Вінницький національний технічний університет
Факультет інтелектуальних інформаційних технологій та автоматизації
Кафедра комп'ютерних наук

Бакалаврська дипломна робота на тему:

ПРОГРАМНИЙ МОДУЛЬ КОМП'ЮТЕРНОЇ ГРИ «Beta»

Виконав: студент 4 курсу, групи 2КН-20б
спеціальності 122 – Комп'ютерні науки
(шифр і назва напряму підготовки, спеціальності)
Шафір Є.М.
(прізвище та ініціали)

Керівник: д.т.н., проф. каф. КН
Іванчук Я. В.
(прізвище та ініціали)
«07» 06 2024 р.

Рецензент: д.т.н., проф., зав. каф. КСУ
Ковтун В. В.
(прізвище та ініціали)
«07» 06 2024 р.

Допущено до захисту
Зав. кафедри проф., д.т.н. Яровий А.А.
«07» 06 2024 р.

Вінниця ВНТУ – 2024 рік

Вінницький національний технічний університет
Факультет інтелектуальних інформаційних технологій та автоматизації
Кафедра комп'ютерних наук
Рівень вищої освіти перший (бакалаврський)
Галузь знань – 12 Інформаційні технології
Спеціальність – 122 Комп'ютерні науки
Освітньо-професійна програма – Комп'ютерні науки

ЗАТВЕРДЖУЮ
Завідувач кафедри КН
проф., д.т.н. Яровий А.А.
«07» 03 2024р.

З А В Д А Н Н Я
НА БАКАЛАВРСЬКУ ДИПЛОМНУ РОБОТУ СТУДЕНТУ
Шафіру Євгенію Михайловичу

1. Тема роботи: Програмний модуль комп'ютерної гри "Beta"
керівник роботи: Іванчук Я. В., проф.
затверджені наказом вищого навчального закладу "11" 03 2024 року №80
2. Строк подання студентом роботи 27 05 2024 р.
3. Вихідні дані до роботи: кількість гравців - не менше 1 чол., частота кадрів – 60 кадр./сек., розширення екрану – 1920/1080 пікселів, кількість рівнів – 1.
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити): дослідження наявних розробок у галузі комп'ютерних ігор; загальний огляд та особливості ігрової індустрії; порівняння та аналіз ігрових рушіїв; розробка гри на платформі Unity; налаштування та конфігурація проекту; розробка сценаріїв гри на мові програмування C#; тестування результатів роботи.
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень): типова схема архітектури комп'ютерної гри; типова схема взаємодії системних елементів комп'ютерної гри; типова схема алгоритму функціонування комп'ютерної гри; типова схема діаграми станів гравця; схема алгоритму ігрового цикла комп'ютерної гри; загальний вигляд інтерфейсного вікна типу «Головне меню» комп'ютерної гри; загальний вигляд геймплею комп'ютерної гри; загальний вигляд завершення комп'ютерної гри.

6. Консультанти розділів проекту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-3	Іванчук Я. В., д.т.н., проф.. каф. КН		

7. Дата видачі завдання 07.03.2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва та зміст етапу	Термін виконання		Прим
		Початок	Закінчення	
1	Визначення мети та задачі дослідження	06.05.24	09.05.24	
2	Загальний огляд та особливості ігрової індустрії	10.05.24	14.05.24	
3	Порівняння та аналіз програмних засобів	15.05.24	19.05.24	
4	Розробка гри на платформі Unity	20.05.24	24.05.24	
5	Розробка сценаріїв гри на мові програмування C#	25.05.24	31.05.24	
6	Тестування та оцінка результатів роботи	01.06.24	06.06.24	

Студент _____ Шафір Є.М.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Іванчук Я. В.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Бакалаврська дипломна робота складається зі 80 сторінок формату А4, на яких є 19 рисунків, список використаних джерел містить 17 найменувань.

У бакалаврській дипломній роботі запропоновано ігрову систему, орієнтовану на прийняття тактичних рішень у битвах з ворожими істотами та дослідження користувачем локацій. Розроблено моделі ігрової системи, які реалізовані в демонстраційному ігровому додатку «Beta». Програмний продукт було створено з використанням мови програмування C# та платформи Unity. В якості середовища розробки програмного забезпечення було обрано Rider.

Розроблені метод та модель гри можуть бути використані при розробці сучасних ігрових систем, у яких передбачена робота з штучним інтелектом.

Ключові слова: комп’ютерна гра, 2D.

ABSTRACT

The bachelor's thesis consists of 80 A4 pages with 19 figures and a bibliography of 17 references.

The bachelor's thesis proposes a game system focused on making tactical decisions in battles with enemy creatures and exploring locations by the user. Models of the game system were developed and implemented in the Beta demo game application. The software product was created using the C# programming language and the Unity platform. Rider was chosen as the software development environment.

The developed method and game model can be used in the development of modern gaming systems that involve work with artificial intelligence.

Keywords: computer game, 2D.

ЗМІСТ

ВСТУП

Актуальність. Індустрія комп'ютерних ігор стрімко розвивається та займає одне з провідних місць у сфері розваг, культурного обміну та навіть освіти. Згідно зі статистичними даними, світовий ринок комп'ютерних ігор перевищив 150 мільярдів доларів у 2023 році і продовжує зростати швидкими темпами [1]. Цей феномен пояснюється не лише високим попитом на розважальний контент, але й можливістю використання ігор у різних сферах життя.

По-перше, комп'ютерні ігри стимулюють розвиток креативного мислення, стратегічного планування та координації, що є важливими навичками у сучасному світі. Вони також можуть слугувати ефективним інструментом для навчання, пропонуючи інтерактивні методи засвоєння матеріалу, які часто виявляються більш ефективними, ніж традиційні методи викладання [2]. Наприклад, навчальні симулятори використовуються для підготовки пілотів, лікарів та інших спеціалістів, що демонструє практичну цінність ігор поза межами розваг.

По-друге, розробка ігор створює численні робочі місця у різних сферах, від програмування та дизайну до маркетингу та менеджменту. Це стимулює розвиток високих технологій та інновацій, сприяє зростанню економіки і формуванню нових ринків.

По-третє, сучасні комп'ютерні ігри є важливим культурним феноменом, що відображає та формує світогляд молодого покоління. Вони слугують засобом соціалізації та взаємодії між гравцями з усього світу, сприяють обміну ідеями та культурними цінностями.

Таким чином, розробка комп'ютерної гри є актуальним і важливим завданням, що вимагає міждисциплінарного підходу та інтеграції знань з різних галузей. У даній дипломній роботі будуть розглянуті ключові аспекти процесу розробки комп'ютерних ігор, зокрема, концептуальне проектування, технічна реалізація та аналіз ефективності кінцевого продукту.

Метою дослідження є розширення можливостей для зняття стресу та розвитку когнітивних здібностей користувача на основі розробленої

комп’ютерної гри «Beta».

Для досягнення мети потрібно розв’язати наступні **задачі**:

1. Проаналізувати жанри комп’ютерних ігор та відомі комп’ютерні ігри відповідного жанру.
2. Виконати порівняльний аналіз програмних рушіїв комп’ютерних ігор.
3. Спроектувати програмний модуль комп’ютерної гри.
4. Розробити програмний модуль комп’ютерної гри з урахуванням вимог до функціональності та продуктивності.
5. Протестувати та оптимізувати програмний модуль комп’ютерної гри для забезпечення стабільної та ефективної роботи.
6. Розробити інструкцію користувача.

Об’єктом дослідження є процес розробки програмного модуля комп’ютерної гри.

Предметом дослідження є алгоритми та програмні засоби комп’ютерної гри.

Апробація була проведена на Всеукраїнська науково-технічна конференція факультету інтелектуальних інформаційних технологій та автоматизації (2024) [3].

Публікації: електронні тези на тему «Сучасні інструменти розробки комп’ютерних ігор» науково-технічної конференції «LIII Всеукраїнська науково-технічна конференція факультету інтелектуальних інформаційних технологій та автоматизації (2024)» [3].

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз жанрів комп’ютерних ігор

Жанри комп’ютерних ігор мають важливе значення для галузі розваг, визначаючи грунтовність гри, спрямованість аудиторії та комерційний успіх [4]. Кожен стиль має свої унікальні риси та механіки, які приваблюють певні групи гравців. Давайте розглянемо головні жанри комп’ютерних ігор, їх характеристики та приклади популярних ігор у кожному жанрі.

Жанр екшену характеризується швидкістю, точністю та реакцією [5]. Гравець повинен швидко орієнтуватися в ситуації та максимально ефективно використовувати бойові механіки, щоб вижити і перемогти ворогів. Дія в іграх цього жанру зазвичай відбувається в реальному часі, що підвищує необхідність швидкого прийняття рішень. Гравець керує персонажем, який має долати перешкоди, боротися з ворогами та виконувати завдання.

Шутери, як піджанр, сфокусовані на стрілянині з різних видів зброї. Від першої особи (FPS) гравець бачить світ очима персонажа, що створює ефект присутності, тоді як від третьої особи (TPS) гравець бачить персонажа повністю, що дозволяє краще контролювати його рухи. Інші піджанри, такі як бійки та файтинг, роблять акцент на рукопашному бою. Вони часто включають комбо-удари, спеціальні атаки та різноманітні стилі бою.

Здоров’я персонажа та ресурси часто обмежені, що додає виклику для гравця. У деяких іграх є часові обмеження, які можуть створювати додатковий тиск. Більшість ігор екшену мають кілька рівнів або етапів з поступово зростаючою складністю, де гравець стикається з різними типами ворогів і перешкодами.

На рисунку 1.1 показано приклад комп’ютерної гри у жанрі екшену.

Рисунок 1.1 – Загальний вигляд комп’ютерної гри жанру «екшен» “Superhot”

Пригодницькі ігри, на відміну від екшену, зосереджені на сюжеті, дослідженні та вирішенні головоломок [6]. Гравець зазвичай керує персонажем, який досліджує світ, спілкується з іншими персонажами, збирає предмети та розкриває таємниці. Сюжет таких ігор має деталі, які допомагають гравцеві заглибитися в історію та краще зрозуміти мотивації персонажів. Часто в пригодницьких іграх сюжет розвивається через діалоги та взаємодію з навколишнім світом.

Гравець повинен вирішувати головоломки та виконувати завдання для просування по сюжету. Головоломки можуть бути різних видів, від простих головоломок зі збіркою предметів до складних логічних завдань. Дослідження світу є ключовим елементом, оскільки гравець повинен уважно вивчати локації та взаємодіяти з об’єктами для отримання підказок або нових завдань.

Темп гри зазвичай повільніший порівняно з екшеном, що дозволяє гравцям заглиблюватися в історію та насолоджуватися дослідженням. Піджанри, такі як Point-and-Click, де гравець взаємодіє з об’єктами через інтерфейс, пропонують інтуїтивний і простий спосіб керування персонажем.

На рисунку 1.2 показано приклад комп’ютерної гри у жанрі пригоди.

Рисунок 1.2 – Загальний вигляд комп’ютерної гри жанру «пригоди» “Tomb Raider”

Рольові ігри дозволяють гравцям зануритися в детально розроблений світ і взяти на себе роль героя, який поступово розвиває свої здібності та навички [7]. Однією з ключових особливостей RPG є розвиток персонажів. Гравець збирає досвід, підвищує рівень і покращує характеристики персонажа, такі як сила, спритність, інтелект чи витривалість. Зазвичай гравець має широкий вибір спорядження, зброї та навичок, які можна вдосконалювати або змінювати, щоб підлаштувати персонажа під свій стиль гри. Цей процес супроводжується збором ресурсів та виконанням квестів.

Сюжетні вибори є важливим елементом рольових ігор. Вони дають гравцям можливість впливати на розвиток історії, взаємини з іншими персонажами та навіть на фінал гри. Такі рішення часто супроводжуються моральними дилемами, змушуючи гравця обирати між різними шляхами розвитку сюжету.

Світ у рольових іграх зазвичай детально розроблений, з різноманітними локаціями, NPC (неігровими персонажами), фракціями та відкритими територіями для дослідження. Однак є також ігри з більш лінійним підходом до сюжету, де гравець поступово просувається через окремі рівні. У відкритому світі гравець може

вільно подорожувати, знаходити нові завдання та досліджувати світ, розкриваючи його історію з різних боків.

Піджанри в рольових іграх різноманітні. Західні рольові ігри (WRPG) надають більше свободи та часто мають відкритий світ, тоді як японські рольові ігри (JRPG) більше зосереджуються на сюжеті та покрокових боях. Масові рольові онлайн-ігри (MMORPG) дозволяють тисячам гравців взаємодіяти один з одним у величезних віртуальних світах, об’єднуючись у гільдії, виконуючи рейди та змагаючись у PvP-боях.

На рисунку 1.3 показано приклад комп’ютерної гри у жанрі рольової гри.

Рисунок 1.3 – Загальний вигляд комп’ютерної гри жанру «рольова гра» “Baldurs Gate 3”

Стратегії вимагають від гравця стратегічного мислення, планування та тактичних рішень [8]. Основною метою є досягнення переваги над суперником шляхом управління ресурсами, будівництва баз та контролю над юнітами. Гравець повинен аналізувати ситуацію, прогнозувати дії ворога та приймати зважені рішення для досягнення перемоги.

Планування та тактика є центральними елементами жанру. Гравець повинен вибудувати стратегію розвитку своєї економіки, військової потужності та оборони, обираючи між агресивним або оборонним стилем гри. Управління ресурсами включає збір і розподіл матеріалів, будівництво будівель та тренування юнітів.

Стратегії поділяються на два основні типи: реального часу (RTS) та покрокові (TBS). У стратегіях реального часу гравці діють одночасно, приймаючи рішення та виконуючи дії у динамічному режимі. Покрокові стратегії дозволяють кожному гравцеві по черзі приймати рішення та виконувати дії, що дає більше часу на обдумування тактики.

4X-стратегії (дослідження, розширення, експлуатація та знищення) — це глобальні стратегії, де гравець керує цілою цивілізацією або імперією. Управління включає дипломатію, економіку, військові дії та наукові дослідження. Гравець повинен досліджувати світ, розширювати свої території, експлуатувати ресурси та знищувати супротивників.

На рисунку 1.4 показано приклад комп’ютерної гри у жанрі стратегії.

Рисунок 1.4 – Загальний вигляд комп’ютерної гри жанру «стратегія» “0 A.D.”

Головоломки пропонують гравцям завдання різного рівня складності, які вимагають логічного мислення, спостережливості та терпіння [9]. Центральним елементом жанру є пошук рішень для виходу з конкретних ситуацій чи завдань, спираючись на логіку або творчі здібності гравця.

Традиційні головоломки включають складання пазлів, вирішення неочевидних проблем або системних завдань. У грі потрібно бути уважним до деталей, думати стратегічно та враховувати підказки, які допомагають знайти рішення.

Фізичні головоломки додають інтерактивності та залучають реалістичну фізику. Гравець повинен вирішувати головоломки шляхом взаємодії з об’єктами, використовуючи їх вагу, траєкторію або інші параметри для досягнення ваших цілей.

Логічні головоломки зазвичай містять системи чи алгоритми, де потрібно правильно вибудувати послідовність дій або зібрати предмети у чіткій комбінації. Гравець вирішує завдання послідовним методом, щоб досягти результату.

На рисунку 1.5 показано приклад комп’ютерної гри у жанрі головоломки.

Рисунок 1.5 – Загальний вигляд комп’ютерної гри жанру «головоломка» “The Talos Principle 2”

Платформери — це один із найстаріших і найвідоміших жанрів комп’ютерних ігор, який став популярним ще за часів аркадних автоматів та перших домашніх консолей. Основна особливість платформерів полягає в тому, що гравець керує персонажем, який повинен пересуватися по рівнях, стрибаючи з однієї платформи на іншу, уникаючи перешкод і збираючи різні предмети [10].

Основними елементами ігрового процесу у платформерах є платформи, які можуть бути як статичними, так і рухомими. Гравець повинен використовувати ці платформи для просування по рівню, стрибаючи між ними. Крім того, в іграх цього жанру часто присутні перешкоди та вороги, яких потрібно уникати або перемагати.

На рисунку 1.6 показано приклад комп’ютерної гри у жанрі платформера.

Рисунок 1.6 – Загальний вигляд комп’ютерної гри жанру «платформер» “SuperTux”

Сюжет у платформерах зазвичай простий і часто зводиться до порятунку персонажа або досягнення якоїсь кінцевої мети. Прогресія через рівні зазвичай лінійна, але іноді може містити елементи відкритого світу.

Платформери мають кілька піджанрів. Класичні платформери зосереджені на основних елементах жанру. Метроїдванія поєднує платформер з елементами

дослідження і розвитку персонажа. Пазл-платформери включають головоломки, які потрібно вирішувати для просування вперед. Раннери є спрощеною версією платформерів, де персонаж автоматично рухається вперед, а гравець контролює лише стрибки та дії.

Після ретельного аналізу та розгляду різних жанрів комп'ютерних ігор, я прийшов до висновку, що платформер є найкращим вибором для моєї дипломної роботи. Цей жанр є популярним та доступним для широкої аудиторії, а також має великий потенціал для творчості та інновацій в дизайні рівнів та ігрової механіки. Крім того, я впевнений, що мої навички та знання в галузі розробки комп'ютерних ігор дозволять мені створити платформер, який буде цікавим, захоплюючим та відрізнятиметься від інших комп'ютерних ігор цього жанру.

1.2 Порівняльний аналіз програмних рушіїв комп'ютерних ігор

Розробка комп'ютерних ігор вимагає не лише базових знань про їх створення та уявлення про те, як вони виглядатимуть, але й належних інструментів та середовища для розробки ігрового інтерфейсу та написання коду, який керує грою.

Для створення комп'ютерних ігор існують спеціалізовані програми, які дозволяють розробникам створювати зовнішній вигляд гри та встановлювати її параметри. Графіка може бути створена як єдиною конструкцією (фоном), так і вручну за допомогою текстур. Фізика рухомих і нерухомих об'єктів у грі встановлюється за допомогою колайдерів, які накладаються на елементи графічного інтерфейсу, щоб дозволити об'єктам гри взаємодіяти один з одним.

Середовища розробки комп'ютерних ігор також дозволяють надавати доступ до свого проекту іншим розробникам, щоб дозволити спільну роботу над розробкою. Перегляд і перевірка гри на працездатність можуть здійснюватися в середовищі розробки, а перегляд коду також доступний, але його написання та редагування відбуваються безпосередньо в середовищах програмування.

Таким чином, розробка комп'ютерних ігор є складним процесом, який вимагає не лише знань та уяви, але й належних інструментів та середовища для розробки.

Спеціалізовані програми та середовища розробки дозволяють розробникам створювати високоякісні ігри, які задовольняють потреби та бажання гравців.

Наприклад, однією з таких програм є Unreal Engine, розроблений Epic Games, є одним з провідних рушіїв для створення ігор і інтерактивних віртуальних світів [11]. Завдяки своїй потужності і гнучкості, він знаходить застосування як у комп'ютерних іграх, так і в кіноіндустрії, архітектурних візуалізаціях і навіть у тренажерах. У цьому підрозділі ми детально розглянемо особливості Unreal Engine, його переваги та недоліки.

На рисунку 1.7 зображено інтерфейс рушія Unity.

Рисунок 1.7 — Загальний вигляд інтерфейсу рушія Unreal Engine 5

Особливості Unreal Engine:

1. Графічні можливості: Unreal Engine славиться своєю здатністю створювати високоякісні, фотореалістичні графічні зображення. Завдяки передовим технологіям рендерингу, таким як Lumen і Nanite.
2. Blueprints: система візуального програмування, яка дозволяє розробникам створювати ігрову логіку без написання коду. Вона ідеально підходить для дизайнерів та художників, які можуть швидко тестувати свої ідеї.

3. Вбудовані інструменти для розробки: Unreal Engine включає широкий набір інструментів для анімації, звукових ефектів, створення візуальних ефектів і навіть інструменти для роботи з віртуальною реальністю.
4. Фізичний рушій: Unreal Engine використовує передовий фізичний рушій, що дозволяє створювати реалістичні симуляції фізичних взаємодій та ефектів.

Переваги Unreal Engine:

1. Висока якість графіки: однією з головних переваг Unreal Engine є його здатність створювати високоякісну графіку. Це робить його ідеальним для розробки ААА-ігор і проектів, що вимагають високого рівня візуальної деталізації.
2. Потужний інструментарій: Unreal Engine надає розробникам безліч потужних інструментів для роботи з анімацією, звуком, світлом і візуальними ефектами. Це дозволяє створювати комплексні проекти без необхідності використання сторонніх інструментів.
3. Кросплатформеність: рушій підтримує експорт проектів на різні платформи, включаючи ПК, консолі, мобільні пристрої і навіть віртуальну реальність. Це дозволяє розробникам охоплювати широку аудиторію.
4. Безкоштовна версія: Unreal Engine доступний безкоштовно, а розробники платять роялті тільки у випадку комерційного успіху проекту. Це робить його доступним для незалежних розробників і малих студій.

Недоліки Unreal Engine:

1. Тяжкий в освоєнні: Unreal Engine має потужні інструменти, але вони можуть бути складними для новачків. Система Blueprints значно полегшує процес, але для повноцінного використання всіх можливостей рушія потрібні знання C++.
2. Вимогливість до ресурсів: Unreal Engine вимагає потужного апаратного забезпечення, що може бути проблемою для невеликих студій або індивідуальних розробників з обмеженими ресурсами.
3. Великий розмір інсталяційних файлів: проекти, створені на Unreal Engine, часто мають великий розмір інсталяційних файлів, що може бути незручним для користувачів з обмеженим дисковим простором або низькою швидкістю

інтернету.

4. Комерційна ліцензія: хоча Unreal Engine доступний безкоштовно, комерційне використання вимагає виплати роялті у розмірі певного відсотка від прибутку. Це може бути недоліком для великих проектів з високими доходами.

Unreal Engine є одним з найпотужніших і найбільш гнучких інструментів для розробки ігор і інтерактивних проектів. Його високоякісні графічні можливості, потужні інструменти і широка кросплатформеність роблять його популярним вибором серед розробників. Однак складність у використанні і вимогливість до ресурсів доволі висока перешкода для новачка.

Інший відомим рушієм є Unity. Unity – це один з найпопулярніших рушіїв для розробки ігор, який використовується розробниками по всьому світу [7]. Його універсальність і доступність роблять його привабливим як для новачків, так і для досвідчених професіоналів. Розглянемо основні риси Unity, його переваги та недоліки.

Unity є багатоплатформним рушієм, що дозволяє створювати ігри для різних операційних систем і пристроїв, включаючи ПК, консолі, мобільні телефони і навіть веб-браузери. Основні характеристики Unity включають:

1. Графічний інтерфейс: Unity має інтуїтивно зрозумілий редактор, що дозволяє розробникам легко маніпулювати об’єктами у тривимірному просторі. Це значно полегшує процес створення сцени і налаштування ігрових об’єктів.
2. Скриптовий інтерфейс: для створення логіки гри Unity використовує мову програмування C#. Це надає розробникам потужні інструменти для реалізації складних ігрових механік.
3. Підтримка плагінів: Unity має вбудовану підтримку плагінів та інструментів, які можна інтегрувати для розширення функціональності. Unity Asset Store пропонує безліч готових рішень, які можуть бути легко інтегровані у ваш проект.
4. Фізичний рушій: вбудований фізичний рушій дозволяє реалізувати реалістичну фізику і колізії, що робить ігри більш захоплюючими.

На рисунку 1.8 зображено інтерфейс рушія Unity.

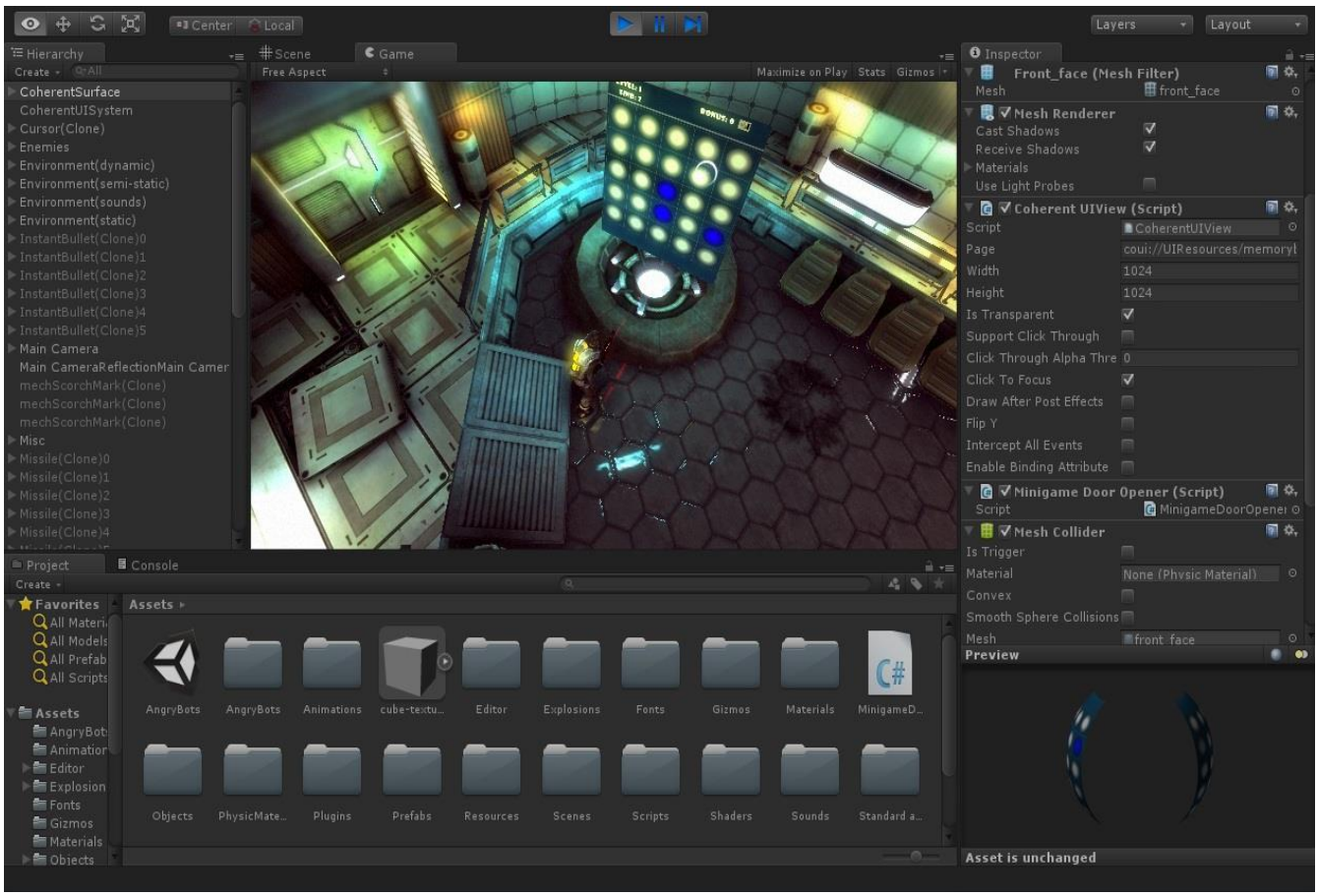

Рисунок 1.8 — Загальний вигляд інтерфейсу рушія Unity

Переваги Unity:

1. Мультиплатформеність: однією з найбільших переваг Unity є його здатність експортувати ігри на безліч платформ без необхідності значних змін у коді. Це дозволяє охопити широку аудиторію гравців.
2. Доступність: Unity надає безкоштовну версію для індивідуальних розробників та невеликих студій, що дозволяє початківцям розробникам створювати свої проекти без значних фінансових вкладень.
3. Активна спільнота: велика і активна спільнота розробників готова допомогти новачкам, а також постійно створює нові ресурси, туторіали та плагіни, які можуть стати в нагоді в процесі розробки.
4. Потужні інструменти: Unity надає широкий спектр інструментів для створення високоякісних ігрових проектів, включаючи інструменти для анімації, візуальних ефектів та звукового супроводу.

Недоліки Unity:

1. Обмеження у графіці: незважаючи на потужний набір інструментів, Unity іноді не дотягує до рівня фотореалістичних рушіїв, таких як Unreal Engine, особливо у високобюджетних проектах.
2. Продуктивність: деякі розробники стикаються з проблемами продуктивності, особливо при створенні великих ігрових світів. Це може вимагати додаткової оптимізації коду та ресурсів.
3. Велика вага інсталяторів: ігри, створені на Unity, можуть мати досить великий розмір інсталяційного файлу, що може бути незручним для користувачів з обмеженим дисковим простором або низькою швидкістю інтернету.
4. Залежність від сторонніх плагінів: Unity Asset Store пропонує багато корисних плагінів, але надмірна залежність від них може призвести до проблем із сумісністю та підтримкою.

Unity є потужним і гнучким інструментом для розробки ігор, який пропонує широкий спектр можливостей для розробників різного рівня. Незважаючи на деякі недоліки, його переваги, такі як мультиплатформеність, доступність та активна спільнота, роблять його одним з найпопулярніших виборів серед розробників ігор. Використання Unity дозволяє створювати різноманітні проекти – від простих інді-ігор до складних комерційних проектів.

1.3 Порівняльний аналіз комп’ютерних ігор у жанрі 2D платформер

2D платформери – це унікальний жанр комп’ютерних ігор, який пропонує гравцям захоплюючу комбінацію простої механіки, складних рівнів, яскравої графіки та елементів дослідження. Цей жанр зародився в епоху аркадних ігор і продовжує еволюціонувати сьогодні, пропонуючи гравцям нові та інноваційні підходи до класичного ігрового процесу [8].

Однією з ключових особливостей 2D платформерів є те, що вони містять унікальні механіки руху, які дозволяють гравцям виконувати неймовірні трюки та комбо. Наприклад, в грі Celeste гравці можуть прилипати до стін та стрибати між ними, в той час як в грі Ori and the Blind Forest персонаж може стрибати між

платформами та використовувати здібності, такі як телепортація та створення енергетичних платформ.

2D платформери також часто пропонують унікальні підходи до оповіді та атмосфери. Багато ігор в цьому жанрі мають глибокі та емоційні сюжети, які розкриваються через діалоги, кат-сцени та навколишнє середовище. Наприклад, в грі Limbo гравець керує хлопчиком, який шукає свою сестру в темному та небезпечному світі, в той час як в грі Hollow Knight гравець досліджує руїни стародавнього королівства, сповненого таємниць та небезпек.

Розглянемо ж декілька прикладів сучасних 2D платформерів більше детально. Першою комп'ютерною грою буде Shovel Knight, розроблена Yacht Club Games, є сучасною класикою у світі незалежних комп'ютерних ігор, що вирізняється вшануванням епохи 8-бітних комп'ютерних ігор, а також сучасним дизайном [9]. Ігровий процес Shovel Knight – це майстерне поєднання ностальгії та інновацій, що відображає глибоке розуміння як ретро, так і сучасної ігрової механіки.

Рисунок 1.9 – Загальний вигляд платформеру "Shovel Knight"

За своєю суттю, Shovel Knight – це 2D-платформер з бічною прокруткою, де гравці керують титулованим лицарем, основною зброєю якого є лопата. Цей, здавалося б, химерний вибір зброї приховує її універсальність, оскільки лопата виконує безліч функцій, як наступальних, так і утилітарних. Вона дозволяє лицареві відкопувати скарби, розбивати блоки і, що особливо важливо, виконувати «Shovel Drop» – атаку вниз, яка також може бути використана для відскоку від ворогів і предметів.

Ігровий дизайн Shovel Knight ретельно продуманий, щоб запропонувати складний, але чесний досвід. Кожен рівень наповнений унікальними ворогами, хитромудрими послідовностями платформ і прихованими секретами, які винагороджують за дослідження та майстерність. Крива складності точно налаштована, що гарантує, що нові гравці поступово знайомляться з механікою, в той же час надаючи досвідченим геймерам достатньо викликів.

Однією з визначних особливостей Shovel Knight є дизайн рівнів, який черпає натхнення в класичних іграх, таких як Mega Man, Castlevania та Super Mario Bros. Кожен етап присвячений певному босу, Order of No Quarter, який представляє унікальний набір завдань і механік, що відповідають його володінню. Наприклад, замок Короля Лицаря має розкішні, повні пасток коридори, а лабораторія Чумного Лицаря рясніє летючими хімікатами та вибухонебезпечними речовинами. Ця тематична узгодженість поширюється на небезпеки навколишнього середовища і типи ворогів, що гарантує, що кожен рівень відчуває себе окремим і цілісним.

Система управління в Shovel Knight чутлива і точна, що дуже важливо для платформера, де час і точність мають першорядне значення. Ідеальне піксельне виявлення зіткнень та жорстке керування забезпечують ігровий процес, який приносить задоволення, але водночас і вимагає багато зусиль. Такий рівень контролю необхідний для опанування різноманітних викликів платформингу та зустрічей з ворогами, які гра підкидає гравцеві.

Битви з босами в Shovel Knight – ще одна родзинка, кожна з яких вимагає глибокого розуміння моделей атаки боса і власних здібностей гравця. Ці сутички призначені для перевірки навичок, які гравці відточували протягом проходження

рівнів, і вимагають швидких рефлексів та стратегічного мислення. Самі боси запам’ятовуються як дизайном, так і тим, як вони кидають виклик гравцеві, часто вимагаючи інноваційного використання здібностей Лицаря Лопати.

На додаток до основної кампанії, Shovel Knight має кілька доповнень, кожне з яких додає нових ігрових персонажів з унікальними здібностями і сюжетними лініями. Ці доповнення, такі як «Чума тіней» та «Привид мук», не лише подовжують тривалість гри, але й пропонують нові перспективи та ігрові механіки, які роблять гру свіжою та захопливою. Кожен персонаж керується по-своєму і привносить у гру свою родзинку, гарантуючи, що гравці, які повертаються до гри, знайдуть нові виклики та насолоду.

Друга комп’ютерна гра, яку я б хотів розглянути буде Inside. Inside – це гра-головоломка-платформер, яка майстерно створює відчуття тривоги та цікавості, занурюючи гравців у таємничий світ, який водночас захоплює та викликає тривогу [10]. Ігровий процес ретельно продуманий, щоб створити відчуття потоку, де кожна частина будується на попередній, щоб створити цілісний і захоплюючий досвід.

Рисунок 1.10 — Загальний вигляд платформеру “Inside”

З самого початку управління Inside є інтуїтивно зрозумілим, що дозволяє гравцям з легкістю керувати рухами героя. Світ гри представлений у 2.5D перспективі, з розумним розташуванням камери для створення відчуття глибини та атмосфери. Провівши героя крізь навколишнє середовище, гравці зіткнуться з різноманітними перешкодами, від навігації підступною місцевістю до уникнення смертельних пасток і ворогів.

Одним із визначних аспектів ігрового процесу Inside є використання головоломок. Ці головоломки розумно інтегровані в навколишнє середовище, часто вимагаючи від гравців творчого мислення та нестандартного використання здібностей головного героя. Наприклад, на початку гри гравці зіткнуться з розділом, де вони повинні використати здатність головного героя керувати зомбі-подібною істотою, щоб дістатися до віддаленої платформи. Ця механіка вводиться тонко, і перша зустріч гравця з істотою більше схожа на експеримент, ніж на навмисну головоломку.

Впродовж гри гравці зіткнуться з різноманітними сценами, від напружених погонь до драматичних протистоянь. Ці епізоди майстерно підібрані за темпом, а звуковий та візуальний ряд гри поєднується, щоб створити справді захоплюючий досвід.

Зрештою, ігровий процес Inside – це майстер-клас з дизайну та темпу. Головоломки, платформи та стелс-послідовності гри працюють разом, створюючи цілісний та захопливий досвід, який водночас кидає виклик і приносить задоволення. Завдяки своїй моторошній атмосфері, продуманому дизайну рівнів та інтуїтивно зрозумілому управлінню, Inside – це гра, яка триматиме гравців на краєчку свого крісла до самого кінця.

Остання ж комп’ютерна гра, яку я хотів би проаналізувати буде Super Meat Boy. Визнаний критиками інді-платформер, розроблений командою Team Meat [11]. Гра відома своєю складністю та жорстким управлінням. Гравець керує м’ясним хлопчиком, червоним персонажем у формі куба, який проходить через низку рівнів, заповнених перешкодами, щоб врятувати свою подругу, дівчину, від злого доктора Фетуса.

Рисунок 1.11 — Загальний вигляд платформеру “Super Meat Boy”

Ігровий процес Super Meat Boy характеризується платформером, що базується на точності. Кожен рівень – це набір смертельних небезпек, таких як пилки, що обертаються, ракети та бездонні ями, через які м’ясник повинен пройти, щоб дістатися до дівчини. Управління дуже просте – гравці можуть змусити м’ясного хлопчика бігти, стрибати і перестрибувати через стіни простими натисканнями кнопок. Однак оволодіння керуванням є важливим для подолання численних викликів у грі.

Однією з унікальних особливостей гри є здатність м’ясного хлопчика стрибати по стіні. Натиснувши кнопку стрибка, коли м’ясний хлопчик торкається стіни, гравці можуть змусити його зістрибнути з неї. Ця механіка є невід’ємною частиною навігації рівнями гри, оскільки багато з них вимагають від гравця поєднати кілька стрибків через стіну, щоб уникнути небезпеки або дістатися до платформ.

Рівні в Super Meat Boy розроблені так, щоб їх можна було пройти швидко, в середньому кожен рівень займає менше хвилини. Однак складність гри полягає в точності, необхідній для навігації по небезпеках кожного рівня. Гравці повинні

точно розраховувати час стрибків і перестрибувань через стіну, щоб уникнути перешкод і дістатися до дівчини. Гра винагороджує гравців, які проходять рівні найшвидше, а найшвидший час відображається в таблиці лідерів для кожного рівня.

Super Meat Boy також має унікальну механіку смерті. Щоразу, коли м'ясний Хлопчик помирає, що трапляється часто через високу складність гри, гра перегравє рівень з самого початку. Однак замість того, щоб починати з нуля, попередні спроби м'ясного хлопчика відображаються як примари, що дозволяє гравцям бачити свої попередні спроби і відповідно коригувати свою стратегію. Ця механіка додає до гри додатковий рівень стратегії, оскільки гравці можуть вчитися на своїх попередніх помилках і відповідно коригувати свій підхід.

Складність гри швидко зростає, з кожним новим світом з'являються нові небезпеки та перешкоди. Пізніші рівні гри відомі тим, що вимагають від гравців точних стрибків і стрибків через стіну у швидкій послідовності, щоб уникнути небезпек і дістатися до дівчини. Однак, завдяки чіткому управлінню та чуйній платформі, ці виклики здаються справедливими і приносять задоволення після їх подолання.

1.4 Висновок до розділу 1

У першому розділі бакалаврської дипломної роботи проведено аналіз предметної області ігрової індустрії, а саме детальний розбір різних ігрових жанрів та їх особливостей. Після аналізу різних ігрових рушіїв, зроблено висновок, що Unity є найкращим вибором для моїх потреб у розробці гри.

Надано опис жанру платформер та детально розглянуто його характерні риси. Було здійснено порівняння існуючих 2D-ігор у жанрі платформер на сьогоднішній день, таких як: Super Meat Boy, Inside, Dead Shovel Knight, виявлено їх ключові особливості.

2 РОЗРОБКА ТА ПРОЄКТУВАННЯ КОМП'ЮТЕРНОЇ ГРИ

2.1 Розробка архітектури комп'ютерної гри

Гнучкість Unity чудово підходить для створення комп'ютерних ігор, а модульний підхід – це мій вибір для того, щоб мій 2D-платформер був не лише організованим, але й легко розширювався та підтримувався в майбутньому [12]. Цей підхід дозволить мені розбити проект на окремі, незалежні компоненти, що значно спростить розробку, тестування та додавання нових функцій. Кожен модуль буде відповідати за свою частину гри, що зробить код більш читабельним та легшим для розуміння. Ось як я планую реалізувати це на практиці:

Основні модулі архітектури комп'ютерної гри:

1. Контролер персонажа: Це серце взаємодії гравця. Він буде обробляти:
    1. Рух: ходьба, біг, стрибки та спеціальні здібності (наприклад, стрибки по стінах або подвійні стрибки).
    2. Зіткнення: виявлення та реагування на платформи, ворогів та небезпеки.
    3. Фізика: застосування гравітації, імпульсу та інших сил, щоб рух відчувався природним.
    4. Анімація: координація спрайтів та анімацій відповідно до дій персонажа.
2. Дизайн рівня:
    1. Тайлмапи: я буду використовувати систему тайлмапів Unity для легкого створення рівнів. Це дозволить мені малювати платформи, фонові елементи та інтерактивні тайли.
    2. Бібліотека префабів: багаторазові префаби для загальних елементів (платформи, вороги, колекційні предмети тощо) прискорять розробку рівнів.
    3. Паралакс-скролінг: кілька фонових шарів, що рухаються з різною швидкістю, створять відчуття глибини та занурення.
3. Менеджер введення: спеціальний модуль для обробки введення гравця. Це

дозволить мені легко перепризначити елементи керування або підтримувати різні пристрої введення (контролери, клавіатури тощо).

4. Менеджер інтерфейсу користувача: обробляє всі елементи інтерфейсу користувача:
    1. HUD: відображення здоров’я, рахунку, часу тощо.
    2. Меню: головне меню, меню паузи, параметри тощо.
    3. Підказки в грі: підручники, повідомлення, діалогові вікна.
5. Менеджер звуку: централізоване управління звуковими ефектами та музикою.
    1. Джерела звуку: керування різними звуковими ефектами (стрибки, звуки ворогів, бонуси).
    2. Управління музикою: поступове згасання музики, перемикання треків на основі ігрових подій.

На рисунку 2.1 зображена архітектура гри.

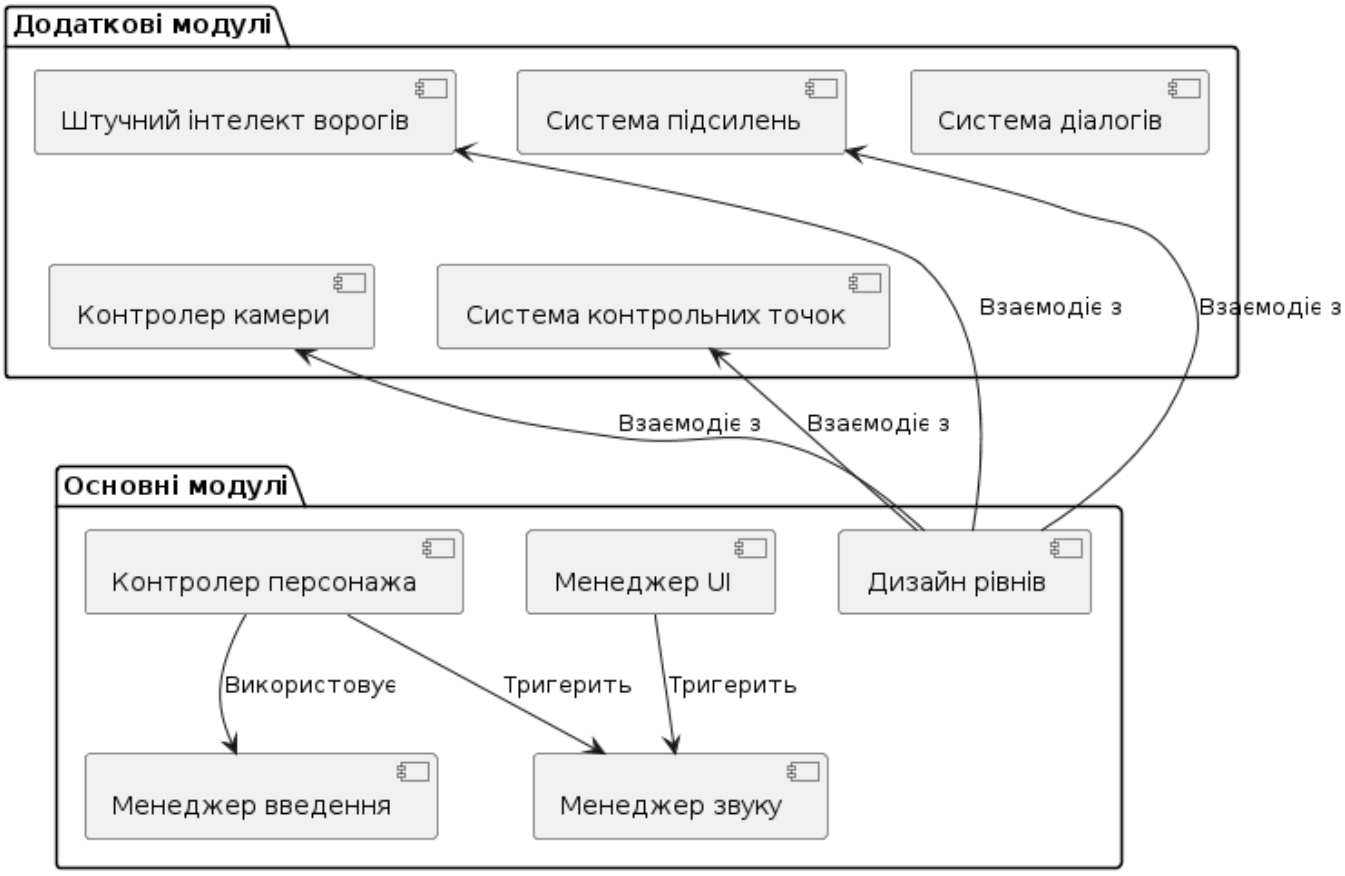

Рисунок 2.1 — Принципова схема архітектури комп’ютерної гри

Можливі додаткові модулі:

1. Штучний інтелект ворога: різна поведінка для різних типів ворогів (патрулювання, переслідування, атака).
2. Система бонусів: обробка ефектів та візуальних ефектів для бонусів.
3. Система контрольних точок: збереження прогресу гравця та обробка відродження.
4. Контролер камери: плавне стеження за гравцем та адаптація до різних ділянок рівня.
5. Система діалогів: якщо в моїй грі є історія, мені знадобиться спосіб відображення розмов.

Причини обрання модульної архітектури:

1. Організація: кожен модуль має свою власну відповідальність, що робить код чистішим і легшим для розуміння.
2. Повторне використання: модулі можна використовувати в інших проектах або легко замінити за потреби.
3. Командна робота: різні люди можуть працювати над різними модулями одночасно.
4. Масштабованість: легше додавати нові функції або розширювати існуючі, не порушуючи роботу всього іншого.

У майбутньому для оптимізації планується використовувати об’єднання спрайтів (Sprite Atlases) для зменшення кількості викликів до графічного процесора (draw calls), що значно покращить продуктивність гри. Використання атласів спрайтів є важливим для досягнення високої ефективності, оскільки це дозволяє зменшити кількість окремих текстур, які потрібно завантажувати та рендерити. Це сприяє більш плавному ігровому процесу та скорочує затримки під час гри.

Ще одним важливим аспектом є пулінг об’єктів (Object Pooling). Замість того, щоб постійно створювати та знищувати об’єкти, такі як кулі ворогів, можна використовувати пул об’єктів для повторного використання цих елементів. Це значно зменшує навантаження на систему та покращує загальну продуктивність. Пулінг об’єктів допомагає знизити кількість викликів до пам’яті та збільшує

ефективність управління ресурсами, що особливо важливо для мобільних платформ та ігор з високою динамікою.

Оптимізація колізій також є критично важливою. Для основних об’єктів можна використовувати прості колайдери, наприклад, прямокутники або кола, що зменшить обчислювальне навантаження. Складні колайдери, такі як полігони, варто використовувати лише для об’єктів з більш складними формами. Такий підхід дозволяє зменшити кількість розрахунків, необхідних для обробки колізій, і тим самим покращити продуктивність.

Кешування даних, тобто зберігання часто використовуваних даних у кеші, також відіграє важливу роль. Це може бути результати обчислень або дані рівня, що дозволяє швидше отримувати доступ до цих даних і зменшувати час очікування під час гри. Таким чином, гравці отримують більш плавний ігровий досвід без затримок.

Інструменти та розширення Unity надають додаткові можливості для розробників. Наприклад, Cinemachine використовується для створення кінематографічних камер та плавного стеження за персонажем, що покращує загальне сприйняття гри. ProBuilder дозволяє швидко створювати та редагувати 3D-моделі безпосередньо в Unity, що значно прискорює процес розробки. DOTween є інструментом для створення плавних анімацій та переходів, що додає динамічності ігровим сценам. Odin Inspector поліпшує візуальний редактор Unity і спрощує налаштування компонентів, роблячи роботу з ними більш інтуїтивно зрозумілою.

Додаткові можливості геймплею також вносять свій вклад у загальне враження від гри. Додавання секретних областей та проходів заохочує гравців до дослідження рівнів, надаючи додаткові виклики та винагороди. Різноманітні типи ворогів з унікальними здібностями та моделями поведінки додають різноманітності ігровому процесу, роблячи його більш цікавим та непередбачуваним. Боси, які представляють собою складні та видовищні бої, також підвищують рівень захоплення гравців. Система досягнень надає додаткові цілі та винагороди за виконання певних завдань, що стимулює гравців до тривалішого перебування у грі.

Для створення гарного сюжету та атмосфери планується розробка унікального світу, який включатиме цікавий сеттинг та захоплюючу історію, що привертатиме

увагу гравців. Візуальний стиль також є ключовим елементом, тому важливо обрати художній стиль, який підкреслить атмосферу гри та зробить її візуально привабливою. Крім того, оригінальна музика та звукові ефекти створюватимуть належний настрій ігрового світу та підсилюватимуть емоції гравців, роблячи ігровий процес більш захопливим та насиченим.

2.2 Розробка та взаємодія функціональних елементів комп'ютерної гри

Комп'ютерна гра складається з кількох ключових функціональних елементів, які взаємодіють між собою, забезпечуючи повноцінний ігровий процес. Ці елементи включають ігрову логіку, обробку вводу користувача, графічний рендеринг, звукове оформлення, фізичний двигун та систему ресурсів.

Ігрова логіка визначає правила гри та управляє станом всіх ігрових об'єктів. Вона включає механіки гри, такі як рухи персонажів, поведінку ворогів, зміни рівнів, взаємодію з об'єктами та іншими персонажами. Логіка гри також відповідає за обробку подій, наприклад, коли персонаж збирає предмет, отримує шкоду або досягає кінця рівня.

Обробка вводу користувача є критично важливою для інтерактивності гри. Вона включає захоплення сигналів від пристроїв вводу, таких як клавіатура, миша, геймпад або сенсорний екран. Ці сигнали перетворюються на команди, які гра розуміє та використовує для керування персонажами, навігації по меню, виконання дій тощо. Наприклад, натискання клавіші «вперед» може призвести до зміни позиції персонажа у відповідному напрямку.

Графічний рендеринг відповідає за візуалізацію всього, що відбувається у грі. Це включає малювання персонажів, об'єктів, фону, ефектів та інтерфейсу користувача. Рендеринг здійснюється з використанням графічного процесора (GPU), який виконує складні обчислення для відображення зображень на екрані з високою швидкістю. Рендеринг може включати 2D або 3D графіку, освітлення, тіні, текстури та інші візуальні ефекти.

Звукове оформлення додає ще один вимір до гри, забезпечуючи

аудіовізуальну взаємодію. Це включає музику, звукові ефекти, голосові команди та інші аудіоелементи. Звуковий двигун управляє відтворенням звуків відповідно до подій у грі. Наприклад, звук кроків персонажа змінюється залежно від поверхні, по якій він ходить.

Фізичний двигун відповідає за симуляцію фізичних законів у грі, таких як гравітація, зіткнення, тертя та інші. Він забезпечує реалістичну взаємодію між об’єктами. Наприклад, коли м’яч стрибає по поверхні, фізичний двигун обчислює траєкторію його руху та визначає, як він буде відскакувати від різних поверхонь.

Система ресурсів управляє завантаженням, зберіганням та звільненням ресурсів, таких як текстури, моделі, звуки та інші медіафайли. Вона забезпечує ефективне використання пам’яті та швидкий доступ до ресурсів, необхідних для гри.

На рисунку 2.2 зображено взаємодія елементів гри.

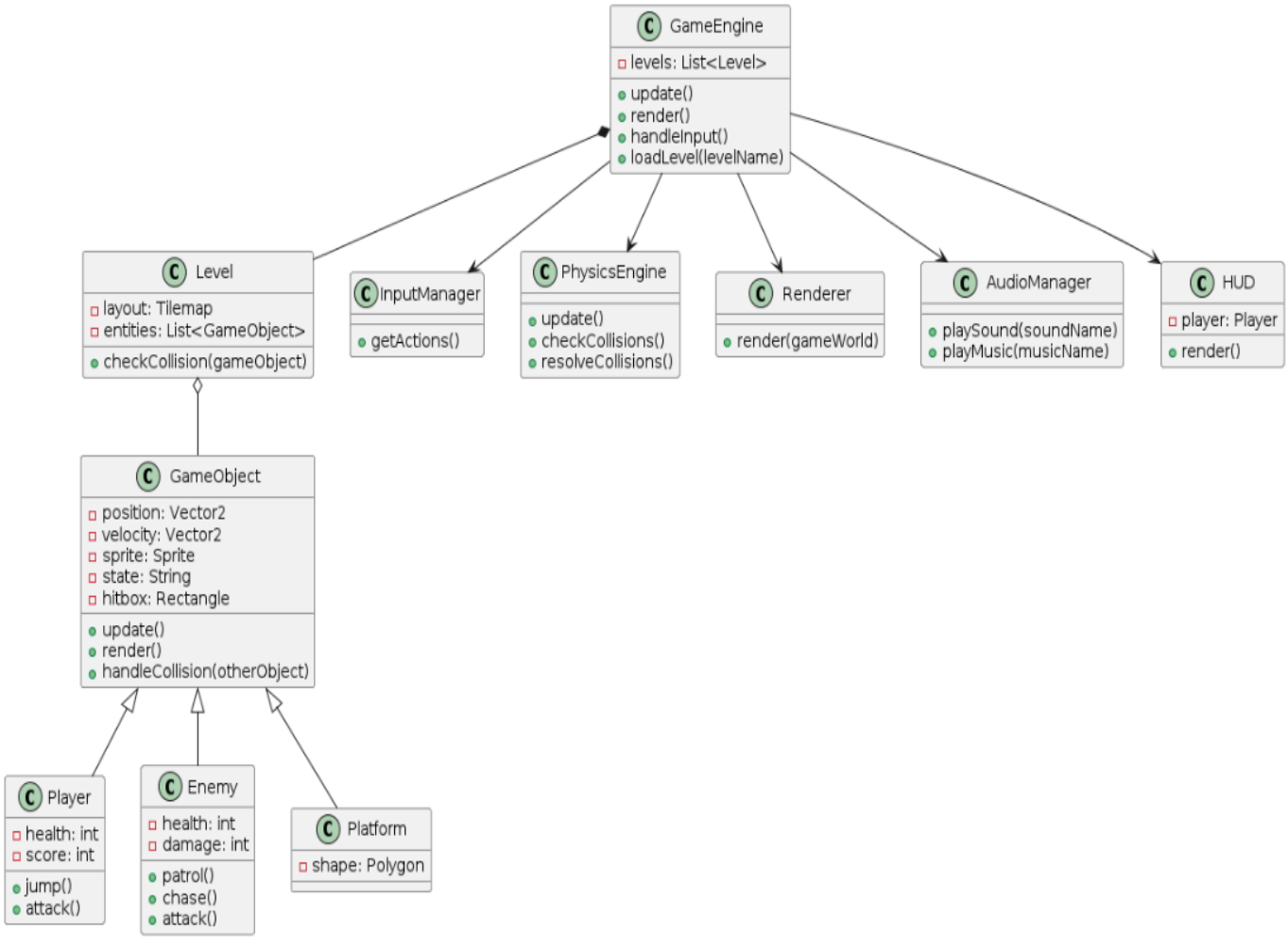

Рисунок 2.2 — Принципова схема взаємодії системних елементів комп’ютерної гри

Взаємодія між цими елементами відбувається через постійний обмін даними та виклики функцій. Наприклад, коли гравець натискає кнопку для стрибка, обробка вводу передає цю команду ігровій логіці. Логіка гри оновлює стан персонажа, а фізичний двигун обчислює траєкторію стрибка. Графічний двигун отримує нові координати персонажа і рендерить його нове положення на екрані, в той час як звуковий двигун відтворює звук стрибка. Всі ці процеси відбуваються синхронно і багаторазово впродовж кожної секунди гри, забезпечуючи плавний ігровий досвід.

GameEngine є центральним елементом управління всією грою. Він відповідає за ініціалізацію гри, завантаження рівнів, керування ігровим циклом (оновлення та рендеринг), обробку вхідних даних від гравця та координацію взаємодії між усіма іншими класами.

1. Змінні:
    1. levels: Список рівнів, доступних у грі.
2. Методи:
    1. update(): Оновлює стан усіх об’єктів на поточному рівні, включаючи гравця, ворогів, платформи та інші елементи. Цей метод викликається на кожному кроці ігрового циклу.
    2. render(): Відповідає за відображення оновленого ігрового світу на екрані. Він викликає метод render() у класі Renderer, передаючи йому необхідні дані для візуалізації.
    3. handleInput(): Отримує вхідні дані від гравця (наприклад, натискання клавіш, рухи миші) та перетворює їх на відповідні дії в грі. Ці дії потім передаються до класу Player.
    4. loadLevel (levelName): Завантажує новий рівень з файлу або іншого джерела даних.

Level представляє окремий рівень у грі. Він містить інформацію про розташування об’єктів на рівні, їх типи та інші дані, необхідні для правильного відображення та взаємодії об’єктів.

3. Змінні:
    1. layout: Використовує клас Tilemap для зберігання інформації про

розташування тайлів (наприклад, стіни, підлога) на рівні.

2. entities: Список усіх ігрових об’єктів (GameObject), присутніх на поточному рівні.

4. Методи:
    1. checkCollision (gameObject): Перевіряє, чи зіткнувся певний ігровий об’єкт (gameObject) з іншими об’єктами на рівні (наприклад, зі стінами, платформами або іншими персонажами).

GameObject слугує базовим класом для всіх об’єктів у грі, таких як гравець, вороги, платформи, бонуси тощо. Він визначає загальні властивості та поведінку, спільні для всіх ігрових об’єктів.

5. Змінні:
    1. position: Вектор, що визначає позицію об’єкта в ігровому світі (x, y координати).
    2. velocity: Вектор, що визначає швидкість та напрямок руху об’єкта.
    3. sprite: Графічне представлення об’єкта (зображення або анімація).
    4. state: Текстовий рядок, що описує поточний стан об’єкта (наприклад, «idle», «walking», «jumping»).
    5. hitbox: Прямокутник або інша геометрична фігура, що визначає межі об’єкта для перевірки зіткнень.
6. Методи:
    1. update(): Оновлює стан об’єкта на кожному кроці ігрового циклу. Це може включати оновлення позиції, швидкості, анімації, стану тощо.
    2. render(): Передає дані про об’єкт (позицію, спрайт) до класу Renderer для відображення на екрані.
    3. handleCollision(otherObject): Визначає, як об’єкт повинен реагувати на зіткнення з іншим об’єктом (otherObject).

Player, Enemy, Platform класи успадковують властивості та методи від GameObject та додають специфічну поведінку для кожного типу об’єкта. Змінні та методи: кожен з цих класів має свої унікальні змінні та методи, що визначають його особливості. Наприклад, Player може мати змінну health, методи jump() та attack(),

тоді як Enemy може мати змінну damage та методи patrol() та chase().

InputManager, PhysicsEngine, Renderer, AudioManager відповідають за окремі аспекти гри, такі як обробка вхідних даних, симуляція фізики, відображення графіки, відтворення звуку та відображення інформації для гравця.

Взаємодія між класами відбувається через виклики методів та передачу даних. Наприклад:

1. GameEngine викликає update() для всіх об’єктів на рівні, щоб оновити їх стан.
2. InputManager зчитує вхідні дані та передає їх до GameEngine, який потім оновлює стан Player.
3. PhysicsEngine перевіряє зіткнення між об’єктами та викликає handleCollision() у відповідних об’єктів.
4. GameEngine викликає render() у всіх об’єктах, а потім передає дані до Renderer для відображення на екрані.

2.3 Розробка алгоритму функціонування комп’ютерної гри

Алгоритм – це послідовність інструкцій, які виконуються для досягнення певної мети. У випадку комп’ютерних ігор, алгоритм функціонування описує послідовність операцій, які відбуваються під час гри від початку до завершення.

Коли гравець запускає гру, спочатку відбувається ініціалізація. Це може включати завантаження ресурсів, ініціалізацію графічного та аудіо-двигунів, створення об’єктів та налаштування гравців.

Після ініціалізації гра чекає на введення користувача. Коли користувач взаємодіє з грою, введені дані обробляються для визначення наступних дій. Наприклад, якщо гравець натискає кнопку для переміщення персонажа, гра реагує на цю дію.

Після обробки введення гра оновлює свій стан. Це означає зміну позицій об’єктів, виконання фізичних обчислень, взаємодію зі світом гри та інші дії, що впливають на гру. Після оновлення стану гри новий стан відображається на екрані. Це включає відображення графічних ефектів, анімації, тексту та іншого візуального

контенту.

Гра продовжує цикл обробки введення, оновлення стану та відображення, доки не буде виконана умова завершення гри. Це може бути досягнення певної мети, програш гравця або інша умова.

Після завершення гри може виводитися результат гри, такий як кількість очок, час гри тощо. Гравець може переглянути свій прогрес та порівняти його з іншими гравцям. Після завершення гри ресурси можуть бути звільнені, графічне вікно закрите, інші додаткові операції завершені, і гра припиняє свою роботу.

Загальна логіка роботи комп’ютерної гри «Beta», включаючи послідовність етапів, взаємозв’язки між компонентами та можливі шляхи розвитку ігрового процесу, детально зображена на схемі 2.3.

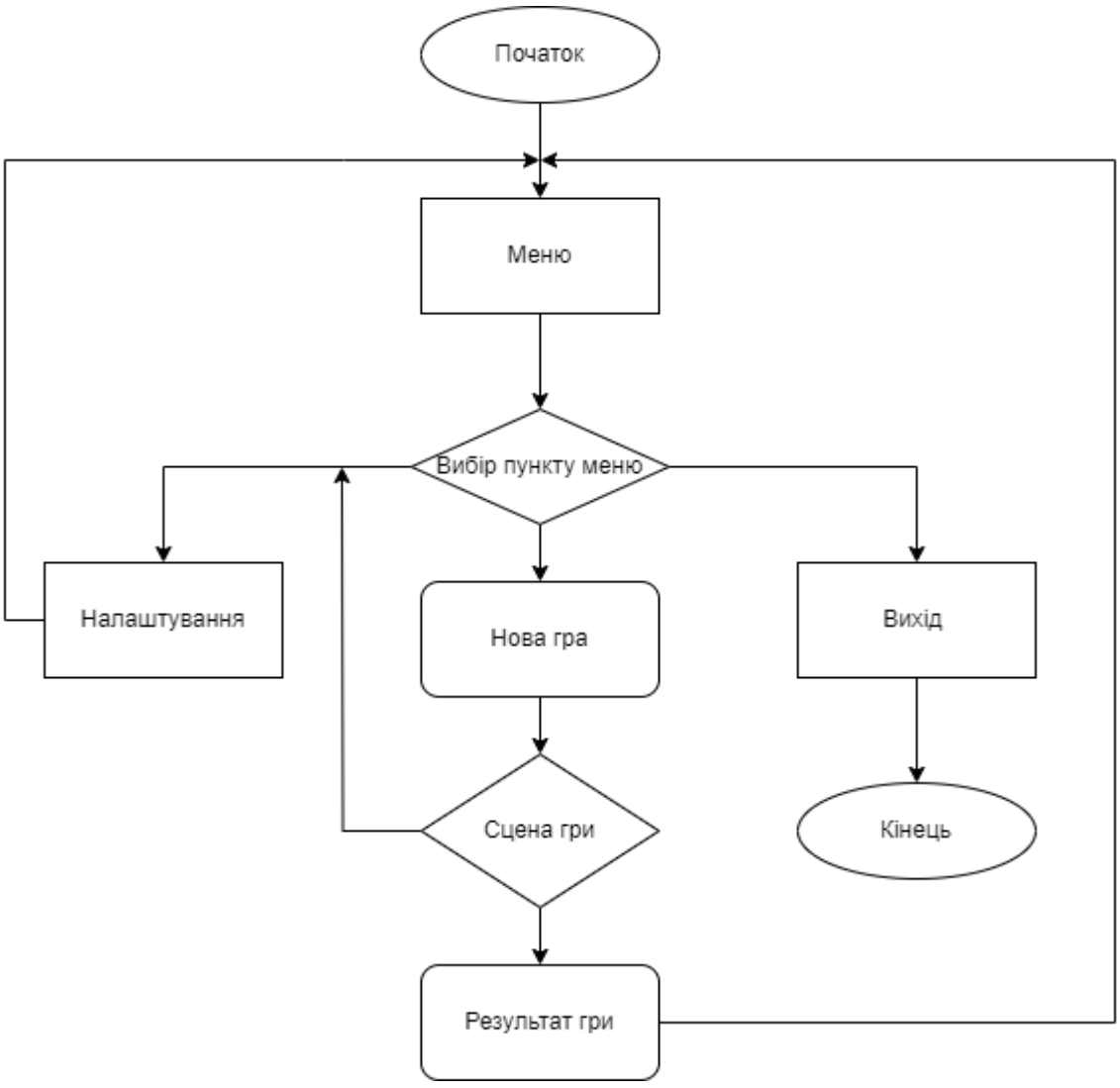

Рисунок 2.3 — Принципова схема алгоритму функціонування комп’ютерної гри

Особливості алгоритму функціонування гри включають в себе можливість взаємодії з користувачем, обробку великої кількості даних та подій в реальному часі, використання різних технологій (графіка, звук, штучний інтелект тощо) та забезпечення плавності гри та відчуття натуральності управління.

Алгоритм складається з кількох блоків і рішень, що представляють етапи від початку до кінця гри. Ось детальний опис кожного елементу:

1. Початок: гра починається з початкової точки, де запускається виконання програми.
2. Меню: після початку гри користувач потрапляє в меню, де він може вибрати одну з декількох опцій.
3. Вибір пункту меню: користувач повинен зробити вибір між трьома можливими опціями:
    1. “Налаштування» .
    2. “Нова гра”.
    3. “Вихід”.
4. Налаштування: якщо користувач вибирає опцію «Налаштування», він переходить до налаштувань гри, де може змінювати різні параметри гри. Після того, як користувач завершив зміну налаштувань, він повертається назад до меню.
5. Нова гра: якщо користувач вибирає опцію «Нова гра», починається нова гра. Користувач переходить до блоку сцени гри.
6. Сцена гри: у цьому блоці відбувається основний ігровий процес. Гравець проходить через різні етапи чи сцени гри.
7. Результат гри: після завершення сцени гри користувач бачить результат гри, наприклад, його рахунок або досягнення. Після перегляду результату гри користувач повертається до меню.
8. Вихід: якщо користувач вибирає опцію «Вихід», виконання програми завершується, і користувач переходить до кінцевої точки.
9. Кінець: це кінцева точка програми, де гра завершує своє виконання.

Загальний потік процесу:

1. Початок -> Меню.
2. Меню -> Вибір пункту меню.
3. Вибір пункту меню:
    1. Якщо вибрано «Налаштування» -> Налаштування -> Меню.
    2. Якщо вибрано «Нова гра» -> Нова гра -> Сцена гри -> Результат гри -> Меню.
    3. Якщо вибрано «Вихід» -> Вихід -> Кінець.

Діаграма станів гравця (або діаграма станів персонажа) є важливим інструментом у розробці відеоігор та інших інтерактивних систем, де є керований персонаж або об’єкт. Вона використовується для візуалізації та визначення різних станів, в яких може перебувати гравець, та переходів між цими станами.

Основні функції та переваги діаграми станів гравця:

1. Візуалізація логіки гри: діаграма станів дозволяє наочно представити складну логіку поведінки гравця. Це полегшує розуміння та аналіз того, як гравець взаємодіє з навколишнім світом гри.
2. Спрощення розробки та налагодження: завдяки діаграмі станів розробники можуть легше реалізувати логіку гри та виявляти помилки у поведінці гравця. Це прискорює процес розробки та підвищує якість кінцевого продукту.
3. Командна робота: діаграма станів є зручним інструментом для комунікації між членами команди розробників. Вона дозволяє всім учасникам мати однакове розуміння логіки гри та узгоджувати свої дії.
4. Можливість розширення та модифікації: діаграма станів може бути легко розширена або змінена у разі потреби. Це дозволяє додавати нові стани, переходи та функціональність без суттєвих змін у коді гри.

Загалом, діаграма станів гравця є незамінним інструментом для розробки складних та інтерактивних ігор. Вона допомагає структурувати логіку гри, спрощує процес розробки та забезпечує узгоджену роботу команди розробників.

Діаграма станів гравця описує різні стани, в яких може знаходитися гравець у грі, а також переходи між цими станами [13]. На рисунок 2.4 зображено докладний опис кожного стану та переходів між ними:

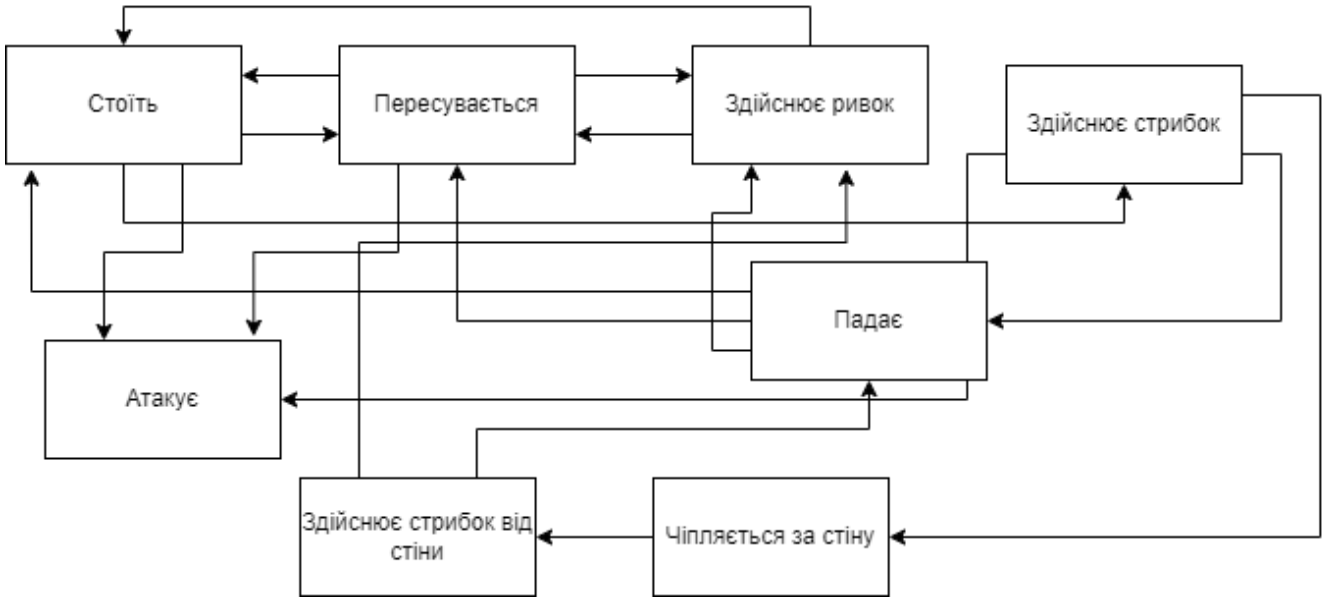

Рисунок 2.4 — Принципова схема діаграми станів гравця

1. Стоїть: гравець перебуває у стані спокою. Переходи:
    1. До стану «Пересувається» (коли гравець починає рухатися).
    2. До стану «Атакує» (коли гравець починає атаку).
    3. До стану «Здійснює стрибок» (коли гравець стрибає).
    4. До стану «Здійснює ривок» (коли гравець здійснює ривок).
    5. До стану «Падає» (коли гравець падає).
2. Пересувається: гравець рухається в будь-якому напрямку. Переходи:
    1. До стану «Стоїть» (коли гравець зупиняється).
    2. До стану «Атакує» (коли гравець починає атаку під час руху).
    3. До стану «Здійснює стрибок" (коли гравець стрибає під час руху).
    4. До стану "Здійснює ривок" (коли гравець здійснює ривок під час руху).
    5. До стану "Падає" (коли гравець падає під час руху).
3. Атакує: гравець виконує атаку. Переходи:
    1. До стану "Стоїть" (коли атака закінчується і гравець зупиняється).
    2. До стану "Пересувається" (коли атака закінчується і гравець продовжує рух).
    3. До стану "Падає" (коли гравець падає під час атаки).
4. Здійснює стрибок: гравець виконує стрибок. Переходи:
    1. До стану "Стоїть" (коли гравець приземляється і зупиняється).

2. До стану "Пересувається" (коли гравець приземляється і починає рух).
3. До стану "Падає" (коли гравець падає під час стрибка).
4. До стану "Чіпляється за стіну" (коли гравець стрибає на стіну).
5. До стану "Здійснює стрибок від стіни" (коли гравець стрибає від стіни).

5. Здійснює ривок: гравець виконує ривок. Переходи:
   1. До стану "Стоїть" (коли ривок закінчується і гравець зупиняється).
   2. До стану "Пересувається" (коли ривок закінчується і гравець продовжує рух).
   3. До стану "Падає" (коли гравець падає під час ривка).
6. Падає: гравець падає вниз. Переходи:
   1. До стану "Стоїть" (коли гравець приземляється і зупиняється).
   2. До стану "Пересувається" (коли гравець приземляється і починає рух).
   3. До стану "Здійснює стрибок" (коли гравець стрибає під час падіння).
7. Чіпляється за стіну: гравець чіпляється за стіну. Переходи:
   1. До стану "Здійснює стрибок від стіни" (коли гравець стрибає від стіни).
   2. До стану "Падає" (коли гравець втрачає хватку і падає).
8. Здійснює стрибок від стіни: гравець виконує стрибок від стіни. Переходи:
   1. До стану "Стоїть" (коли гравець приземляється і зупиняється).
   2. До стану "Пересувається" (коли гравець приземляється і починає рух).
   3. До стану "Падає" (коли гравець падає під час стрибка від стіни).
   4. До стану "Чіпляється за стіну" (коли гравець знову чіпляється за стіну після стрибка).

Ця діаграма допомагає зрозуміти, як різні стани гравця взаємопов'язані і які умови призводять до переходів між цими станами. Вона є основою для програмування логіки керування персонажем у грі.

Діаграма станів гравця ілюструє введені можливості та обмеження. Вона також сприятиме розробці дерева станів для анімацій у ігровому рушії.

2.4 Проектування ігрового механізму комп’ютерної гри

Game loop (ігровий цикл) є основною структурою програмного коду, яка керує

процесом виконання комп'ютерних ігор. Це центральний механізм, що забезпечує постійне оновлення стану гри та відображення змін на екрані, створюючи ілюзію безперервного ігрового процесу.

У своїй суті, game loop відповідає за постійну перевірку і обробку користувацького вводу, оновлення внутрішньої логіки гри та відображення результатів на екрані. Це означає, що незалежно від того, що робить гравець (рухає персонажа, стріляє, взаємодіє з об'єктами тощо), game loop забезпечує, щоб всі ці дії були своєчасно оброблені та правильно відображені.

Основна структура game loop зазвичай включає такі компоненти:

1. Ініціалізація: на початку роботи гри виконується підготовка всіх необхідних ресурсів і налаштувань. Це включає завантаження текстур, звуків, початкових параметрів гри тощо.
2. Вхідний цикл: основна частина циклу, яка повторюється постійно під час роботи гри. Цей цикл включає:
    1. Обробку вводу: захоплення та обробка вводу від користувача (клавіатура, миша, геймпад тощо).
    2. Оновлення: зміна стану гри на основі обробленого вводу та внутрішньої логіки. Це може включати рухи персонажів, анімації, фізичні взаємодії та інші аспекти ігрового процесу.
    3. Візуалізація: очищення екрану та відображення оновленого стану гри.
3. Завершення: коли гравець виходить з гри або гра закінчується з інших причин, виконується очищення пам'яті та закриття всіх ресурсів, що були використані під час гри.

Цикл повторюється на кожному кадрі (frame), і зазвичай у сучасних іграх він виконується з частотою 30-60 разів на секунду (30-60 FPS), що забезпечує плавність і реалістичність рухів та анімацій.

Ігровий цикл також може включати додаткові механізми для управління часом, такі як обмеження частоти кадрів або адаптивне регулювання швидкості виконання логіки гри для підтримання стабільної продуктивності незалежно від потужності апаратного забезпечення.

Таким чином, game loop є ключовим елементом, що дозволяє іграм реагувати на дії гравців і динамічно змінюватися, забезпечуючи інтерактивний та захоплюючий ігровий процес.

На рисунку 2.5 показана діаграма, яка представляє собою цикл гри для 2D платформера і складається з кількох етапів.

Рисунок 2.5 – Принципова схема алгоритму ігрового циклу комп’ютерної гри

Починається все з ініціалізації гри, де налаштовуються початкові параметри, встановлюються значення змінних, готуються ресурси до завантаження, налаштовуються гравці, ігрові об'єкти та інші необхідні компоненти. Після цього відбувається завантаження ресурсів: текстур, спрайтів, звуків, музики, шрифтів та інших медіа файлів, які зберігаються в пам'яті для подальшого використання в процесі гри.

Далі йде налаштування сцени, що включає розміщення ігрових об'єктів, налаштування рівнів, розташування ворогів, платформ та інших елементів, які будуть використовуватися в грі. Після цього починається основний цикл гри, який повторюється доти, доки гра активна.

Перший крок у цьому циклі - обробка вводу користувача. Це включає натискання клавіш, рух миші або інші дії, які змінюють параметри гри, такі як рух гравця або взаємодія з об'єктами. Далі відбувається оновлення логіки гри, де змінюються стани ігрових об'єктів, оновлюються лічильники часу, керуються анімації та виконуються інші логічні операції.

На наступному етапі відбувається оновлення положення об'єктів, де обчислюються нові позиції для ігрових об'єктів на основі їх швидкостей, напрямків руху та інших параметрів. Після цього перевіряються колізії, тобто взаємодії між об'єктами. Якщо два об'єкти стикаються, в залежності від їх типів можуть виконуватися різні дії, такі як відскакування, отримання шкоди або збирання предметів.

Потім оновлюється стан камери, щоб вона стежила за гравцем або іншими важливими об'єктами, забезпечуючи правильне відображення сцени на екрані. Далі йде очищення екрану, під час якого екран очищується для підготовки до малювання нового кадру, щоб запобігти залишкам зображень від попереднього кадру.

Після цього відбувається малювання сцени, де всі ігрові об'єкти та елементи інтерфейсу малюються на екран у правильному порядку. Це включає фони, персонажів, ворогів, предмети та інші графічні елементи. Останній крок в циклі - оновлення екрану для відображення всіх змін, що забезпечує плавність анімацій та реакції гри на дії користувача.

Коли гра завершується, завантажені ресурси вивантажуються з пам'яті. Це включає видалення текстур, звуків та інших файлів, які більше не потрібні. Нарешті, відбувається остаточне завершення гри, включаючи виведення повідомлень користувачу, збереження прогресу, статистики або інших даних.

2.5 Висновок до розділу 2

У даному розділі були розглянуті ключові аспекти розробки гри, включаючи архітектуру гри, розробку функціональних елементів, створення алгоритмів та тестування.

Розроблено модульну архітектуру, яка включає основні компоненти гри: контролер персонажа, ігровий рушій, менеджер вхідних даних, фізичний рушій, рендерер та аудіо менеджер. Така архітектура забезпечує гнучкість, зручність у розробці та легкість у підтримці та розширенні гри в майбутньому.

У рамках розробки функціональних елементів гри були детально розроблені класи для кожного типу ігрових об'єктів, включаючи гравця, ворогів та платформи. Реалізовано основні методи та змінні для управління ігровими об'єктами, такі як рух, зіткнення, відображення та обробка вхідних даних.

Було створено алгоритм, який включає послідовність дій від завантаження гри до взаємодії гравця з ігровим світом. Також була представлена діаграма станів гравця, яка візуально демонструє всі можливі стани персонажа та переходи між ними, що дозволяє спростити розробку та тестування гри.

Таким чином, успішно реалізовано ключові етапи розробки гри "Beta", що включають проєктування архітектури, розробку функціональних елементів, створення алгоритмів. Це закладає міцну основу для подальшої розробки та вдосконалення гри, що сприятиме створенню якісного ігрового продукту, який буде цікавим та захоплюючим для гравців.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ МОДУЛЯ КОМП'ЮТЕРНОЇ ГРИ

3.1 Обґрунтування вибору мови програмування та середовища розробки

Для розробки сценарію гри важливо вибрати мову програмування, яка буде використовуватися для написання коду. Кожна мова програмування має свої особливості та переваги для різних завдань. У професійній розробці ігор часто застосовуються кілька мов програмування для досягнення оптимальних результатів. Найпоширенішими мовами для розробки ігор є C# і Java, які забезпечують високу продуктивність і збалансовані системні вимоги.

Крім цих мов, для створення ігор можуть бути використані й інші мови програмування, такі як Python і PHP. Вони часто застосовуються для реалізації окремих компонентів ігрового коду.

C++ є однією з найпотужніших мов програмування, яка підтримує об'єктно-орієнтоване, процедурне та узагальнене програмування. Це означає, що розробники можуть використовувати різні підходи та методики для створення своїх відеоігор, та знайти найкращий підхід для конкретного проекту. Крім того, C++ має велику стандартну бібліотеку, яка включає функції для введення-виведення, багатопоточності та контейнерів.

C++ широко використовується на різних платформах, включаючи ПК, консолі, мобільні пристрої та інші пристрої. Це робить його відмінним вибором для розробників, які хочуть створювати відеоігри для багатьох різних платформ. Крім того, C++ добре поєднується з іншими інструментами для розробки відеоігор, такими як ігрові рушії, графічні бібліотеки та інші програмні інструменти.

C# є популярною мовою програмування для розробки відеоігор, особливо для використання з ігровим рушієм Unity. Ця мова є повністю об'єктно-орієнтованою, що дозволяє розробникам створювати та використовувати об'єкти та класи для створення своїх відеоігор. Крім того, C# має простий та зрозумілий синтаксис, що робить його відмінним вибором для нових розробників, які тільки починають свою кар'єру в розробці відеоігор.

C# широко використовується на платформі .NET Framework, що дозволяє розробникам створювати відеоігри для ПК. Крім того, C# добре інтегрується з іншими інструментами для розробки відеоігор, такими як ігрові рушії, графічні бібліотеки та інші програмні інструменти. Це дозволяє розробникам використовувати найкращі інструменти для конкретного проекту, та створювати відеоігри швидше та ефективніше.

Java є популярною мовою програмування для розробки відеоігор, особливо для мобільних пристроїв, таких як смартфони та планшети. Ця мова є багатопоточною, що дозволяє розробникам створювати відеоігри, які можуть виконувати багато завдань одночасно, та ефективно використовувати ресурси мобільних пристроїв. Крім того, Java має вбудовані функції для роботи з пам'яттю, що дозволяє розробникам створювати відеоігри, які не вимагають великого об'єму пам'яті для свого виконання.

Java також підтримує взаємодію з графічними та звуковими рушіями, що робить її відмінним вибором для створення повноцінних ігрових додатків. Це дозволяє розробникам використовувати найкращі інструменти для створення графіки та звукового супроводу, та створювати відеоігри з високою якістю графіки та звуку.

Сьогодні існує безліч середовищ для розробки 2D ігор та мов програмування, які можна використовувати для написання сценаріїв. Unity є однією з найпоширеніших платформ для розробки 2D та 3D ігор. Її інтуїтивно зрозумілий інтерфейс і широкий функціонал роблять Unity оптимальним вибором для створення ігор будь-якої складності. Оскільки Unity добре співпрацює з мовою програмування C#, сценарій гри буде реалізовано на цій мові.

Для написання коду потрібно мати середовище розробки. IDE (Integrated Development Environment) – це програмне забезпечення, що забезпечує всебічну підтримку для розробки програмного забезпечення. IDE зазвичай включає в себе редактор коду, компілятор або інтерпретатор, засоби для налагодження, а також часто інші інструменти для полегшення процесу розробки, такі як управління версіями та інтеграція з системами збірки.

Коли йдеться про розробку на C#, три основні IDE заслуговують на особливу увагу: Visual Studio, JetBrains Rider та Visual Studio Code.

Visual Studio – це потужна IDE від Microsoft, яка забезпечує всеосяжну підтримку для розробки на C#. Вона пропонує інтуїтивно зрозумілий інтерфейс, гнучкі інструменти для налагодження, інтеграцію з системами контролю версій, такими як Git, а також великий набір розширень, які дозволяють розширити функціональність IDE відповідно до потреб розробника. Visual Studio підтримує не тільки C#, але й багато інших мов програмування, що робить її універсальним інструментом для розробників.

JetBrains Rider – це ще одна популярна IDE для C#, яка вирізняється своєю швидкістю та зручністю. Rider базується на IntelliJ IDEA і забезпечує високу продуктивність завдяки інтелектуальному аналізу коду та потужним засобам для налагодження. Вона добре інтегрується з платформою .NET і підтримує різні проекти, такі як ASP.NET, Unity, Xamarin та інші. Rider також відома своєю кросплатформеністю, оскільки працює на Windows, macOS та Linux.

Visual Studio Code – це легка, але потужна редакторка коду від Microsoft, яка завоювала популярність серед розробників завдяки своїй гнучкості та багатству розширень. Хоча VS Code не є повноцінною IDE в традиційному розумінні, вона пропонує безліч розширень, що дозволяють додати підтримку для C#. Завдяки своїй легкості та швидкості, VS Code ідеально підходить для розробників, які шукають швидке та налаштовуване середовище для написання коду.

JetBrains Rider буде оптимальним вибором. Вона поєднує в собі високу продуктивність, зручність використання та широкі можливості для інтеграції з різними платформами і проектами. Rider забезпечує швидкий і інтелектуальний досвід розробки, що дозволяє максимально ефективно використовувати ваш час і ресурси.

3.2 Програмна реалізація модулю комп’ютерної гри

Для роботи даного програмного модуля слід розробити декілька скриптів, а

саме: Events, ExitTrigger, Game Manager, Heath Manager, pickup, PlayerController, UIManager.

Events клас містить три публічні методи: Menu(), Level(), та Quit(). Ці методи використовуються для обробки різних подій в грі, таких як завантаження різних сцен або виходу з додатку:

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

public class Events : MonoBehaviour
{
    public void Menu()
    {
        SceneManager.LoadScene(0);
    }
    public void Level()
    {
        SceneManager.LoadScene(1);
    }
    public void Quit()
    {
        Application.Quit();
    }
};
```

Клас ExitTrigger - скрипт додається до ігрового об'єкту у сцені Unity, який призначений для обробки функції виходу з рівня:

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

```
using UnityEngine.SceneManagement;

public class ExitTrigger : MonoBehaviour
{
    //public Animator anim;
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.gameObject.tag == "Player")
        {
            StartCoroutine("LevelExit");
        }
    }

    IEnumerator LevelExit()
    {
        //anim.SetTrigger("Exit");
        yield return new WaitForSeconds(0.1f);

        UIManager.instance.fadeToBlack = true;

        yield return new WaitForSeconds(2f);
        // Do something after flag anim
        GameManager.instance.LevelComplete();
    }
}.
```

Клас PlayerController –відповідає за керування рухом, стрибками та стрільбою гравця у 2D-грі:

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

```
using UnityEngine.UI;
public enum Controls { mobile,pc}

public class PlayerController : MonoBehaviour
{
    public float moveSpeed = 5f;
    public float jumpForce = 10f;
    public float doubleJumpForce = 8f;
    public LayerMask groundLayer;
    public Transform groundCheck;
    private Rigidbody2D rb;
    private bool isGroundedBool = false;
    private bool canDoubleJump = false;
    public Animator playeranim;
    public Controls controlmode;
    private float moveX;
    public bool isPaused = false;

    public ParticleSystem footsteps;
    private ParticleSystem.EmissionModule footEmissions;
    public ParticleSystem ImpactEffect;
    private bool wasonGround;

   // public GameObject projectile;
   // public Transform firePoint;

    public float fireRate = 0.5f; // Time between each shot
    private float nextFireTime = 0f; // Time of the next allowed shot

    private void Start()
```

```
{
    rb = GetComponent<Rigidbody2D>();
    footEmissions = footsteps.emission;

    if (controlmode == Controls.mobile)
    {
        UIManager.instance.EnableMobileControls();
    }
}
private void Update()
{
    isGroundedBool = IsGrounded();

    if (isGroundedBool)
    {
        canDoubleJump = true; // Reset double jump when grounded

        if (controlmode == Controls.pc)
        {
            moveX = Input.GetAxis("Horizontal");
        }
        if (Input.GetButtonDown("Jump"))
        {
            Jump(jumpForce);
        }
    }

    else
    {
        if (canDoubleJump && Input.GetButtonDown("Jump"))
```

```
            {
                Jump(doubleJumpForce);
                canDoubleJump = false; // Disable double jump until grounded again
            }
        }

        if (!isPaused)
        {
            // Calculate rotation angle based on mouse position
            Vector3 mousePosition = Camera.main.ScreenToWorldPoint(Input.mousePosition);
            Vector3 lookDirection = mousePosition - transform.position;
            float angle = Mathf.Atan2(lookDirection.y, lookDirection.x) * Mathf.Rad2Deg;
        }.
```

UIManager керує користувацьким інтерфейсом гри. Він відповідає за відображення та оновлення елементів UI, таких як здоров'я гравця, набрані очки, таймери, інвентар та інші елементи. UIManager також може обробляти користувацькі введення та взаємодії з інтерфейсом:

```
using UnityEngine;
using UnityEngine.UI;

public class UIManager : MonoBehaviour
{
    public static UIManager instance;
    public GameObject mobileControls;
    public bool fadeToBlack, fadeFromBlack;
    public Image blackScreen;
    public float fadeSpeed = 2f;
    //player reference
```

```
public PlayerController playerController;
private void Awake()
{
    instance = this;
}
public void DisableMobileControls()
{
    mobileControls.SetActive(false);
}
public void EnableMobileControls()
{
    mobileControls.SetActive(true);
}
private void Update()
{
    UpdateFade();
}
private void UpdateFade()
{
    if (fadeToBlack)
    {
        FadeToBlack();
    }
    else if (fadeFromBlack)
    {
        FadeFromBlack();
    }
}
private void FadeToBlack()
{
```

```
        FadeScreen(1f);
        if (blackScreen.color.a >= 1f)
        {
            fadeToBlack = false;
        }
    }
    private void FadeFromBlack()
    {
        FadeScreen(0f);
        if (blackScreen.color.a <= 0f)
        {
            if (playerController.controlmode == Controls.mobile)
            {
                EnableMobileControls();
            }
            fadeFromBlack = false;
        }
    }
    private void FadeScreen(float targetAlpha)
    {
        Color currentColor = blackScreen.color;
        float newAlpha = Mathf.MoveTowards(currentColor.a, targetAlpha, fadeSpeed
* Time.deltaTime);
        blackScreen.color = new Color(currentColor.r, currentColor.g, currentColor.b,
newAlpha);
    }
}.
```

Game Manager відповідає за загальне керування грою, включаючи управління станами гри, збереження прогресу, перехід між рівнями та загальну логіку гри. Він часто виступає центральним контролером, до якого звертаються інші скрипти для

виконання різних завдань.

Health Manager займається управлінням здоров'ям гравця або інших об'єктів у грі. Він відстежує поточний рівень здоров'я, обробляє пошкодження та відновлення здоров'я, а також реагує на зміни стану здоров'я, наприклад, коли здоров'я досягає нуля.

Pickup відповідає за логіку взаємодії з об'єктами, які можна підібрати в грі. Це можуть бути предмети, які додають здоров'я, боєприпаси, бонуси або інші ресурси. Скрипт обробляє події підбору, додає відповідні бонуси гравцеві і може відображати ефекти або повідомлення про підбір.

3.3 Тестування програмного модулю комп’ютерної гри

При тестуванні програмного модуля комп'ютерної гри необхідно перевірити правильність введення, збереження та відображення даних, а також взаємодію з іншими компонентами системи. Зокрема, варто перевірити коректність роботи процесу старту і завершення гри, оновлення та відображення рахунку, а також роботу кожного з ігрових блоків.

Тестування почнемо з того що відкриємо Unity та запустимо проект (рис 3.1).

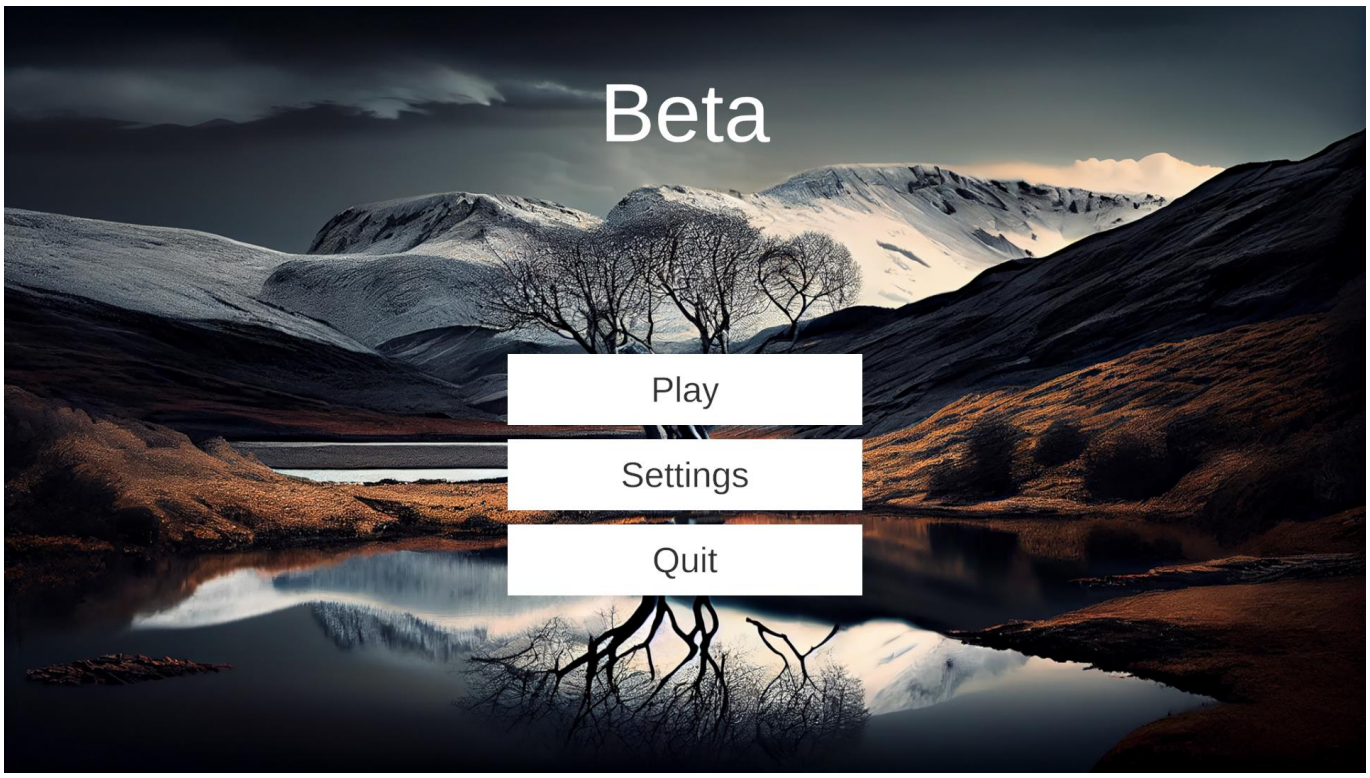

Рисунок 3.1 – Загальний вигляд інтерфейсного вікна типу «Головне меню» комп’ютерної гри

Обираємо “PLAY” та можемо переходити до процесу гри (рис. 3.2).

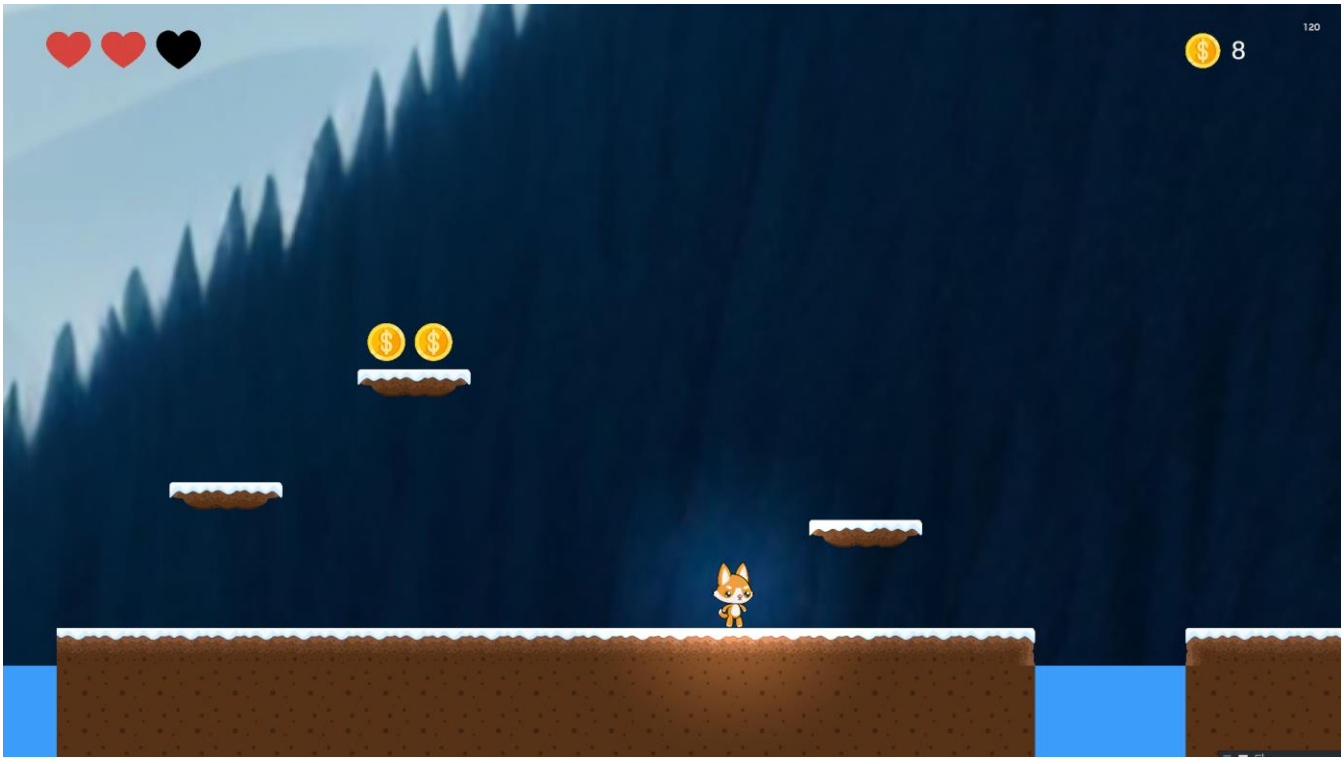

Рисунок 3.2 – Загальний вигляд геймплею комп’ютерної гри

Метою гри є дійти до кінця рівня та назбирати найбільше монет. Підрахунок монет ведеться з правого кутка. У випадку смерті гравець повертається до початкової точки.

Після завершення гри є можливість переглянути кількість зібраних монет та повернутись у меню або вийти з гри (рис. 3.3).

Рисунок 3.3 – Загальний вигляд завершення комп’ютерної гри

3.4 Висновок до розділу 3

У третьому розділі було проведено аналіз і обґрунтування вибору засобів для реалізації програмного засобу. Мовою програмування обрано C# та рушій Unity в якості набору інструментів для розробки ігор, IDE – Rider.

Для зручного керування створені такі класи: Events, ExitTrigger, Game Manager, Heath Manager, Pickup, PlayerController, UIManager. Пояснено за що відповідають дані класи.

Протестовано ігровий процес гри, щоб перевірити її функціональність, стабільність і загальну якість. Тести охопили різні аспекти геймплею, включаючи механіку, графіку, звук та взаємодію з користувачем.

ВИСНОВКИ

Поставлені перед бакалаврською дипломною роботою задачі виконано в повному обсязі, а саме:

– проаналізовано жанри комп’ютерних ігор та відомі комп’ютерні ігри відповідного жанру;

– виконано порівняльний аналіз програмних рушіїв комп’ютерних ігор;

– спроектовано програмний модуль комп’ютерної гри;

– розроблено програмний модуль комп’ютерної гри з урахуванням вимог до функціональності та продуктивності;

– протестовано та оптимізовано програмний модуль комп’ютерної гри для забезпечення стабільної та ефективної роботи;

– розроблено інструкцію користувача.

У ході виконання бакалаврської роботи були досліджені різноманітні жанри комп’ютерних ігор та виявлено їх особливості. Виконано порівняльний аналіз програмних рушіїв комп'ютерних ігор, що дозволило вибрати найбільш оптимальний варіант для реалізації проекту.

Розроблено структуру програмного модуля та UML-діаграму. Створений алгоритм функціонування програмного модуля, який гарантує послідовність логічних подій у грі та правильну роботу всіх її компонентів. Спроектовано ігровий механізм комп'ютерної гри, який враховує різноманітні аспекти ігрового дизайну.

Виконана програмна реалізація програмного модуля за допомогою обраних раніше програмних засобів. Після розробки модуля його функціональність перевірена шляхом тестування, щоб забезпечити стабільну та ефективну роботу гри. Була розроблена інструкція користувача, яка надає докладні пояснення щодо управління та функцій гри.

Мета дослідження - реалізація дозвілля людини на основі розробленої комп'ютерної гри «Beta», яка дозволяє привести в норму емоційний та психічний стан людини, а також покращити її розумові здібності. Мета була досягнуто за рахунок розширення та удосконалення існуючих рішень.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Statista [Електронний ресурс] Режим доступу до ресурсу: https://www.statista.com/statistics/292056/video-game-market-value-worldwide/.
2. Користь комп'ютерних ігор у навчанні [Електронний ресурс] Режим доступу до ресурсу: https://direct.mit.edu/books/monograph/4028/Computer-Games-for-LearningAn-Evidence-Based.
3. Шафір Є. М., Іванчук Я. В. «Сучасні інструменти розробки комп'ютерних ігор»: матеріали конференції «LI Науково-технічна конференція підрозділів Вінницького національного технічного університету (2024)», Вінниця, 2024. [Електронний ресурс] – Режим доступу до ресурсу: https://conferences.vntu.edu.ua/index.php/all-fksa/all-fksa-2024/paper/view/20927.
4. Жанри комп'ютерних ігор [Електронний ресурс] Режим доступу до ресурсу: https://www.hp.com/us-en/shop/tech-takes/video-game-genres.
5. Ігри у жанрі екшену [Електронний ресурс] Режим доступу до ресурсу: https://www.computerhope.com/jargon/a/action-game.htm.
6. Ігри у жанрі пригоди [Електронний ресурс] Режим доступу до ресурсу: https://adventuregamers.com/articles/view/17547.
7. Рольові ігри [Електронний ресурс] Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Role-playing_game.
8. Ігри у жанрі стратегії [Електронний ресурс] Режим доступу до ресурсу: https://www.kathleenmercury.com/what-is-a-strategy-game.html.
9. Ігри у жанрі головоломка [Електронний ресурс] Режим доступу до ресурсу: https://www.linkedin.com/pulse/what-puzzle-games-ten-pixel-studio/.
10. Unreal Engine [Електронний ресурс] Режим доступу до ресурсу: https://www.unrealengine.com.
11. Unity [Електронний ресурс] Режим доступу до ресурсу: https://unity.com.
12. 2D платформери [Електронний ресурс] Режим доступу до ресурсу: https://gamicus.fandom.com/wiki/2D_platform_video_games.
13. Shovel Knight [Електронний ресурс] Режим доступу до ресурсу:

https://store.steampowered.com/app/250760/Shovel_Knight_Treasure_Trove.

14. Inside [Електронний ресурс] Режим доступу до ресурсу: https://www.theguardian.com/technology/2016/jul/05/inside-review-limbo-developer-playdead-xbox-pc-platform-puzzle.
15. Super meat boy [Електронний ресурс] Режим доступу до ресурсу: https://www.gameinformer.com/review/super-meat-boy-forever/super-meat-boy-forever-review-hardcore-hurdling.
16. Модульний дизайн [Електронний ресурс] Режим доступу до ресурсу: https://ebrahim-karimi.medium.com/unity3d-modular-development-principle-umdp-revolutionizing-collaborative-game-development-0de10e2c4652.
17. Діграма станів [Електронний ресурс] Режим доступу до ресурсу: https://www.maxzosim.com/state-modelling/.

ДОДАТОК А

ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ НА НАЯВНІСТЬ ТЕКСТОВИХ ЗАПОЗИЧЕНЬ

Назва роботи: Програмний модуль комп'ютерної гри "Beta"

Тип роботи: бакалаврська дипломна робота
(БДР, МКР)

Підрозділ кафедра комп'ютерних наук, ФІТА
(кафедра, факультет)

Показники звіту подібності Unicheck

Оригінальність 97,06% Схожість 2,94%

Аналіз звіту подібності (відмітити потрібне):

- ✓ Запозичення, виявлення у роботі, оформлені коректно і не містять ознак плагіату.
- □ Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її виконання автором. Роботу направити на розгляд експертної комісії кафедри.
- □ Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.

Особа, відповідальна за перевірку Озеранський В.С.

Ознайомлені з повним звітом подібності, який був згенерований системою Unicheck щодо роботи.

Автор роботи Шафір Є.М.

Керівник роботи Іванчук Я.В.

ДОДАТОК Б

Інструкція користувача

1. Відкриття Unity:
    1. Відкрийте Unity на вашому комп'ютері.
    2. Завантажте проект гри, якщо він ще не завантажений.
2. Збірка проекту (Build):
    1. Перейдіть у меню File та виберіть Build Settings
    2. Виберіть платформу (наприклад, PC, Mac & Linux) та натисніть Switch Platform.
    3. Натисніть Build і виберіть папку для збереження збірки.
    4. Дочекайтеся завершення процесу збірки.
3. Запуск зібраного проекту:
    1. Перейдіть до папки, де ви зберегли збірку проекту.
    2. Запустіть виконуваний файл гри (наприклад, GameName.exe для Windows).

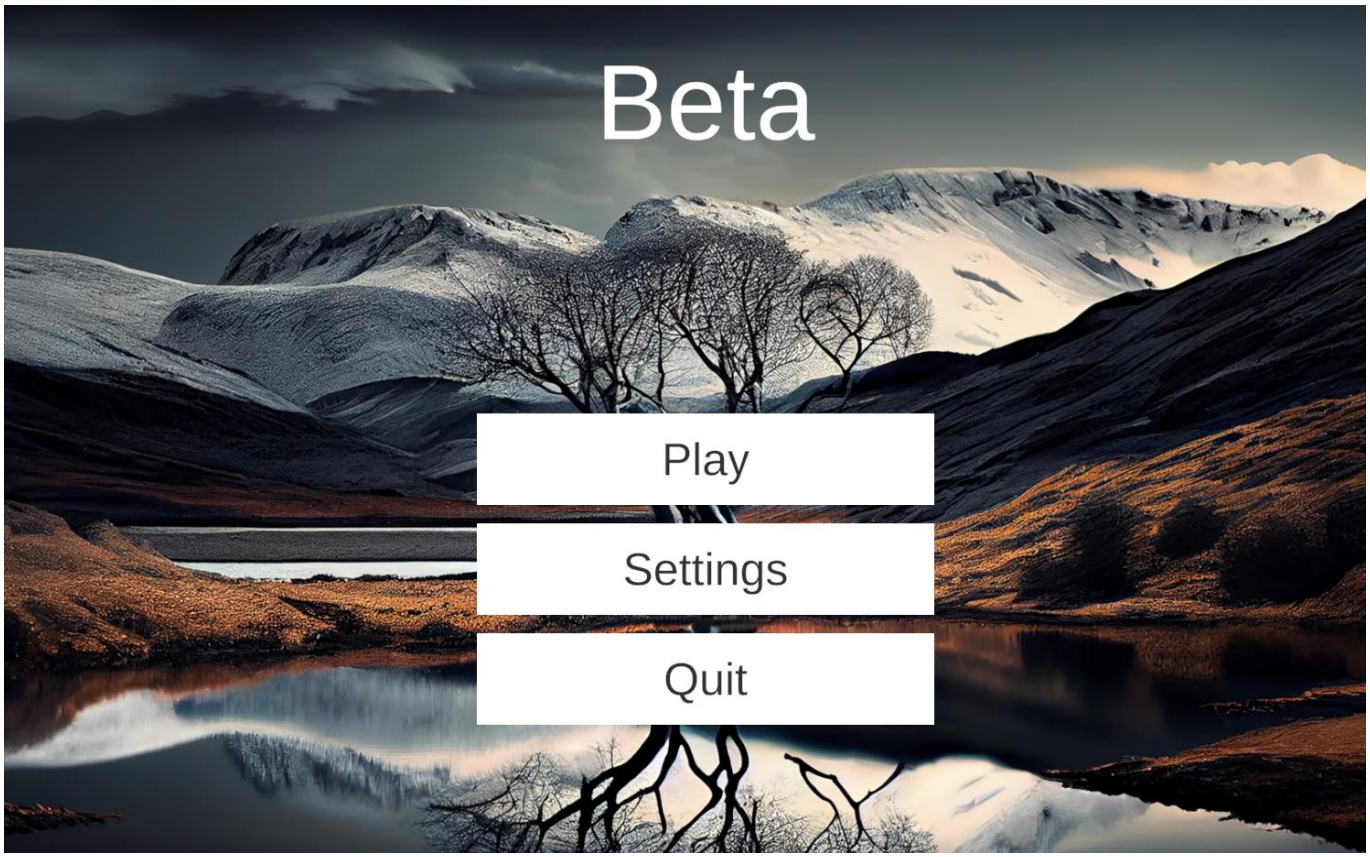

Рисунок Б.1 – Загальний вигляд вікна “меню” при запуску гри

ДОДАТОК В

Лістинг програми

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

public class Events : MonoBehaviour
{
    public void Menu()
    {
        SceneManager.LoadScene(0);
    }

    public void Level()
    {
        SceneManager.LoadScene(1);
    }

    public void Quit()
    {
        Application.Quit();
    }
}
```

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

public class ExitTrigger : MonoBehaviour
{
    //public Animator anim;
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.gameObject.tag == "Player")
        {
            StartCoroutine("LevelExit");
        }
    }

    IEnumerator LevelExit()
    {
        //anim.SetTrigger("Exit");
        yield return new WaitForSeconds(0.1f);

        UIManager.instance.fadeToBlack = true;

        yield return new WaitForSeconds(2f);
        // Do something after flag anim
```

```
        GameManager.instance.LevelComplete();
    }
}

using UnityEngine;
using TMPro;
using System.Collections;
using UnityEngine.SceneManagement;

public class GameManager : MonoBehaviour
{
    public static GameManager instance;


    [SerializeField] private TMP_Text coinText;

    [SerializeField] private PlayerController playerController;

    private int coinCount = 0;
    private int gemCount = 0;
    private bool isGameOver = false;
    private Vector3 playerPosition;

    //Level Complete

    [SerializeField] GameObject levelCompletePanel;
    [SerializeField] TMP_Text leveCompletePanelTitle;
    [SerializeField] TMP_Text levelCompleteCoins;

    private int totalCoins = 0;


    private void Awake()
    {
        instance = this;
        Application.targetFrameRate = 60;
    }

    private void Start()
    {
        UpdateGUI();
        UIManager.instance.fadeFromBlack = true;
        playerPosition = playerController.transform.position;

        FindTotalPickups();
    }

    public void IncrementCoinCount()
    {
        coinCount++;
        UpdateGUI();
    }
```

```
public void IncrementGemCount()
{
    gemCount++;
    UpdateGUI();
}

private void UpdateGUI()
{
    coinText.text = coinCount.ToString();
}

public void Death()
{
    if (!isGameOver)
    {
        // Disable Mobile Controls
        UIManager.instance.DisableMobileControls();
        // Initiate screen fade
        UIManager.instance.fadeToBlack = true;

        // Disable the player object
        playerController.gameObject.SetActive(false);

        // Start death coroutine to wait and then respawn the player
        StartCoroutine(DeathCoroutine());

        // Update game state
        isGameOver = true;

        // Log death message
        Debug.Log("Died");
    }
}

public void FindTotalPickups()
{
    pickup[] pickups = GameObject.FindObjectsOfType<pickup>();

    foreach (pickup pickupObject in pickups)
    {
        if (pickupObject.pt == pickup.pickupType.coin)
        {
            totalCoins += 1;
        }
    }
}

public void LevelComplete()
{
    levelCompletePanel.SetActive(true);
    leveCompletePanelTitle.text = "LEVEL COMPLETE";
```

```
        levelCompleteCoins.text = "COINS COLLECTED: " + coinCount.ToString() + " / " +
totalCoins.ToString();
    }

    public IEnumerator DeathCoroutine()
    {
        yield return new WaitForSeconds(1f);
        playerController.transform.position = playerPosition;

        // Wait for 2 seconds
        yield return new WaitForSeconds(1f);

        // Check if the game is still over (in case player respawns earlier)
        if (isGameOver)
        {
            SceneManager.LoadScene(1);
        }
    }
}

using UnityEngine;
using UnityEngine.UI;

public class HealthManager : MonoBehaviour
{
    public static HealthManager instance;
    public GameObject damageEffect;

    private int MaxHealth = 6;
    public int currentHealth;

    [SerializeField] private Image[] hearts;
    [SerializeField] private Sprite FullHeartSprite;
    [SerializeField] private Sprite HalfHeartSprite;
    [SerializeField] private Sprite EmptyHeartSprite;

    private GameObject Player;

    private void Awake()
    {
        instance = this;
    }

    private void Start()
    {
        Player = GameObject.FindObjectOfType<PlayerController>().gameObject;
        currentHealth = MaxHealth;
        DisplayHearts();
    }


    public void HurtPlayer()
```

```
    {
        if (currentHealth > 0)
        {
            currentHealth--;
            DisplayHearts();
            //Player.GetComponent<PlayerController>().Knockback();
        }
        else if (currentHealth <= 0)
        {
            GameManager.instance.Death();
        }

        Instantiate(damageEffect, Player.transform.position, Quaternion.identity);
    }

    public void DisplayHearts()
    {
        int fullHeartsCount = currentHealth / 2; // Calculate the number of full hearts
        bool hasHalfHeart = (currentHealth % 2) == 1; // Check if there's a half heart needed

        for (int i = 0; i < hearts.Length; i++)
        {
            if (i < fullHeartsCount)
            {
                // Heart should be full
                hearts[i].sprite = FullHeartSprite;
            }
            else if (hasHalfHeart && i == fullHeartsCount)
            {
                // Heart should be half
                hearts[i].sprite = HalfHeartSprite;
            }
            else
            {
                // Heart should be empty
                hearts[i].sprite = EmptyHeartSprite;
            }
        }
    }
}

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

public class pickup : MonoBehaviour
{
    public enum pickupType
    {
        coin,
        gem,
        health
```

```
    }

    public pickupType pt;
    [SerializeField] GameObject PickupEffect;

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (pt == pickupType.coin)
        {
            if (collision.gameObject.tag == "Player")
            {
                GameManager.instance.IncrementCoinCount();

                Instantiate(PickupEffect, transform.position, Quaternion.identity);

                Destroy(this.gameObject, 0.2f);
            }
        }

        if (pt == pickupType.gem)
        {
            if (collision.gameObject.tag == "Player")
            {
                GameManager.instance.IncrementGemCount();

                Instantiate(PickupEffect, transform.position, Quaternion.identity);

                Destroy(this.gameObject, 0.2f);
            }
        }
    }
}

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

public enum Controls
{
    mobile,
    pc
}

public class PlayerController : MonoBehaviour
{
    public float moveSpeed = 5f;
    public float jumpForce = 10f;
    public float doubleJumpForce = 8f;
    public LayerMask groundLayer;
    public Transform groundCheck;
```

```
private Rigidbody2D rb;
private bool isGroundedBool = false;
private bool canDoubleJump = false;

public Animator playeranim;

public Controls controlmode;


private float moveX;
public bool isPaused = false;

public ParticleSystem footsteps;
private ParticleSystem.EmissionModule footEmissions;

public ParticleSystem ImpactEffect;
private bool wasonGround;


// public GameObject projectile;
// public Transform firePoint;

public float fireRate = 0.5f; // Time between each shot
private float nextFireTime = 0f; // Time of the next allowed shot

private void Start()
{
   rb = GetComponent<Rigidbody2D>();
   footEmissions = footsteps.emission;

   if (controlmode == Controls.mobile)
   {
      UIManager.instance.EnableMobileControls();
   }
}

private void Update()
{
   isGroundedBool = IsGrounded();

   if (isGroundedBool)
   {
      canDoubleJump = true; // Reset double jump when grounded

      if (controlmode == Controls.pc)
      {
         moveX = Input.GetAxis("Horizontal");
      }


      if (Input.GetButtonDown("Jump"))
      {
```

```
            Jump(jumpForce);
        }
    }
    else
    {
        if (canDoubleJump && Input.GetButtonDown("Jump"))
        {
            Jump(doubleJumpForce);
            canDoubleJump = false; // Disable double jump until grounded again
        }
    }

    if (!isPaused)
    {
        // Calculate rotation angle based on mouse position
        Vector3 mousePosition = Camera.main.ScreenToWorldPoint(Input.mousePosition);
        Vector3 lookDirection = mousePosition - transform.position;
        float angle = Mathf.Atan2(lookDirection.y, lookDirection.x) * Mathf.Rad2Deg;

        // ... (your existing code for rotation)

        // Handle shooting
        if (controlmode == Controls.pc && Input.GetButtonDown("Fire1") && Time.time >=
nextFireTime)
        {
            Shoot();
            nextFireTime = Time.time + 1f / fireRate; // Set the next allowed fire time
        }
    }

    SetAnimations();

    if (moveX != 0)
    {
        FlipSprite(moveX);
    }

    //impactEffect

    if (!wasonGround && isGroundedBool)
    {
        ImpactEffect.gameObject.SetActive(true);
        ImpactEffect.Stop();
        ImpactEffect.transform.position =
            new Vector2(footsteps.transform.position.x, footsteps.transform.position.y - 0.2f);
        ImpactEffect.Play();
    }

    wasonGround = isGroundedBool;
}

public void SetAnimations()
```

```
    {
        if (moveX != 0 && isGroundedBool)
        {
            playeranim.SetBool("run", true);
            footEmissions.rateOverTime = 35f;
        }
        else
        {
            playeranim.SetBool("run", false);
            footEmissions.rateOverTime = 0f;
        }

        playeranim.SetBool("isGrounded", isGroundedBool);
    }

    private void FlipSprite(float direction)
    {
        if (direction > 0)
        {
            // Moving right, flip sprite to the right
            transform.localScale = new Vector3(1, 1, 1);
        }
        else if (direction < 0)
        {
            // Moving left, flip sprite to the left
            transform.localScale = new Vector3(-1, 1, 1);
        }
    }

    private void FixedUpdate()
    {
        // Player movement
        if (controlmode == Controls.pc)
        {
            moveX = Input.GetAxis("Horizontal");
        }

        rb.velocity = new Vector2(moveX * moveSpeed, rb.velocity.y);
    }

    private void Jump(float jumpForce)
    {
        rb.velocity = new Vector2(rb.velocity.x, 0); // Zero out vertical velocity
        rb.AddForce(Vector2.up * jumpForce, ForceMode2D.Impulse);
        playeranim.SetTrigger("jump");
    }

    private bool IsGrounded()
    {
        float rayLength = 0.25f;
        Vector2 rayOrigin = new Vector2(groundCheck.transform.position.x,
groundCheck.transform.position.y - 0.1f);
```

```
        RaycastHit2D hit = Physics2D.Raycast(rayOrigin, Vector2.down, rayLength, groundLayer);
        return hit.collider != null;
    }

    private void OnCollisionEnter2D(Collision2D collision)
    {
        if (collision.gameObject.tag == "killzone")
        {
            GameManager.instance.Death();
        }
    }


    //mobile;
    public void MobileMove(float value)
    {
        moveX = value;
    }

    public void MobileJump()
    {
        if (isGroundedBool)
        {
            // Perform initial jump
            Jump(jumpForce);
        }
        else
        {
            // Perform double jump if allowed
            if (canDoubleJump)
            {
                Jump(doubleJumpForce);
                canDoubleJump = false; // Disable double jump until grounded again
            }
        }
    }

    public void Shoot()
    {
        //GameObject fireBall = Instantiate(projectile, firePoint.position, Quaternion.identity);
        //fireBall.GetComponent<Rigidbody2D>().AddForce(firePoint.right * 500f);
    }

    public void MobileShoot()
    {
        if (Time.time >= nextFireTime)
        {
            Shoot();
            nextFireTime = Time.time + 1f / fireRate; // Set the next allowed fire time
        }
    }
}
```

ДОДАТОК Г

Графічний матеріали

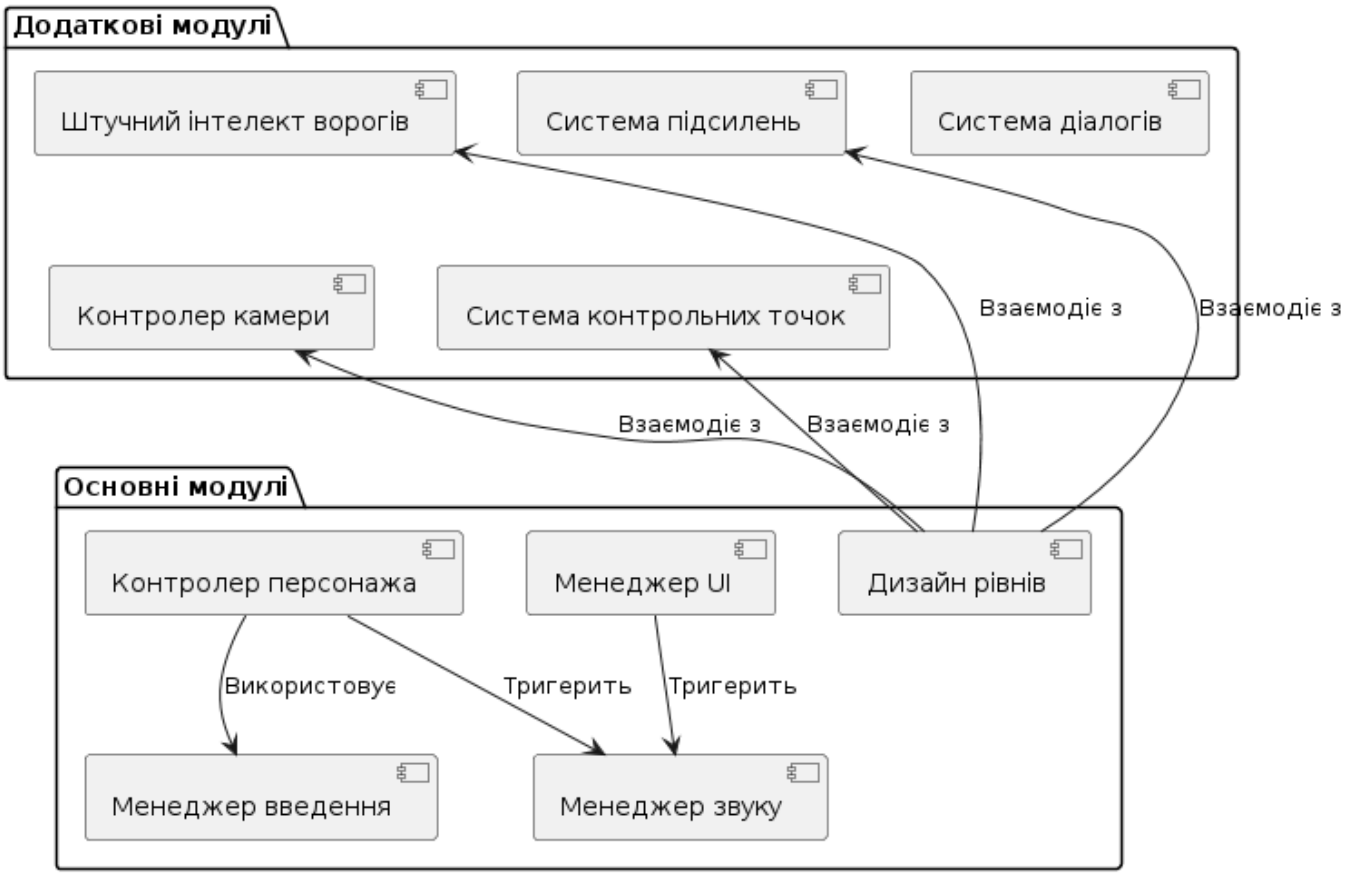

Рисунок Г.1 — Принципова схема архітектури комп’ютерної гри

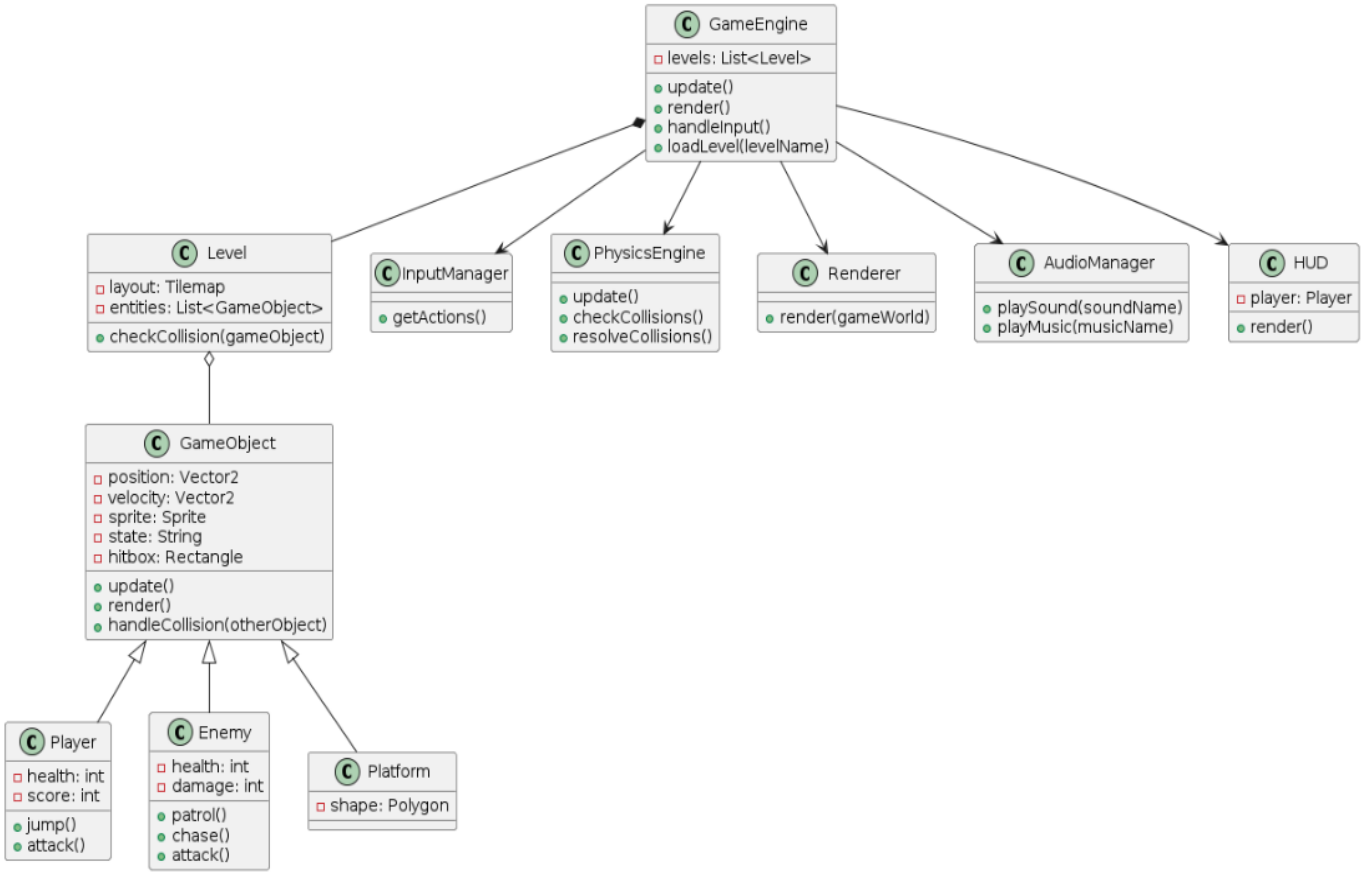

Рисунок Г.2 — Принципова схема взаємодії системних елементів комп’ютерної гри

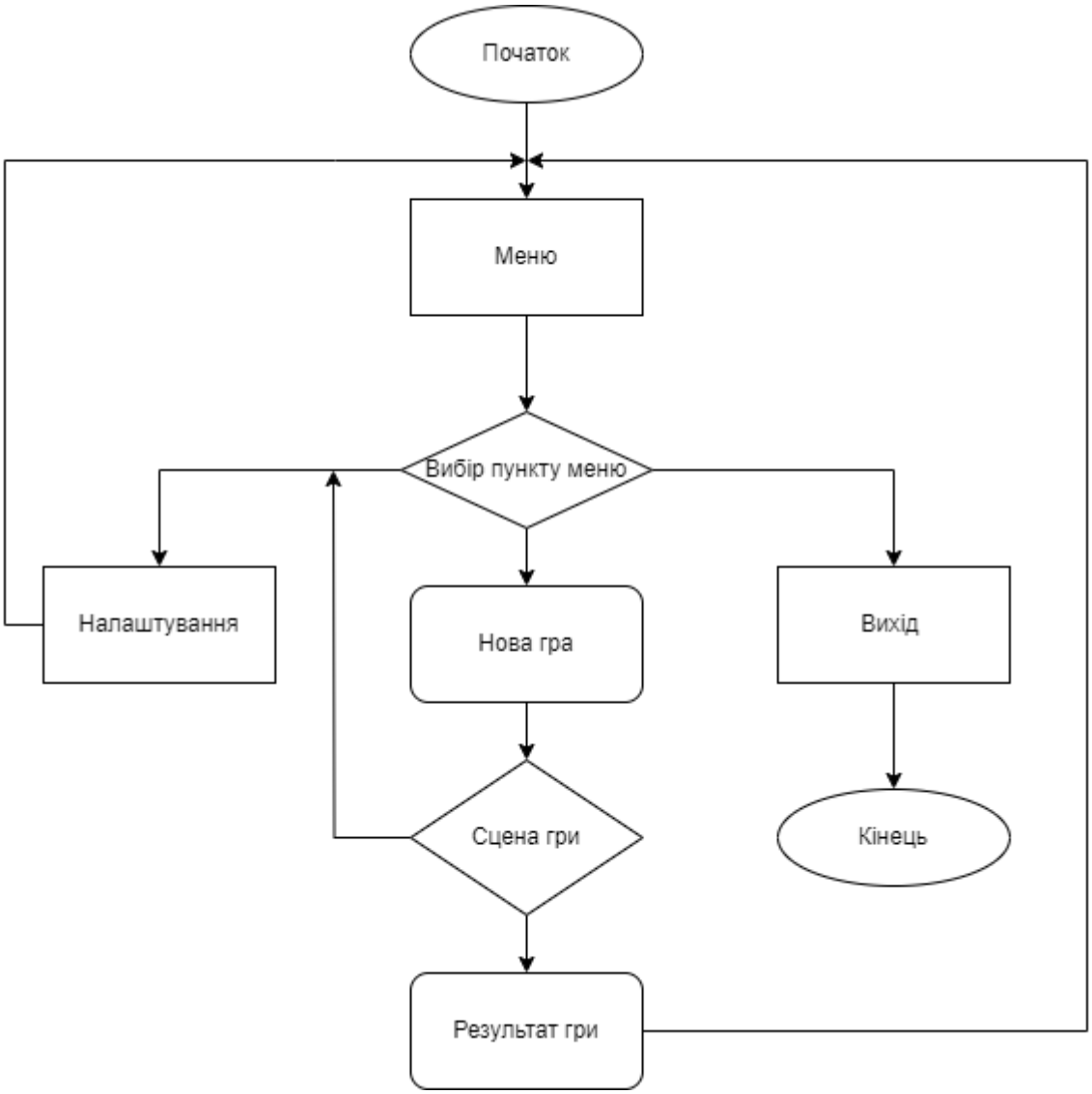

Рисунок Г.3 — Принципова схема алгоритму функціонування комп’ютерної гри

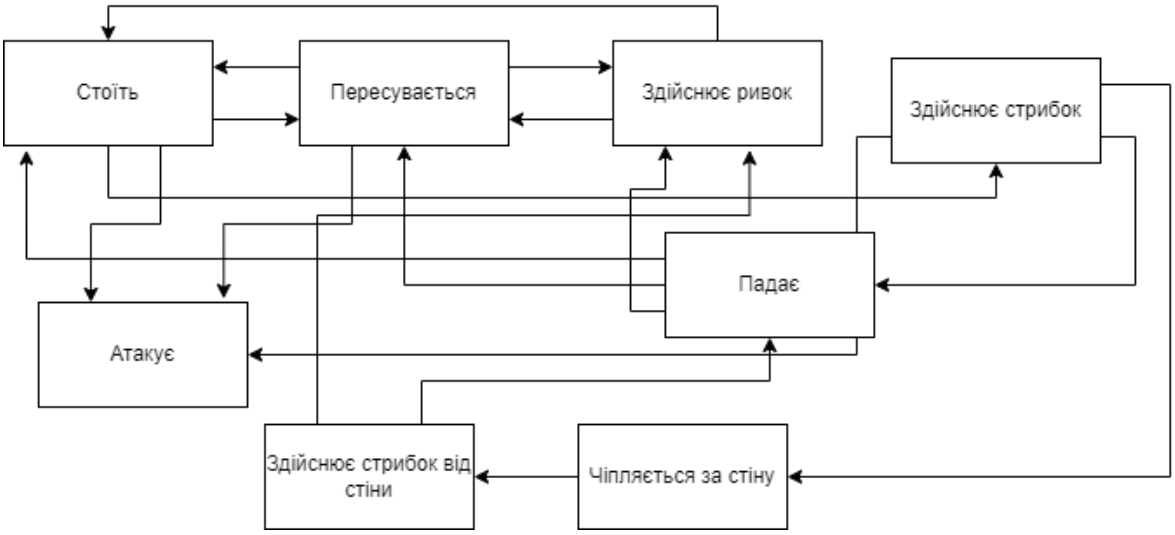

Рисунок Г.4 — Принципова схема діаграми станів гравця

Рисунок Г.5 – Принципова схема алгоритму ігрового цикла комп’ютерної гри

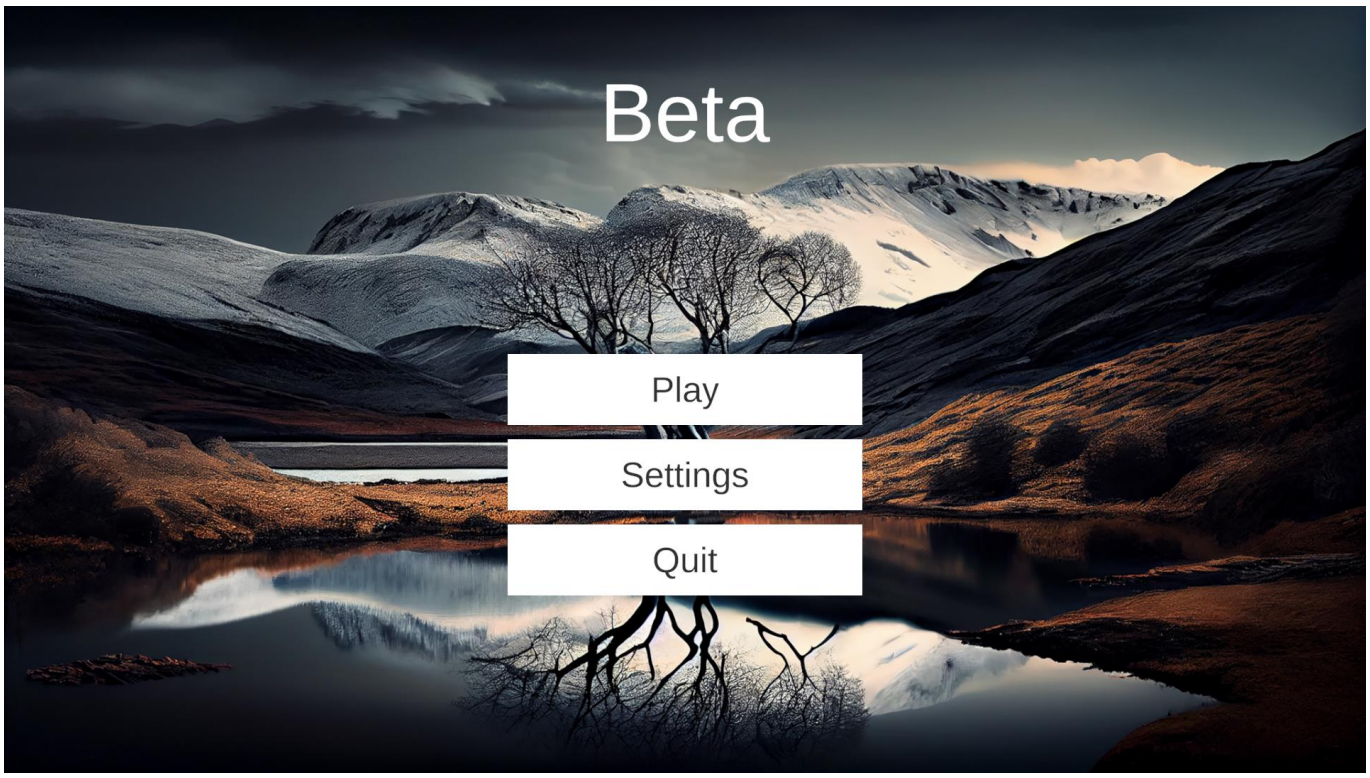

Рисунок Г.6 – Загальний вигляд інтерфейсного вікна типу «Головне меню» комп’ютерної гри

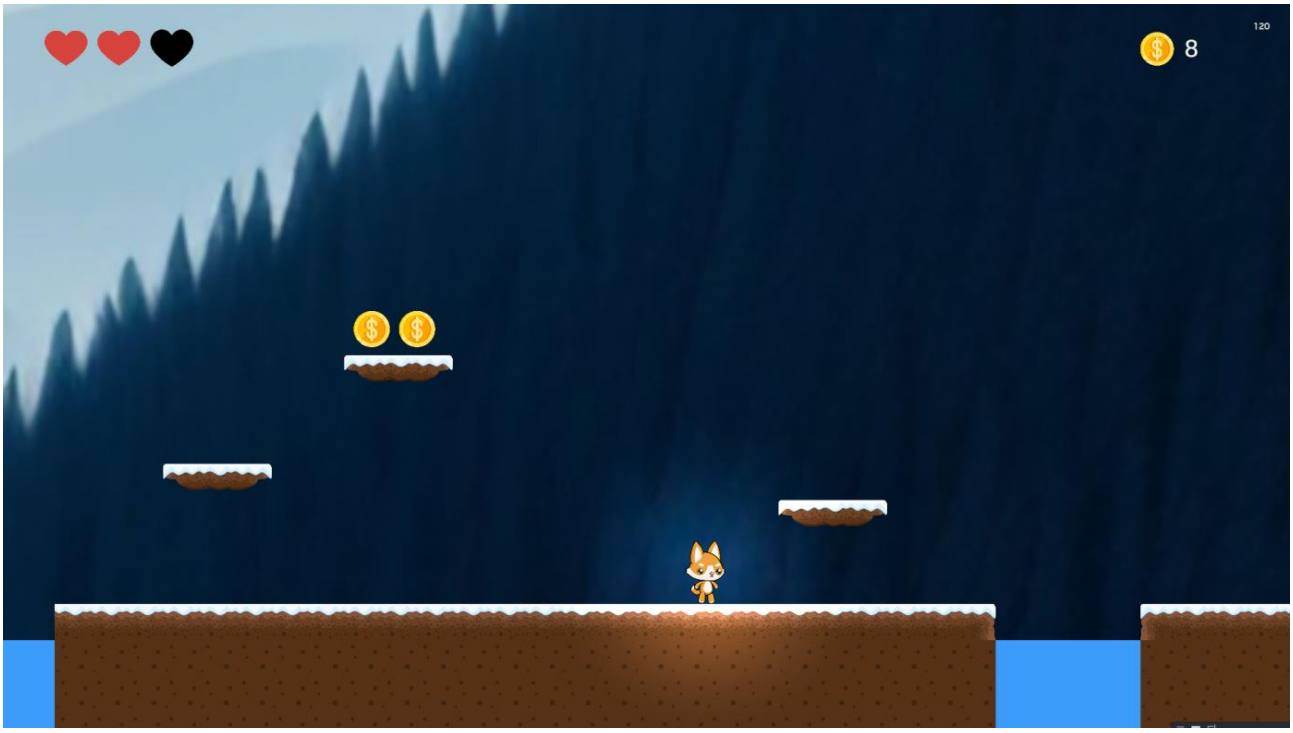

Рисунок Г.7 – Загальний вигляд геймплею комп’ютерної гри

Рисунок Г.8 – Загальний вигляд завершення комп’ютерної гри