Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет центр післядипломної освіти
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

Ігровий програмний застосунок в жанрі “Platformer” на рушії Unity

(тема)

Виконав:
студент 4 курсу, групи ПЗПІп-22-1

Буренко О.П.
(прізвище, ініціали)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)
Тип програми освітньо-професійна
Освітня програма Програмна інженерія
(повна назва освітньої програми)

Керівник доц. Кириченко І.В.
(посада, прізвище, ініціали)

Допускається до захисту
Зав. кафедри ____________ З.В. Дудар
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Факультет центр післядипломної освіти

Кафедра програмної інженерії

Рівень вищої освіти перший (бакалаврський)

Спеціальність 121 – Інженерія програмного забезпечення

Тип програми Освітньо-професійна

Освітня програма Програмна Інженерія

(шифр і назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _________
(підпис)
«____» __________ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Буренку Олексію Павловичу

(прізвище, ім’я, по батькові)

1. Тема роботи Ігровий програмний застосунок в жанрі “Platformer” на рушії Unity

Затверджена наказом по університету від 17.06. 2024р. № 588 Ст.

2. Термін подання студентом роботи до екзаменаційної комісії 22.07.2024

3. Вихідні дані до роботи Розробити ігровий застосунок в жанрі “Platformer”, Мова програмування C# використовуючи ігровий двигун Unity 2D.

4. Перелік питань, що потрібно опрацювати в роботі

Вступ, аналіз предметної галузі, формування вимог до програмної системи, архітектура та проектування програмного забезпечення, опис прийнятих програмних рішень, тестування розробленого програмного забезпечення, висновки, додатки.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз предметної галузі	20.05.2024	*виконано*
2	Створення специфікації ПЗ	22.05.2024	*виконано*
3	Проектування ПЗ	24.05.2024	*виконано*
4	Розробка ПЗ	28.05.2024	*виконано*
5	Тестування ПЗ	30.05.2024	*виконано*
6	Оформлення пояснювальної записки	14.07.2024	*виконано*
7	Підготовка презентації та доповіді	15.06.2024	*виконано*
8	Попередній захист	16.07.2024	*виконано*
9	Нормоконтроль, рецензування	16.07.2024	*виконано*
10	Здача роботи у електронний архів	16.07.2024	*виконано*
11	Допуск до захисту у зав. кафедри	19.07.2024	*виконано*

Дата видачі завдання 6 травня 2024р.

Студент ______________________ Буренко О.П.
(підпис)

Керівник роботи ________________ доц. Кириченко І.В.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка до передатестаційної практики бакалавра, 60 с., 15 рис., 10 джерел.

ІГРОВИЙ ПРОГРАМНИЙ ЗАСТОСУНОК, PLATFORMER C#, UNITY3D

Об’єкт розробки – ігровий програмний застосунок в жанрі Platformer.

Мета розробки – створення ігрового програмного застосунку 2D Platformer.

Метод рішення – середовище розробки Unity2D, мова програмування C#.

У результаті розробки створено ігровий програмний застосунок, в якому ми рухаємося по рівню знищуємо ворогів та пригаємо по платформах.

GAME SOFTWARE APPLICATION, PLATFORMER C#, UNITY2D

The object of development is a game software application in the Platformer genre.

The purpose of the development is to create a 2D Platformer gaming software application.

The solution method is the Unity2D development environment, the C# programming language.

As a result of the development, a gaming software application was created, in which we move through the level, destroy enemies and jump on platforms.

Я, Буренко Олексій Павлович, студент гр. ПЗПІп-22-1, здобувач вищої освіти на першому (бакалаврському) рівні кафедри «Програмна інженерія», заявляю: моя кваліфікаційна робота на тему «Ігровий програмний модуль інвентаря для комп’ютерної гри у жанрі RPG», що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений із діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

ЗМІСТ

Вступ 7
1 Аналіз предметної галузі 8
1.1 Аналіз предметної галузі 8
1.2 Виявлення та вирішення проблем 10
1.3 Постановка задачі 12
1.3.1 Цільова аудиторія 12
1.3.2 Монетизація 12
2 Формування вимог до програмної системи 14
3 Архітектура та проєктування програмного забезпечення 16
3.1 UML проєктування ПЗ 16
3.2 Проєктування архітектури ПЗ 28
3.3 Проєктування структури зберігання даних 29
3.4 Приклади найцікавіших алгоритмів та методів 29
3.5 Створення UI/UX 33
4 Опис прийнятих програмних рішень 37
4.1 Вибір інструментів програмної реалізації 37
4.2 Архітектура проєкту 37
4.3 Цікаві програмні рішення 38
5 Тестування програмного забезпечення 43
5.1 Тестування ігрового застосунку 43
5.2 Приклади критичних помилок 43
Висновок 48
Перелік джерел посилання 49
Додаток А 50
Додаток Б 51
Додаток В 56

ПЕРЕЛІК СКОРОЧЕНЬ

ASCII – American Standard Code for Information Interchange

FIX – Financial Information Exchange

FAST – FIX Adapted for Streaming

FTP – File Transfer Protocol

SBE – Simple Binary Encoding

SOH – Simple Open Header

HTTP – Hypertext Transfer Protocol

MVD – Model-View-Delegate

SDK – Serial Development Kit

TCP – Transmission Control Protocol

WPF – Windows Presentation Foundation

XML – Extensible Markup Language

NPC – Non-Playable Character

ВСТУП

Темою кваліфікаційної роботи є ігровий програмний застосунок в жанрі ”Platformer”. Основне завдання ігрового програмного застосунку в жанрі “Platformer” полягає в тому, щоб гравець керував персонажем, який пересувається по рівнях, стрибаючи з платформи на платформу, уникаючи перешкод і збираючи різноманітні предмети або бонуси.

Гра включає в себе складні рівні з різними перешкодами, ворогами та головоломками, які гравець повинен подолати, використовуючи стрибки, навички керування та іноді спеціальні предмети або здібності персонажа. Суть гри полягає в успішному завершенні кожного рівня, досягненні кінцевої мети або зібранні всіх необхідних предметів.

Метою роботи є розробка програмного застосунку в жанрі платформера, створення цікавої та захоплюючої гри, яка буде привертати увагу гравців.

Для розробки продукту використовувався ігровий рушій Unity2D та мови програмування C#. Відповідно до технологій використовуються середовища розробки Unity2D, VS Code.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Аналіз предметної галузі

Платформери є одним з найпопулярніших і найстаріших жанрів відеоігор, який зародився наприкінці 1970-х років. Першими платформерами вважаються Frogs (1978), де були стрибки, але не було платформ, і Space Panic (1980), де були платформи, але не було стрибків. Революційною грою став Donkey Kong (1981), де вперше з'явився скролінг. Першим справжнім платформером вважається гра Pitfall! (1982) від Activision, але справжня слава прийшла до жанру після виходу Super Mario Bros. (1985) від Nintendo. [1] (див. рис. 1.1).

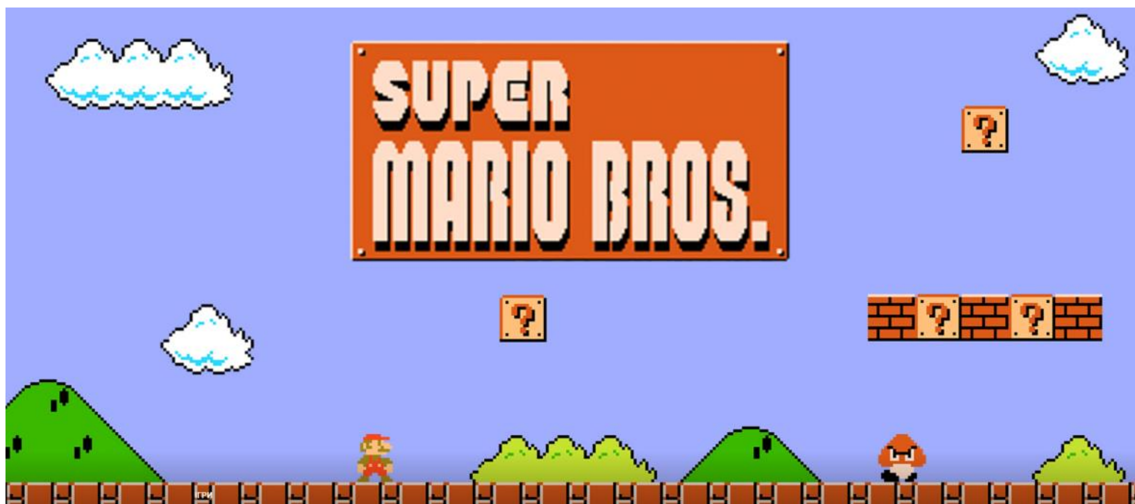


Рисунок 1.1 – Super Mario Bros. (1985) від Nintendo. (за даними [1])

Frogs та Space Panic були важливими кроками у розвитку жанру. У Frogs гравці контролювали жабу, яка стрибала, щоб ловити комах, але відсутність платформ обмежувала ігрову механіку. Space Panic ввела концепцію платформ, де гравці пересувалися по статичних рівнях і уникали ворогів, але без можливості стрибати.

Donkey Kong став справжнім проривом завдяки введенню скролінгу, що дозволило створювати більш динамічні та складні рівні. Гра вперше представила персонажа Маріо, який став іконою в світі відеоігор. Гравці керували Маріо, який пересувався вертикально і горизонтально по рівнях, уникаючи перешкод і рятуючи принцесу.

Наступним важливим кроком у розвитку жанру стала гра Pitfall! від Activision. У цій грі гравці керували персонажем Гаррі, який досліджував джунглі, стрибаючи через ями і уникаючи небезпек. Гра була однією з перших, що використовувала багатоекранні ігрові світи з горизонтальним проходженням, що стало стандартом для майбутніх платформерів.

Справжній бум популярності платформерів відбувся з виходом Super Mario Bros. від Nintendo. Гра запропонувала гравцям великі, складні рівні з різноманітними ворогами, пастками і пауер-апами. Вона встановила стандарти для жанру, включаючи скролінг рівнів, плавне управління персонажем і інтерактивне середовище. Super Mario Bros. стала однією з найвпливовіших і найуспішніших ігор в історії, створивши цілу франшизу і перетворивши Маріо на символ компанії Nintendo.

У 1980-х і 1990-х роках з'явилися інші значущі платформери, такі як Manic Miner (1983) і Jet Set Willy (1984), які стали популярними серед власників домашніх комп'ютерів. Вони продовжили розвиток жанру, вводячи нові механіки та складні рівні.

У 1991 році компанія Sega випустила Sonic the Hedgehog, яка стала одним з найуспішніших платформерів і створила конкуренцію для Super Mario Bros. Гра використовувала можливості 16-бітної консолі Sega Genesis, пропонуючи швидкий геймплей і яскраву графіку, що стало її відмінною рисою.

Платформери продовжують розвиватися і до сьогодні, зберігаючи популярність серед гравців. Сучасні ігри в цьому жанрі поєднують класичні елементи з новими технологіями, такими як тривимірна графіка і інтерактивні середовища. Жанр постійно еволюціонує, пропонуючи нові виклики і враження для гравців.

1.2 Виявлення та вирішення проблем

У процесі розробки платформерів розробники стикаються з різноманітними проблемами, які впливають на якість ігрового процесу, технічну стабільність та загальне враження від гри. Ці проблеми охоплюють технічні, дизайнерські та користувацькі аспекти. Вирішення цих проблем є критично важливим для створення успішного ігрового продукту. Проблеми можуть бути технічними, проблеми геймдизайну та проблеми користувацького досвіду.

Технічні проблеми включають в себе оптимізацію продуктивності, сумісність з різними пристроями, проблеми з геймдизайном та проблеми з користувацьким досвідом.

Платформери часто мають великий ігровий світ, який потребує ефективного управління ресурсами. Це включає оптимізацію графіки, анімацій та фізичних моделей, щоб гра могла працювати плавно навіть на старіших та слабких компютерах. Для вирішення цієї проблеми необхідно проводити тестування на різних конфігураціях апаратного забезпечення і застосовувати методи оптимізації, такі як рівні деталізації (LOD), динамічне завантаження ресурсів та використання ефективних алгоритмів рендерингу.

Різноманітність апаратних конфігурацій ПК створює виклики для забезпечення стабільної роботи гри на всіх можливих системах. Це вимагає широкого тестування і створення версій гри, оптимізованих для різних типів обладнання. Важливо забезпечити підтримку різних роздільних здатностей екранів та форматів співвідношення сторін.

Один з ключових елементів успішного платформера - різноманітність рівнів і викликів, які постають перед гравцем. Створення цікавих і різнопланових рівнів вимагає творчого підходу і значних зусиль від команди розробників. Важливо забезпечити баланс між складністю і цікавістю гри, щоб гравці не втрачали інтересу.[2]

Гравці очікують від платформерів точного і чуйного управління. Навіть незначні затримки або неточності можуть зіпсувати враження від гри. Для вирішення цієї проблеми розробники повинні ретельно налаштовувати фізику руху

персонажів та механіку стрибків, а також проводити численні тестування для досягнення оптимальної чутливості управління.

Хоча платформери зазвичай фокусуються на геймплеї, сюжет і атмосфера також грають важливу роль. Важливо створити захоплюючий сюжет і цікавих персонажів, які залучатимуть гравців і спонукатимуть їх проходити гру до кінця. Крім того, атмосфера гри повинна бути підтримана якісною графікою та звуковим супроводом.

Нові гравці можуть мати труднощі з освоєнням механік гри, тому важливо забезпечити ефективне навчання. Це може бути досягнуто через інтерактивні підказки, навчальні рівні або інші способи поступового введення нових елементів геймплею.

Балансування рівня складності є критично важливим для утримання інтересу гравців. Гра повинна пропонувати достатньо викликів, щоб бути цікавою, але не настільки складною, щоб викликати розчарування. Розробники можуть впроваджувати різні режими складності або адаптивні системи, що підлаштовуються під навички гравця.

Зручна система збереження прогресу є важливим елементом, який дозволяє гравцям продовжувати гру з місця останнього збереження. Це особливо актуально для складних платформерів з великими рівнями. Можливість автоматичного збереження або ручного збереження повинна бути добре реалізована і зрозуміла для користувачів.

Загалом, успішний розвиток платформера вимагає комплексного підходу до вирішення технічних, дизайнерських та користувацьких проблем. Вирішення цих проблем допоможе створити якісний продукт, який буде цікавий і доступний широкій аудиторії гравців.

1.3 Постановка задачі

Слід створити унікальну концепцію гри, яка включатиме основний сюжет, механіки геймплею та візуальний стиль. Проектування рівнів має забезпечити різноманітність ігрових викликів, що включають перешкоди, ворогів та головоломки. Розробка ігрових механік передбачає реалізацію рухів персонажа, а також взаємодії з об'єктами та ворогами в грі. Не менш важливо забезпечити технічну реалізацію, написавши оптимізований код на C# та використовуючи Unity2D для створення графіки, фізики та анімацій.

1.3.1 Цільова аудиторія

Гра повинна бути доступною та цікавою для різних вікових груп. Платформери зазвичай приваблюють широку аудиторію, від дітей до дорослих. Важливо створити контент, який буде зрозумілий і прийнятний для кожної з цих груп. Цільова аудиторія може включати як новачків, так і досвідчених гравців. Необхідно збалансувати рівень складності таким чином, щоб новачки могли легко освоїти основи гри, а досвідчені гравці мали достатньо викликів.

Гра орієнтовна на аудиторію 16-45 років чоловіки та жінки. Також ця гра сподобається тим хто полюбляє ігри про вампірів.

1.3.2 Монетизація

Монетизація гри є важливим аспектом, який впливає на фінансову успішність проекту. Існує кілька стратегій монетизації, які можна застосувати.

Традиційний спосіб монетизації, коли гравці купують гру за одноразову плату. Цей підхід дозволяє отримати дохід на етапі випуску гри. Важливо встановити конкурентну ціну, яка відповідає якості та тривалості гри.

Впровадження мікротранзакцій дозволяє гравцям купувати додатковий контент, такий як скіни для персонажів, додаткові рівні. Важливо збалансувати мікротранзакції, щоб вони не впливали негативно на ігровий процес та не створювали відчуття "плати за перемогу".

Випуск додаткового контенту (DLC) після релізу основної гри може стимулювати гравців повертатися до гри та витрачати більше грошей. Це можуть бути нові рівні, персонажі, режими гри тощо.

Вбудована реклама може бути додатковим джерелом доходу, особливо якщо гра безкоштовна. Важливо, щоб реклама не заважала ігровому процесу і не відштовхувала гравців.

Впровадження системи підписок може запропонувати гравцям регулярний доступ до нового контенту або ексклюзивних можливостей. Це забезпечує стабільний потік доходу протягом тривалого часу.

Розробка стратегії монетизації повинна враховувати потреби та очікування цільової аудиторії, а також особливості гри, щоб забезпечити оптимальний баланс між отриманням доходу і задоволенням гравців.[3]

2 ФОРМУВАННЯ ВИМОГ ДО ПРОГРАМНОЇ СИСТЕМИ

Мета розробки гри полягає в створенні захоплюючого та цікавого ігрового досвіду для користувачів. Гра повинна мати якісну графіку, плавний геймплей, інтуїтивно зрозумілий інтерфейс та достатню кількість рівнів і викликів, щоб утримувати інтерес гравців і бажання повертатися в гру.

Гра повинна включати в себе головоломки, секретні рівні, колекційні предмети та цікаві та складні досягнення. Також потрібно пропрацювати лор гри. З цікавою історією та складними завданнями грати цікавіше. Забезпечити гравцям задоволення та виклики через різні рівні, унікальних ворогів та складні перешкоди.

Створити гру з високоякісною графікою, звуком та геймплеєм.

Забезпечити інтерактивні елементи, такі як збирання предметів, взаємодія з ворогами, NPS та навколишнім середовищем.

Гра повинна бути доступна на різних платформах, включаючи ПК, консолі та мобільні пристрої.

Загальний опис. Гравець керує персонажем, що пересувається по рівнях, змагається з ворогами спілкується з NPS та збирає предмети. Кожен рівень має свій набір перешкод, ворогів та завдань, які гравець повинен виконати, щоб перейти на наступний рівень.

Використання високоякісної піксельної графіки, яка відповідає стилю гри та створює атмосферу ретро-ігор. Супровід ігрового процесу звуковими ефектами та музикою, що створюють приємну атмосферу гри.

Загальні обмеження:

– гра повинна працювати плавно на різних пристроях, включаючи старі моделі ПК та мобільних пристроїв;
– розмір гри не повинен бути занадто великим, щоб її можна було легко завантажити та встановити;
– гра повинна бути сумісною з основними операційними системами, такими як Windows, macOS, iOS, Android;
– проект повинен бути завершений в межах встановленого терміну;

- обмеження на бюджет, що включає витрати на розробку, тестування, маркетинг та підтримку гри;
- гра повинна бути відповідною для цільової аудиторії;
- гра повинна відповідати правовим вимогам, таким як авторське право на використовувані ресурси (музику, графіку).

Ці характеристики допоможуть структуровано підходити до розробки гри, забезпечуючи її якість, відповідність очікуванням гравців та своєчасне завершення проекту.

3 АРХІТЕКТУРА ТА ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 UML проєктування ПЗ

Розробка ігрового застосунку в жанрі “Platformer” починається з UML проектування, що включає кілька ключових етапів для визначення основної структури та функціональності гри.[4] В процесі проектування розробки були визначені вигляд головного меню гри. Після аналізу було створено Use-case діаграму головного меню. (див. рис. 3.1.1).

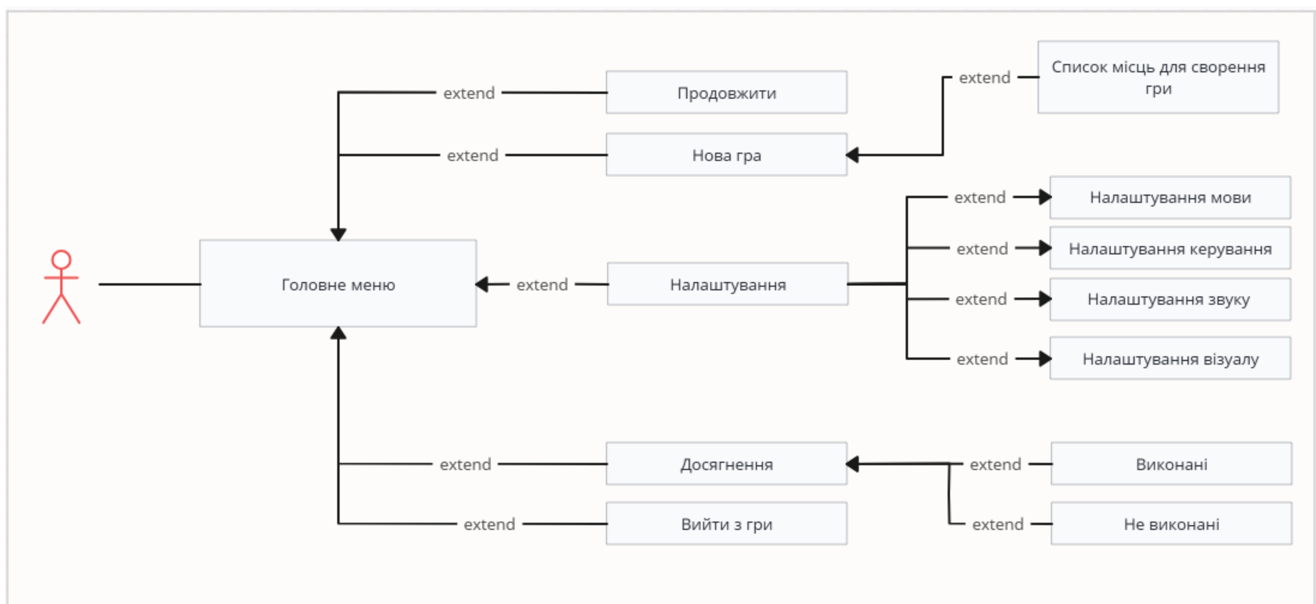


Рисунок 3.1.1 – Use-case діаграма головного меню ігрового застосунку (рисунок виконаний самостійно)

Діаграма визначення випадків використання: Опис основних дій, які можуть виконувати користувачі. Наприклад, "Гравець рухається", "Гравець стрибає", "Гравець збирає предмети", "Гравець взаємодіє з ворогами".

Опис основних дій (випадків використання), які можуть виконувати користувачі.

Гравець рухається (Player Moves). Гравець використовує клавіші керування для переміщення персонажа вліво або вправо по екрану.

Основний потік подій:

– гравець натискає клавішу вліво або вправо;

– система визначає напрямок руху;
– персонаж гравця переміщується у вказаному напрямку;
– система перевіряє колізії з об'єктами та оновлює позицію персонажа на екрані.

Гравець стрибає (Player Jumps). Гравець натискає клавішу стрибка для підняття персонажа вгору.

Основний потік подій:

– гравець натискає клавішу стрибка;
– система активує анімацію стрибка;
– персонаж гравця піднімається вгору;
– система перевіряє колізії з платформами та іншими об'єктами;
– персонаж гравця приземляється на платформу або інший об'єкт.

Гравець збирає предмети (Player Collects Items). Гравець переміщує персонажа до предметів, щоб їх зібрати.

Основний потік подій:

– гравець переміщує персонажа до предмета.
– система перевіряє колізію між персонажем та предметом.
– при зіткненні предмет додається до інвентарю гравця.
– система оновлює інтерфейс для відображення зібраних предметів.

Гравець взаємодіє з ворогами (Player Interacts with Enemies). Гравець вступає у взаємодію з ворогами, що може включати атаку або блокування.

Основний потік подій:

– гравець переміщується до ворога;
– система перевіряє колізію між персонажем гравця та ворогом;
– якщо гравець атакує, система зменшує здоров'я ворога;
– якщо ворог атакує, система зменшує здоров'я персонажа гравця;
– персонаж гравця може заблокувати атаку або контратакувати. (див. рис. 3.1.2).

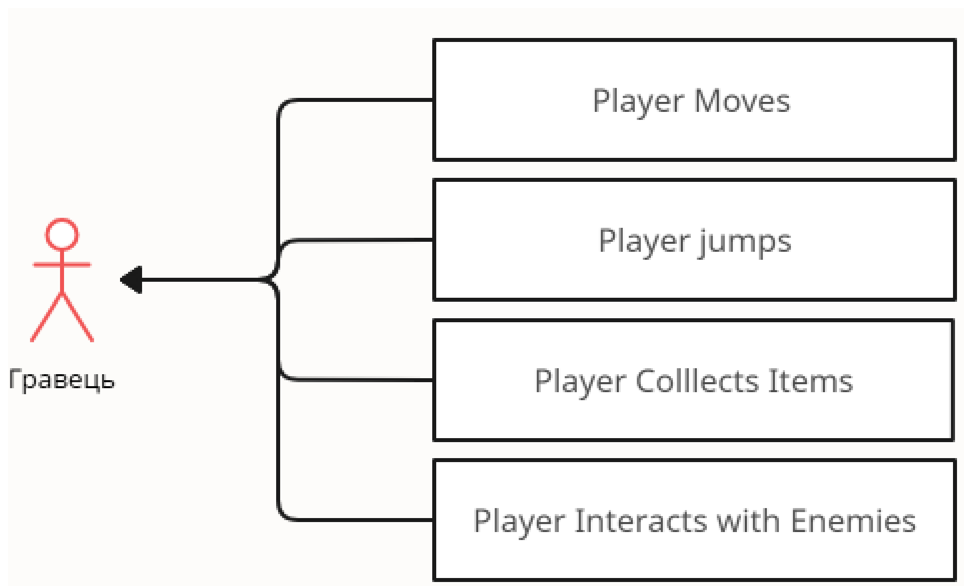


Рисунок 3.1.2 – Use-case діаграма основних дій гравця (рисунок виконаний самостійно)

Ця діаграма відображає основні випадки використання (Use-cases) для гри платформера, ілюструючи взаємодію гравця з системою. Кожен випадок використання показує певну дію, яку може виконувати гравець, допомагаючи візуалізувати та деталізувати функціональні вимоги до гри.

Для успішного проєктування гри важливо визначити основні об'єкти та їхні взаємодії в системі. Це допоможе структурувати програмне забезпечення та забезпечити його ефективне функціонування. Основні класи гри відображають ключові компоненти, з якими взаємодіє гравець.

Основні класи гри:

– персонаж гравця (PlayerCharacter);
– вороги (Enemy);
– платформи (Platform);
– предмети (Item);
– рівні (Level).

Визначення атрибутів та методів класів.

Клас “Персонаж гравця (PlayerCharacter)”

а) атрибути:

1) position: координати позиції персонажа на екрані;
2) velocity: швидкість руху персонажа;
3) health: рівень здоров'я персонажа.

б) методи:

1) move(direction): переміщення персонажа в заданому напрямку;
2) jump(): стрибок персонажа;
3) attack(): атака ворогів;
4) takeDamage(amount): зменшення здоров'я персонажа при отриманні ушкоджень.

Клас “Вороги (Enemy)”

а) атрибути:

1) position: координати позиції ворога на екрані;
2) velocity: швидкість руху ворога;
3) health: рівень здоров'я ворога;
4) damage: рівень ушкоджень, які завдає ворог.

б) методи:

1) move(direction): переміщення ворога в заданому напрямку;
2) attack(): атака персонажа гравця;
3) takeDamage(amount): зменшення здоров'я ворога при отриманні ушкоджень.

Клас “Платформи (Platform)”

а) атрибути:

1) position: координати позиції платформи на екрані;
2) size: розміри платформи.

б) методи:

1) isSolid(): визначення, чи є платформа твердої.

Клас "Предмети (Item)"

а) атрибути:

1) position: координати позиції предмета на екрані;

2) type: тип предмета (наприклад, збільшення здоров'я, зброя).

б) методи:

1) collect(): збирання предмета персонажем.

Клас "Рівні (Level)"

а) атрибути:

1) number: номер рівня;

2) layout: розташування елементів рівня (платформи, вороги, предмети).

б) методи:

1) load(): завантаження рівня;

2) complete(): завершення рівня.

Визначення зв'язків між класами. Наслідування (Inheritance). Клас "Базовий персонаж (Character)" може бути базовим класом для класів "Персонаж гравця (PlayerCharacter)" і "Вороги (Enemy)".

Асоціація (Association). Клас "Рівень (Level)" асоціюється з класами "Персонаж гравця (PlayerCharacter)", "Вороги (Enemy)", "Платформи (Platform)" і "Предмети (Item)". (див. рис. 3.1.3).

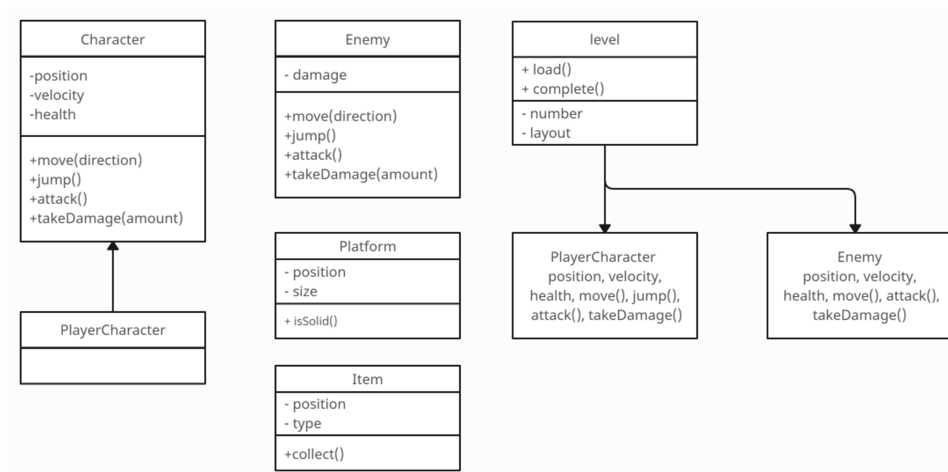


Рисунок 3.1.3 – Use-case діаграма ілюструє зв'язки між основними класами та показує наслідування, атрибути й методи для кожного класу

(рисунок виконаний самостійно)

Діаграми послідовності є важливими інструментами в UML проєктуванні, які допомагають зрозуміти, як відбуваються взаємодії між об'єктами в межах певних сценаріїв. Для розробки гри слід визначити ключові сценарії використання та показати послідовність дій, що виконуються для кожного з них.

Визначення сценаріїв використання:

а) початок рівня (Level Start):

1) гравець починає новий рівень. Система завантажує всі необхідні ресурси, ініціалізує об'єкти рівня та встановлює початкову позицію гравця;

б) взаємодія з ворогом (Interaction with Enemy):

1) гравець стикається з ворогом. Система визначає результат взаємодії, який може включати атаки, зменшення здоров'я, тощо;

в) завершення рівня (Level Completion):

1) гравець досягає кінцевої точки рівня. Система перевіряє умови завершення, зберігає прогрес та переходить до наступного рівня або відображає екран завершення гри.

Опис послідовності дій.

Початковий стан: гравець знаходиться на платформі. Гравець натискає клавішу стрибка, і система отримує сигнал про це натискання. Вона викликає метод jump() у об'єкта гравця, збільшуючи вертикальну швидкість персонажа для імітації стрибка.

Після цього система перевіряє наявність колізій з іншими об'єктами, такими як платформи та вороги, оновлює позицію персонажа на екрані та відображає анімацію стрибка. (див. рис. 3.1.4).

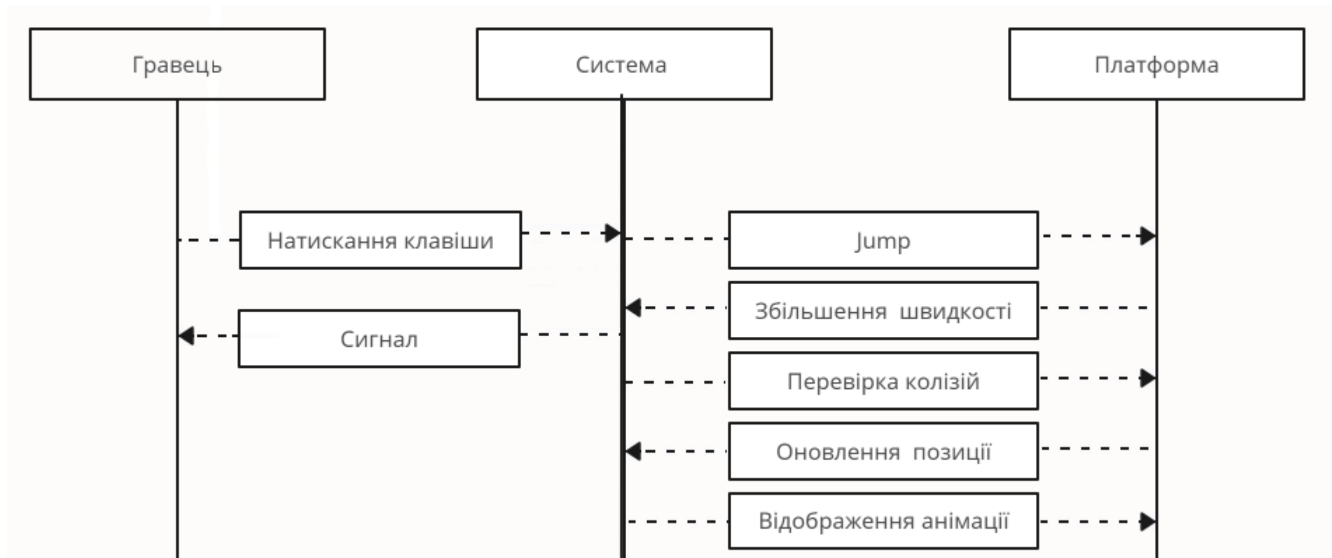


Рисунок 3.1.4 – Діаграма послідовності для стрибка персонажа
(рисунок виконаний самостійно)

Діаграма послідовності для початку рівня

Початковий стан: гравець вибирає рівень для початку гри. Гравець натискає кнопку "Почати рівень", і система отримує сигнал про початок рівня. Система завантажує необхідні ресурси для рівня, ініціалізує об'єкти рівня (персонажі, вороги, платформи, предмети), встановлює початкову позицію персонажа та відображає рівень на екрані. (див. рис. 3.1.5).

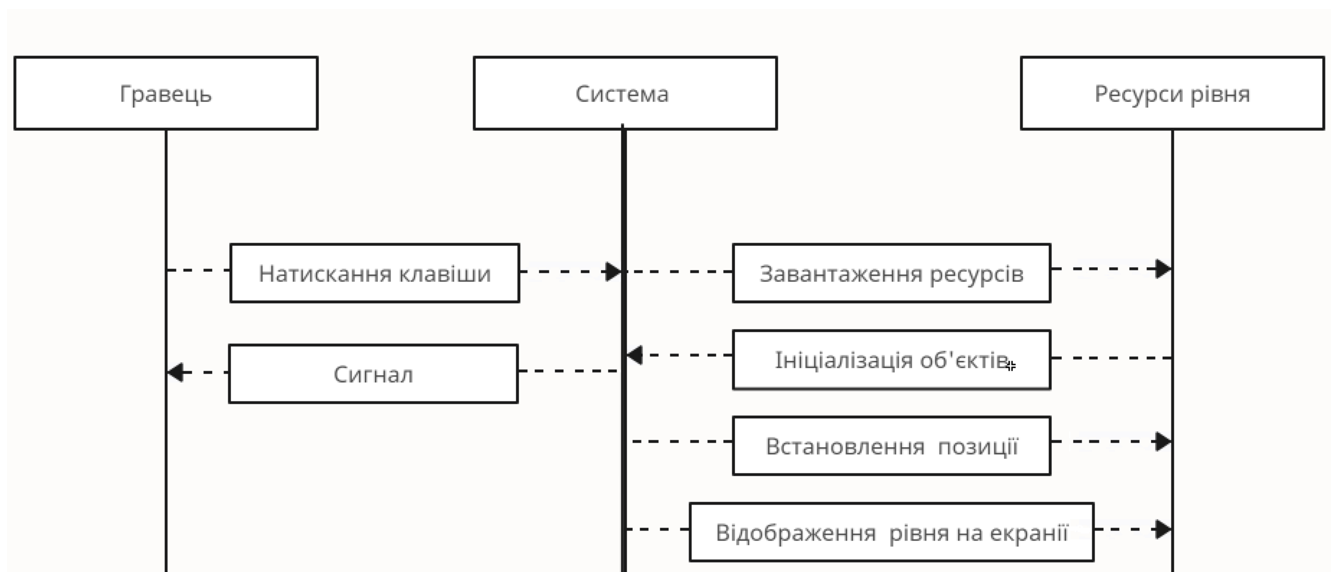


Рисунок 3.1.5 – Діаграма послідовності для початку рівня
(рисунок виконаний самостійно)

Приклад діаграми послідовності для завершення рівня.

Початковий стан: гравець досягає кінцевої точки рівня. Гравець взаємодіє з об'єктом завершення рівня, і система отримує сигнал про завершення рівня. Система перевіряє умови завершення рівня, зберігає прогрес гравця та відображає екран завершення рівня або переходу до наступного рівня. (див. рис. 3.1.6).

Рисунок 3.1.6 – Діаграма послідовності для завершення рівня
(рисунок виконаний самостійно)

Діаграми станів є важливими для моделювання поведінки об'єктів у грі, показуючи, як вони реагують на різні події та переходять між станами. Для створення діаграм станів необхідно визначити можливі стани об'єктів та описати переходи між ними.

Основні об'єкти гри та їхні стани:

а) персонаж гравця (PlayerCharacter):

1) платформи (Platform);
2) на землі (OnGround): Персонаж стоїть або рухається по платформі;
3) у стрибку (Jumping): Персонаж знаходиться в повітрі після стрибка;
4) у падінні (Falling): Персонаж падає вниз без підтримки під ногами;
5) пошкоджений (Damaged): Персонаж отримав удар від ворога або впав з висоти;

6) мертвий (Dead): Персонаж втратив всі очки здоров'я і гра закінчилася.

б) вороги (Enemy):

1) патрулювання (Patrolling): Ворог рухається за заданою траєкторією;
2) атакує (Attacking): Ворог атакує персонажа гравця;
3) пошкоджений (Damaged): Ворог отримав удар від гравця;
4) мертвий (Dead): Ворог знищений гравцем.

в) предмети (Item):

1) доступний (Available): Предмет може бути зібраний гравцем;
2) зібраний (Collected): Предмет зібраний гравцем і більше не доступний на рівні.

Опис переходів між станами.

а) персонаж гравця (PlayerCharacter):

1) на землі (OnGround): У цьому стані персонаж знаходиться на твердій поверхні. Подія, що переводить його до наступного стану, – це натискання кнопки стрибка, що ініціює перехід до стану "У стрибку (Jumping)";
2) у стрибку (Jumping): Персонаж піднімається в повітря після натискання кнопки стрибка. Коли він досягає піку стрибка, відбувається перехід до стану "У падінні (Falling)";
3) у падінні (Falling): Персонаж починає падати вниз після досягнення піку стрибка. Перехід до стану "На землі (OnGround)" відбувається, коли персонаж досягає поверхні платформи або землі. Якщо під час падіння персонаж отримує пошкодження, він переходить до стану "Пошкоджений (Damaged)";
4) пошкоджений (Damaged): У цьому стані персонаж зазнає пошкодження. Після завершення анімації пошкодження персонаж переходить до стану "На землі (OnGround)" або "У падінні (Falling)", залежно від поточної ситуації. Якщо персонаж втрачає всі очки здоров'я, відбувається перехід до стану "Мертвий (Dead)".

б) вороги (Enemy):

1) патрулювання (Patrolling): У цьому стані ворог переміщається по визначеному маршруту. Коли гравець входить у зону видимості ворога, відбувається перехід до стану "Атакує (Attacking)".
2) атакує (Attacking): Ворог переслідує або атакує гравця. Якщо ворог більше не бачить гравця, він повертається до стану "Патрулювання (Patrolling)". Якщо ворог отримує пошкодження, він переходить до стану "Пошкоджений (Damaged)".
3) пошкоджений (Damaged): Ворог зазнає пошкодження. Якщо ворог знищений гравцем, відбувається перехід до стану "Мертвий (Dead)".

в) предмети (Item):

1) доступний (Available): У цьому стані предмет доступний для збору. Коли гравець збирає предмет, відбувається перехід до стану "Зібраний (Collected)". (PlayerCharacter). (див. рис. 3.1.7).

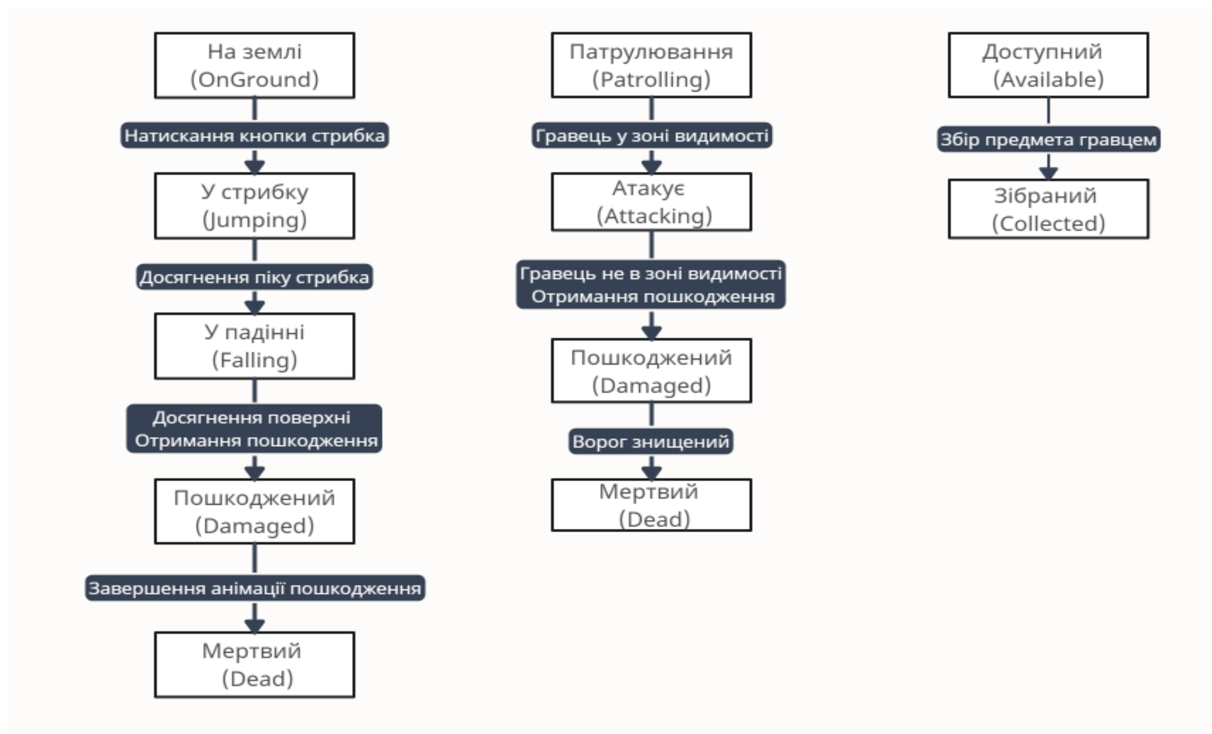


Рисунок 3.1.7 – Діаграма станів

(рисунок виконаний самостійно)

Створення діаграм активностей.

Діаграми активностей у UML допомагають візуалізувати потоки управління та даних у різних процесах гри. Вони показують, як різні дії пов'язані між собою і які умови або події спричиняють ці дії.

Процес завантаження рівня включає кілька ключових етапів. Спочатку ініціалізується завантажувач рівня, що підготовлює системи та об'єкти для завантаження. Далі відображається екран завантаження, який повідомляє гравцеві про поточний стан. Відбувається завантаження ресурсів рівня, включаючи текстури, моделі, звуки та специфічні дані рівня. Паралельно завантажуються асети та дані структури рівня.

Після цього ініціалізуються ігрові об'єкти, такі як гравець, вороги та платформи, і налаштовуються початкові параметри гравця та камери. Екран завантаження приховується, і запускається основний ігровий цикл, дозволяючи гравцеві почати взаємодію з новим рівнем.

Основні процеси гри можуть включати:

Процес завантаження рівня. (див. рис. 3.1.8).

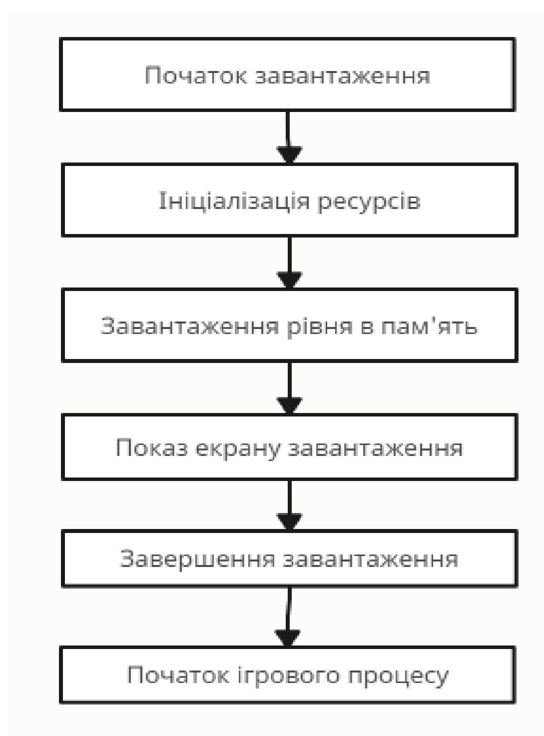


Рисунок 3.1.8 – Діаграма активностей початку завантаження рівня (рисунок виконаний самостійно)

Ігровий процес включає кілька ключових аспектів. Гравець керує персонажем, який пересувається по платформах, стрибає, уникає перешкод і збирає різні предмети. Система фізики Unity обробляє рухи персонажа та взаємодії з об'єктами на рівні. Камера слідує за персонажем, забезпечуючи оптимальний огляд ігрового простору.

Гра може включати ворогів та інші небезпеки, яких потрібно уникати або перемагати. Крім того, на рівнях можуть бути розміщені головоломки або завдання, які гравець має вирішити для просування вперед. Ігровий цикл оновлює стан гри, обробляє користувацький ввід та керує іншими аспектами ігрового процесу в реальному часі. (див. рис. 3.1.9).

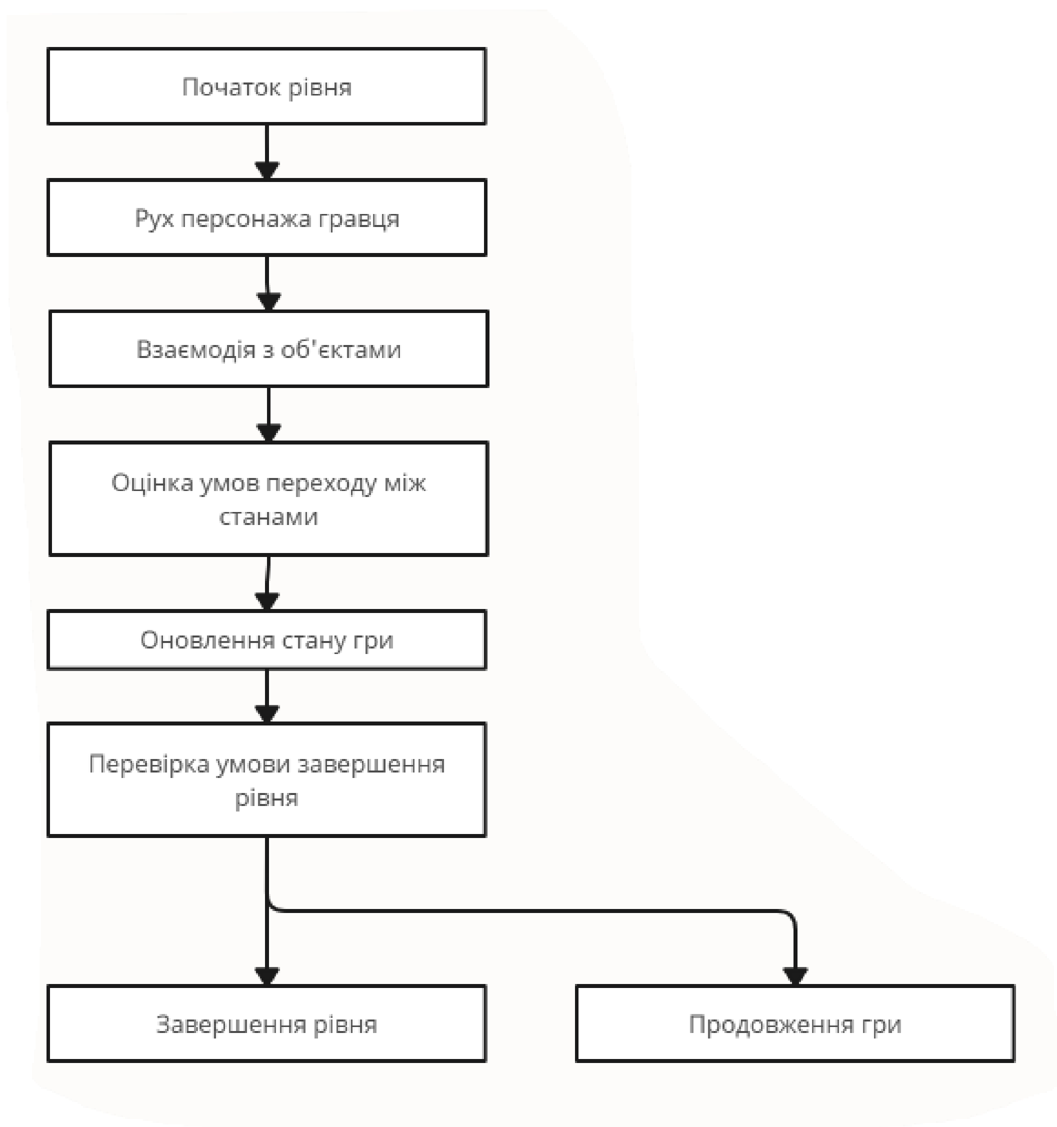


Рисунок 3.1.9 – Діаграма активностей ігровий процес
(рисунок виконаний самостійно)

Завершення рівня включає кілька ключових етапів. Спочатку гравець досягає кінцевої точки рівня, яка може бути відмічена спеціальним об'єктом або зоною. При досягненні цієї точки тригер викликає завершення рівня. Гра зберігає прогрес гравця, включаючи зібрані предмети, досягнуті цілі та інші важливі дані.

Відображається екран завершення рівня, який може показувати статистику рівня, таку як час проходження, кількість зібраних предметів та бали. Після цього гравець може перейти до наступного рівня, повторити поточний рівень або повернутися до головного меню. Всі ці дії відбуваються плавно, щоб забезпечити безперервність ігрового досвіду. (див. рис. 3.1.10).

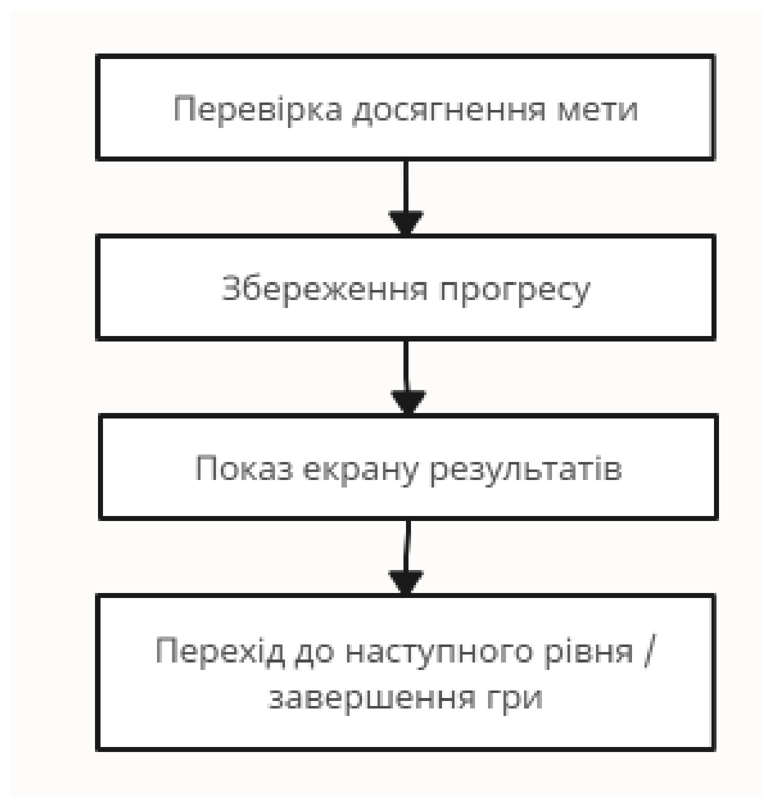


Рисунок 3.1.10 – Діаграма активностей завершення рівня
(рисунок виконаний самостійно)

3.2 Проєктування архітектури ПЗ

Проєктування архітектури програмного забезпечення (ПЗ) для гри є ключовим етапом, що визначає загальну структуру системи. Включає в себе вибір паттернів проєктування, таких як MVC (Model-View-Controller) або ECS (Entity-

Component-System), що найкраще відповідають потребам гри. Архітектура повинна уможливлювати ефективне керування ресурсами, включаючи обробку графіки, фізики та штучного інтелекту, забезпечувати легкість підтримки нових функцій і розширень.

Крім того, важливо розглядати питання безпеки даних і оптимізації продуктивності, зокрема під час обробки великої кількості об'єктів на екрані і управління станами гри. Важливо також розглядати питання підтримки кросплатформеності і можливості інтеграції з різними сервісами Unity для розширення функціоналу гри.

3.3 Проєктування структури зберігання даних

Розробка гри на Unity також вимагає осмисленого підходу до структури зберігання даних. Основними аспектами є вибір оптимального механізму зберігання і завантаження ресурсів, таких як графічні файли, звуки та інші медіа-елементи. Unity надає можливості для роботи з різними типами баз даних і системами файлового зберігання, що дозволяє розробникам адаптувати підходи в залежності від конкретних потреб проекту. Для оптимізації швидкодії гри важливо враховувати кешування і попереднє завантаження ресурсів, а також забезпечити механізми для збереження і обміну станом гри між різними сценами і рівнями. Також слід враховувати можливість інтеграції з різними сервісами хмарного зберігання для забезпечення резервного копіювання і синхронізації даних між різними пристроями користувачів.

3.4 Приклади найцікавіших алгоритмів та методів

У розробці гри можна застосовувати різноманітні алгоритми та методи, що покращують ігровий процес, оптимізують роботу гри та забезпечують захоплюючий ігровий досвід. Гра у жанрі платформер вимагає врахування безлічі аспектів, таких як фізика руху, поведінка ворогів, генерація рівнів, обробка анімацій, оптимізація продуктивності та створення інтерактивних середовищ.

Використання ефективних алгоритмів та методів дозволяє забезпечити плавність і реалістичність гри, роблячи її більш цікавою та захоплюючою для гравців. Нижче наведено деякі з найцікавіших алгоритмів та методів, які використані у процесі розробки гри.

Рух головного героя є одним з найважливіших елементів, який значно впливає на ігровий процес та загальний досвід гравця.

Клас HeroKnight відповідає за рух, атаки, стрибки та інші дії головного героя в грі. Нижче наведено короткий опис алгоритмів та методів, використаних для реалізації функціональності головного героя.

Алгоритм обробки вводу гравця отримує та обробляє натискання клавіш, визначаючи напрямок руху та різні дії (стрибок, атака, блокування, перекат). Він забезпечує плавність руху та своєчасне виконання команд.

```
float inputX = Input.GetAxis("Horizontal");

 // Swap direction of sprite depending on walk direction
 if (inputX > 0)
 {
     GetComponent<SpriteRenderer>().flipX = false;
     m_facingDirection = 1;
 }

 else if (inputX < 0)
 {
     GetComponent<SpriteRenderer>().flipX = true;
     m_facingDirection = -1;
 }

 // Move
 if (!m_rolling )
     m_body2d.velocity = new Vector2(inputX * m_speed, m_body2d.velocity.y);
}
```

Алгоритм стрибка реалізує можливість стрибка персонажа шляхом додавання вертикальної сили до Rigidbody2D, коли гравець натискає кнопку стрибка.

```
else if (Input.GetKeyDown("space") && m_grounded && !m_rolling)
{
    m_animator.SetTrigger("Jump");
    m_grounded = false;
    m_animator.SetBool("Grounded", m_grounded);
```

```
    m_body2d.velocity = new Vector2(m_body2d.velocity.x, m_jumpForce);
    m_groundSensor.Disable(0.2f);
}
```

Обробка станів персонажа включає визначення, чи знаходиться персонаж на землі, чи здійснює стрибок, чи виконує інші дії. Це досягається за допомогою сенсорів та булевих змінних.

```
if (!m_grounded && m_groundSensor.State())
 {
     m_grounded = true;
     m_animator.SetBool("Grounded", m_grounded);
 }

if (m_grounded && !m_groundSensor.State())
 {
     m_grounded = false;
     m_animator.SetBool("Grounded", m_grounded);
 }
```

Анімація руху реалізується за допомогою компоненту Animator, який перемикає анімації залежно від станів персонажа та вводу гравця.

```
else if(Input.GetMouseButtonDown(0) && m_timeSinceAttack > 0.25f &&
!m_rolling)
 {
     m_currentAttack++;

     if (m_currentAttack > 3)
         m_currentAttack = 1;

     if (m_timeSinceAttack > 1.0f)
         m_currentAttack = 1;

     m_animator.SetTrigger("Attack" + m_currentAttack);

     m_timeSinceAttack = 0.0f;
 }
```

Обробка взаємодії з анімаційними подіями. Метод AE_SlideDust відповідає за створення ефекту пилу при ковзанні. Він викликається в анімаціях ігрового персонажа.

```
void AE_SlideDust()
{
    Vector3 spawnPosition;

    if (m_facingDirection == 1)
        spawnPosition = m_wallSensorR2.transform.position;
    else
        spawnPosition = m_wallSensorL2.transform.position;

    if (m_slideDust != null)
    {
        // Set correct arrow spawn position
        GameObject dust = Instantiate(m_slideDust, spawnPosition,
gameObject.transform.localRotation) as GameObject;
        // Turn arrow in correct direction
        dust.transform.localScale = new Vector3(m_facingDirection, 1, 1);
    }
}
```

Алгоритм роботи порталу, портали у грі реалізуються за допомогою алгоритму, який визначає взаємодію персонажа з порталами, його переміщення та збереження кінетичної енергії.

Ініціалізація порталу.

У методі Start() змінна tpActive встановлюється у значення true, що дозволяє порталу бути активним одразу після початку гри.

```
private void Start()
 {
     tpActive = true;
 }
```

Обробка входу в портал.

Метод OnTriggerEnter2D(Collider2D other) викликається при вході колайдера іншого об'єкта в тригерний колайдер порталу. У цьому методі виконується перевірка на наявність компонента Rigidbody2D у об'єкта та стан активності порталу.

```
private void OnTriggerEnter2D(Collider2D other)
 {
     Rigidbody2D rb = other.GetComponent<Rigidbody2D>();
     if (tpActive && rb != null)
     {
```

```
          tpActive = false;
          float magnitude = rb.velocity.magnitude;
          rb.velocity = Vector3.zero;
          Vector3 direction =
toPortal.transform.TransformDirection(Vector3.right) -
transform.TransformDirection(Vector3.left);
          other.transform.position = toPortal.transform.position;
          rb.AddForce(direction * magnitude, ForceMode2D.Impulse);
      }
      else tpActive = true;
 }
```

Переміщення персонажа через портал.

Коли об'єкт з Rigidbody2D входить у портал, виконується наступний процес:

– вимикається активність порталу, щоб уникнути зацикленого переміщення;
– зберігається величина поточної швидкості об'єкта;
– швидкість об'єкта встановлюється на нуль для уникнення проблем при телепортації;
– визначається напрямок переміщення від поточного порталу до цільового.
– об'єкт телепортується до положення цільового порталу;
– застосовується сила до об'єкта, щоб зберегти його кінетичну енергію при переміщенні через портал.

Цей алгоритм забезпечує плавне і реалістичне переміщення персонажа між двома точками у просторі гри, що робить ігровий процес більш динамічним і захоплюючим.

3.5 Створення UI/UX

Створення інтерфейсу користувача (UI) та користувацького досвіду (UX) для гри включає кілька важливих аспектів, які допоможуть зробити гру привабливою та зручною для гравців.[5]

Розробка головного меню з простим та зрозумілим розташуванням кнопок, таких як "Продовжити", "Нова гра", "Налаштування", "Досягнення" та "Вийти з гри". Візуальний стиль меню повинен відповідати тематиці. Створення інтерфейсу

користувача (UI) та користувацького досвіду (UX) для гри включає розробку простого та зрозумілого головного меню в Unity для цього є всі необхідні інструменти. [6]

Ігровий інтерфейс має бути мінімалістичним і інтегрованим у візуальний стиль гри, зручним для гравця. Меню налаштувань повинно дозволяти змінювати ключові параметри гри з інтерактивними елементами, а секція інструкцій має надавати всю необхідну інформацію про управління та механіки.

Інтерфейс повинен забезпечувати миттєвий зворотний зв'язок на дії гравця і бути адаптованим до різних платформ, щоб забезпечити комфортне управління і відображення на різних екранах. (див. рис. 3.5.1).

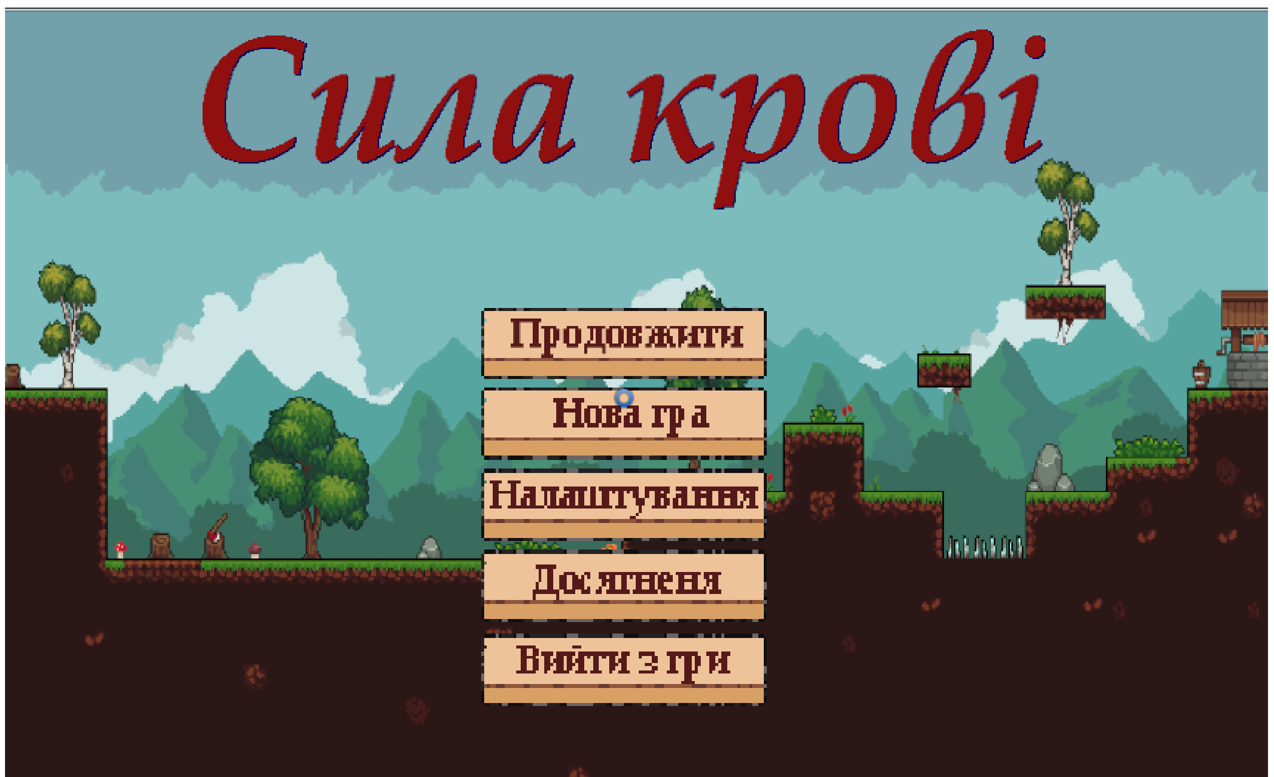


Рисунок 3.5.1 – Головне меню

(рисунок виконаний самостійно)

Меню налаштування надає можливості гравцям налаштовувати графіку, звук, управління та інші параметри гри через інтуїтивно зрозуміле меню налаштувань. Якщо гравець захоче змінити кнопки керування чи змінити графіку він це зможе налаштувати самостійно в "Налаштуваннях керування", та в "Налаштуваннях візуальних" (див. рис. 3.5.2).

Рисунок 3.5.2 – Меню налаштування
(рисунок виконаний самостійно)

Розробка меню паузи, яке дозволяє гравцям зупиняти гру, зберігатися, змінювати налаштування або виходити в головне меню. (див. рис. 3.5.3).

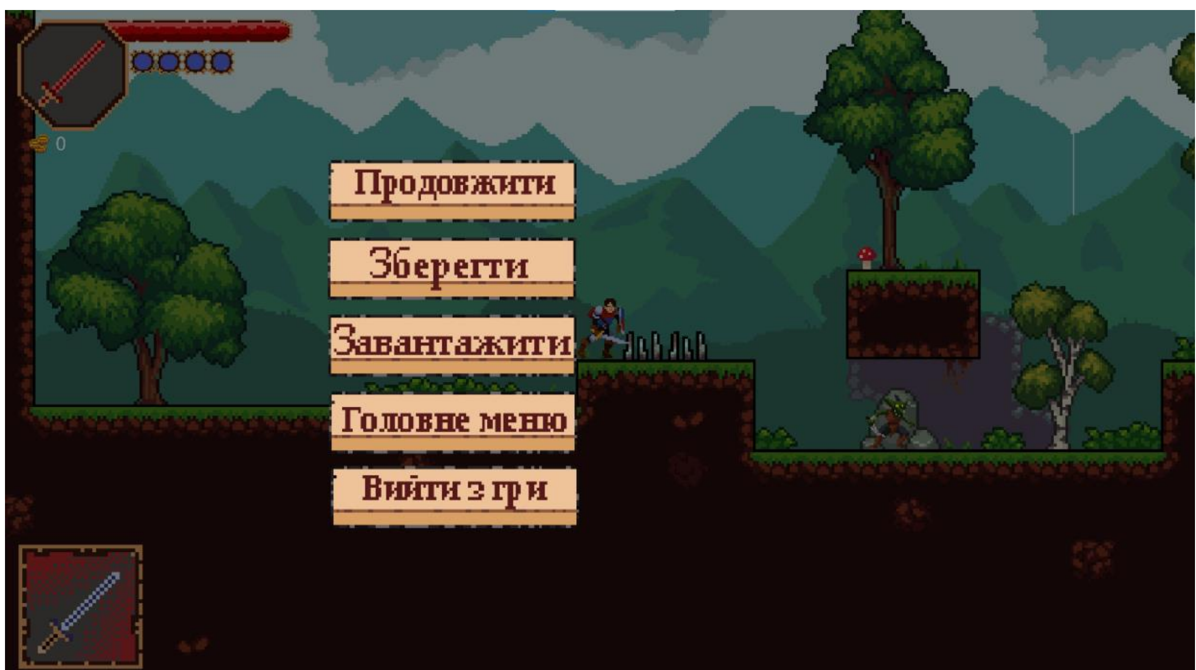


Рисунок 3.5.3 – Меню паузи
(рисунок виконаний самостійно)

Візуальні індикатори.

Індикатори здоров'я, мани та навичок. Відображення стану здоров'я, мани персонажа гравця за допомогою візуальних індикаторів, таких як шкала здоров'я, кристали мани, іконка здібностей. (див. рис. 3.5.4).

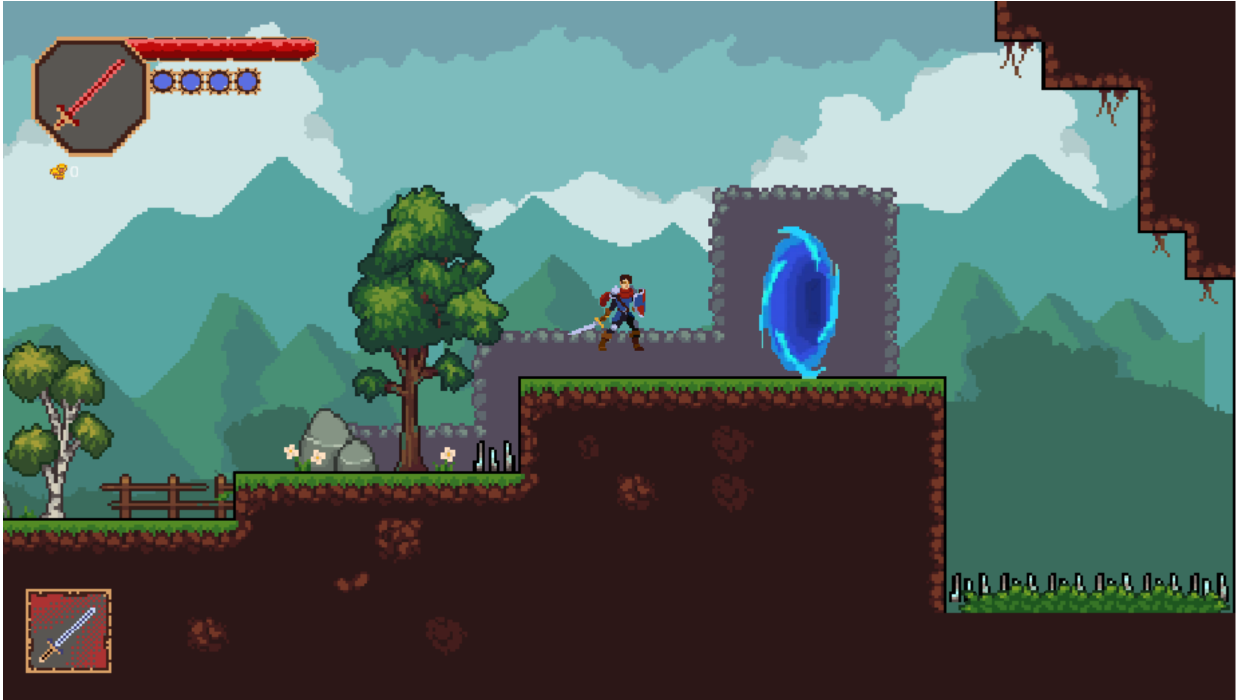


Рисунок 3.5.4 – Індикатори здоров’я, мани та навичок
(рисунок виконаний самостійно)

Відповідальний підхід до розробки інтерфейсу користувача забезпечує інтуїтивну навігацію, зручне управління та візуальну привабливість, що, в свою чергу, підвищує залученість гравців.

Надання гравцям можливості налаштовувати ключові параметри гри, легкий доступ до інструкцій і інтеграція навчальних підказок сприяють кращому освоєнню гри. Забезпечення миттєвого зворотного зв'язку та адаптація інтерфейсу до різних платформ є важливими для створення комфортного ігрового досвіду. В цілому, добре продуманий UI/UX є невід'ємною частиною успішної гри, яка не тільки привертає, але й утримує інтерес гравців.

4 ОПИС ПРИЙНЯТИХ ПРОГРАМНИХ РІШЕНЬ

4.1 Вибір інструментів програмної реалізації

Вибір ігрового рушія є одним з найважливіших рішень на початковому етапі розробки гри, оскільки від цього залежить якість та ефективність всього процесу розробки. Після детального аналізу ринку ігрових рушіїв було вирішено використовувати Unity від Unity Technologies версії 2022.3.36f1. Unity є одним з найпопулярніших і найпотужніших рушіїв для розробки 2D та 3D ігор завдяки своїй гнучкості, широкому набору інструментів та активній спільноті розробників.[7]

Основною перевагою Unity є його багато платформенність. Unity підтримує експорт ігор на різноманітні платформи, включаючи Windows, macOS, iOS, Android, а також консолі, такі як PlayStation, Xbox і Nintendo Switch. Це дозволяє розробникам охопити широку аудиторію та забезпечити доступність гри для користувачів різних пристроїв. Крім того, Unity надає інтегровані інструменти для оптимізації продуктивності на різних платформах, що допомагає забезпечити плавний ігровий процес без затримок та лагів.

Ще однією важливою перевагою Unity є потужні інструменти для розробки ігор, включаючи фізику, анімацію, звукові ефекти та інтеграцію сторонніх бібліотек. Unity також має зручний редактор, який дозволяє швидко створювати та налаштовувати сцени, персонажів та об'єкти. Активна спільнота розробників та велика кількість доступних ресурсів, таких як документація, форуми та навчальні матеріали, дозволяють швидко знаходити рішення проблем та отримувати підтримку, що значно спрощує процес розробки та підвищує його ефективність.

4.2 Архітектура проєкту

Архітектура проєкту базується на модульному підході, що забезпечує гнучкість та легкість в підтримці і розвитку гри. Цей підхід дозволяє розділити гру на окремі, незалежні компоненти, кожен з яких виконує конкретну функцію. Така

структура спрощує роботу з кодом, полегшує тестування , а також дозволяє легко додавати нові функціональні можливості без значних змін у існуючому коді.

Одним з ключових модулів є основний геймплейний модуль, який відповідає за основну логіку гри. Цей модуль включає механіки управління персонажем, взаємодії з оточенням, ворогами та об'єктами. Управління персонажем реалізовано через обробку введення з клавіатури та геймпада, що забезпечує плавний і точний контроль над діями героя. Модуль також включає в себе обробку фізики, що дозволяє реалізувати реалістичні взаємодії, такі як стрибки, зіткнення та гравітація.

Система управління рівнями є ще одним важливим компонентом архітектури проєкту. Вона відповідає за завантаження та управління ігровими рівнями, включаючи перехід між рівнями та збереження прогресу гравця. Цей модуль забезпечує плавний перехід між різними частинами гри, зберігає досягнення гравця та дозволяє відновити гру з того ж місця, де вона була зупинена. Це сприяє покращенню загального досвіду гравця та підтримці його зацікавленості.

Інші важливі компоненти включають систему анімації та систему звуку. Система анімації відповідає за створення та управління анімаціями персонажів та об'єктів, забезпечуючи плавні переходи між різними станами та синхронізацію анімацій з геймплейними подіями. Це дозволяє створити реалістичні та привабливі візуальні ефекти. Система звуку забезпечує відтворення звукових ефектів та музики, що додає атмосферу грі та підвищує рівень занурення. Всі ці модулі працюють разом, створюючи цілісний та захоплюючий ігровий досвід.

4.3 Цікаві програмні рішення

У розробці використання анімаційних тригерів і перехідних станів є ключовим аспектом для створення реалістичного та ефективного керування анімаціями персонажа.[8] Анімаційні тригери, такі як Attack, Jump, Roll, Hurt, активуються при відповідних подіях у грі, що дозволяє забезпечити миттєве переключення між анімаціями у відповідь на дії гравця, такі як натискання клавіш або миші. Це не лише робить реакцію персонажа більш плавною і природною, але й збільшує іммерсію гравця в ігровому процесі.

Перехідні стани важливі для створення плавного переходу між анімаціями, зокрема між рухом, стрибками, атаками та стоячим станом. Це дозволяє уникнути різких змін анімацій, забезпечуючи гладкість і природність рухів персонажа. Наприклад, при переході від руху до стрибка персонаж може плавно активувати анімацію стрибка, що додає реалістичності його поведінці.

Управління рухом і взаємодія з оточенням також відіграє важливу роль у реалізації анімацій. Наприклад, виявлення колізій зі стінами може змінювати анімацію персонажа на стінному ковзанні або реакцію на перешкоди, що додає глибини ігровому світу. Реакція на взаємодію гравця з оточенням, така як відтворення анімацій ураження, блокування атак або виконання спеціальних рухів, додає відчуття взаємодії та динамічності до геймплею.

Нарешті, анімація смерті та відродження персонажа є важливою частиною створення відчуття наслідків у грі. Вона не тільки відтворює втрату здоров'я і смерть персонажа, але й демонструє його повернення до життя після відродження, що робить ігровий процес більш цікавим і динамічним для гравця. Використання всіх цих елементів сприяє створенню іммерсивного досвіду і підвищує візуальну привабливість, забезпечуючи приємне ігрове середовище для користувачів.

```
 else if (Input.GetMouseButtonDown(1) && !m_rolling)
    {
        m_animator.SetTrigger("Block");
        m_animator.SetBool("IdleBlock", true);
    }
    else if (Input.GetMouseButtonUp(1))
    {
        m_animator.SetBool("IdleBlock", false);
    }
    else if (Input.GetKeyDown("left shift") && !m_rolling &&
!m_isWallSliding)
    {
        m_rolling = true;
        m_animator.SetTrigger("Roll");
        m_body2d.velocity = new Vector2(m_facingDirection * m_rollForce,
m_body2d.velocity.y);
    }
    else if (Input.GetKeyDown("space") && m_grounded && !m_rolling)
    {
        m_animator.SetTrigger("Jump");
        m_grounded = false;
        m_animator.SetBool("Grounded", m_grounded);
        m_body2d.velocity = new Vector2(m_body2d.velocity.x, m_jumpForce);
```

```
            m_groundSensor.Disable(0.2f);
        }
        else if (Mathf.Abs(inputX) > Mathf.Epsilon)
        {
            m_delayToIdle = 0.05f;
            m_animator.SetInteger("AnimState", 1);
        }
        else
        {
            m_delayToIdle -= Time.deltaTime;
            if (m_delayToIdle < 0)
                m_animator.SetInteger("AnimState", 0);
        }
}
```

Штучний інтелект ворогів. Розробка алгоритмів поведінки ворогів, таких як патрулювання, напад.

Патрулювання ворогів в ігровому середовищі зазвичай реалізується за допомогою системи точок маршруту. Вороги пересуваються між заданими точками на карті, виконуючи циклічний обхід або статично розміщені маршрути. Під час патрулювання вороги можуть випадково вибирати наступну точку маршруту або чекати певний час у кожній точці перед продовженням маршруту. Цей підхід забезпечує ворогам незалежність від гравця і додає ігрову глибину, дозволяючи створювати варіативність у поведінці ворогів.

```
void Patrol()
{
    transform.position = Vector2.MoveTowards(transform.position,
patrolPoints[currentPointIndex].position, speed * Time.deltaTime);
    if (Vector2.Distance(transform.position,
patrolPoints[currentPointIndex].position) < 0.2f)
    {
        currentPointIndex = (currentPointIndex + 1) % patrolPoints.Length;
        FlipIfNeeded();
    }
     }
```

Напад ворогів спрямований на гравця у випадку, коли той потрапляє в радіус дії ворога. Після виявлення гравця вороги переходять у режим нападу, під час якого вони активно переслідують гравця і намагаються нанести шкоду. Вороги можуть використовувати різноманітні атаки, такі як стрільба або ближній бій, залежно від їхньої типової поведінки і характеристик. Після звільнення гравця від радіусу

нападу або після досягнення цілі вороги можуть повертатися до попереднього стану, такого як патрулювання або очікування на точці.

Ці два аспекти програмування поведінки ворогів дозволяють створювати живу, динамічну ігрову екосистему, де вороги взаємодіють з гравцем і між собою в залежності від ігрових умов і задач.

Ворог використовує дві різні атаки залежно від поточного лічильника атак. Це реалізовано через перевірку ‘attackCount % 2 == 0’, що дозволяє чергувати анімації атаки. Перша атака завдає меншу шкоду (attackDamage1), а друга - більшу (attackDamage2). Кожна атака спровокована входом гравця в радіус атаки ворога, що визначається колайдером.

Код також містить логіку оновлення часу до наступної атаки (nextAttackTime = Time.time + attackCooldown), що контролює частоту атак. Це забезпечує, що ворог не атакуватиме гравця безперервно, а замість цього застосовуватиме паузу між атаками, яка визначається значенням ‘attackCooldown’.

Кожен вихід гравця з радіусу атаки викликає зміну стану ворога на очікування (Idle_Mushroom), під час чого використовується анімація, що відповідає за спокійний стан ворога.

Цей підхід дозволяє створити ворога з динамічними і реактивними атаками, які залежать від розташування гравця та часу між атаками, що зроблює його поведінку більш реалістичною та викликає у гравця стратегічні дії.

```
void Attack()
{
    if (attackCount % 2 == 0)
    {
        enemyAnimator.PlayAnimation("Attack1_Mushroom");
        HealthSystem playerHealth = player.GetComponent<HealthSystem>();
        if (playerHealth != null)
        {
            playerHealth.TakeDamage(attackDamage1);
        }
    }
    else
    {
        enemyAnimator.PlayAnimation("Attack2_Mushroom");
        HealthSystem playerHealth = player.GetComponent<HealthSystem>();
        if (playerHealth != null)
        {
```

```
                playerHealth.TakeDamage(attackDamage2);
            }
        }
        attackCount++;
        Debug.Log("Attacking player!");

}
```

Портал в грі може бути цікавим програмним рішенням, що додає додатковий вимір до геймплею. Портали можуть використовуватися для переміщення гравця між різними частинами рівня або навіть між різними рівнями, створюючи цікаві та непередбачувані виклики.

```
using UnityEngine;
public class Portal : MonoBehaviour
{
    [SerializeField] private Portal toPortal;
    public static bool tpActive;
    private void Start()
    {
        tpActive = true;
    }
    private void OnTriggerEnter2D(Collider2D other)
    {
        Rigidbody2D rb = other.GetComponent<Rigidbody2D>();
        if (tpActive && rb != null)
        {
            tpActive = false;
            float magnitude = rb.velocity.magnitude;
            rb.velocity = Vector3.zero;
            Vector3 direction =
toPortal.transform.TransformDirection(Vector3.right) -
transform.TransformDirection(Vector3.left);
            other.transform.position = toPortal.transform.position;
            rb.AddForce(direction * magnitude, ForceMode2D.Impulse);
        }
        else tpActive = true;
    }
```

Гравці можуть використовувати портали для вирішення головоломок, уникнення перешкод або швидкого переміщення по карті.

5 ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

5.1 Тестування ігрового застосунку

Тестування ігрового застосунку є важливим етапом, оскільки воно спрямоване на виявлення помилок, що можуть вплинути на геймплей, стабільність і загальний користувацький досвід. Під час тестування важливо перевірити, що управління персонажем працює коректно і зручно, що не виникає затримок чи проблем з фізикою гри. Також слід перевірити, що усі ігрові механіки, такі як збирання предметів, взаємодія з ворогами та рішення головоломок, функціонують так, як було заплановано.

Особлива увага приділяється тестуванню графіки і звуку[9]. Графічний двигун повинен забезпечувати плавність анімацій, адаптивність до різних екранних роздільностей і відповідність естетичним вимогам проєкту. Звуковий двигун повинен відтворювати всі звукові ефекти та музику без затримок чи перебоїв, створюючи іммерсивну атмосферу гри.

5.2 Приклади критичних помилок

Під час тестування можуть бути виявлені наступні критичні помилки.

Перш за все, збої програми можуть виникати під час інтенсивних сцен, таких як багато ворогів на екрані або складні взаємодії з оточенням. Це може призвести до непередбачуваного виходу з гри або фризів, що вимагають негайного виправлення.

Друга типова проблема - помилки зі звуком і музикою. Наприклад, неправильне відтворення звукових ефектів під час стрілянини або відсутність фонової музики під час ключових моментів гри може значно погіршити іммерсію гравців.[10]

Також серйозними є проблеми з фізикою гри. Наприклад, неправильність реакції персонажа на гравітацію чи некоректна обробка зіткнень можуть призвести до неприродних рухів або можливості пройти через стіни, що порушує логіку гри.

Значна увага також приділяється проблемам сумісності із різними платформами. Гра повинна однаково добре працювати на різних операційних системах та апаратних платформах без відчутних різниць у продуктивності чи стабільності.

Ці критичні помилки потребують негайного усунення перед випуском гри на ринок, щоб забезпечити якісний користувацький досвід та позитивні відгуки від гравців.[10]

Таблиця 5.2 – Список тестових випадків програмного застосунку

Тест № 1	
Назва тесту:	Перевірка переходу між рівнями
Опис тесту:	Перевірка коректності переходу між рівнями гри платформера.
Компонент системи:	Level Manager
Пріоритет:	P2
Критичність:	S3
Кроки відтворення:	Завантажити гру та перейти до головного меню. Вибрати опцію "Нова гра" та розпочати перший рівень. Завершити перший рівень, щоб активувати перехід до наступного рівня. Перейти до наступного рівня.
Очікуваний результат:	Гравець успішно переходить до наступного рівня.
Фактичний результат:	Перехід не відбувається.
Результат:	Знайдено баг, не виправлено 05.06.2024

Продовження таблиці 5.2

Тест № 2	
Назва тесту:	Перевірка руху персонажа та колізій
Опис тесту:	Перевірка руху персонажа та взаємодії з об'єктами на рівні гри платформера.
Компонент системи:	Character Controller
Пріоритет:	P1
Критичність:	S2
Кроки відтворення:	Запустити гру та перейти до рівня. Рухатись по рівню, включаючи стрибки та переміщення. Спробувати пересунути або взаємодіяти з об'єктами, які мають колізії.
Очікуваний результат:	Персонаж взаємодіє з об'єктами правильно без непередбачених проникнень або блокувань.
Фактичний результат:	Рух та колізії персонажа відпрацьовують коректно.
Висновок:	Знайдено баг, виправлено 05.07.2024
Тест № 3	
Назва тесту:	Перевірка анімацій персонажа
Опис тесту:	Перевірка відтворення та коректності анімацій персонажа в різних сценаріях гри платформера.
Компонент системи:	Animation Controller
Пріоритет:	P2
Критичність:	S3
Кроки відтворення:	Запустити гру та перейти до рівня з різними викликами анімацій персонажа (біг, стрибок, атака тощо). Виконувати дії, які викликають зміну анімацій персонажа. Перевірити, чи анімації відповідають діям персонажа.
Очікуваний результат:	Анімації персонажа відтворюються коректно та без розривів.

Продовження таблиці 5.2

Фактичний результат:	Анімації персонажа відпрацьовують правильно у всіх сценаріях.
Висновок:	Знайдено баг, виправлено 06.07.2024
Тест № 4	
Назва тесту:	Перевірка на отримання шкоди ворогами
Опис тесту:	Перевірка коректності отримання шкоди ворогами від атаки головного персонажа
Компонент системи:	Enemy Damage
Пріоритет:	P1
Критичність:	S2
Кроки відтворення:	Запустити гру та перейти до рівня з ворогами. Підвести головного персонажа до ворога та здійснити атаку (натискання відповідної клавіші). Перевірити реакцію ворога на атаку(втрата здоров'я). Повторити атаку декілька разів, поки ворог не буде повністю переможений.
Очікуваний результат:	Ворог отримує шкоду під час атаки головного персонажа, відображається відповідна анімація та звук, зменшується рівень здоров'я ворога, і після досягнення нуля здоров'я ворог знищується.
Фактичний результат:	Ворог отримує шкоду, правильно відображаються анімації та звуки, рівень здоров'я ворога зменшується, ворог знищується після досягнення нуля здоров'я.
Висновок:	Пройдено 10.07.2024

Продовження таблиці 5.2

Тест № 5	
Назва тесту:	Перевірка отримання шкоди героєм від шипів
Опис тесту:	Перевірка коректності отримання шкоди головним героєм при контакті з шипами у грі платформера.
Компонент системи:	DamageOnContact
Пріоритет:	P1
Критичність:	S2
Кроки відтворення:	Запустити гру та перейти до рівня з шипами. Перемістити головного героя на шипи. Перевірити реакцію головного героя на контакт з шипами (втрата здоров'я, анімація). Відійти від шипів та повторити крок декілька разів.
Очікуваний результат:	Головний герой отримує шкоду при кожному контакті з шипами, відображається відповідна анімація, зменшується рівень здоров'я героя.
Фактичний результат:	Головний герой отримує шкоду при контакті з шипами, правильно відображаються анімації, рівень здоров'я героя зменшується.
Висновок:	Знайдено баг, виправлено 21.06.2024

Усі інші тести були виконані під час розробки програмного коду. Вони були спрямовані на перевірку коректної роботи окремих механік та геймплейних елементів, що стосуються функціональності демонстраційного додатку.

ВИСНОВКИ

У процесі розробки ігрового застосунку в жанрі "Platformer" було реалізовано комплексний підхід, що включав декілька ключових етапів. Ми ретельно планували та реалізували концепцію гри, забезпечуючи структурованість ідей з урахуванням ігрового дизайну, механік, візуального стилю та атмосфери.

Важливою частиною процесу було розуміння потреб та очікувань цільової аудиторії, що допомогло створити гру, яка відповідає їхнім вподобанням і вимогам. Проведений аналіз ринку та опитування серед потенційних гравців дозволили визначити ключові аспекти гри, які є найбільш важливими для нашої аудиторії.

Висока якість технічної реалізації була досягнута завдяки оптимізації коду, використанню Unity2D та мови програмування C#. Цей досвід показав, що комплексний підхід, увага до деталей та врахування зворотного зв'язку є ключовими факторами успіху в розробці ігрових застосунків.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Платформер [Електронний ресурс] – URL: https://uk.wikipedia.org/wiki/%D0%9F%D0%BB%D0%B0%D1%82%D1%84%D0%BE%D1%80%D0%BC%D0%B5%D1%80 (Дата звернення 29.05.2024)

2. Лугова, Т. А. Проектування комп'ютерних ігор для навчання: навч. підручник / Т. А. Лугова, О. А. Блажко ; Одес. нац. політехн. ун-т. – Одеса, 2018. – 209 с.

3. Способи монетизації ігор [Електронний ресурс] – URL: https://avada-media.ua/ua/services/sposoby-monetizacii-igr/ (Дата звернення 01.06.2024)

4. UML-діаграми [Електронний ресурс] – URL: https://dou.ua/forums/topic/40575/ (Дата звернення 10.06.2024)

5. Jesse Schell. The Art of Game Design - CRC Press, 2014. – 600 с.

6. Unity Documentation User interface (UI) [Електронний ресурс] – URL: https://docs.unity3d.com/2022.3/Documentation/Manual/UIToolkits.html (Дата звернення 9.06.2024)

7. Unity Engine [Електронний ресурс] – URL: https://unity.com/products/unity-engine (Дата звернення 11.07.2024)

8. Unity Documentation Animator [Електронний ресурс] – URL: https://docs.unity3d.com/ScriptReference/Animator.SetTrigger.html (Дата звернення 11.07.2024)

9. Тестування ігор: типи та способи тестування мобільних/настільних програм [Електронний ресурс] – URL: https://www.guru99.com/uk/game-testing-mobile-desktop-apps.html (Дата звернення 11.07.2024)

10. Пропустити не можна пофіксити: баги в іграх і чому їх не уникнути [Електронний ресурс] – URL: https://vokigames.com/ua/propustyty-ne-mozhna-pofiksyty-bagy-v-igrah-i-chomu-yih-ne-unyknuty/ (Дата звернення 11.07.2024)

ДОДАТОК А

Звіт результатів перевірки на унікальність тексту в базі ХНУРЕ

Дата звіту 7/12/2024
Дата редагування ---

Звіт не був оцінений.

метадані

Заголовок
Кваліфікаційна робота Буренко О.П. гр. ПЗПІп-22-1

Автор
Буренко Олексій Павлович

Науковий керівник / Експерт
Вадим Юрійович Нечволод

підрозділ
Харківський національний університет радіоелектроніки

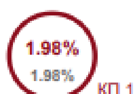
Тривога

У цьому розділі ви знайдете інформацію щодо текстових спотворень. Ці спотворення в тексті можуть говорити про МОЖЛИВІ маніпуляції в тексті. Спотворення в тексті можуть мати навмисний характер, але частіше характер технічних помилок при конвертації документа та його збереженні, тому ми рекомендуємо вам підходити до аналізу цього модуля відповідально. У разі виникнення запитань, просимо звертатися до нашої служби підтримки.

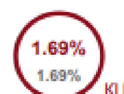
Заміна букв	ß	0
Інтервали	A→	0
Мікропробіли	▯	0
Білі знаки	ß	0
Парафрази (SmartMarks)	a	10

Обсяг знайдених подібностей

Коефіцієнт подібності визначає, який відсоток тексту по відношенню до загального обсягу тексту було знайдено в різних джерелах. Зверніть увагу, що високі значення коефіцієнта не автоматично означають плагіат. Звіт має аналізувати компетентна / уповноважена особа.



КП 1



КЦ

25	6925	54729
Довжина фрази для коефіцієнта подібності 2	Кількість слів	Кількість символів

Подібності за списком джерел

Нижче наведений список джерел. В цьому списку є джерела із різних баз даних. Колір тексту означає в якому джерелі він був знайдений. Ці джерела і значення Коефіцієнту Подібності не відображають прямого плагіату. Необхідно відкрити кожне джерело і проаналізувати зміст і правильність оформлення джерела.

10 найдовших фраз — Колір тексту

ПОРЯДКОВИЙ НОМЕР	НАЗВА ТА АДРЕСА ДЖЕРЕЛА URL (НАЗВА БАЗИ)	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)	
1	123_Rudenko Yurii Andriiovych 6/27/2024 Odessa I.I.Mechnikov National University (Odessa I.I.Mechnikov National University)	42	0.61 %

ДОДАТОК Б

СЛАЙДИ ПРЕЗЕНТАЦІЇ

Міністерство освіти і науки України Харківський національний університет радіоелектроніки

Кваліфікаційна робота бакалавра

Ігровий програмний застосунок в жанрі "Platformer" на рушії Unity

Виконав:
студент гр. ПЗПІп-22-1
Буренко О.П.

Науковий керівник:
к.т.н., доцент каф. ПІ
Кириченко І. В

Рисунок Б.1 – Слайд 1

Мета роботи

- Розробка ігрового програмного застосунку в жанрі "Platformer".
- Створення ігрового процесу, який викличе інтерес та задоволення.
- Забезпечення високого рівня інтерактивності та зворотного зв'язку.
- Забезпечення плавного ігрового процесу без затримок та помилок.
- Забезпечення можливості збереження ігрового прогресу.

Рисунок Б.2 – Слайд 2

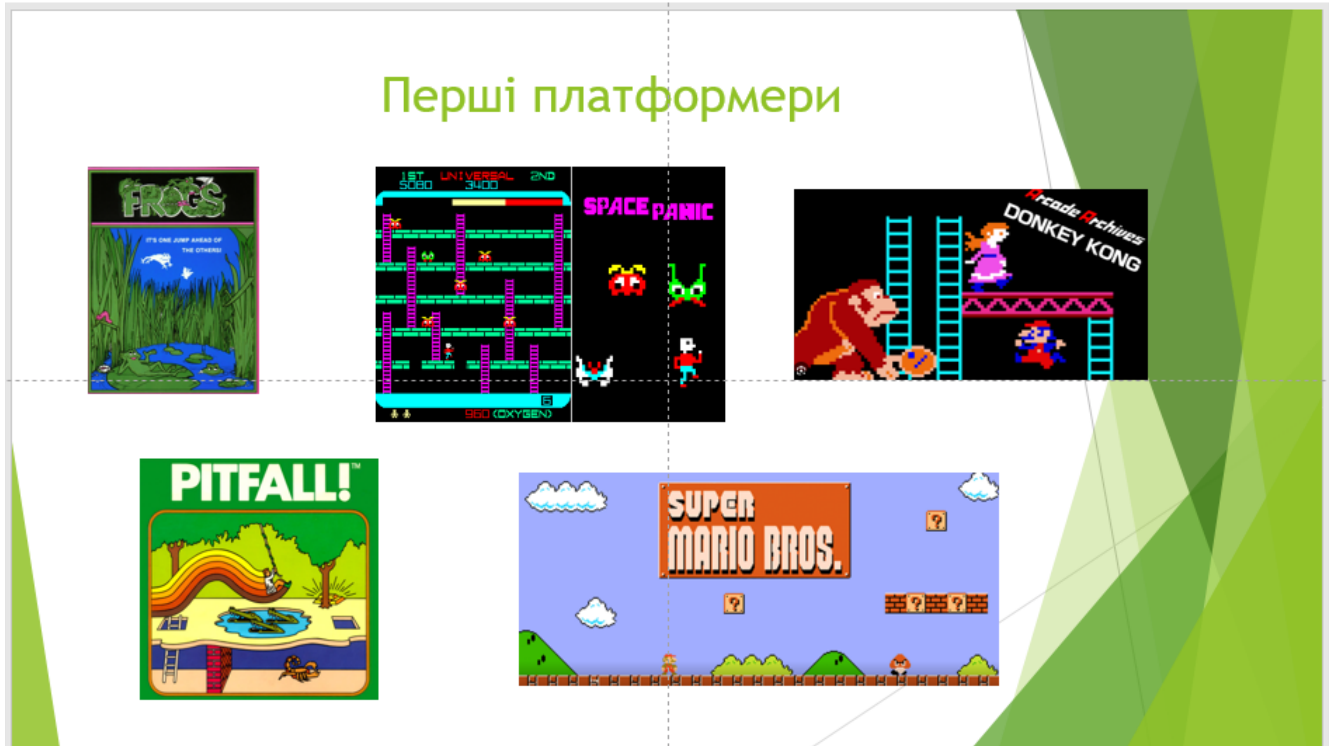


Рисунок Б.3 – Слайд 3

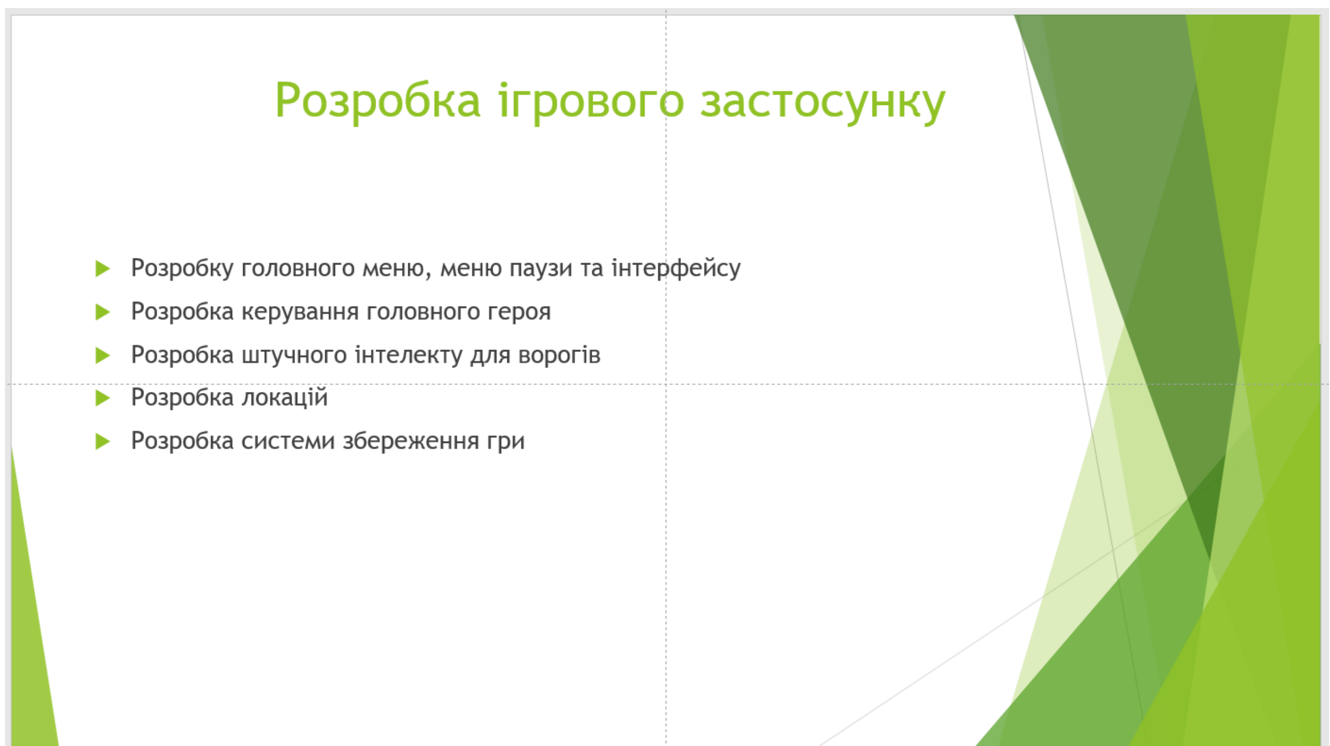


Рисунок Б.4 – Слайд 4

Розробка інтерфейсу гри

Рисунок Б.5 – Слайд 5

Методи керування персонажем у грі:

- Ходьба (Walk/Run): метод для пересування персонажа горизонтально.
- Стрибок (Jump): метод для виконання стрибка.
- Ковзання по стіні (Wall Slide): метод для ковзання по стіні, коли персонаж доторкається до стіни під час падіння.
- Перекат (Roll): метод для виконання перекату, що дозволяє персонажу швидко переміщуватися на коротку відстань.
- Атака (Attack): метод для виконання атаки.
- Блокування (Block): метод для блокування атак ворогів.
- Прийом шкоди (Take Damage): метод для обробки отримання шкоди від ворогів.
- Смерть (Die): метод для обробки стану смерті персонажа.

Рисунок Б.6 – Слайд 6

Штучний інтелект для ворогів

Метод Patrol() відповідає за автоматичний рух ворога між заданими точками патрулювання з певною швидкістю, перевірку досягнення цих точок і зміну напрямку руху при необхідності.

```
void Patrol()
{
    transform.position = Vector2.MoveTowards(transform.position,
patrolPoints[currentPointIndex].position, speed * Time.deltaTime);
    if (Vector2.Distance(transform.position,
patrolPoints[currentPointIndex].position) < 0.2f)
    {
        currentPointIndex = (currentPointIndex + 1) % patrolPoints.Length;
        FlipIfNeeded();
    }
}
```

Рисунок Б.7 – Слайд 7

Розробка локацій

Ми розробляли локації в Unity з використанням Tile Palette для створення деталізованих і інтерактивних середовищ для нашої гри.

- Використовуючи цей інструмент, ми імпортували або створювали набори плиток, розміщували їх на Tilemap, налаштовували колізії та інші властивості, такі як анімація, що дозволяло створювати різноманітні та живі локації.

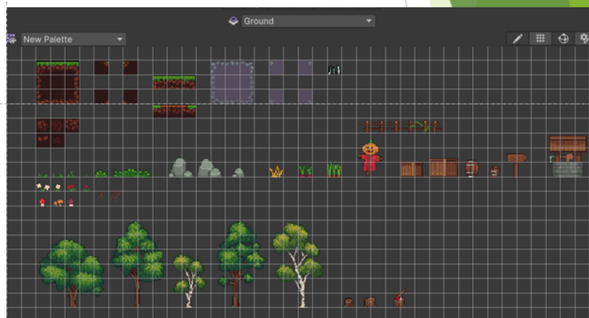


Рисунок Б.8 – Слайд 8

Збереження гри

За збереження гри відповідають три скрипти:

- PlayerData: Цей скрипт відповідає за створення об'єкта, який містить дані про гравця для зберігання. Включає в себе рівень гравця, його здоров'я та позицію у просторі.
- Player: Цей скрипт представляє самого гравця в грі. Він має методи для зберігання та завантаження даних гравця. Використовується для оновлення рівня, здоров'я та позиції гравця під час завантаження гри.
- SaveSystem: Цей скрипт відповідає за зберігання та завантаження даних гравця на диск. Використовує бінарні файли для зберігання об'єкта PlayerData і його пізнішого відновлення під час завантаження гри.

Рисунок Б.9 – Слайд 9

Висновок

- Успішна розробка гри платформера вимагає комплексного підходу, що включає декілька ключових етапів.
- Перш за все, необхідно ретельно планувати та реалізовувати концепцію гри. Це означає, що ідеї мають бути добре структурованими, з урахуванням ігрового дизайну, механік, візуального стилю та атмосфери.
- Концепція повинна бути цікавою і унікальною, щоб привернути увагу гравців і виділитися на фоні численних конкурентів.
- Після випуску гри важливо слідкувати за відгуками користувачів, виправляти помилки та додавати новий контент. Це допоможе утримувати інтерес гравців та забезпечить довготривалу популярність гри.

Рисунок Б.10 – Слайд 10

ДОДАТОК В

Приклад програмного коду

Скрипт здоров’я персонажа

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

public class HealthSystem : MonoBehaviour
{
    public float maxHealth = 100f;
    private float currentHealth;
    public Image healthBar;
    public GameObject deathScreenCanvas;
    public Button restartButton;
    private Animator animator;

    private bool isDead = false;

    private bool isVampirismActive = false;
    private float vampirismDuration = 10.0f;
    private float vampirismEffectInterval = 1.0f;
    private float vampirismHealAmount = 5.0f;
    private float vampirismTimer = 0.0f;

    void Start()
    {
        currentHealth = maxHealth;
        UpdateHealthBar();

        animator = GetComponent<Animator>();

        if (deathScreenCanvas != null)
        {
            deathScreenCanvas.SetActive(false);
        }
        if (restartButton != null)
        {
            restartButton.onClick.AddListener(RestartGame);
        }
    }

    void Update()
    {
        if (isVampirismActive)
        {
            vampirismTimer += Time.deltaTime;

            if (vampirismTimer >= vampirismEffectInterval)
            {
                HealPlayer(vampirismHealAmount);
                vampirismTimer = 0.0f;
```

```
            }

            if (vampirismTimer >= vampirismDuration)
            {
                isVampirismActive = false;
            }
        }
    }

    public void TakeDamage(float amount)
    {
        if (isDead) return;

        currentHealth -= amount;
        if (currentHealth < 0)
        {
            currentHealth = 0;
        }
        UpdateHealthBar();

        if (currentHealth == 0)
        {
            isDead = true;
            animator.SetTrigger("Death");
            ShowDeathScreen();
        }
    }

    void HealPlayer(float amount)
    {
        currentHealth += amount;
        if (currentHealth > maxHealth)
        {
            currentHealth = maxHealth;
        }
        UpdateHealthBar();
    }

    private void UpdateHealthBar()
    {
        if (healthBar != null)
        {
            healthBar.fillAmount = currentHealth / maxHealth;
        }
    }

    private void ShowDeathScreen()
    {
        if (deathScreenCanvas != null)
        {
            deathScreenCanvas.SetActive(true);
        }
    }

    private void RestartGame()
    {
        SceneManager.LoadScene("Level_1");
    }
```

```
    public void ActivateVampirism()
    {
        isVampirismActive = true;
        vampirismTimer = 0.0f;
    }
}
```

Скрипт атаки персонажа

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

public class PlayerAttack : MonoBehaviour
{
    public float attackDamage = 10f;

    void Update()
    {
        if (Input.GetMouseButtonDown(0))
        {
            Attack();
        }
    }

    void Attack()
    {
        Collider2D[] hitEnemies =
Physics2D.OverlapCircleAll(transform.position, 1f);

        foreach (Collider2D enemy in hitEnemies)
        {
            if (enemy.CompareTag("Enemy"))
            {
                enemy.GetComponent<EnemyHealth>().TakeDamage(attackDamage);
            }
        }
    }

    void OnDrawGizmosSelected()
    {
        Gizmos.color = Color.red;
        Gizmos.DrawWireSphere(transform.position, 1f);
    }
}
```

Скрипт патрулювання ворога

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

public class Patroler : MonoBehaviour
{

    public float speed;

    public int positionOfPatrol;
    public Transform point;
    bool moveingRight = true;

    Transform player;
    public float stoppingDistance;

    bool chill = false;
    bool angry = false;
    bool goBack = false;

    void Start()
    {
        player = GameObject.FindGameObjectWithTag("Player").transform;
    }


    void Update()
    {
        if (Vector2.Distance(transform.position, point.position) < positionOfPatrol && angry == false)
        {
            chill = true;
        }

        if (Vector2.Distance(transform.position, player.position) < stoppingDistance)
        {
            angry = true;
            chill = false;
            goBack = false;
        }
        if (Vector2.Distance(transform.position, player.position) > stoppingDistance)
        {
            goBack = true;
            angry = false;
        }

        if (chill == true)
        {
            Chill();
        }
         else if (angry == true)
        {
            Angry();
        }
        else if (goBack == true)
        {
            GoBack();
        }
    }
```

```
    void Chill()
    {
        speed = 1;
        if (transform.position.x > point.position.x + positionOfPatrol)
        {
            moveingRight = false;
        }
        else if (transform.position.x < point.position.x - positionOfPatrol)
        {
            moveingRight = true;
        }

        if (moveingRight)
        {
            transform.position = new Vector2(transform.position.x + speed * Time.deltaTime, transform.position.y);
        }
        else
        {
            transform.position = new Vector2(transform.position.x - speed * Time.deltaTime, transform.position.y);
        }

    }
        void Angry()
        {
        speed = 2;
        transform.position = Vector2.MoveTowards(transform.position, player.position, speed * Time.deltaTime);

        }

        void GoBack()
        {

          transform.position = Vector2.MoveTowards(transform.position, point.position, speed * Time.deltaTime);
        }

}
```