as
EndScreen
	add_child(end_screen_instance)
```

Основними змінними скрипту є два визначення, а саме end_screen_scene: PackedScene та player: CharacterBody2D. Перша змінна є експортованою, що очікує на присвоєння упакованої сцени (PackedScene) екрану завершення гри, що дозволяє легко налаштувати, який екран буде показаний після поразки, безпосередньо з редактора Godot (рисунок 3.4). Друга ж змінна отримує посилання на вузол гравця (%player) при готовності сцени (@onready). Це забезпечує доступ до об'єкта гравця для взаємодії.

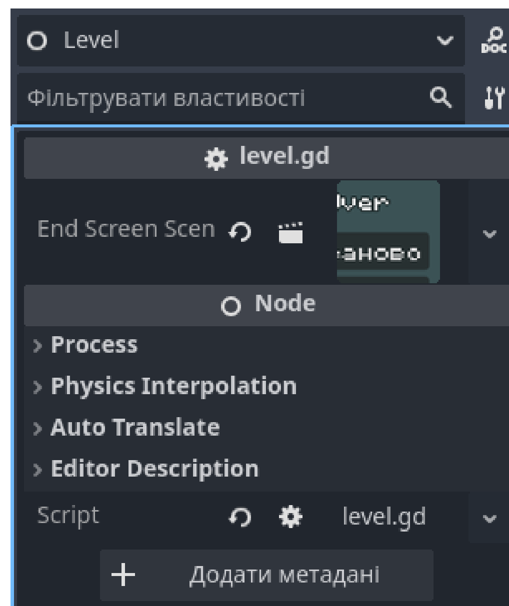


Рисунок 3.4 – Налаштування сцени та присвоєння сцени поразки

Окрім змінних, в скрипті є декілька методів. Функція _ready() викликається один раз, коли вузол Level та його дочірні вузли входять до дерева активних сцен. У цьому методі відбувається підключення до сигналу died компонента здоров'я гравця (player.health_component.died). Сигнал died спрацьовує, коли здоров'я гравця досягає нуля.

Метод _process(delta: float) на даний метод не використовується, однак, він може бути використаний у майбутньому для реалізації логіки, що потребує постійного оновлення.

Метод on_died() викликається, коли гравець гине (отримано сигнал died). Також тут створюється екземпляр сцени екрану завершення гри

(end_screen_scene.instantiate()). Результат приводиться до типу EndScreen, що передбачає наявність однойменного скрипта у сцени завершення. Окрім цього створений екземпляр екрану завершення (end_screen_instance) додається як дочірній вузол до сцени Level, що робить його видимим та активним.

3.4 Структура сцени Player

Розробка ігрового персонажа є ключовим аспектом створення будь-якої гри. Сцена гравця (Player) в проєкті реалізована як CharacterBody2D, що є стандартним вузлом в Godot Engine для створення персонажів, які можуть рухатися та взаємодіяти з фізичним світом гри. Сцена гравця (рисунок 3.5) має ієрархічну структуру, що складається з кількох вузлів, кожен з яких відповідає за певну функціональність персонажа:

- Player: кореневий вузол сцени, що визначає фізичну поведінку гравця, такий як рух та зіткнення. До цього вузла прикріплений основний скрипт, що керує логікою гравця;
- AbitlityManager: вузол типу Node, призначений для управління здібностями гравця. Він слугує контейнером для екземплярів різних здібностей, які гравець може отримати або активувати протягом гри;
- AttackController: цей вузол відповідає за логіку атаки гравця;
- HealthComponent: компонент, що інкапсулює логіку здоров'я гравця. Він містить інформацію про максимальне та поточне здоров'я, а також методи для отримання та втрати здоров'я;
- GraceTime: вузол Timer, що використовується для реалізації "часу невразливості" (grace time) після отримання гравцем пошкодження. Це запобігає отриманню кількох пошкоджень поспіль за короткий час;
- PickUpArea: вузол типу Area2D, що визначає зону, в межах якої гравець може підбирати предмети;
- CollisionShape2D: дочірній вузол PickUpArea, що визначає фізичну

форму зони підбору;

- AnimatedSprite2D: відповідає за візуальне представлення гравця за допомогою анімацій (наприклад, анімація простою, бігу);

- CollisionShape2D: основний фізичний колайдер гравця, що прикріплений безпосередньо до CharacterBody2D. Він визначає фізичні межі гравця для зіткнень зі світом гри;

- PlayerHurtBox: вузол Area2D, представляє зону, при попаданні в яку ворогів гравець отримує пошкодження;

- CollisionShape2D: дочірній вузол PlayerHurtBox, що визначає форму цієї зони пошкодження;

- ProgressBar: візуальний елемент, що відображає поточний рівень здоров'я гравця.

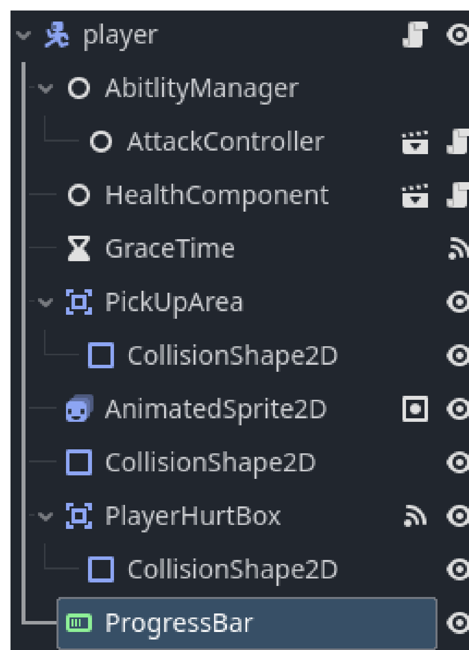


Рисунок 3.5 – Структура сцени Player

3.4.1 Скрипт сцени Player

Основний скрипт, що керує поведінкою гравця, реалізує комплексну взаємодію персонажа з ігровим світом та його внутрішніми станами, має назву player.gd. При ініціалізації сцени, у функції _ready(), встановлюються

необхідні зв'язки між компонентами гравця. Наприклад, підключається реакція на смерть персонажа та зміну його здоров'я. Також підключаються обробники глобальних подій, такі як додавання нових покращень здібностей. Початкове значення здоров'я гравця одразу відображається на відповідному індикаторі. У кожному кадрі гри у функції _process(delta) скрипт обробляє введення користувача для визначення напрямку руху. Це реалізовано через окрему функцію movement_vector() (лістинг 3.2).

Лістинг 3.2 – Визначення вектора руху

```
func movement_vector() -> Vector2:
     var movement_X = Input.get_action_strength("move_right") -
Input.get_action_strength("move_left")
     var movement_Y = Input.get_action_strength("move_down") -
Input.get_action_strength("move_up")
     return Vector2(movement_X, movement_Y)
```

На основі цього вектору розраховується цільова швидкість, до якої плавно інтерполюється поточна швидкість персонажа, забезпечуючи плавний рух. Загальне переміщення та обробка зіткнень здійснюється за допомогою вбудованої функції move_and_slide(). Візуалізація гравця динамічно змінюється: залежно від наявності руху активуються анімації бігу або простою, а напрямок руху по горизонталі визначає, чи потрібно віддзеркалювати спрайт персонажа.

Механіка отримання пошкоджень реалізована через спеціальну зону навколо гравця (PlayerHurtBox). Скрипт відстежує кількість ворогів, що увійшли в цю зону. При контакті з ворогом гравцю завдається шкода, що ілюструється у функції check_if_damaged() (лістинг 3.3).

Лістинг 3.3 – Логіка отримання пошкоджень

```
func check_if_damaged():
     if enemies_colliding == 0 || !grace_time.is_stopped():
          return
     health_component.take_damage(1)
     grace_time.start()
```

Після отримання шкоди запускається таймер невразливості (grace_time), що запобігає миттєвому отриманню численних пошкоджень. Після закінчення цього таймера, якщо вороги все ще знаходяться в зоні контакту, гравець може знову отримати шкоду.

Скрипт також відповідає за реакцію на зміну стану здоров'я. При отриманні пошкоджень візуальний індикатор здоров'я гравця (ProgressBar) оновлюється. У випадку, коли здоров'я гравця досягає нуля, спрацьовує логіка смерті, яка призводить до видалення об'єкта гравця з ігрової сцени.

Окрім базових функцій, скрипт інтегрований із системою здібностей. При отриманні сигналу про додавання нового покращення, скрипт перевіряє тип цього покращення. Якщо це нова здібність, відповідна сцена здібності інстанціюється та додається до менеджера здібностей гравця, розширюючи його можливості. Цей процес відбувається у функції on_ability_upgrade_added() (лістинг 3.4).

Лістинг 3.4 – Логіка додавання нової здібності

```
func on_ability_upgrade_added(upgrade:AbilityUpgrade, 
current_upgrades:Dictionary):
	if not upgrade is NewAbility:
		return
	var new_ability = upgrade as NewAbility
	ability_manager.add_child(new_ability.new_ability_scene.ins
tantiate())
```

3.5 Архітектура компонентів

Для забезпечення гнучкості та перевикористання коду в проєкті застосовується компонентний підхід до розробки ігрових сутностей (рисунок 3.6). Замість створення монолітних класів, що описують всю поведінку об'єкта, функціональність розбивається на окремі, незалежні модулі. Ці компоненти потім додаються до ігрових об'єктів (вузлів) для надання їм специфічної поведінки. Такий підхід дозволяє легко комбінувати різні функціональні блоки, створюючи різноманітних персонажів та об'єкти.

Наприклад, компонент здоров'я (health_component) інкапсулює всю логіку, пов'язану з отриманням пошкоджень, зміною поточного здоров'я та сигналізацією про смерть об'єкта. Цей компонент може бути використаний як для гравця, так і для ворогів.

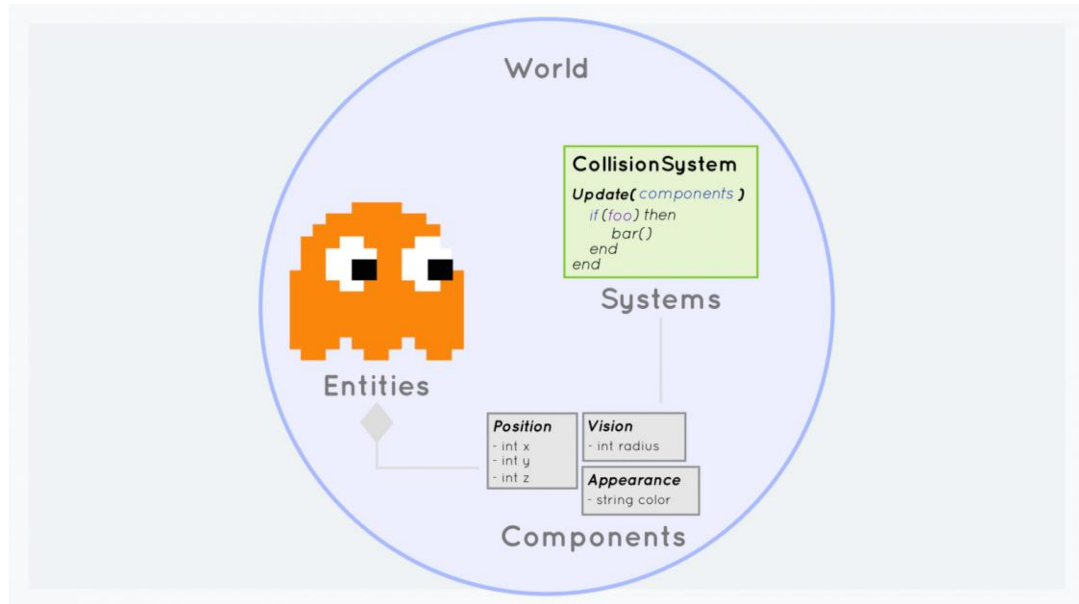


Рисунок 3.6 – Компонентний підхід до розробки

Інші компоненти, такі як компонент випадіння предметів (exp_bottle_drop_component), компонент для обробки влучань (hit_box_component та hurt_box_component) або компонент руху (movement_component), також реалізують специфічну, чітко визначену частину функціональності. Наприклад, компонент випадіння предметів може бути приєднаний до будь-якого ворога, і він буде відповідати за створення екземпляра пляшки з досвідом після смерті цього ворога, базуючись на сигналі від його компонента здоров'я.

Компоненти часто взаємодіють між собою через систему сигналів Godot або через прямі посилання, якщо це доцільно. Наприклад, компонент, що відповідає за отримання шкоди (HurtBoxComponent) при виявленні зіткнення з атакуючим об'єктом (HitBoxComponent) звертається до приєднаного до нього компонента здоров'я для реєстрації пошкодження

(лістинг 3.5). Це сприяє слабкому зв'язуванню між модулями, що полегшує їх тестування, модифікацію та заміну.

Лістинг 3.5 – Приклад взаємодії між компонентами

```
extends Area2D
class_name HurtBoxComponent
@export var health_component: HealthComponent
# Called when the node enters the scene tree for the first time.
func _ready() -> void:
	pass # Replace with function body.
func _on_area_entered(area: Area2D) -> void:
	if not area is HitBoxComponent:
		return
	if health_component == null:
		return
	var hit_box_component = area as HitBoxComponent
	health_component.take_damage(hit_box_component.damage)
```

Структура файлів у каталозі components (рисунок 3.7), де кожен компонент представлений як скриптом (.gd), так і окремою сценою (.tscn), свідчить про те, що компоненти можуть мати не тільки логіку, а й власну візуальну або вузлову структуру. Це дозволяє створювати складні, але керовані ігрові сутності шляхом об’єднання простих та зрозумілих будівельних блоків.

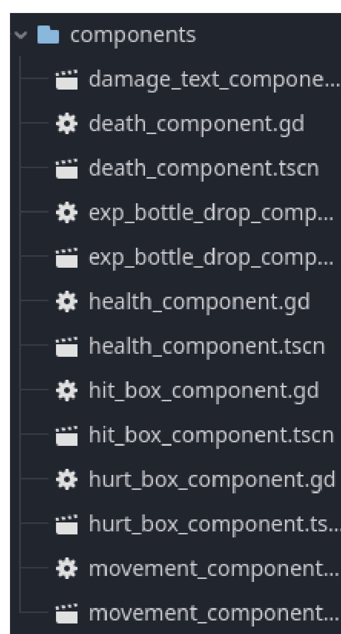


Рисунок 3.7 – Каталог components

3.6 Система генерації ворогів

Для забезпечення постійного ігрового виклику в проєкті реалізована система генерації ворогів, представлена сценою EnemySpawner (рисунок 3.8). Ця система відповідає за періодичне створення екземплярів ворогів на ігровій арені, а також за адаптацію частоти їх появи залежно від поточного рівня складності гри.

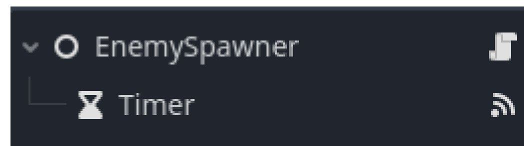


Рисунок 3.8 – Структура сцени EnemySpawner

В основі EnemySpawner лежить вузол типу Node, до якого прикріплений скрипт, що керує логікою генерації. Важливим елементом сцени є дочірній вузол Timer, який визначає інтервали між появою нових ворогів. При ініціалізації EnemySpawner створюється пул ворогів (EnemyPool), до якого додаються різні типи ворогів (наприклад, орки) із зазначенням їх відносної ймовірності появи. Система також підписується на глобальну подію збільшення рівня складності, що дозволяє динамічно коригувати час між генераціями.

Коли спрацьовує таймер, система вибирає випадковий тип ворога з пулу та створює його екземпляр. Позиція для появи нового ворога визначається спеціальною функцією, яка намагається розмістити ворога поза прямою видимістю гравця, але на певній відстані від нього. Після створення, екземпляр ворога додається на відповідний ігровий шар.

Зі зростанням рівня складності гри, базовий час очікування таймера зменшується (але не нижче певного мінімального значення), що призводить до частішої появи ворогів. Також, при досягненні певних рівнів складності, до пулу можуть додаватися нові, сильніші типи ворогів (наприклад, слайми),

що урізноманітнює ігровий процес та підвищує його динаміку.

Особливу увагу слід звернути на механізм визначення позиції для появи ворога, реалізований у функції get_spawn_position() (лістинг 3.6).

Лістинг 3.6 – Механізм визначення позиції ворогів

```
func get_spawn_position():
	var player = get_tree().get_first_node_in_group("player")
as Node2D
	var spawn_pos = Vector2.ZERO
	var random_direction = Vector2.RIGHT.rotated(randf_range(0,
TAU))
	var random_distance = randi_range(380, 500)
	for i in 24:
		spawn_pos = player.global_position + (random_direction
* random_distance)
		var raycast =
PhysicsRayQueryParameters2D.create(player.global_position,
spawn_pos, 1)
		var intersection =
get_tree().root.world_2d.direct_space_state.intersect_ray(raycas
t)
		if intersection.is_empty(): break
		else: random_direction =
random_direction.rotated(deg_to_rad(15))
	return spawn_pos
```

Робота цієї функції полягає в пошуку підходящої точки навколо гравця. Спочатку отримується посилання на об'єкт гравця. Далі генерується випадковий напрямок (random_direction) та випадкова відстань (random_distance) від гравця в заданому діапазоні (від 380 до 500 пікселів). Потенційна позиція для появи (spawn_pos) розраховується на основі позиції гравця, випадкового напрямку та відстані.

Після цього виконується перевірка за допомогою кидання променю (raycast) від позиції гравця до розрахованої spawn_pos. Мета цієї перевірки – впевнитися, що між гравцем та потенційною точкою появи ворога немає перешкод у вигляді стін, колон тощо. Якщо промінь нічого не перетинає (intersection.is_empty()), це означає, що позиція вважається "чистою" та підходить для генерації ворога, тому цикл пошуку переривається.

Якщо ж промінь перетинає якусь перешкоду, це означає, що обрана

позиція не є оптимальною (наприклад, ворог може з'явитися за стіною, недосяжним для гравця). У такому разі випадковий напрямок random_direction повертається на 15 градусів, і робиться нова спроба знайти підходящу позицію. Цей процес повторюється до 24 разів. Якщо за 24 спроби "чиста" позиція не знайдена, повертається остання розрахована spawn_pos. Такий підхід дозволяє генерувати ворогів у зонах, які будуть доступні для гравця, уникаючи появи ворогів всередині стін або інших непрохідних об'єктів, забезпечуючи при цьому певну випадковість.

4 ІНСТРУКЦІЯ КОРИСТУВАЧА

При першому запуску гри користувач потрапляє до головного меню (рисунок 4.1). Це стартовий екран, що надає основні опції для початку взаємодії з грою. Дизайн меню виконаний у мінімалістичному стилі, характерному для загальної візуальної концепції проєкту.

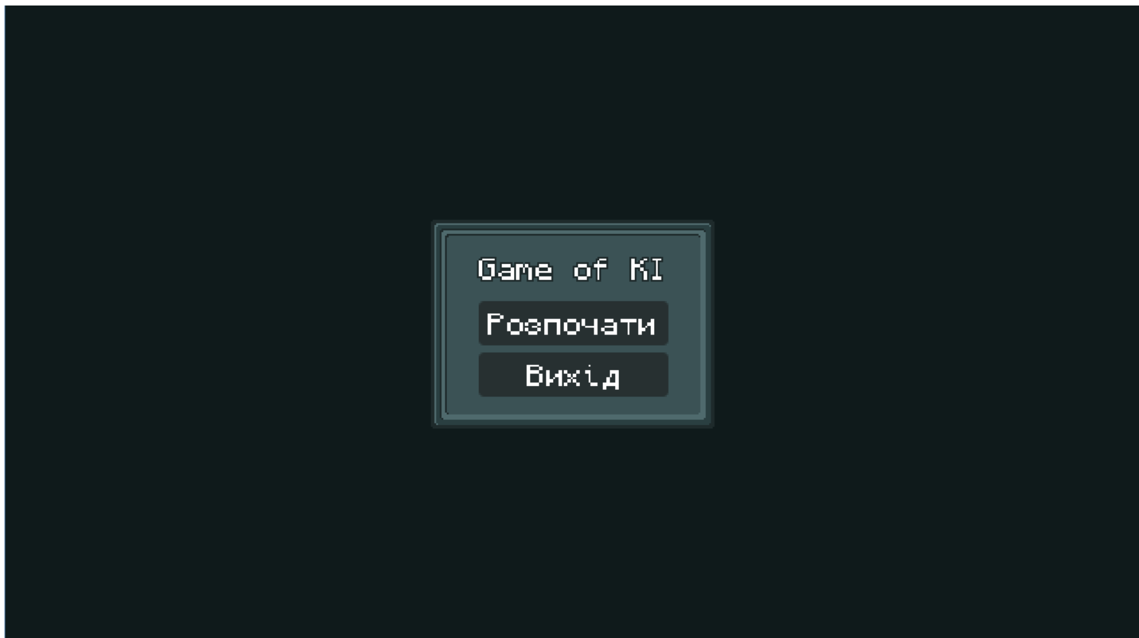


Рисунок 4.1 – Головне меню гри

На екрані головного меню присутні наступні елементи:

- назва гри ("Game of KI"): відображається у верхній частині меню;

- кнопка "Розпочати": основна кнопка, натискання на яку ініціює перехід до безпосередньо ігрового процесу. Це є точкою входу в основний геймплей;

- кнопка "Вихід": натискання на цю кнопку призводить до закриття гри та повернення користувача до операційної системи.

Для початку гри користувачеві необхідно натиснути на кнопку "Розпочати". Після цього відбудеться завантаження ігрової сцени, і користувач зможе керувати персонажем та взаємодіяти з ігровим світом.

Якщо ж користувач бажає закрити гру, йому слід обрати кнопку "Вихід".

Після вибору опції "Розпочати" в головному меню, користувач переноситься безпосередньо на ігрову арену, де починається основний ігровий процес (рисунок 4.2).

Рисунок 4.2 – Ігровий процес

Керування персонажем здійснюється за допомогою клавіатури. Користувач може вільно переміщувати свого героя по ігровій карті, використовуючи стандартні клавіші руху "WASD". Основна задача гравця – виживати якомога довше, маневруючи між численними ворогами, що з'являються на арені, та уникаючи їхніх атак.

Важливою особливістю гри є автоматична система атаки. Персонаж гравця самостійно атакує найближчих ворогів, і напрямок атаки не залежить від напрямку руху гравця. Це дозволяє користувачеві зосередитися на ухиленні та позиціонуванні, в той час як його герой автоматично веде вогонь по супротивниках.

На екрані присутні декілька ключових елементів інтерфейсу, що надають користувачеві важливу інформацію:

- таймер: розташований у верхній частині екрана, він відраховує час,

що минув з початку поточної ігрової сесії. Це дозволяє гравцеві відстежувати тривалість свого виживання;

- індикатор здоров'я персонажа: невелика червона смужка, розташована безпосередньо під спрайтом гравця, візуально відображає поточний рівень його здоров'я. Зменшення цієї смужки свідчить про отримання пошкоджень;

- шкала досвіду: знаходиться в нижній частині екрана і показує прогрес гравця в наборі досвіду. Прогрес заповнення можна зрозуміти по золотистому кольору шкали. Заповнення цієї шкали призводить до підвищення рівня персонажа та можливості отримати нові покращення;

- відображення шкоди: при успішному нанесенні шкоди ворогу, над ним з'являється числове значення, що показує кількість завданих пошкоджень. Це дозволяє гравцеві оцінювати ефективність своєї зброї та атак.

Важливою частиною ігрового процесу є система прогресії персонажа, яка базується на зборі досвіду та виборі покращень. З деяким шансом зі знищеного ворога на ігровій арені випадає спеціальний предмет – пляшечка з досвідом (рисунок 4.3).

Рисунок 4.3 – Пляшечка з досвідом

Для того, щоб підібрати цю пляшечку, користувачеві необхідно наблизити свого персонажа до неї. Після підбирання, шкала досвіду, розташована внизу екрана, частково заповнюється. Коли шкала досвіду заповнюється повністю, персонаж гравця отримує новий рівень.

Отримання нового рівня супроводжується тимчасовою зупинкою ігрового процесу. На екрані з'являється спеціальне меню вибору покращень (рисунок 4.4), яке надає користувачеві можливість посилити свого персонажа.

Рисунок 4.4 – Екран вибору покращень

У цьому меню гравцеві пропонується на вибір кілька варіантів покращень. Це можуть бути:

- покращення характеристик існуючої зброї: наприклад, підвищення шкоди основної атаки ("Sword Damage" – підвищує шкоду меча на 10%) або швидкості атаки ("Sword Speed" – підвищує швидкість меча на 10%);
- отримання нового виду зброї або здібності: наприклад, можливість використовувати вогняні кулі, так звані фаєрболи ("Fireball" – наносить шкоду ворогам навколо гравця).

Користувачеві необхідно уважно ознайомитися з описом кожного

запропонованого покращення та обрати те, яке найкраще відповідає його поточному стилю гри або стратегічним цілям. Вибір здійснюється кліком лівої кнопки миші по відповідній картці покращення. Після того, як гравець зробить свій вибір, меню покращень зникає, ігровий процес відновлюється, а обране покращення негайно застосовується до персонажа. Цей цикл збору досвіду та вибору покращень є ключовим для посилення гравця та його здатності протистояти все сильнішим хвилям ворогів.

Невід'ємною частиною ігрового процесу є можливість поразки. Якщо персонаж гравця отримує критичну кількість пошкоджень і його здоров'я опускається до нуля, він гине. Ця подія супроводжується появою на екрані спеціального меню "Game Over" (рисунок 4.5), що сигналізує про завершення поточної ігрової сесії.

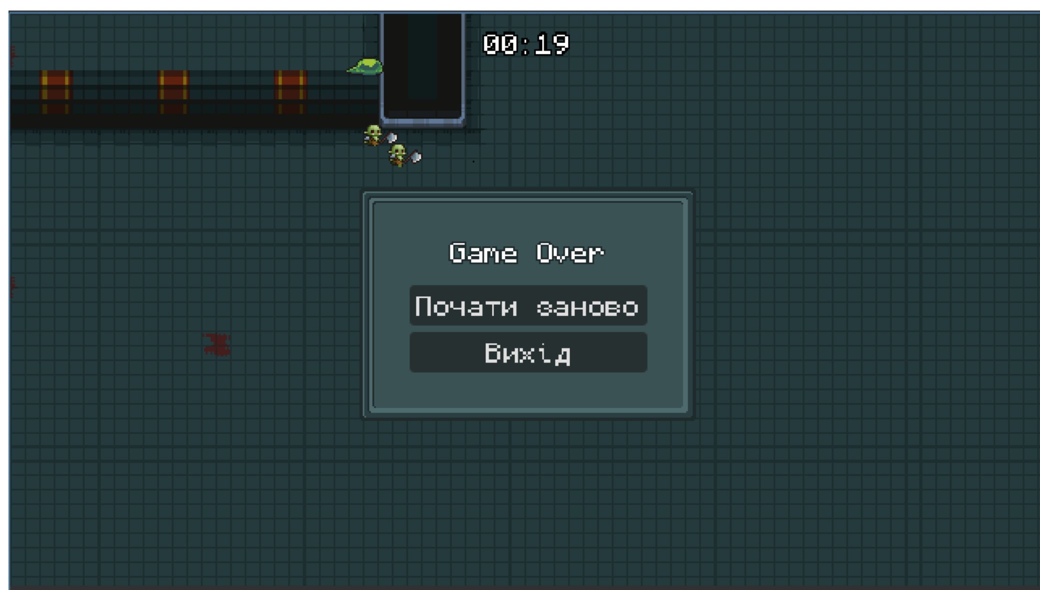


Рисунок 4.5 – Екран поразки

У цей момент ігровий процес повністю зупиняється. Таймер, розташований у верхній частині екрана, також припиняє свій відлік, фіксуючи загальний час, протягом якого гравець зміг протриматися на арені. Це значення слугує своєрідним показником успішності поточної спроби.

Екран "Game Over" пропонує користувачеві два варіанти подальших дій:

- кнопка "Почати заново": натискання на цю кнопку дозволяє гравцеві негайно розпочати нову ігрову сесію. При цьому весь прогрес поточної спроби, включаючи здобуті покращення та здібності, обнуляється. Гравець починає гру з початковими характеристиками та зброєю, що відповідає принципам жанру roguelike;
- кнопка "Вихід": ця опція дозволяє користувачеві вийти з гри та повернутися до головного меню або повністю закрити ігровий застосунок, залежно від реалізації.

Вибір однієї з цих опцій здійснюється стандартним кліком лівої кнопки миші. Таким чином, навіть у випадку поразки, гра надає можливість негайно спробувати ще раз, застосовуючи отриманий досвід для досягнення кращих результатів у наступних спробах, або ж завершити ігрову сесію.

ВИСНОВКИ

У ході виконання даної кваліфікаційної роботи було успішно розв'язано всі поставлені задачі, що дозволило створити функціональний прототип гри в жанрі roguelike.

На початковому етапі було проведено детальний аналіз предметної області, включаючи вивчення історії розвитку ігрових застосунків, особливостей жанрів top-down та roguelike, а також аналіз популярних ігор-представників, що дозволило визначити ключові механіки, успішні дизайнерські рішення та потенційні напрямки для розробки власного проєкту. Також було проаналізовано основні технології, обрані для реалізації: ігровий рушій Godot Engine, мову програмування GDScript, графічний редактор Aseprite та ключові патерни проєктування, що сприяли створенню гнучкої та масштабованої архітектури.

На основі проведеного аналізу було розроблено концепцію гри, включаючи основні ігрові механіки, систему прогресії персонажа, взаємодію з ворогами та інтерфейс користувача. Було створено дизайн ключових ігрових сцен та розроблено скрипти, що реалізують логіку ключових аспектів гри, створено необхідні анімації для персонажів та ігрових ефектів.

Проведене тестування розробленого програмного продукту підтвердило стабільну роботу основних функціональних блоків та відповідність поведінки гри поставленим вимогам. Критичних помилок, що перешкоджали б основному ігровому процесу, виявлено не було.

Попри успішну реалізацію запланованого функціоналу, слід зазначити потенційні напрямки для вдосконалення. Зокрема, деякі анімації потребують подальшої оптимізації для забезпечення більш стабільного візуального відображення на різних конфігураціях обладнання. Також, для підвищення реіграбельності та візуального розмаїття, доцільно розглянути впровадження процедурної генерації ігрових рівнів, що значно збагатить ігровий досвід та загальне враження від гри.

Подальший розвиток проєкту включає такі напрямки:

- розширення контенту: додавання нових ігрових персонажів з унікальними стартовими здібностями, нових типів ворогів з різноманітними моделями поведінки та атаками, а також значне розширення переліку доступних здібностей та покращень;

- покращення ігрового світу: реалізація процедурної генерації рівнів для забезпечення унікальності кожної ігрової сесії;

- технічна оптимізація: подальша оптимізація продуктивності гри, особливо для мобільних платформ або менш потужних пристроїв;

- покращення досвіду користувача: впровадження більш розгорнутої системи навчання гравця, можливо, через інтерактивні підказки або окремий навчальний рівень;

- додавання мета-прогресії: розробка системи постійних покращень, які гравець може розблоковувати між ігровими сесіями.

Таким чином, виконана робота є міцним фундаментом для подальшого розвитку та створення повноцінного ігрового продукту. Досягнуті результати демонструють ефективність обраних технологій та підходів до розробки.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Kent S. L. The Ultimate History of Video Games: From Pong to Pokémon and Beyond – The Story Behind the Craze That Touched Our Lives and Changed the World. Three Rivers Press, 2001. 624 p.

2. Hill-Whittall R. The Indie Game Developer Handbook. 2015. 278 p.

3. Shaker N., Togelius J., Nelson M. J. Procedural Content Generation in Games. Springer, 2016. 235 p.

4. Schell J. The Art of Game Design: A Book of Lenses. CRC Press, 2019. 648 p.

5. Bradfield C. Godot 4 Game Development Projects: Build five cross-platform 2D and 3D games using one of the most powerful open source game engines. 2nd ed. 2023. 264 p.

6. Godot Engine Documentation. URL: https://docs.godotengine.org.

7. Bradfield C. Godot Engine Game Development Projects. 2018. 298 p.

8. McGuire M. Godot 4 GDScript 2.0 Programming. 2023. 342 p.

9. Hervieux M. Learning Pixel Art. 2015. 208 p.

10. Humphries M., Dawe J. Make Your Own Pixel Art: Create Graphics for Games, Animations, and More!. No Starch Press, 2019. 200 p.

11. Aseprite Documentation. URL: https://www.aseprite.org/docs.

12. Cox C. Eradicating Singleton Anti-Patterns: Ensuring Efficient and Scalable Code. Independently published, 2023. 47 p.

13. Trkulja M. Mastering Godot: Complete instructions for making video games. 2021. 224 p.

14. Nystrom R. Game Programming Patterns. Apress, 2011. 300 p.