МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Чорноморський національний університет імені Петра Могили

Факультет комп'ютерних наук

Кафедра інженерії програмного забезпечення

ДОПУЩЕНО ДО ЗАХИСТУ

Завідувач кафедри_____ Є. О. Давиденко

підпис

«___» _____2024 р.

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

Розробка ігрового застосунку в жанрі платформера на рушії Unity

Спеціальність «Інженерія програмного забезпечення»

121 – КРБ.1 – 408.22010814

Здобувач _____М. Д. Мядзелець

підпис

«__» _____ 2024 р.

Керівник PhD, ст. викладач _____ І. О. Кандиба

підпис

«__» _____ 2024 р.

Консультант канд. техн. наук, доцент _____ А. О. Алексєєва

підпис

«__» _____ 2024 р.

Миколаїв – 2024 рік

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Чорноморський національний університет імені Петра Могили
Факультет комп'ютерних наук
Кафедра інженерії програмного забезпечення

ЗАТВЕРДЖУЮ

Зав. кафедри _____ Є. О. Давиденко

« 22 » ____грудня_____ 2023 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи бакалавра

Видано здобувачу групи 408 факультету комп'ютерних наук

_____Мядзелець Максим Денисович_____

(прізвище, ім'я, по батькові здобувача)

1. Тема кваліфікаційної роботи

_____Розробка ігрового застосунку в жанрі платформера на рушії Unity_____

_____

Затверджена наказом по ЧНУ від «22» грудня 2023 р. № 269

2. Строк представлення кваліфікаційної роботи «____» _____ 20__ р.

3. Очікуваний результат роботи та початкові дані, якщо такі потрібні

_____Очікуваним результатом є ігровий застосунок у жанрі платформера на основі рушія Unity2D_____

4. Перелік питань, що підлягають розробці

Проведення аналізу предметної галузі, включаючи вивчення популярних платформерів, їхніх особливостей та недоліків. Розробка концепції та проектування ігрового платформера з використанням Unity. Реалізація ігрового платформера, включаючи розробку графіки, програмування геймплею та обробку звукового супроводу. Проведення тестування розробленого ігрового платформера з метою виявлення і усунення помилок та недоліків.

5. Перелік графічних матеріалів

Презентація

6. Завдання до спеціальної частини

_____

_____

_____

7. Консультанти:

Консультант	Кафедра (організація) Частина роботи	Кафедра (організація) Частина роботи
Алєксєєва А.О.	Кафедра екології	Спеціальна частина з охорони праці

Керівник роботи ___PhD, ст. викладач Кандиба Ігор Олександрович_____

(посада, прізвище, ім'я, по батькові)

_____

(підпис)

Завдання прийнято до виконання

Мядзелець Максиму Денисовичу

(прізвище, ім'я, по батькові здобувача)

_____

(підпис)

Дата видачі завдання « 22 » ____грудня_____ 2023 р.

КАЛЕНДАРНИЙ ПЛАН виконання кваліфікаційної роботи

Тема:_____Розробка ігрового застосунку в жанрі платформера на рушії Unity

№	Найменування роботи	Початок	Закінчення	Примітки
1	Розробка та затвердження завдання на виконання КРБ	21.12.23	21.12.23	виконано
2	Огляд літератури за темою роботи	24.12.23	28.12.23	виконано
3	Складання календарного плану КРБ	15.01.24	15.01.24	виконано
4	Аналіз предметної області	16.01.24	17.01.24	виконано
5	Розробка проєктних рішень	17.01.24	20.01.24	виконано
6	Моделювання та конструювання ПЗ	21.01.24	27.01.24	
7	Кодування та тестування розробленого ПЗ	27.01.24	10.02.24	виконано
8	Розробка керівництва користувача	10.02.24	22.02.24	виконано
9	Розробка спеціальної частини з охорони праці	20.05.24	26.05.24	виконано
10	Відгук керівника КРБ	27.05.24	27.05.24	виконано
11	Оформлення КРБ та презентації	26.05.24	03.06.24	виконано
12	Попередній захист	03.06.24	05.06.24	виконано
13	Рецензування	15.06.24	18.06.24	виконано
14	Захист кваліфікаційної роботи	26.06.24	26.06.24	виконано

Розробив здобувач ____Мядзелець Максим Денисович_____ _____

(прізвище, ім'я, по батькові) *(підпис)*

«15» _січня___ 2024 р.

Керівник роботи __ст. викладач Кандиба Ігор Олександрович____ _____

(посада, прізвище, ім'я, по батькові) *(підпис)*

«16» _січня___ 2024 р.

АНОТАЦІЯ

до кваліфікаційної роботи бакалавра

«Розробка ігрового застосунку в жанрі платформера на рушії Unity»

Здобувач 408 гр.: Мядзелець Максим Денисович

Керівник: PhD, ст. викладач Кандиба Ігор Олександрович

Кваліфікаційна робота присвячена розробці ігрового застосунку в жанрі платформеру на рушії Unity.

Об’єктом кваліфікаційної роботи є процес створення ігрового застосунку на базі Unity.

Предметом є інструментарії розробки ігрового застосунку на базі Unity.

Метою кваліфікаційної роботи є створення ігрового платформера на базі Unity з метою привернення уваги гравців та стимулювання розвитку цього жанру в ігровій індустрії.

Для досягнення цієї мети необхідно вирішити наступні завдання:

1) дослідити особливостей жанру платформер та провести аналіз сучасних ігор цього жанру;

2) проаналізувати можливості рушія Unity для розробки ігор та використання його функцій для реалізації особливостей гри в жанрі платформер;

3) спроєкутвати гру в жанрі платформер;

4) розробка дизайну гри, включаючи геймплей, рівні, персонажів та анімації;

5) реалізація гри на рушії Unity та тестування її функцій та ефективності.

У першому розділі було проведено аналіз різних жанрів ігрових застосунків. Кожен з них був проаналізований окремо, визначено їх основні риси. Також були виявлені переваги та недоліки кожного ігрового застосунку. У завершальній частині розділу 1 була сформульована специфікація вимог для мобільної гри в жанрі платформер на платформі Unity.

У другому розділі було проведено створення та аналіз діаграм. Отримано цінний інструмент для аналізу та моделювання взаємодії між системою та її користувачами. Створено діаграми що були потрібні для подальшого проектування системи.

У третьому розділі було проведено аналіз ключових інструментів та технологій для розробки ігор, таких як .NET (зокрема мова програмування C#), Visual Studio та Unity. На основі характеристик та переваг кожного з інструментів було обрано найбільш підходящий для проєкту, забезпечуючий успішну та ефективну розробку гри.

У четвертому розділі було розроблено 2D ігровий застосунок жанру Platform, з аналізом існуючих рішень, дизайном програмного забезпечення, створенням інтерфейсу користувача, розробкою ігрових елементів, вибором рушія Unity 3D, тестуванням ігрової механіки, що дозволило досягти мети проекту та створити функціональну й привабливу гру.

КРБ викладена на 92 сторінки, вона містить 4 розділи, 64 ілюстрацій, 3 таблиці, 20 джерел в переліку посилань

Ключові слова: ігровий застосунок, платформер, рівні, звукове супроводження, гравці, unity.

ABSTRACT

to the bachelor's qualification work

«Development of a platformer game application on the Unity engine»

Student of group 408: Myadzelets Maksym Denysovych

Supervisor: PhD, senior lecturer Kandyba Igor Oleksandrovych

Work is devoted to the development of a game application in the genre of platformer on the Unity engine.

The object of qualification work is the process of creating a game application based on Unity.

The subject is the tools for developing a game application based on Unity.

The purpose of this qualification work is to create a platformer game based on Unity in order to attract the attention of players and stimulate the development of this genre in the gaming industry.

To achieve this goal, the following tasks need to be solved:

1) to study the features of the platformer genre and analyse modern games of this genre;

2) to analyse the capabilities of the Unity engine for game development and the use of its functions to implement the features of a platformer game;

3) to design a game in the platformer genre;

4) develop game design, including gameplay, levels, characters and animations;

5) implementing the game on the Unity engine and testing its functions and efficiency.

The first section analysed different genres of gaming applications. Each of them was analysed separately and their main features were identified. The advantages and disadvantages of each game application were also identified. In the final part of Chapter 1, a specification of requirements for a mobile platformer game on the Unity platform was formulated.

In second chapter, diagrams were created and analysed. We obtained a valuable tool for analysing and modelling the interaction between the system and its users. The diagrams that were needed for further system design were created.

In the third chapter, we analysed key tools and technologies for game development, such as .NET (in particular, the C# programming language), Visual Studio, and Unity. Based on the characteristics and advantages of each tool, the most suitable for the project was chosen to ensure successful and efficient game development.

In the fourth chapter, we developed a 2D Platform game application, including analysis of existing solutions, software design, creation of the user interface, development of game elements, selection of the Unity 3D engine, and testing of game mechanics, which allowed us to achieve the project goal and create a functional and attractive game.

The bachelor's qualification work is set out on 92 pages, it contains 4 sections, 64 illustrations, 3 tables, 20 sources in the list of references

Keywords: game application, platformer, levels, sound, players, unity.

ЗМІСТ

ВСТУП

Актуальність обраної теми полягає в тому, що в сучасному світі ігрова індустрія стикається з викликом стагнації, що виникає через перенасиченість ринку платформерів. Велика кількість готових продуктів, в основному ремастери та продовження існуючих ігор, призводить до втрати цікавості гравців та зменшення їхнього інтересу до нових проектів. Така ситуація негативно позначається на розвитку галузі, адже платформери є одним із найпопулярніших жанрів, і необхідно шукати способи для їхнього оновлення та вдосконалення. Розробка нового платформера на базі Unity надасть можливість привернути увагу гравців та стимулювати розвиток індустрії.

2) Об’єкт та предмет дослідження

Об’єктом кваліфікаційної роботи є процес створення ігрового застосунку на базі Unity.

Предметом є інструментарії розробки ігрового застосунку на базі Unity.

Метою кваліфікаційної роботи є створення ігрового платформера на базі Unity з метою привернення уваги гравців та стимулювання розвитку цього жанру в ігровій індустрії.

Завданням роботи є:

– Проведення аналізу предметної галузі, включаючи вивчення популярних платформерів, їхніх особливостей та недоліків.

– Розробка концепції та проєктування ігрового платформера з використанням Unity.

– Реалізація ігрового застосунку, включаючи розробку графіки, програмування геймплею та обробку звукового супроводу.

– Проведення тестування розробленої гри з метою виявлення і усунення помилок та недоліків.

1 АНАЛІЗ ІГРОВОГО ЖАНРУ ПЛАТФОРМЕРІВ

1.1 Історія та еволюція жанру платформера

Початки і перші кроки (1980-і роки)

Жанр платформер бере свій початок у 80-х роках. Цей період характеризується зародженням відеоігор та експериментами з геймплеєм. «Donkey Kong» від Nintendo та «Mario Bros» від Shigeru Miyamoto стали піонерами цього жанру, встановивши основні механіки, такі як стрибки, рухи по платформам та уникання перешкод (рис. 1.1). Ці ігри стали символами тогочасної індустрії відеоігор і відіграли ключову роль у формуванні жанру платформера.

Епоха 16-бітних консолей та золота середина (1990-і роки)

90-ті роки принесли з собою нові можливості завдяки введенню 16-бітних консолей. Цей період відзначився збільшенням деталізації графіки та розширенням геймплею. Гра «Sonic the Hedgehog» від Sega стала інноваційною в жанрі, зосередившись на швидкості та динаміці, що відрізняли її від класичних платформерів (рис. 1.2). Ігри як «Super Mario World» від Nintendo продовжували радувати гравців захоплюючими пригодами, зберігаючи при цьому класичний стиль геймплею.

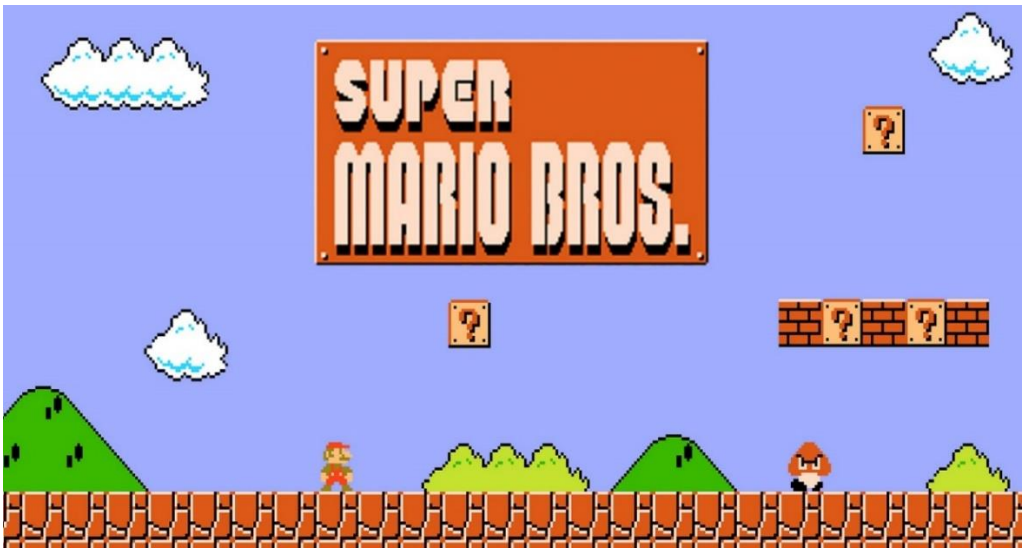

Рисунок 1.1 – Інтерфейс відеогри Super Mario World

Рисунок 1.2 – Інтерфейс відеогри Sonic the Hedgehog

3D-революція (початок 2000-х)

З появою 3D-графіки в кінці 90-х років, жанр платформера пережив нову еру. Гра «Super Mario 64» від Nintendo стала культовою, перенеся геймплей Маріо у тривимірний світ (рис. 1.3). Це відкрило необмежені можливості для дослідження, дозволяючи гравцям вільно рухатися та взаємодіяти з оточуючим світом. Інші ігри, такі як «Banjo-Kazooie» від Rare, також розширили можливості жанру, вводячи складніші головоломки та інноваційні механіки геймплею.

Рисунок 1.3 – Інтерфейс відеогри Super Mario 64

Сучасність і нові тенденції (з 2010-х років)

У сучасний період, жанр платформера продовжує розвиватися та адаптуватися до сучасних реалій. Ігри як «Celeste», «Hollow Knight» та «Super Meat Boy» стали символами сучасної інді-сцени, вводячи нові геймплейні механіки та естетику (рис. 1.4), (рис. 1.5). З появою мобільних платформ та віртуальної реальності, жанр платформера знову переживає період інновацій. Ігри, такі як «Moss» та «Astro Bot Rescue Mission», демонструють, як платформери можуть використовувати нові технології для створення іммерсивного та захоплюючого досвіду для гравців.

Таким чином, історія та еволюція жанру платформера свідчить про його неперервний розвиток та здатність адаптуватися до змін у технологіях та смаках гравців. Жанр продовжує залишатися важливою складовою індустрії відеоігор, пропонуючи нові інновації та захоплюючі пригоди для гравців усього світу.

Рисунок 1.4 – Інтерфейс відеогри Super Meat Boy

Рисунок 1.5 – Інтерфейс відеогри Celeste

Таким чином, історія та еволюція жанру платформера свідчить про його неперервний розвиток та здатність адаптуватися до змін у технологіях та смаках гравців. Жанр продовжує залишатися важливою складовою індустрії відеоігор, пропонуючи нові інновації та захоплюючі пригоди для гравців усього світу.

1.2 Огляд існуючих ігрових застосунків у жанрі платформера

Класичні платформери:

Ця категорія включає в себе ігри, які віддзеркалюють класичний стиль платформера, який був у перших іграх серії «Super Mario» та «Sonic the Hedgehog». Вони часто відрізняються яскравою, кольоровою графікою, простими, але ефективними механіками геймплею та веселими, запам’ятовуваними персонажами. Найбільш відомі представники цієї категорії - це ігри «Super Mario Bros.», «Super Mario World», «Sonic the Hedgehog», а також останні версії у серіях, такі як «Super Mario Odyssey» та «Sonic Mania» (рис. 1.6).

Рисунок 1.6 – Інтерфейс відеогри Super Mario Odyssey

Метроїдванії:

Ця підкатегорія поєднує елементи платформера з метроїдваній (англ. Metroidvania) - піджанром пригодницьких ігор, які характеризуються великим відкритим світом для дослідження та необхідністю отримання нових навичок для просування в грі. Вони зазвичай мають складні лабіринти, головоломки та секрети, що можуть бути розкриті з розвитком гравця. Прикладами цієї категорії є «Hollow Knight», де гравець досліджує великий містикум, збираючи нові

навички та зброю, або «Ori and the Blind Forest», яка поєднує красиву графіку з емоційною сюжетною лінією (рис. 1.12), (рис. 1.13).

Рисунок 1.7 – Інтерфейс відеогри Sundered

Інді-платформери:

Інді-платформери - це ігри, розроблені невеликими студіями або навіть одним розробником, які часто відзначаються оригінальними механіками геймплею, унікальною візуальною естетикою та глибокими сюжетами. Вони можуть включати в себе інноваційні механіки, які змушують гравців переосмислити звичні аспекти платформера. Наприклад, «Celeste» - інді-платформер, який не лише пропонує складні рівні для пройдення, але й дотикується до теми психічного здоров’я та особистої боротьби. Іншим відомим прикладом є «Fez», який відзначається використанням унікальної графіки та складних головоломок (рис. 1.9).

Рисунок 1.9 – Інтерфейс відеогри Fez

3D платформери:

У цій категорії представлені платформери у тривимірному просторі, які дозволяють гравцям рухатися в різних площинах та відчувати глибину та просторовість гри. Вони часто відрізняються більшою свободою руху та складнішими головоломками. Популярні представники цієї категорії - це «Ratchet & Clank», «Spyro the Dragon» та «Super Mario 3D World» (рис. 1.10).

Рисунок 1.10 – Інтерфейс відеогри Spyro the Dragon

Експериментальні платформери:

Ця категорія включає в себе ігри, які експериментують зі стандартними механіками платформера та впроваджують нові ідеї та концепції. Вони можуть включати в себе нестандартні механіки геймплею, унікальні контролі або навіть незвичайні жанрові гібриди. Наприклад, «Limbo» від Playdead - це платформер, який відрізняється своєю темною атмосферою та головоломками, а «Braid» від Number None, Inc. пропонує інноваційні механіки часу та простору (рис. 1.11).

Рисунок 1.11 – Інтерфейс відеогри Limbo

Hollow Knight

Таблиця 1.1 – Характеристика гри Hollow Knight[19]

Назва характеристики	Опис
Назва	Hollow Knight
Розробник	Team Cherry
Архітектура	Гра є двомірною грою з видом збоку.
Мова реалізації	C#
Ігровий процес	Hollow Knight - це гра про дослідження великого та загадкового світу, боротьбу з ворогами та розгадування секретів. Головний герой використовує меч для бою, накопичуючи душу з кожним ударом. У разі смерті втрачається вся валюта, але можна відновити її, подолавши власну тінь. Кожна зона пов’язана з іншими шляхами, що дає гравцям відчуття свободи.
Аналіз переваг	– Захоплюючий світ зі своїми таємницями; – глибокий геймплей зі складними боями та головоломками; – унікальний атмосферний дизайн персонажів та локацій;

Закінчення таблиці 1.1 – Характеристика гри Hollow Knight

Аналіз недоліків	– висока складність гри, що може відлякати деяких гравців; – можливість загубитися в складному світі без належного навігаційного підказу; – велика кількість випадкових помилок та багів, що можуть вплинути на геймплей.

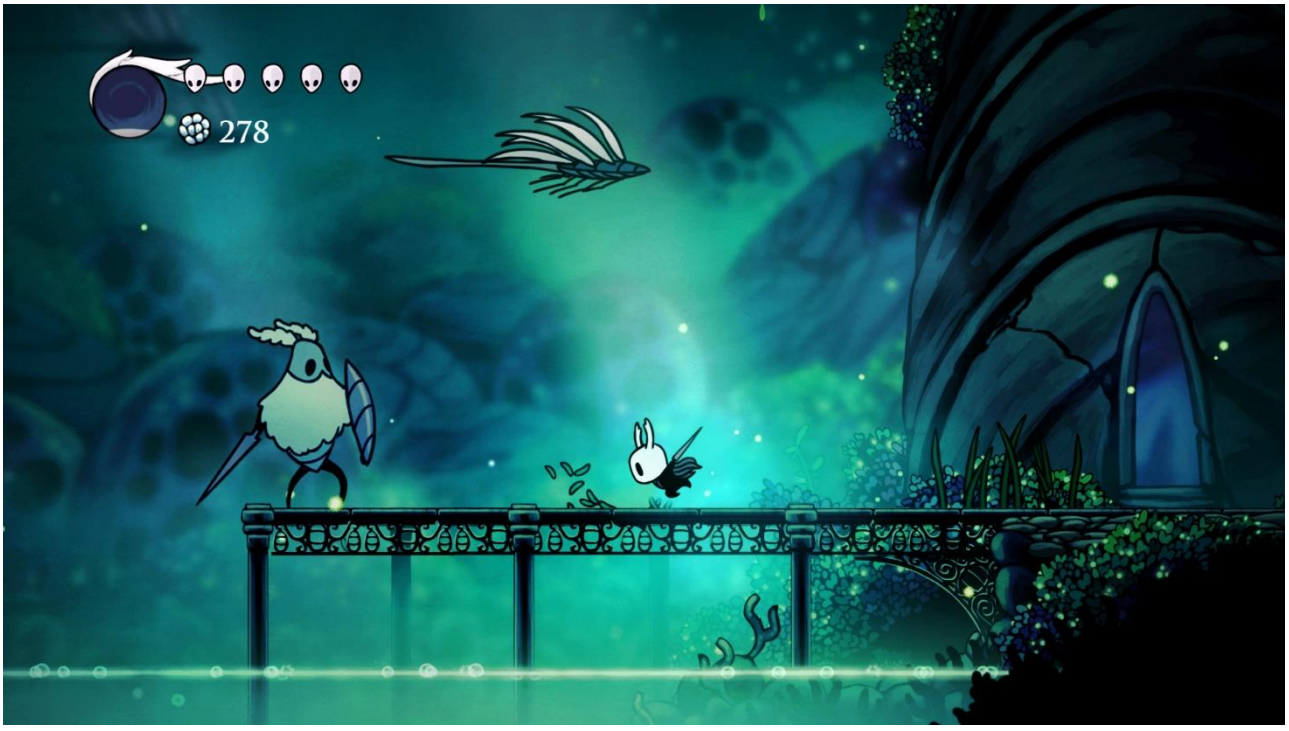

Рисунок 1.12 – Інтерфейс відеогри Hollow Knight

Ori and the Blind Forest

Таблиця 1.2 – Характеристика гри Ori and the Blind Forest[20]

Назва характеристики	Опис
Назва	Ori and the Blind Forest
Розробник	Moon Studios
Архітектура	Гра є двомірною грою з видом збоку.
Мова реалізації	C#
Ігровий процес	Ori and the Blind Forest - це двовимірний платформер з елементами тривимірного світу. Гравець управляє Орі - казковою істотою, яка шукає вихід з кожного рівня, збираючи чарівні камені для відкриття брам та знищення ворогів. Орі має запас життя та енергії, які можуть бути використані на посилені атаки та створення «духовних зв’язків» - точок збереження гри. Він навчається різним навичкам, таким як лазіння по стінах, пірнання під воду та відштовхування від ворожих атак. Рівні у грі прямо пов’язані між собою, що дозволяє гравцеві вільно переміщатися в світі та повертатися на вже пройдені рівні для пошуку прихованих предметів.

Закінчення таблиця 1.2 – Характеристика гри Ori and the Blind Forest

Аналіз переваг	– Дивовижна мистецька графіка та чутлива музика; – емоційна історія, що привертає увагу гравців; – чудова анімація персонажів та рухів;
Аналіз недоліків	– можливість деяким гравцям знайти гру занадто сумірною; – деякі головоломки можуть бути занадто складними для деяких гравців; – періодичні проблеми з оптимізацією, що можуть вплинути на продуктивність гри.

Рисунок 1.13 – Інтерфейс відеогри Ori and the Blind Forest

Rayman Origins

Таблиця 1.3 – Характеристика гри Rayman Origins[14]

Назва характеристики	Опис
Назва	Rayman Origins
Розробник	Ubisoft
Архітектура	Гра є двомірною грою з видом збоку.
Мова реалізації	C#
Ігровий процес	Шлях Рейман пролягає через безліч красивих і цікавих рівнів: через Тарабарські джунглі, пустелю Діджіріду, Ласу землю, Море прозорливості і Хмурні хмари. Головним завданням героя в першій половині є знаходження і звільнення фей, а в другій — королів цих земель. На рівні герой може підібрати безліч бонусів: монетки, електуни і маленькі світлячки — люми.
Аналіз переваг	– Веселий та динамічний геймплей; – багато різноманітних рівнів та викликів; – велика кількість різноманітних персонажів з унікальними властивостями;

Закінчення таблиці 1.3 – Характеристика гри Rayman Origins

Аналіз недоліків	– відсутність глибокої сюжетної лінії, що може робити гру менш привабливою для деяких гравців; – можливість відчувати певну монотонність через повторюваність дій; – нерідко зустрічаються помилки у поведінці штучного інтелекту персонажів.

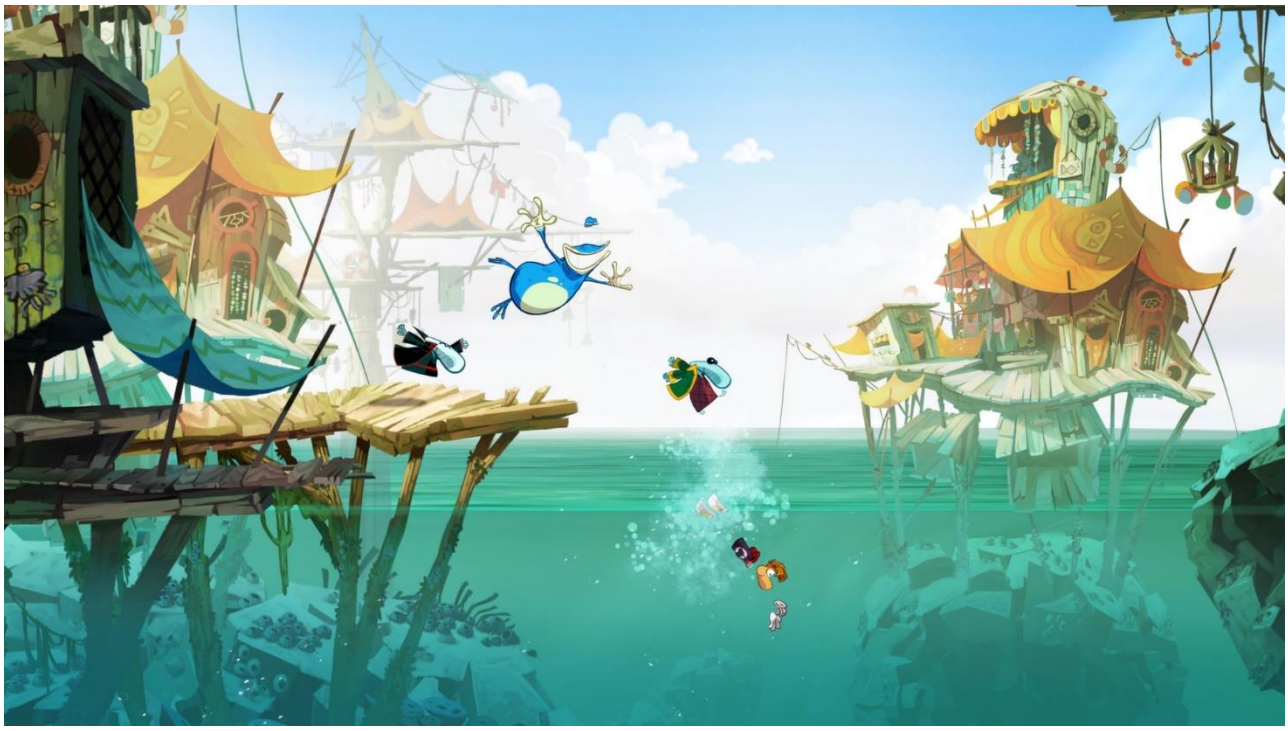

Рисунок 1.13 – Інтерфейс відеогри Rayman Origins

Rayman Origins пропонує веселий і легкий платформінг з акцентом на кооперативний геймплей, Ori and the Blind Forest зачаровує своєю емоційною історією та візуальною красою, в той час як Hollow Knight надає глибокий, складний і детально продуманий досвід дослідження. Кожна з цих ігор

виділяється своїми унікальними рисами, що роблять їх видатними у своєму жанрі і приваблюють різні категорії гравців.

1.3 Переваги 2D ігор

2D графіка має свої власні переваги, які варто врахувати при розробці ігор. Перш за все, вона забезпечує чіткі та деталізовані графічні елементи, що полегшує сприйняття інтерфейсу для користувачів. Багато гравців також відчувають особливе зв’язок з класичним стилем 2D ігор, що спонукає їх насолоджуватись грою через відчуття ностальгії.

Проте існує ризик, що це почуття може бути перебільшеним, адже ностальгія може спотворювати сприйняття минулого. Подібно до ситуації, коли людина відчуває приємну тугу за часами, коли вона не була присутньою. Така романтизація минулого може призвести до переоцінки простоти та безтурботності 2D ігор, що, у свою чергу, може призвести до певних очікувань щодо сучасних ігор.

Варто також зазначити, що 2D ігри мають свій особливий шар культурного значення, які вони здобули з часом. Вони стали культовими символами геймінгу і залишаються такими й досі. Їх простота та невимушеність привертають до себе нових гравців навіть у сучасній ері ігрової індустрії.

Ще одна перевага 2D ігор полягає в їх простоті управління. Оскільки гра працює в двовимірному просторі, вона зазвичай пропонує менше варіантів переміщення, що робить управління героями більш простим та зрозумілим для гравців.

Також важливо відзначити, що для запуску 2D ігор не потрібно мати дуже потужний комп’ютер або ігрову консоль. Це забезпечує більшу доступність для гравців з менш потужним обладнанням.

Нарешті, 2D ігри можуть бути способом втечі від агресії та жорстокості, які часто присутні у 3D іграх. Вони створюють безпечне середовище, де відсутні

насильство та жорстокі сцени, що робить їх більш привабливими для широкої аудиторії.

1.4 Специфікація вимог до гри в жанрі 2D платформер

Призначення системи: створення захоплюючого ігрового середовища, в якому гравці можуть насолоджуватися захоплюючими пригодами, стрибками та вирішенням головоломок у двовимірному світі.

Сфера застосування: розваги, відпочинок, розвиток реакції та просторового мислення.

Характеристики користувачів: користувачі будь-якого віку, які мають смартфон або планшет із підтримкою сенсорного введення.

Функції системи:

– Система головного меню, що містить список рівнів, налаштування та досягнення;

– система управління персонажем, яка реагує на торкання екрану для руху, стрибків та взаємодії з об’єктами;

– система рівнів з різноманітними платформами, перешкодами та ворогами;

– система збору предметів та бонусів;

– система збереження та завантаження прогресу гри;

– система анімації та візуальних ефектів;

– система досягнень та лідербордів;

– система магазину для придбання додаткових предметів та покращень;

– система звуку та музичного супроводу.

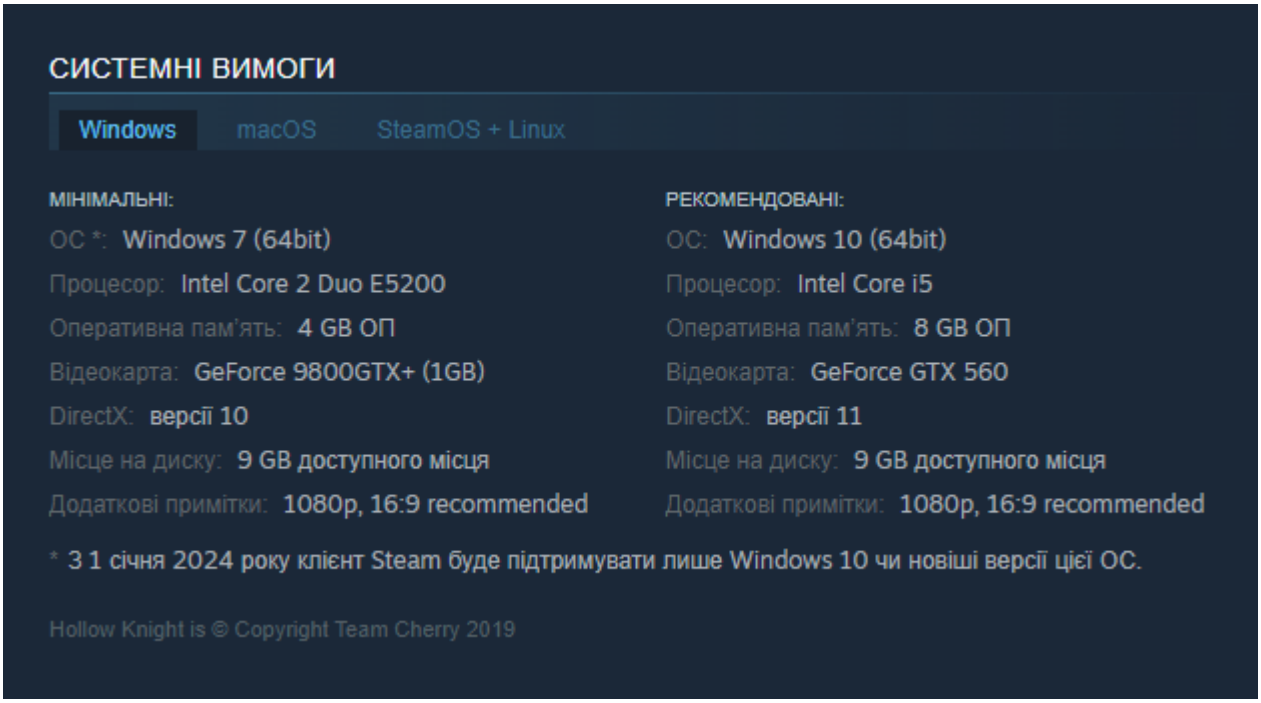

Рисунок 1.14 – Приклад вимог для гри Hollow Knight

Вимоги до технічного забезпечення:

– Операційна система: Android або iOS;

– процесор: підтримка сенсорних операцій, без особливих вимог до характеристик;

– вільне місце на пристрої: від 100 МБ;

– оперативна пам’ять: від 1 ГБ;

– мінімальна версія ОС: Android 4.4 або iOS 9.0.

Висновки до розділу 1

У розділі 1 було проведено аналіз різних жанрів ігрових застосунків, де були виокремлені три аналоги гри-застосунку. Кожен з них був проаналізований окремо, визначено їх основні риси, такі як розробник, видавник, мова випуску, перелік функцій і т. д. Також були виявлені переваги та недоліки кожного ігрового застосунку. У завершальній частині розділу 1 була сформульована специфікація вимог для мобільної гри в жанрі платформер на платформі Unity.

2 МОДЕЛЮВАННЯ ТА ПРОЄКТУВАННЯ ГРИ В ЖАНРІ ПЛАТФОРМЕР

2.1 Створення Usecase

Використання системи може бути описане за допомогою текстових сценаріїв, які вказують на можливі результати взаємодії користувача з системою для досягнення певних мет цілей. Сценарій - це послідовність кроків, які користувач виконує в системі для виконання певних операцій. Написання варіантів використання дозволяє чітко визначити ролі користувачів, їхні дії та цілі використання системи. Вони визначають функціональні та поведінкові вимоги до системи, показуючи, що саме вона має робити. Є три форми опису:

– Короткий - це звичайно один абзац, що описує коротко один із сценаріїв, який, як правило, приводить до успішного результату.

– Поверхневий - це загальний огляд усіх сценаріїв у вільній формі (основний та альтернативний) одного з варіантів використання.

– Повний - детально описуються всі кроки та дії, включаючи попередні та післяумови, необхідні для виконання сценарію. Варіанти використання вибираються з повного списку на етапі аналізу, зокрема коротких та поверхневих описів, для вирішення конкретних проблем, критичних для функціонування системи, наприклад, використання касових апаратів.

Таблиця 2.1 – Повна форма «Створення рівня»

Usecase section	Comment
Use Case 1:	Створення Нового Рівня
Scope:	Система розробки гри на Unity
Level:	User-goal
Primary Actor:	Розробник гри

Закінчення таблиці 2.1 – Повна форма «Створення рівня»

Stakeholders and Interests:	Розробник гри: Зацікавлений у створенні нового рівня з визначеними характеристиками та об’єктами. Команда розробників: Може мати інтерес у спільному редагуванні та рецензуванні нового рівня. Preconditions: Користувач ввійшов у систему розробки гри та обрав опцію «Створити новий рівень».
Success Guarantee:	Новий рівень успішно доданий до проекту та може бути відредагований.
Main Success Scenario:	– Розробник вибирає опцію «Створити новий рівень»; – обирає розмір та фоновий малюнок для нового рівня; – розміщує об’єкти, перешкоди та персонажів на сцені; – налаштовує параметри об’єктів (розмір, швидкість, поведінка); – зберігає новий рівень у проекті.
Extensions:	Розробник вирішує не зберігати створений рівень та виходить із режиму редагування.

2.2 Діаграми варіантів використання

Діаграма використання – це графічне зображення взаємодії між системою та її користувачами. Вона відображає різні варіанти використання системи (юзкейси) та акторів (користувачів або інші системи), які здійснюють ці варіанти використання. Діаграма використання надає високорівневий огляд функціональності системи, її можливостей та взаємодії з користувачами (рис. 2.1).

Основні компоненти діаграми використання:

– Юзкейси (Use Cases): Це функціональні можливості системи, які використовуються для досягнення конкретних цілей користувачів або інших систем. Кожен юзкейс представляє собою конкретний сценарій використання системи.

– Актори (Actors): Це сутності, які взаємодіють з системою. Актори можуть бути користувачами, зовнішніми системами або іншими чинниками, які викликають або сприймають дії в системі.

– Відношення між юзкейсами та акторами: Вказують, які актори взаємодіють з якими юзкейсами. Це допомагає визначити, хто виконує певні дії в системі.

Діаграми використання використовуються для наступних цілей:

– Уточнення вимог до системи: Вони допомагають в ідентифікації та уточненні функціональних вимог до системи шляхом визначення можливих варіантів використання.

– Визначення області впливу системи: Вони дозволяють визначити, які актори взаємодіють з системою та які функціональні можливості вони мають доступ до.

– Комунікація з зацікавленими сторонами: Діаграми використання можуть використовуватися для спілкування з зацікавленими сторонами та командою розробників для розуміння та узгодження вимог до системи.

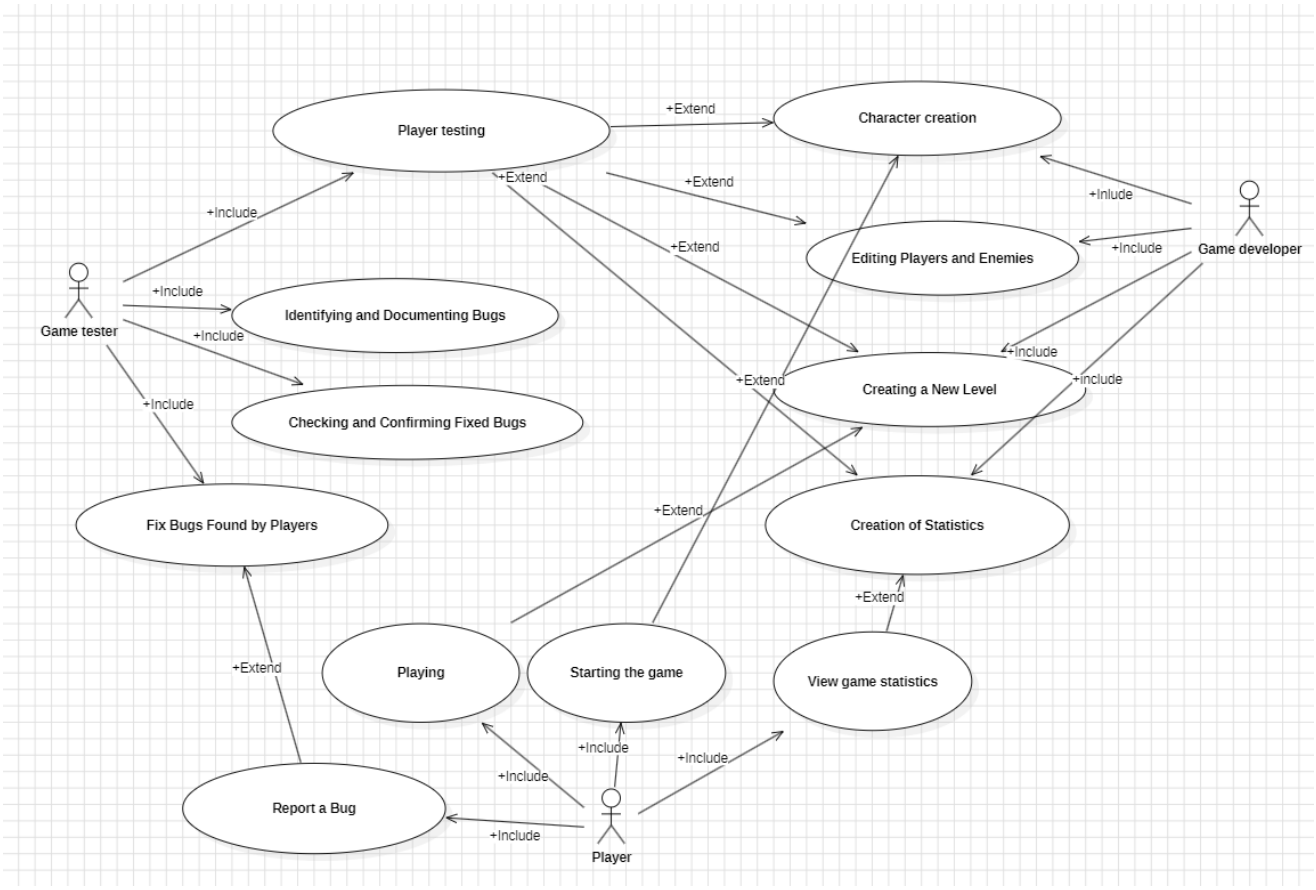

Рисунок 2.1 – Діаграма варіантів використання

Ця діаграма представляє взаємодію між трема основними акторами у процесі розробки та тестування гри: розробником, тестером та гравцем.

– Розробник відповідає за створення основних елементів гри, таких як рівні, вороги, гравець та збір статистики.

– Тестер перевіряє готовність гри, тестуючи її функціонал, документуючи знайдені баги і перевіряючи виправлені проблеми. Він також відповідає за фіксацію та перевірку багів, які знаходять гравці.

– Гравець запускає гру, грає в неї, а потім дивиться на статистику своєї гри. Якщо гравець знаходить баги або проблеми під час гри, він повідомляє розробникам про це.

2.3 Діаграма класів

Діаграма класів - це структурна діаграма, що відображає класи системи програмного забезпечення, їх атрибути та методи, а також взаємозв’язки між

ними (рис. 2.2). Вона є важливим інструментом при проектуванні програмного забезпечення, оскільки дозволяє моделювати архітектуру системи та визначити способи взаємодії між її компонентами.

Основна мета діаграми класів - це візуалізація структури програми та взаємозв’язків між її складовими частинами. Вона допомагає розробникам отримати загальне уявлення про систему, її класи та їх взаємозв’язки, що полегшує розуміння коду, сприяє аналізу та виявленню можливих проблем.

Діаграма класів (рис. 2.2) також використовується для комунікації між членами розробницького колективу, оскільки вона надає зручний спосіб візуалізації архітектурних концепцій та обміну ідеями щодо організації програмного забезпечення.

Отже, діаграма класів є потужним інструментом при аналізі, проектуванні та розробці програмного забезпечення, оскільки дозволяє систематизувати інформацію про структуру системи та зручно візуалізувати її складові частини.

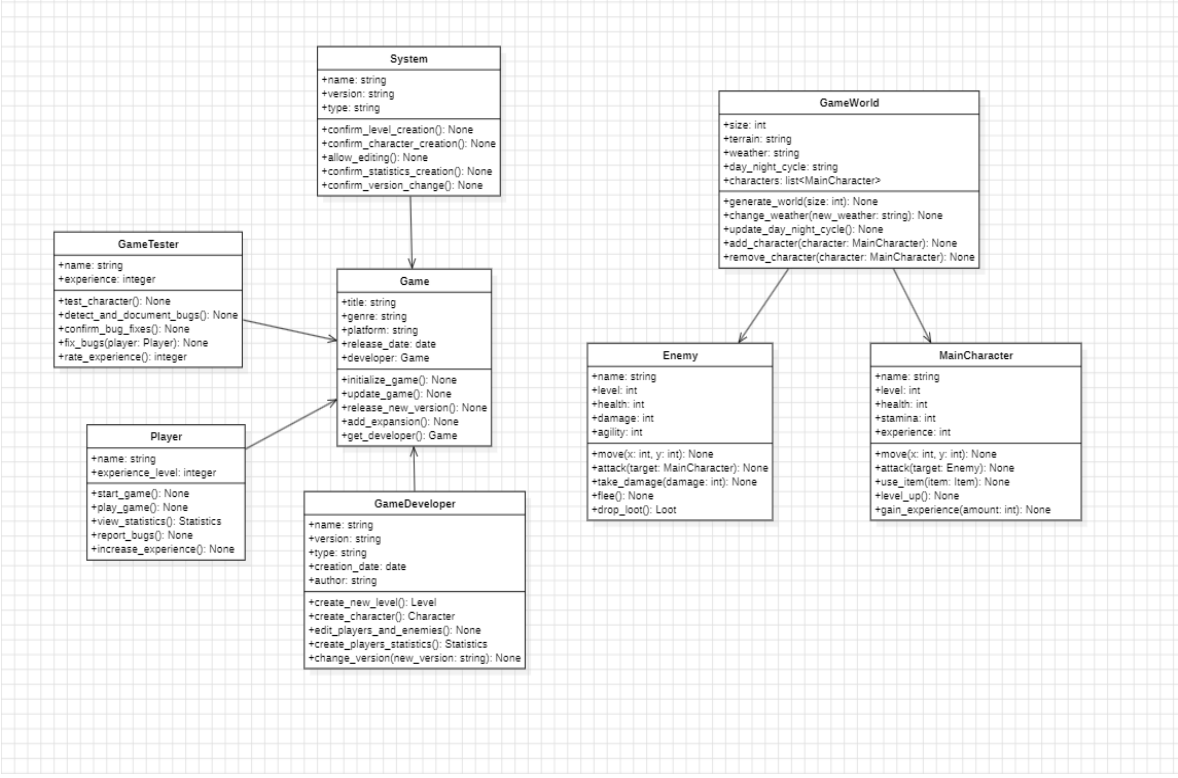

Рисунок 2.2 – Діаграма класів

2.4 Діаграми станів та переходів

Діаграма станів та переходів - це графічне зображення конкретного об’єкту або системи та всіх можливих станів, в яких він може перебувати, разом з усіма можливими переходами між цими станами (рис. 2.3). Вона використовується для моделювання поведінки системи або об’єкту, який може переходити з одного стану в інший в залежності від внутрішніх або зовнішніх подій.

Діаграма станів та переходів відображається у вигляді графа, де вузли представляють стани, а стрілки - можливі переходи між ними (рис. 2.3), (рис. 2.4). Кожен стан може мати власні властивості або дії, які відбуваються в цьому стані. Переходи між станами відбуваються при виникненні певних подій або умов, які визначаються моделлю.

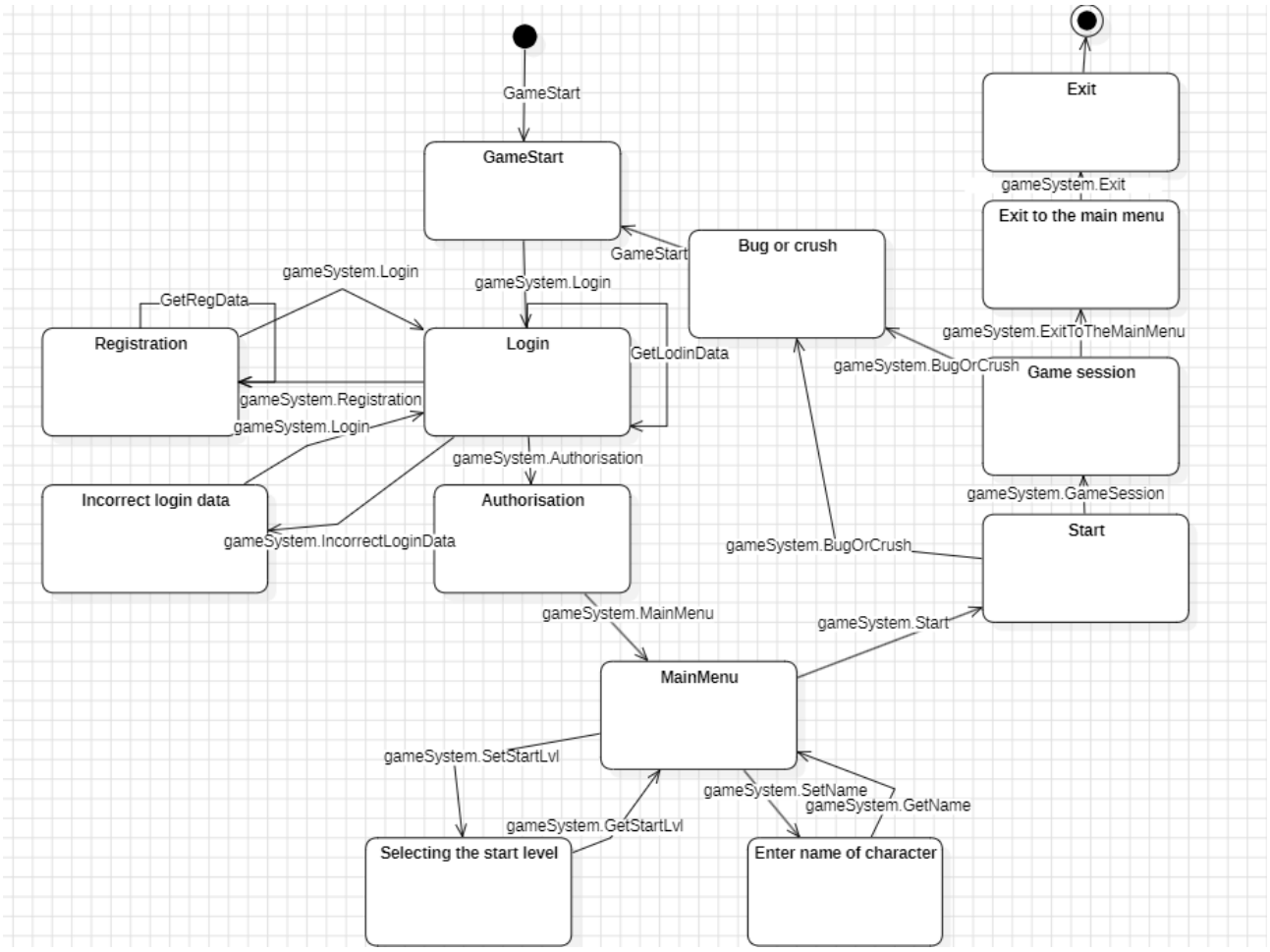

Рисунок 2.3 – Діаграма стану

Діаграми станів та переходів широко використовуються в різних галузях, таких як розробка програмного забезпечення, системи керування, автоматизація виробництва тощо. Вони дозволяють розробникам аналізувати та визначати поведінку системи, виявляти можливі проблеми та оптимізувати її роботу. Також ці діаграми є потужним інструментом для комунікації між членами розробницького колективу, оскільки вони надають зручний спосіб візуалізації і взаєморозуміння поведінки системи.

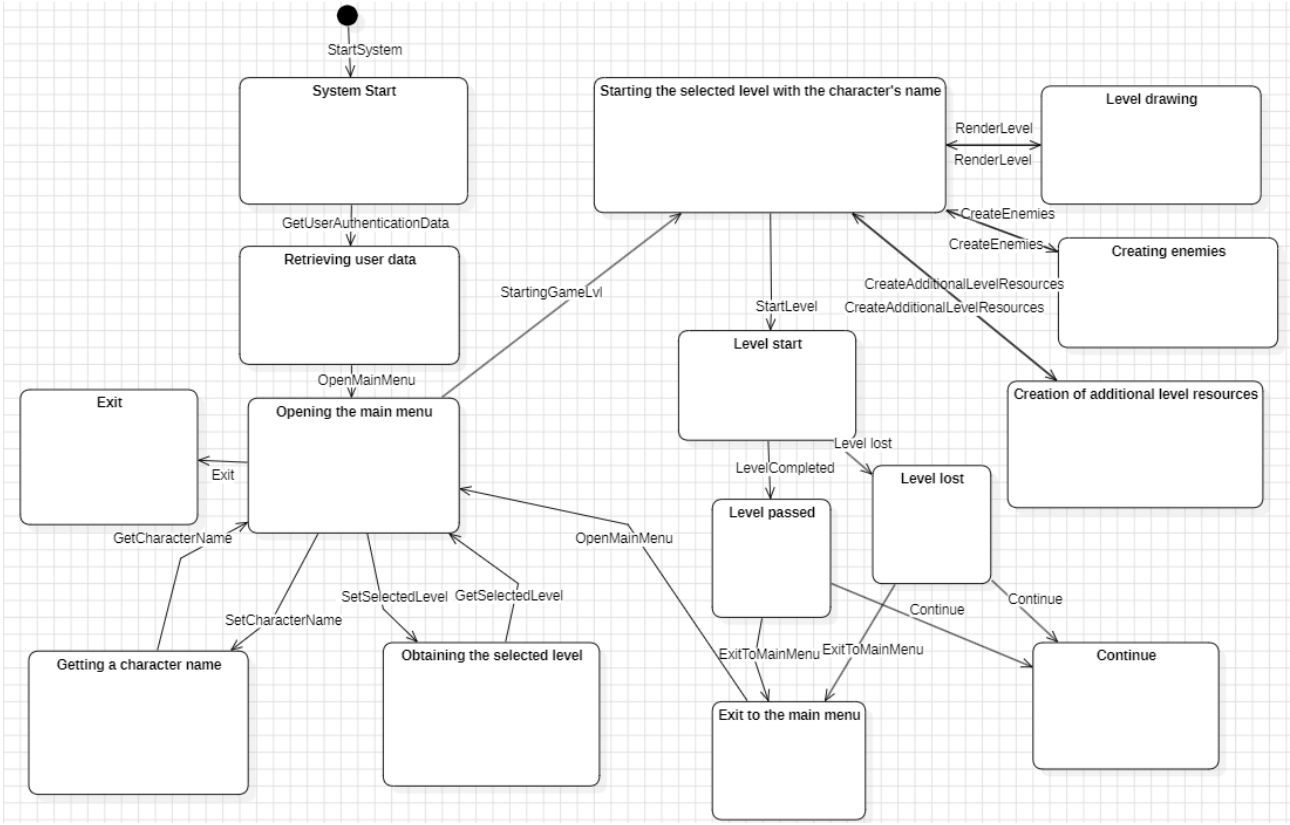

Рисунок 2.4 – Діграма переходів

2.5 Сценарій використання

Сценарій використання (use case) — це опис взаємодії користувача з системою для досягнення певної мети (рис. 2.5). Сценарії використання допомагають розробникам зрозуміти, як користувачі взаємодіють із продуктом або системою.

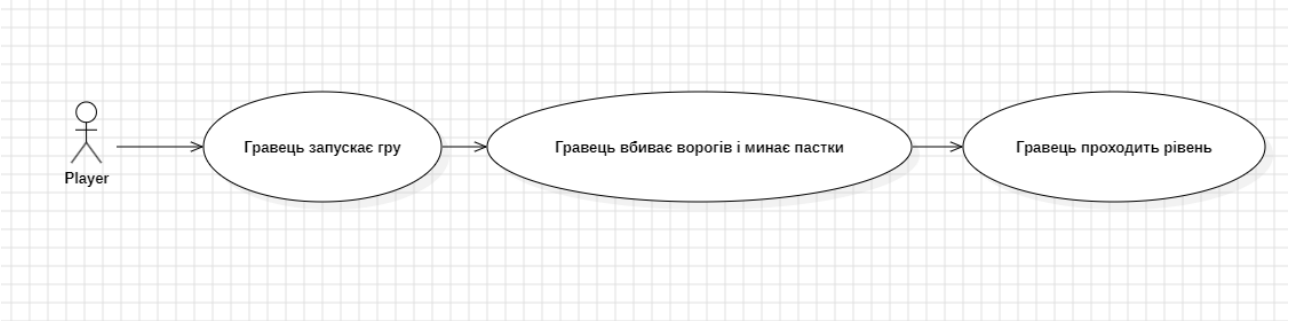

Рисунок 2.5 – Сценарій використання

Цей сценарій використання описує конкретний процес від початку до кінця, що допомагає розробникам створювати функціонал, який відповідає потребам користувачів.

Висновки до розділу 2

У розділі 2 було проведено створення та аналіз діаграм. Отримано цінний інструмент для аналізу та моделювання взаємодії між системою та її користувачами. Діаграми використання допомагають уточнити вимоги до системи, визначити ролі акторів та ідентифікувати функціональні можливості системи. Вони створюють базу для подальшого проектування системи та сприяють зрозумінню її функціональності як для розробників, так і для зацікавлених сторін.

3 ІНСТРУМЕНТАЛЬНІ ЗАСОБИ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Платформа .NET та мова програмування C#

Платформа .NET та мова програмування C# є надзвичайно потужними інструментами для створення різноманітних програм, включаючи 2D ігри. Платформа .NET широко використовується серед програмістів завдяки своїм перевагам. Основою платформи є середовище виконання Common Language Runtime (CLR), яке забезпечує виконання коду, написаного на різних мовах програмування, включаючи C#. CLR відповідає за управління пам’яттю, збірку сміття, безпеку та інші аспекти виконання програм.

Мова програмування C# є однією з найпопулярніших для розробки програмного забезпечення, включаючи ігри. Вона поєднує в собі простоту використання з потужними можливостями і є статично типізованою, що дозволяє виявляти помилки на етапі компіляції, що полегшує відлагодження програм. Також вона підтримує об’єктно-орієнтований підхід до програмування, що сприяє організації коду у логічні блоки.

Платформа .NET має широку підтримку та велику базу розробників, що активно використовують цю платформу і мови програмування, такі як C#. Це забезпечує швидкий розвиток навичок програмування та можливість обміну досвідом у спільноті.

Однією з найважливіших переваг платформи .NET є її масштабованість, яка дозволяє використовувати її для розробки як невеликих, так і великих проєктів, включаючи 2D ігри різної складності. Тут можна використовувати різні бібліотеки та фреймворки, такі як MonoGame або Unity, які надають широкі можливості для роботи з графікою, анімацією, фізикою та іншими аспектами гри.

Крос-платформеність платформи .NET дозволяє розробляти 2D ігри та запускати їх на різних операційних системах, таких як Windows, macOS та Linux,

що розширює аудиторію ігрових користувачів та сприяє поширенню гри на різних платформах.

Платформа .NET також має багато інструментів для підтримки розробки ігор, таких як Visual Studio[1], яке надає потужні можливості для створення, відлагодження та тестування програм на базі .NET. Крім того, існують численні сторонні бібліотеки та ресурси, які полегшують процес розробки ігор на платформі .NET (рис. 3.1).

Узагальнюючи, платформа .NET та мова програмування C# є відмінними виборами для розробки 2D ігор, оскільки вони надають потужність, швидкість розробки, масштабованість та крос-платформену підтримку. З їхньою допомогою можна створювати вражаючі ігри, використовуючи різноманітні інструменти та ресурси, доступні в цьому середовищі.

Рисунок 3.1 – Логотип .NET

Мова програмування C# є однією з найпопулярніших мов для розробки програмних продуктів, у тому числі ігор. Вона має безліч переваг, які роблять її привабливим вибором для створення 2D ігор. Однією з ключових переваг C# є її простота і зрозумілість. Синтаксис C# логічний і схожий на інші мови програмування, що робить її легко засвоюваною, особливо для початківців у галузі розробки ігор.

Ще одна перевага C# - це його масштабованість та підтримка об’єктно-орієнтованого програмування (ООП). Це дозволяє створювати структурований та модульний код, що полегшує розробку та підтримку гри.

Крім того, C# має потужну систему типів і автоматичне управління пам’ятю, що сприяє уникненню багатьох типових помилок, пов’язаних з управлінням пам’ятю, і знижує ризик виникнення помилок у великих проєктах.

Для розробки 2D ігор мова C# має розширені функції інтегрованих розробничих середовищ (IDE) та інструментів, таких як Visual Studio, що надають розширений набір інструментів для аналізу коду, налагодження та автоматизованого тестування. Це сприяє вдосконаленню продуктивності та оптимізації роботи гри.

Крім того, мова C# має розширені фреймворки та бібліотеки, такі як Unity[2], які спеціально розроблені для роботи з графікою, анімацією та фізикою, що полегшує процес створення гри.

Інтеграція мови C# з іншими інструментами та технологіями Microsoft, такими як Windows Forms, WPF і UWP, дозволяє створювати інтерфейс користувача для гри, використовуючи широкий набір елементів керування та стильових можливостей.

Загалом, мова C# є потужною та гнучкою мовою програмування, яка має безліч переваг для розробки 2D ігор. Вона поєднує простоту синтаксису, підтримку ООП, розширені функціональні можливості розробничих середовищ та багато іншого, що дозволяє ефективно працювати над проєктами та забезпечує якість та гнучкість у розробці.

3.2 Середовище розробки Visual Studio

Середовище розробки Visual Studio вважається одним з найпопулярніших і потужних інструментів для створення програмного забезпечення, включаючи 2D ігри (рис. 3.2). Воно пропонує широкий спектр функціональних можливостей та інструментів, що робить його перевагою перед іншими середовищами розробки.

По-перше, Visual Studio надає потужну підтримку мови програмування C#, яка є однією з найбільш поширених мов для розробки ігор. Інтегрована

підтримка C# в Visual Studio дозволяє легко створювати, редагувати, налагоджувати та відлагоджувати код гри, використовуючи багаті функціональні можливості мови, такі як об’єктно-орієнтоване програмування, делегати, події та інші.

Visual Studio надає потужні інструменти для управління проектом та збірки, дозволяючи створювати різноманітні типи проектів, такі як бібліотеки класів, модулі, компоненти та інші, що сприяє організації проекту розробки 2D гри.

Одним з важливих переваг Visual Studio є його потужна система налагодження, яка дозволяє ефективно відлагоджувати гру, виявляти та виправляти помилки, перевіряти значення змінних тощо.

Крім того, Visual Studio має інтеграцію з різноманітними фреймворками та бібліотеками, які широко використовуються в галузі розробки ігор, такими як Unity. Це дає можливість використовувати потужні функції цих фреймворків для створення 2D ігор.

Також важливо відзначити активну спільноту розробників, яка підтримує Visual Studio і надає безліч ресурсів, форумів, підручників, блогів та інших джерел для отримання допомоги, відповідей на запитання та порад від досвідчених розробників.

Загалом, використання Visual Studio для розробки 2D ігор має безліч переваг, від широкого набору функціональних можливостей та інструментів до потужної системи налагодження та інтеграції з різноманітними фреймворками. Це середовище розробки надає всі необхідні інструменти для створення високоякісних 2D ігор.

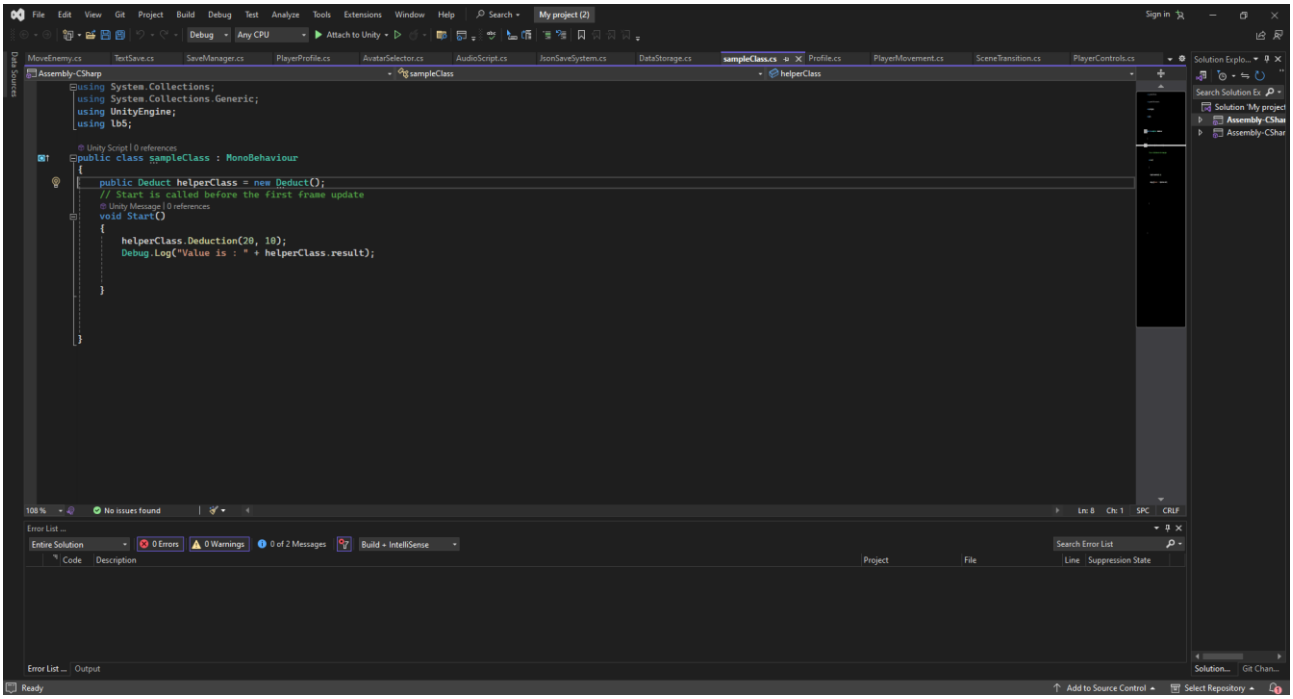

Рисунок 3.2 – Інтерфейс Visual Studio

3.3 Рушій для розробки відеоігор Unity

Unity вважається одним з найпопулярніших рушіїв для розробки ігор і має безліч переваг, особливо у сфері створення ігор. Ось кілька ключових переваг Unity (рис. 3.3):

Крос-платформеність: Unity дозволяє розробляти ігри, які працюють на різних платформах, включаючи Windows, macOS, Linux, iOS, Android та інші. Це забезпечує широке охоплення аудиторії і дозволяє розповсюджувати гру на різних пристроях без необхідності переписування коду.

Готові компоненти та ресурси: Unity надає велику кількість готових компонентів, ресурсів та інструментів, які полегшують процес створення гри. Наприклад, ви можете використовувати готові 2D фізичні рушії, системи анімації, системи частинок, системи колізій та інші, що дозволяє прискорити процес розробки та створити професійні графічні ефекти.

Легкість використання: Unity має дружній інтерфейс користувача, який дозволяє легко розуміти та працювати з усіма аспектами розробки гри. Його інтерфейс складається зі зручних панелей, вікон та редакторів, що дозволяє

візуально налаштовувати графічні об’єкти, налаштовувати фізику, створювати скрипти, управляти сценами та багато іншого.

Мови програмування: Unity підтримує кілька мов програмування, включаючи C#, яка є однією з найпоширеніших мов у галузі розробки ігор. Використання C# дозволяє вам використовувати всі потужні можливості цієї мови для створення складної логіки гри та взаємодії з іншими системами Unity.

Велика спільнота розробників: Unity має широку та активну спільноту розробників, яка надає поради, розв’язує проблеми та надає велику кількість ресурсів для розвитку навичок в розробці гри.

Рисунок 3.3 – Логотип Unity

Unity є потужним та гнучким інструментом для розробки ігор, який надає всі необхідні ресурси для реалізації творчих ідей. Відмінні графічні можливості, підтримка багатьох платформ, різноманітність інструментів для роботи з анімаціями, звуком та інтерфейсом, а також велика спільнота розробників роблять Unity чудовим вибором як для початківців, так і для досвідчених розробників.

Після встановлення Unity постає необхідність встановити одну із запропонованих його версій, а після цього створити новий проєкт.

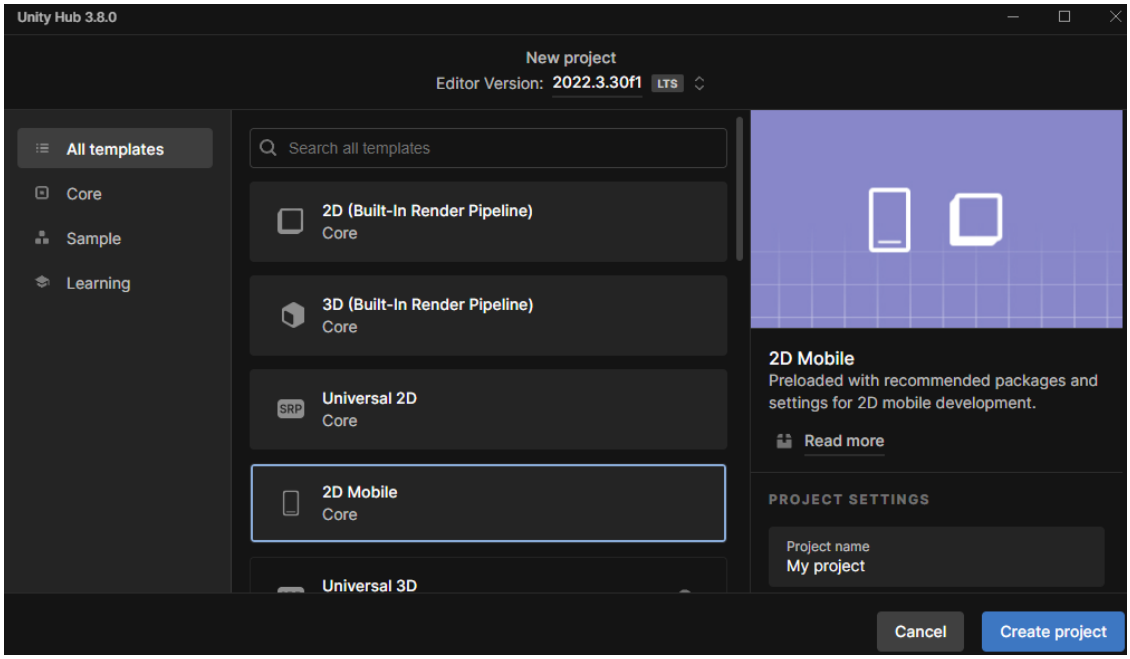

Рисунок 3.4 – Вікно створення порожнього застосунку для рушія Unity

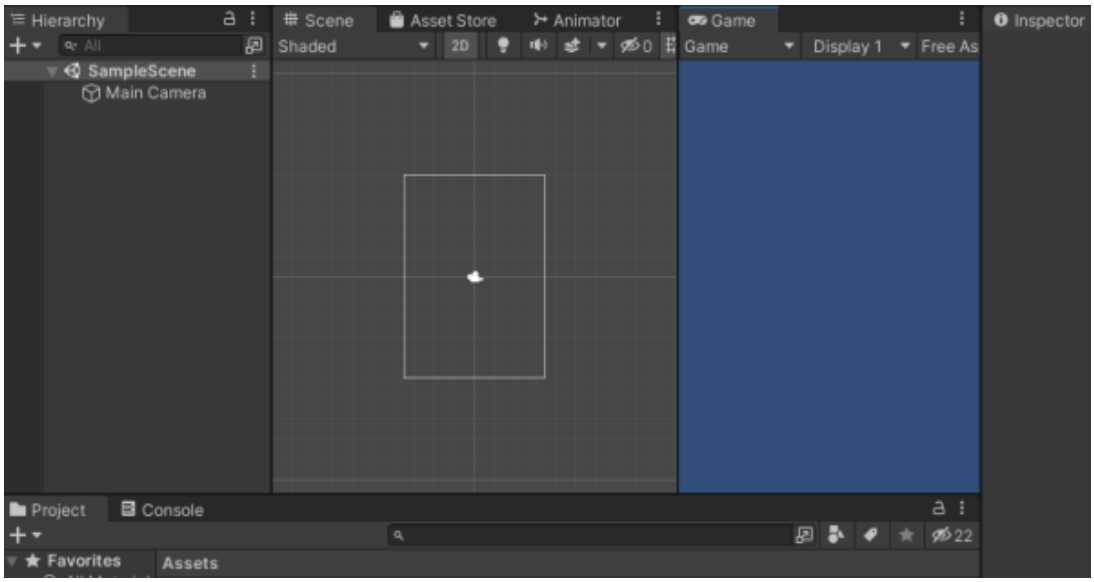

Рисунок 3.5 – Вікно рушія Unity

Спочатку вибираємо шаблон гри, та придумаємо назву для проекту (не самого додатку) і обрати папку, де буде зберігатися гру (рис. 3.4). Після цього, у в Unity відкритому проекті, бачимо кілька основних вікон, піктограм та параметрів (рис. 3.5).

– Hierarchy: Зазвичай тут відображається довгий список усіх ігрових об’єктів у сцені. Це полегшує пошук конкретного об’єкта для зміни його властивостей або додавання компонентів.

– Scene: Це найбільше вікно, зазвичай розташоване посередині, де можна переглядати поточний рівень гри або меню. Тут можна вільно переміщати, масштабувати та змінювати розміри об’єктів з ієрархії.

– Game: У цьому вікні бачимо гру з точки зору камери, і не можемо переміщувати об’єкти.

– Asset Store: Це вікно містить магазин активів для розробників.

– Inspector: Після вибору об’єкта з ієрархії, в цьому вікні можна змінювати його властивості, додавати скрипти, змінювати розмір, переміщувати та додавати готові компоненти.

– Project: Це вікно зазвичай розташоване внизу та містить всі наявні сцени, звуки, папки, зображення, анімації тощо. Тут можна створювати нові сценарії C#.

– Console: Тут показуються помилки, попередження та інші повідомлення від редактора.

3.4 Створення сцен, скриптів та об’єктів в Unity

Процес створення нової сцени в Unity полягає у переході до меню «File», де користувач вибирає «New Scene» (рис. 3.6). Сцена є основою для розробки ігор в Unity і може містити різні об’єкти, такі як герої, вороги, декорації, освітлення, звуки та інші елементи, необхідні для створення ігрового середовища.

Додавання об’єктів до сцени є важливим аспектом розробки ігор у цьому рушії. Об’єкти можна створювати вручну, імпортувати з зовнішніх джерел або брати з вбудованого магазину «Asset Store». Для створення нового об’єкта потрібно перейти в меню «GameObject» і вибрати тип об’єкта.

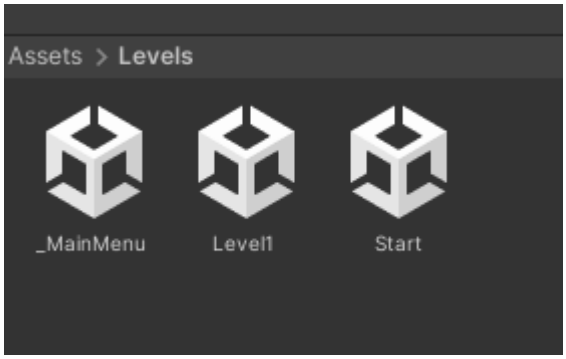

Рисунок 3.6 – Сцени Unity

Об’єкти в Unity мають різні властивості, які можна налаштовувати за допомогою інспектора. Властивості включають параметри, такі як позиція, розмір, колір, текстура, матеріал, фізичні властивості та інші.

Успішна розробка гри вимагає не лише створення та розміщення об’єктів, але й контролю за їхньою поведінкою. Для цього потрібно писати скрипти, які визначають, як об’єкти взаємодіють один з одним та з ігровим середовищем.

Однією з фундаментальних частин розробки в Unity є програмування. Unity використовує мову програмування C# для створення скриптів, що контролюють поведінку об’єктів в ігровому середовищі.

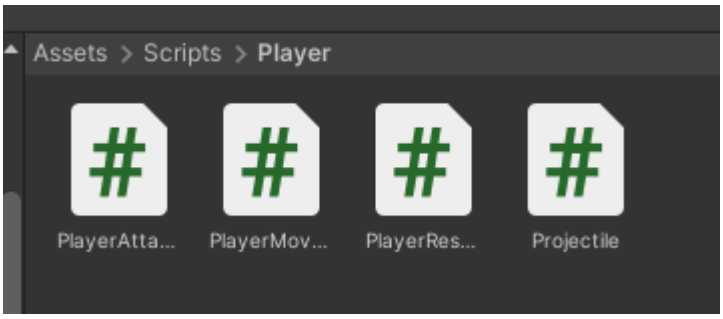

Рисунок 3.7 – C# скрипти

C# є об’єктно-орієнтованою мовою програмування, яка є популярною в індустрії розробки ігор завдяки своїй гнучкості, потужності та зрозумілості. За допомогою цієї мови розробники можуть створювати високоякісні скрипти для взаємодії з різними компонентами гри.

Процес програмування в Unity починається зі створення нового скрипту через меню «Assets», «Create», а потім «C# Script» (рис. 3.7). Новий скрипт можна

приєднати до об’єкта в ігровій сцені, перетягнувши його на об’єкт у вікні Ієрархії або через Інспектор.

Усередині скрипта визначаються методи, які представляють дії, що виконуються об’єктом. Наприклад, методи «Start» та «Update» використовуються для виконання коду на етапах початку гри та під час кожного кадру відповідно.

3.5 Методика тестування ігор

Тестування ігор - це процес перевірки функціональності, геймплею, графіки та інших аспектів гри з метою виявлення помилок, удосконалення гри та забезпечення високої якості продукту. Тестування також поділяється на різні принципи тестування (рис. 3.8).

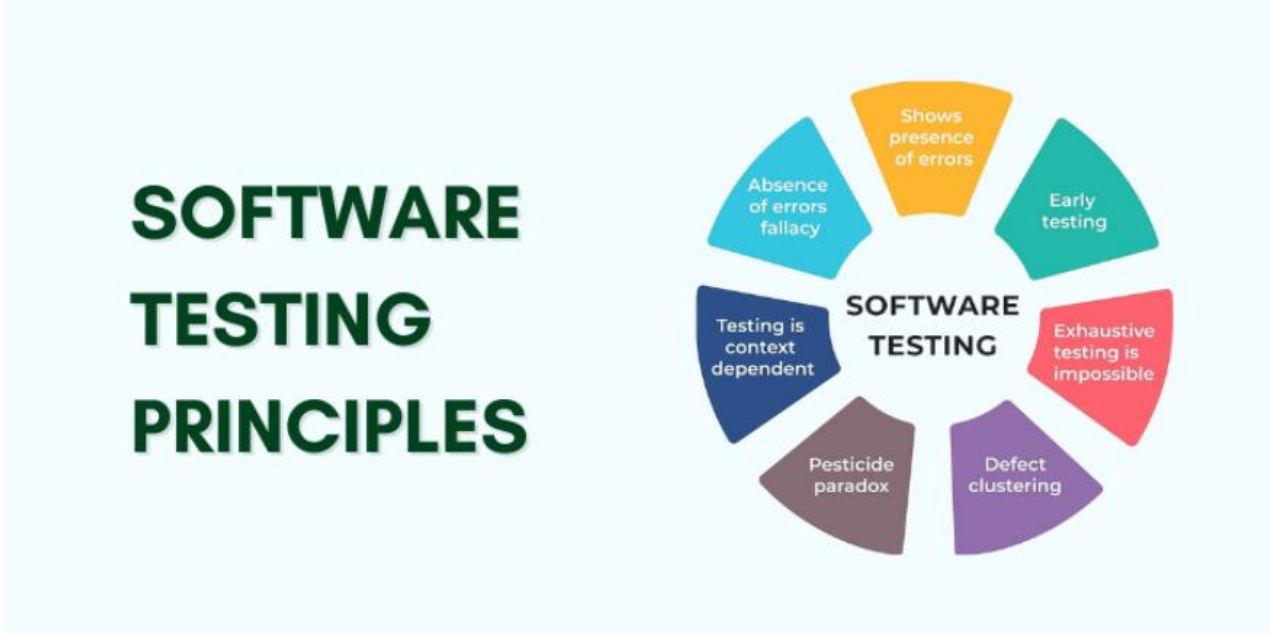

Рисунок 3.8 – Принципи тестування

1. Функціональне тестування:

– Перевірка основних функцій гри, таких як керування персонажем, здібність взаємодії з оточенням, виконання завдань тощо.

– Тестування функціональності може включати перевірку меню, налаштувань, інвентаря, системи здоров’я тощо.

2. Тестування геймплею:

– Оцінка загального враження від гри, балансу між рівнем складності та задоволенням, емоційної залученості гравця.

– Перевірка рівнів, сценаріїв, квестів та інших елементів, які впливають на геймплей.

3. Тестування сумісності:

– Випробування гри на різних пристроях (комп’ютерах, консолях, мобільних пристроях) та операційних системах.

– Перевірка взаємодії гри з різними конфігураціями обладнання та драйверами.

4. Тестування мережі:

– Перевірка стабільності мережевого з’єднання та затримок при онлайн-грі.

– Тестування обміну даними між гравцями, синхронізації подій тощо.

5. Тестування на відповідність вимогам:

– Перевірка того, чи відповідає гра встановленим вимогам до функціональності, продуктивності, сумісності тощо.

– Виконання тестування на різних етапах розробки для забезпечення відповідності вимогам замовника та стандартам галузі.

6. Тестування безпеки:

– Виявлення та виправлення потенційних вразливостей, які можуть бути використані для злому гри або порушення приватності гравців.

– Тестування на наявність вразливостей у вбудованих системах захисту, таких як антивіруси та фаєрволи.

7. Тестування локалізації:

– Перевірка коректності перекладів гри на різні мови та адаптації інтерфейсу до місцевих культурних особливостей.

– Оцінка зрозумілості та прийнятності перекладів для місцевих аудиторій.

8. Тестування продуктивності:

– Вимірювання продуктивності гри на різних пристроях та конфігураціях обладнання.

– Виявлення проблем з витратою ресурсів (пам’яті, процесора, графіки) та оптимізація для підвищення продуктивності.

9. Тестування стабільності:

– Перевірка стійкості гри до збоїв, витоку пам’яті, падінь чи інших проблем, що можуть вплинути на досвід гравця.

– Виявлення та виправлення програмних помилок та вразливостей, які можуть призвести до некоректної роботи гри.

Ці методики можуть використовуватися окремо або в поєднанні залежно від конкретних вимог та особливостей гри. Важливо, щоб тестування було систематичним, і проводилося на різних етапах розробки, щоб виявляти помилки на ранніх стадіях та запобігти їх поширенню в кінцевій версії гри.

Висновки до розділу 3

У розділі 3 було проведено аналіз ключових інструментів та технологій для розробки ігор, таких як .NET (зокрема мова програмування C#), Visual Studio та Unity. Мова програмування C#, забезпечуючи простоту синтаксису та підтримку об’єктно-орієнтованого програмування, є ключовою для розробки ігор у середовищі .NET. Visual Studio, як потужне середовище розробки, сприяє зручності та продуктивності завдяки інтегрованій підтримці мови C#, інструментам управління проєктом та системі налагодження. У той же час Unity вирізняється своєю крос-платформеністю, готовими компонентами та ресурсами, легкістю використання та широкою спільнотою розробників.

Розглянувши ці три ключові компоненти, можна визначити, що вибір залежить від потреб та специфіки проєкту. Використання .NET з мовою C# в Visual Studio може бути ідеальним для більш традиційних, складних проєктів, де важлива продуктивність та масштабованість. З іншого боку, Unity забезпечує швидкість початку роботи, зручність у використанні та готові компоненти, що робить його привабливим для швидкого прототипування та розробки ігор.

Отже, на основі характеристик та переваг кожного з інструментів можна ефективно вибрати найбільш підходящий для конкретного проєкту, забезпечуючи успішну та ефективну розробку гри.

4 ПРОГРАМНА РЕАЛІЗАЦІЯ ІГРОВОГО ЗАСТОСУНКУ

4.1 Реалізація головного меню

Для створення головного меню з двома кнопками «Грати» та «Вихід», додавання бекграунду та тексту з назвою гри в Unity, виконуємо наступні кроки (рис. 4.1).

Спершу створюємо нову сцену для головного меню. Зберігаємо цю сцену в папці «Scenes» і називаємо її «MainMenu». Після цього додаємо Canvas, натискаючи правою кнопкою миші в Hierarchy і вибираючи UI > Canvas. Canvas є контейнером для всіх елементів інтерфейсу.

Додаємо бекграунд для головного меню. В Canvas натискаємо правою кнопкою миші, вибираємо UI > Image і називаємо цей елемент «Background». У вікні Inspector для Background вибираємо зображення, яке хочемо використовувати як фон, і налаштовуємо розмір зображення, щоб воно заповнювало весь екран.

Додаємо текст з назвою гри. В Canvas натискаємо правою кнопкою миші, вибираємо UI > Text і називаємо цей елемент «GameTitle». У вікні Inspector вводимо текст, який буде відображати назву гри, та налаштовуємо його шрифт, розмір і колір. Розміщуємо текст у верхній частині екрану.

Додаємо кнопку «Грати». В Canvas натискаємо правою кнопкою миші, вибираємо UI > Button і називаємо цей елемент «PlayButton». У вікні Inspector змінюємо текст кнопки на «Грати». Для цього розгортаємо PlayButton, вибираємо Text і змінюємо текст у вікні Inspector. Розміщуємо кнопку в центрі екрану.

Рисунок 4.1 – Головне меню

Додаємо кнопку «Вихід». Копіюємо PlayButton, вставляємо його і називаємо новий елемент «ExitButton» (рис. 4.2). Змінюємо текст кнопки на «Вихід» аналогічно попередньому кроку. Розміщуємо кнопку під кнопкою «Грати».

Створюємо C# скрипт для керування головним меню і називаємо його «MainMenu» (рис. 4.3).

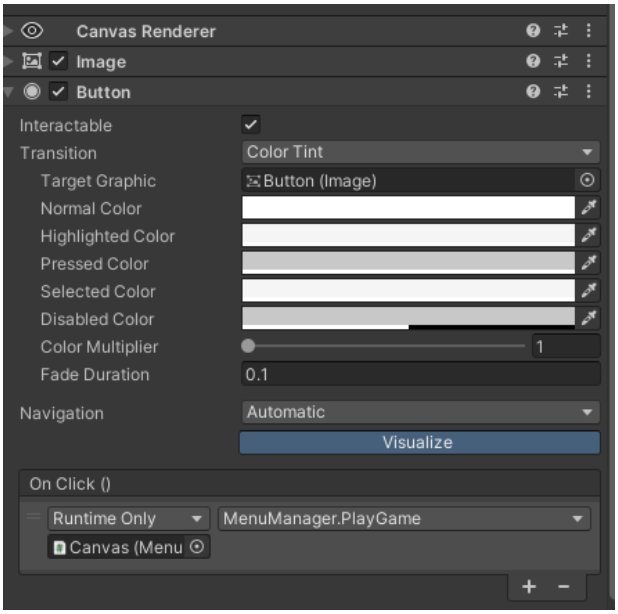

Рисунок 4.2 – Кнопка початку гри

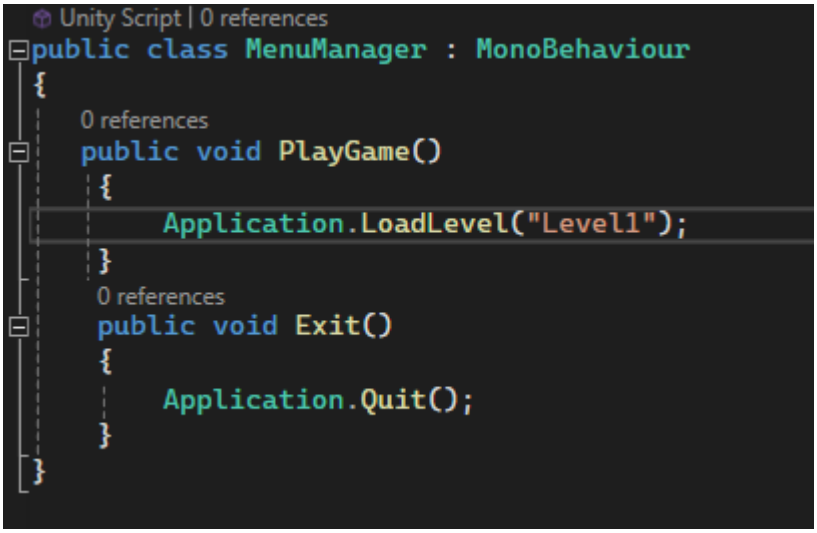

Рисунок 4.3 – MenuManager скрипт

Коли головне меню гри готово, можно переходити до створення рівнів, героя, та ворогів.

4.2 Реалізація керування героєм

Для створення моделі гравця потрібно створити 2D модель гравця чи імпортувати її в проект Unity з завантажених бібліотек (рис. 4.4). Після цього перетягніть модель на сцену. Коли модель знаходиться на сцені, створіть нову папку в проекті, наприклад, «Prefabs», і перетягніть модель зі сцени в цю папку, щоб створити префаб.

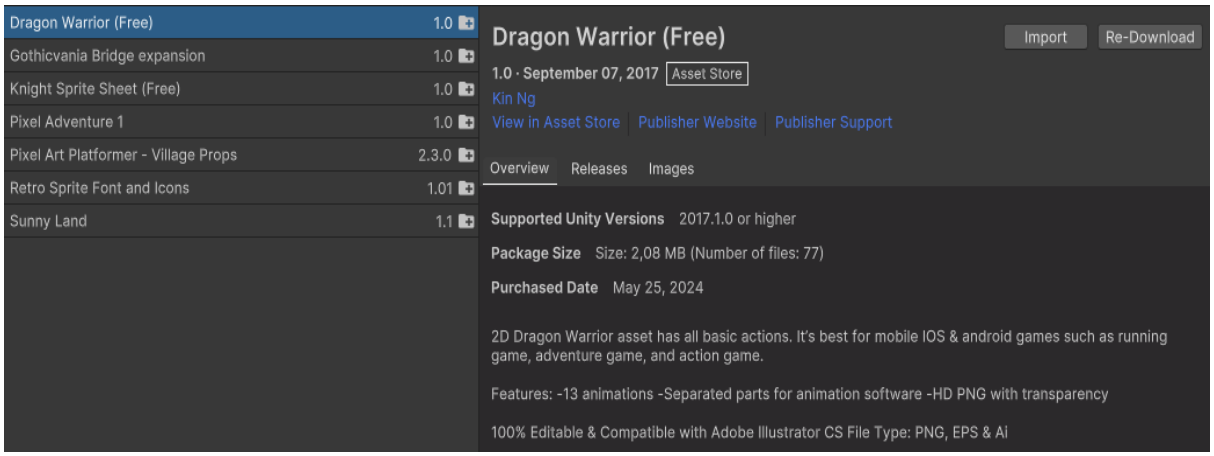

Рисунок 4.4 – Завантажені асети

Асети в Unity - це ресурси, які використовуються для створення гри. Вони можуть включати зображення, звуки, відео, 3D-моделі, тексти, шрифти, анімації

та інше. Асети важливі, оскільки вони дозволяють розробникам створювати та редагувати різноманітні об’єкти та ефекти у грі без необхідності написання коду з нуля або створення ресурсів вручну.

Використання асетів дозволяє розробникам швидко прототипувати ідеї, швидко змінювати вигляд та функціональність гри, а також полегшує спільну роботу розробників та дизайнерів. Крім того, асети можуть бути використані для створення багатьох ефектів, таких як анімація, освітлення та звукові ефекти, що додає грі реалізм та іммерсію.

Рисунок 4.5 – Prefab гравця

Додаємо необхідні компоненти до префабу гравця (рис. 4.5). Вибираємо префаб гравця у вікні сцени. У вкладці Inspector натискаємо Add Component і додаємо компонент Rigidbody, який відповідає за фізику об’єкта.

Налаштовуємо параметри компонента, такі як маса і гравітація, за потребою. Далі додаємо компонент Collider, наприклад, Capsule Collider, для обробки зіткнень. Налаштовуємо розмір і форму колайдера відповідно до моделі гравця. Для анімації додаємо компонент Animator та призначаємо контролер анімації, який відповідає за анімаційні переходи і стану.

```
Unity Message | 0 references
private void Update()
{
    horizontalInput = Input.GetAxis("Horizontal");


    if (horizontalInput > 0.01f)
        transform.localScale = Vector3.one;
    else if (horizontalInput < -0.01f)
        transform.localScale = new Vector3(-1, 1, 1);


    anim.SetBool("run", horizontalInput != 0);
    anim.SetBool("grounded", isGrounded());


    if (Input.GetKeyDown(KeyCode.Space))
        Jump();


    if (Input.GetKeyUp(KeyCode.Space) && body.velocity.y > 0)
        body.velocity = new Vector2(body.velocity.x, body.velocity.y / 2);

    if (onWall())
    {
        body.gravityScale = 0;
        body.velocity = Vector2.zero;
    }
    else
    {
        body.gravityScale = 7;
        body.velocity = new Vector2(horizontalInput * speed, body.velocity.y);

        if (isGrounded())
        {
            coyoteCounter = coyoteTime;
            jumpCounter = extraJumps;
        }
        else
            coyoteCounter -= Time.deltaTime;
    }
```

Рисунок 4.6 – PlayerMovement скрипт

Щоб додати логіку руху, атаки та управління здоров’ям, створюємо новий C# скрипт і називаємо його PlayerMovement (рис. 4.6). У цьому скрипті спочатку створюємо логіку для руху. Для цього обробляємо введення з клавіатури та змінюємо швидкість і напрямок руху гравця, використовуючи компонент Rigidbody. Додаємо поворот гравця у напрямку руху.

4.3 Реалізація атаки гравця

Далі реалізуємо логіку атаки. Додаємо метод для обробки введення з клавіатури або миші для атаки. Використовуємо префаб снаряду або іншого об’єкта атаки, щоб створювати його екземпляри під час атаки, і задаємо початкову швидкість для снаряду (рис. 4.7).

Рисунок 4.7 – Prefab снаряду

Спершу створюємо префаб снаряду. Створюємо новий 2D об’єкт, який буде нашим снарядом, наприклад, сферу. Налаштовуємо розмір і зовнішній вигляд снаряду відповідно до потреб. Додаємо компонент Rigidbody до снаряду для забезпечення фізичної взаємодії. Налаштовуємо параметри Rigidbody, зокрема вимикаємо використання гравітації, якщо це необхідно. Додаємо Animator та Box Collider 2D.

```
private void Awake()
{
    anim = GetComponent<Animator>();
    boxCollider = GetComponent<BoxCollider2D>();
}
Unity Message | 0 references
private void Update()
{
    if (hit) return;
    float movementSpeed = speed * Time.deltaTime * direction;
    transform.Translate(movementSpeed, 0, 0);

    lifetime += Time.deltaTime;
    if (lifetime > 5) gameObject.SetActive(false);
}

Unity Message | 0 references
private void OnTriggerEnter2D(Collider2D collision)
{
    hit = true;
    boxCollider.enabled = false;
    anim.SetTrigger("explode");

    if (collision.tag == "Enemy")
        collision.GetComponent<Health>().TakeDamage(1);
}
1 reference
public void SetDirection(float _direction)
{
    lifetime = 0;
    direction = _direction;
    gameObject.SetActive(true);
    hit = false;
    boxCollider.enabled = true;

    float localScaleX = transform.localScale.x;
    if (Mathf.Sign(localScaleX) != _direction)
        localScaleX = -localScaleX;

    transform.localScale = new Vector3(localScaleX, transform.localScale.y, transform.localScale.z);
}
0 references
private void Deactivate()
{
    gameObject.SetActive(false);
}
```

Рисунок 4.8 – Progectile скрипт

У методі Update додаємо перевірку для введення з клавіатури або миші для атаки (рис. 4.8). Додаємо новий метод Attack, який буде створювати екземпляр снаряду і задавати йому початкову швидкість. Використовуємо клавішу миші для атаки.

Тепер при натисканні кнопки миші (або іншого заданого вводу) гравець буде стріляти снарядом, який рухатиметься в напрямку спавну з заданою швидкістю. Це забезпечує базову функціональність атаки, яку можна далі налаштовувати і розширювати відповідно до потреб гри.

4.4 Реалізація ігрового рівня

Спочатку створюємо моделі блоків у програмі для моделювання або використовуємо вбудовані інструменти в Unity для цього. Ці моделі можуть мати різні форми та розміри в залежності від концепції нашої гри (рис. 4.9).

Рисунок 4.9 – Текстура землі

Після створення моделей імпортуємо їх у проект Unity і перетягуємо на сцену. Розміщуємо блоки так, щоб вони утворювали бажану структуру ігрового рівня, використовуючи різні розміри та форми для створення цікавого та різноманітного ландшафту (рис. 4.10).

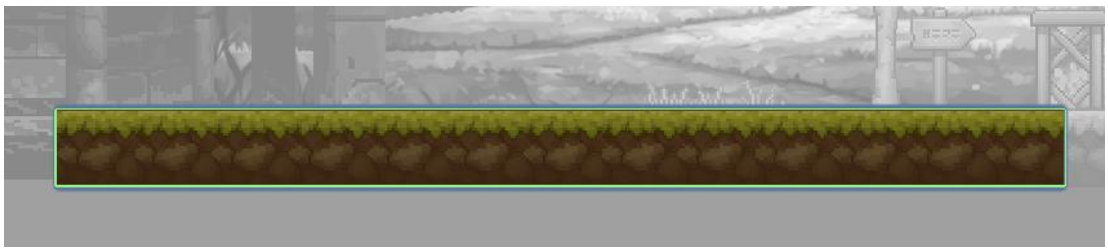

Рисунок 4.10 – Струтура ігрового рівня

Після розміщення блоків на сцені додаємо колайдери до кожного блоку, щоб забезпечити взаємодію гравця з ними. Колайдери визначають область, з якою може взаємодіяти гравець, тому налаштовуємо їх так, щоб вони відповідали формі блоку.

Рисунок 4.11 – Перевірка взаємодії ігрока з блоками

Після додавання колайдерів перевіряємо, чи правильно взаємодіє гравець з блоками під час тестування ігрового рівня (рис. 4.11). Запускаємо сцену у режимі гри та перевіряємо, чи гравець може взаємодіяти з блоками відповідно до очікувань.

4.5 Налаштування камери

Для зручного гемплею треба прив’язати камеру до гравця. Спочатку вибираємо камеру у вікні Hierarchy або створюємо нову камеру, якщо її ще немає на сцені. Впевнюємося, що камера знаходиться на сцені разом з гравцем (рис. 4.12).

Після цього створюємо новий C# скрипт, наприклад, називаємо його CameraController, і відкриваємо його у редакторі коду. У цьому скрипті будемо писати логіку для прив’язки камери до гравця (рис. 4.13).

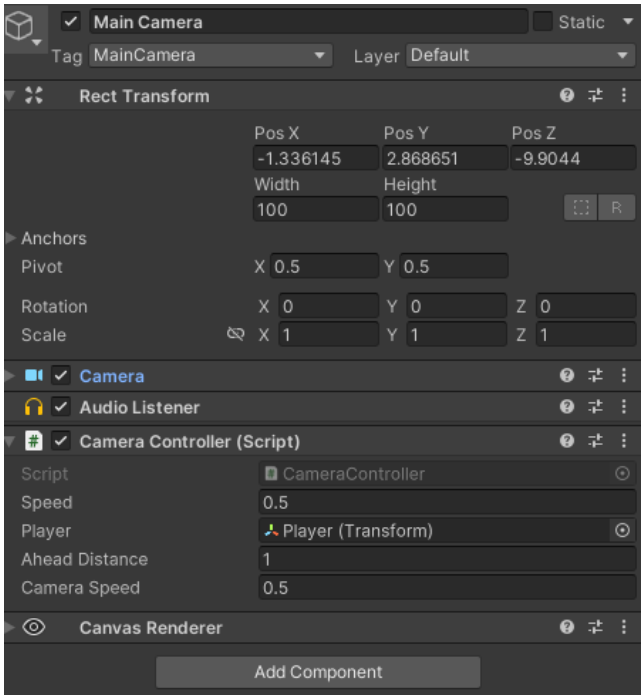

Рисунок 4.12 – Налаштування камери

```
public class CameraController : MonoBehaviour
{

    [SerializeField] private float speed;
    private float currentPosX;
    private Vector3 velocity = Vector3.zero;


    [SerializeField] private Transform player;
    [SerializeField] private float aheadDistance;
    [SerializeField] private float cameraSpeed;
    private float lookAhead;

    Unity Message | 0 references
    private void Update()
    {

        transform.position = new Vector3(player.position.x + lookAhead, transform.position.y, transform.position.z);
        lookAhead = Mathf.Lerp(lookAhead, (aheadDistance * player.localScale.x), Time.deltaTime * cameraSpeed);
    }

    1 reference
    public void MoveToNewRoom(Transform _newRoom)
    {
        currentPosX = _newRoom.position.x;
    }
}
```

Рисунок 4.13 – CameraController скрипт

Після цього зберігаємо зміни і запускаємо гру в режимі Play, щоб перевірити, чи камера правильно слідує за гравцем. Камера повинна плавно слідувати за гравцем, зберігаючи заданий офсет і забезпечуючи стабільний огляд під час гри.

4.6 Реалізація ворогів

Спочатку створюємо 2D модель ворога та імпортуємо її в проект Unity (рис. 4.14). Після імпортування перетягуємо модель на сцену і розміщуємо її в бажаній початковій позиції. Створюємо нову папку в проекті, наприклад, «Prefabs», і перетягуємо модель зі сцени в цю папку, щоб створити префаб ворога.

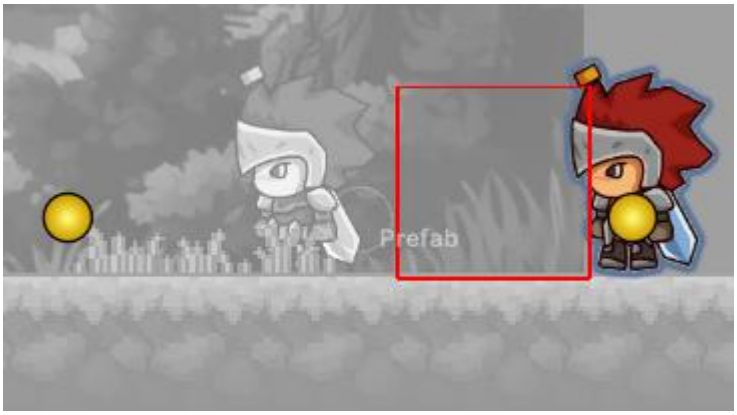

Рисунок 4.14 – Prefab ворога

Потім додаємо необхідні компоненти до префабу ворога. Вибираємо префаб ворога у вікні Hierarchy. У вікні Inspector натискаємо Add Component і додаємо компонент Rigidbody, який відповідає за фізику об’єкта. Налаштовуємо параметри компонента, такі як маса і гравітація, за потребою. Далі додаємо компонент Collider, наприклад, Capsule Collider, для обробки зіткнень. Налаштовуємо розмір і форму колайдера відповідно до моделі ворога.

Для патрулювання місцевості додаємо дві точки, які визначатимуть напрямки патрулювання ворога. Створюємо дві порожні GameObject і називаємо їх, наприклад, PatrolPoint1 і PatrolPoint2. Розміщуємо ці об’єкти у вікні сцени в місцях, де ворог повинен змінювати напрямок руху.

Далі створюємо C# скрипт і додаємо логіку патрулювання (рис. 4.15). У цьому скрипті визначаємо змінні для швидкості руху, точок патрулювання та стану ворога.

```
Unity Message | 0 references
private void Update()
{
    cooldownTimer += Time.deltaTime;

    //Attack only when player in sight?
    if (PlayerInSight())
    {
        if (cooldownTimer >= attackCooldown)
        {
            cooldownTimer = 0;
            anim.SetTrigger("meleeAttack");
        }
    }

    if (enemyPatrol != null)
        enemyPatrol.enabled = !PlayerInSight();
}

3 references
private bool PlayerInSight()
{
    RaycastHit2D hit =
        Physics2D.BoxCast(boxCollider.bounds.center + transform.right * range * transform.localScale.x * colliderDistance,
        new Vector3(boxCollider.bounds.size.x * range, boxCollider.bounds.size.y, boxCollider.bounds.size.z),
        0, Vector2.left, 0, playerLayer);

    if (hit.collider != null)
        playerHealth = hit.transform.GetComponent<Health>();

    return hit.collider != null;
}
Unity Message | 0 references
private void OnDrawGizmos()
{
    Gizmos.color = Color.red;
    Gizmos.DrawWireCube(boxCollider.bounds.center + transform.right * range * transform.localScale.x * colliderDistance,
        new Vector3(boxCollider.bounds.size.x * range, boxCollider.bounds.size.y, boxCollider.bounds.size.z));
}

0 references
private void DamagePlayer()
{
    if (PlayerInSight())
        playerHealth.TakeDamage(damage);
```

Рисунок 4.15 – MeleeEnemy скрипт

Для атаки гравця додаємо перевірку на наявність гравця в межах видимості ворога. Якщо гравець знаходиться в полі зору ворога, змінюємо стан ворога на атаку та реалізуємо рух ворога в напрямку гравця. Додаємо перевірку на відстань між ворогом та гравцем, щоб визначити, коли ворог повинен почати атаку.

Далі реалізуємо систему здоров’я ворога (рис. 4.16). У скрипті додаємо змінні для максимального та поточного здоров’я ворога. Реалізуємо метод для зменшення здоров’я при отриманні ушкоджень і перевірку, чи здоров’я не зменшилося до нуля. Якщо здоров’я ворога вичерпано, викликаємо метод, що обробляє смерть ворога, наприклад, відображення анімації смерті або видалення ворога зі сцени.

```
public void TakeDamage(float _damage)
{
    if (invulnerable) return;
    currentHealth = Mathf.Clamp(currentHealth - _damage, 0, startingHealth);

    if (currentHealth > 0)
    {
        anim.SetTrigger("hurt");
        StartCoroutine(Invunerability());
        SoundManager.instance.PlaySound(hurtSound);
    }
    else
    {
        if (!dead)
        {
            //Deactivate all attached component classes
            foreach (Behaviour component in components)
                component.enabled = false;

            anim.SetBool("grounded", true);
            anim.SetTrigger("die");

            dead = true;
            SoundManager.instance.PlaySound(deathSound);
        }
    }
}
2 references
```

Рисунок 4.16 – Health скрипт

Для додавання анімацій до ворога створюємо Animator Controller і додаємо анімації для різних станів ворога, таких як патрулювання, атака, отримання ушкоджень та смерть (рис. 4.17). Прив’язуємо Animator Controller до ворога, додаємо відповідні параметри та налаштовуємо переходи між анімаціями в залежності від стану ворога.

Рисунок 4.17 – Атака ворога

Рисунок 4.18– Анімація смерті ворога

Після написання та налаштування всіх компонентів і логіки ворога зберігаємо зміни і тестуємо гру, щоб переконатися, що ворог правильно патрулює місцевість, атакує гравця, реагує на ушкодження та вмирає, коли його здоров’я вичерпується. Ці дії забезпечують функціональність ворога, яка включає патрулювання, атаку гравця, управління здоров’ям та анімаціями, що покращує геймплей вашої гри.

4.7 Реалізація пасток на рівні

Спочатку створюємо або імпортуємо 2D моделі пасток. Це можуть бути, наприклад, шипи, падаючі камені, вогняні пастки або будь-які інші об’єкти, які відповідають концепції гри. Після створення або імпортування моделей перетягуємо їх на сцену.

Рисунок 4.19 – Prefab пастки піли

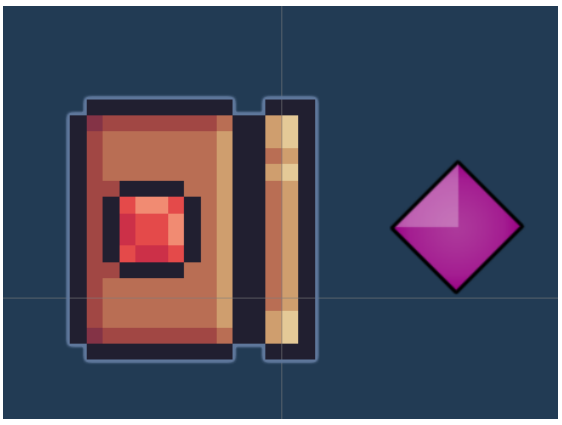

Рисунок 4.20 – Prefab пастки зі стрілами

Рисунок 4.21 – Prefab вогняної пастки

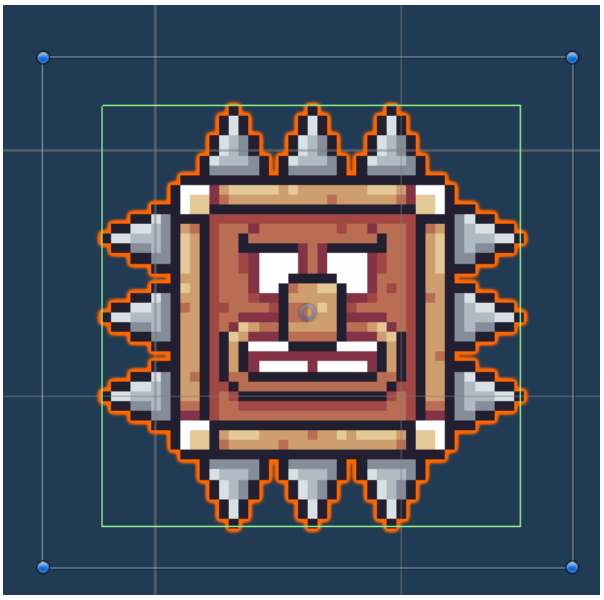

Рисунок 4.22 – Prefab переслідуючої пастки

Рисунок 4.23 – Prefab пастки з шипами

Для кожної пастки додаємо компонент Collider, який відповідає за виявлення зіткнень з гравцем. Вибираємо модель пастки у вікні Hierarchy і

додаємо відповідний Collider (наприклад, Box Collider або Sphere Collider). Встановлюємо прапорець Is Trigger, щоб колайдер реагував на проходження через нього гравця, але не створював фізичних взаємодій.

Далі створюємо C# скрипт, який буде відповідати за взаємодію пастки з гравцем (рис. 4.24). У цьому скрипті визначаємо метод OnTriggerEnter, який буде викликатися, коли гравець входить у колайдер пастки. В цьому методі додаємо логіку, яка визначає, що станеться з гравцем, коли він активує пастку (наприклад, зменшення здоров'я гравця, гра звукового ефекту, або анімація пастки).

```
public class EnemyDamage : MonoBehaviour
{
    [SerializeField] protected float damage;

    Unity Message | 2 references
    protected void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.tag == "Player")
            collision.GetComponent<Health>().TakeDamage(damage);
    }
}
```

Рисунок 4.24 – EnemyDamage скрипт

Також додаємо звукові та візуальні ефекти для пасток, щоб зробити їх більш помітними і інтуїтивно зрозумілими для гравця. Для цього імпортуємо відповідні аудіо файли та анімації і налаштовуємо їх у компонентах AudioSource та Animator, які додаємо до пасток.

Рисунок 4.25 – Розташування пасток на рівні

Розміщуємо пастки на рівні так, щоб вони створювали виклики для гравця, але при цьому не були занадто складними або неможливими для уникнення (рис. 4.25). Тестуємо рівень, щоб переконатися, що пастки працюють належним чином, взаємодіють з гравцем та додають цікавості до ігрового процесу.

Нарешті, оптимізуємо розташування та поведінку пасток на рівні відповідно до результатів тестування, щоб забезпечити баланс між викликом і задоволенням від гри. Це включає налаштування частоти спрацьовування пасток, їх кількість і розташування, а також будь-які додаткові ефекти, які можуть покращити загальний геймплей.

4.8 Реалізація системи чекпоінтів

Спочатку створюємо або імпортуємо 3D модель або простий об’єкт для чекпоінта (рис. 4.26). Це може бути прапорець, стовпчик або будь-який інший об’єкт, який буде легко помітити на рівні. Після імпортування перетягуємо модель на сцену і розміщуємо її в потрібному місці, де гравець повинен зберегти прогрес.

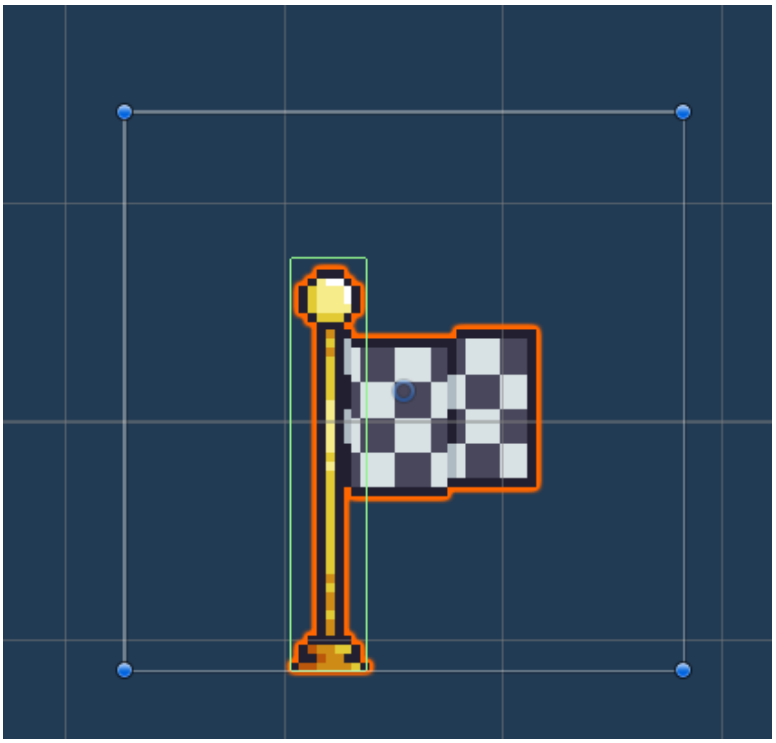

Рисунок 4.26 – Prefab чекпоінту

Вибираємо об’єкт чекпоінта у вікні Hierarchy. У вікні Inspector натискаємо Add Component і додаємо компонент Collider, наприклад, Box Collider, щоб визначити область, в яку гравець повинен потрапити для активації чекпоінта. Встановлюємо прапорець Is Trigger, щоб колайдер реагував на проходження через нього гравця, але не створював фізичних взаємодій.

```
    1 reference
    public void Respawn()
    {
        AddHealth(startingHealth);
        anim.ResetTrigger("die");
        anim.Play("Idle");
        StartCoroutine(Invunerability());
        dead = false;

        foreach (Behaviour component in components)
            component.enabled = true;
    }
}
```

Рисунок 4.27 – Health скрипт

Створюємо новий C# скрипт (рис. 4.27). У цьому скрипті визначаємо метод OnTriggerEnter, який буде викликатися, коли гравець входить у колайдер чекпоінта. Додаємо логіку для зберігання позиції чекпоінта як останньої збереженої точки гравця:

Розміщуємо кілька чекпоінтів на рівні в ключових місцях, де гравець повинен зберегти прогрес (рис. 4.28). Важливо розташувати їх так, щоб вони були легко помітні та досяжні для гравця під час проходження рівня.

Рисунок 4.28 – Чекпоінт на рівні

Запускаємо гру і тестуємо роботу чекпоінтів. Перевіряємо, чи правильно зберігається позиція гравця при проходженні через чекпоінт та чи відновлюється гравець з останньої збереженої точки після смерті або перезапуску рівня.

4.9 Реалізація додаткових елементів оточення

Додаваємо додаткові елементів оточення, такі як дерева, трава, вказівники та інші об’єкти, для більш приємного досвіду у грі.

Спочатку створюємо або імпортуємо 2D моделі елементів оточення. Це можуть бути моделі дерев, трави, вказівників, каменів, квітів та інших об’єктів.

Після імпортування моделей у проект Unity, перетягуємо їх на сцену. Для кожного типу елементів оточення створюємо окрему папку в проекті, щоб зберігати їх у впорядкованому вигляді.

Розміщення дерев: Використовуємо інструмент Terrain для розміщення дерев (рис. 4.30). Вибираємо об’єкт Terrain у вікні Hierarchy і відкриваємо вкладку Paint Trees у вікні Inspector. Натискаємо кнопку Edit Trees і додаємо нову дерев’яну модель у список. Після цього за допомогою пензля розміщуємо дерева на території, налаштовуючи щільність і розмір дерев.

Розміщення трави: Для розміщення трави використовуємо вкладку Paint Details у вікні Inspector (рис. 4.29). Додаємо нову модель трави або текстуру у список деталей. За допомогою пензля розміщуємо траву на території, налаштовуючи щільність, розмір та різноманітність трав’яного покриття.

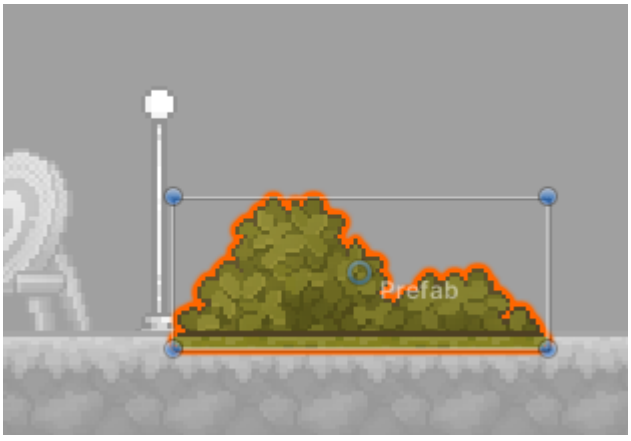

Рисунок 4.29 – Додатковий елемент кущ

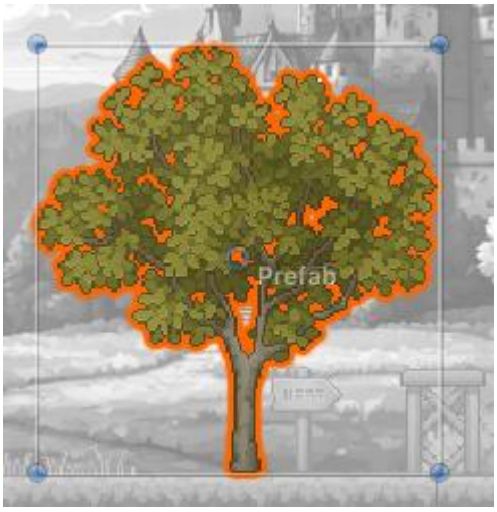

Рисунок 4.30 – Додатковий елемент дерево

Рисунок 4.31 – Додатковий елемент вказівник

Додавання вказівників і інших елементів (рис. 4.31). Розміщення вказівників: Вибираємо моделі вказівників і перетягуємо їх на сцену. Розміщуємо вказівники у відповідних місцях, де вони можуть допомогти гравцеві орієнтуватися у просторі. Налаштовуємо їх положення, масштаб і орієнтацію. Додавання інших елементів оточення: Вибираємо моделі інших елементів, таких як камені, квіти, лавки тощо, і перетягуємо їх на сцену. Розміщуємо ці елементи у відповідних місцях для додання різноманітності та деталізації ландшафту. Налаштовуємо їх положення, масштаб і орієнтацію.

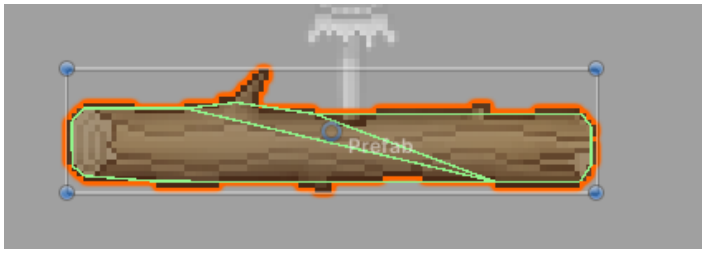

Рисунок 4.32 – Додатковий елемент поліно

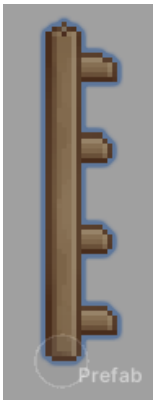

Рисунок 4.33 – Додатковий елемент сходи

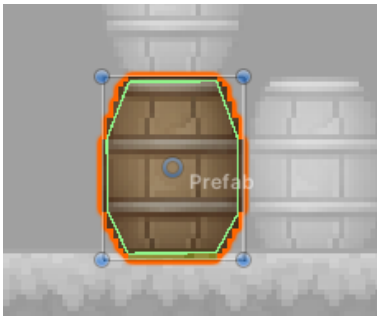

Рисунок 4.34 – Додатковий елемент діжка

Для великих об’єктів, таких як дерева, камені або вказівники, додаємо компоненти Collider, щоб гравець міг з ними взаємодіяти. Вибираємо об’єкт на сцені, у вікні Inspector натискаємо Add Component і додаємо відповідний колайдер (наприклад, Box Collider, Capsule Collider або Mesh Collider). Налаштовуємо розмір і форму колайдера відповідно до об’єкта.

Рисунок 4.35 – Тестування додаткових елементів

Після розміщення всіх елементів оточення тестуємо гру, щоб переконатися, що вони правильно взаємодіють з іншими об’єктами на сцені і не викликають проблем з продуктивністю (рис. 4.35). Оптимізуємо розташування та кількість елементів для забезпечення плавної роботи гри.

4.10 Реалізація анімацій

Перейдемо до використання піксель арту безпосередньо в процесі розробки відеоігор. У будь-якій комп’ютерній грі ігрові об’єкти мають бути інтерактивними: персонажі повинні рухатись, вороги – нападати, і в цілому щось повинно відбуватись на екрані. Все це забезпечується за допомогою анімацій.

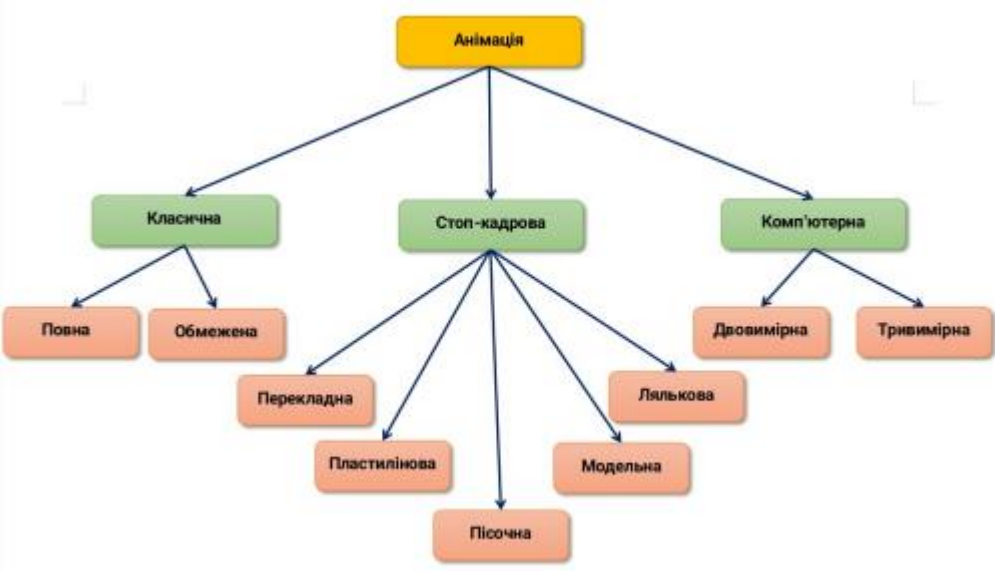

Рисунок 2.6 – Види анімацій

Анімація виникла ще у XIX столітті у вигляді стробоскопічного диску, на якому зображення під час швидкого обертання створювали ефект руху, що і заклало основу сучасної анімації (рис. 2.6). Вже на початку XX століття почали з’являтись перші фільми та мультфільми, які використовували плівку та проектори; у випадку мультфільмів плівка складалась з фотографій малюнків.

Основний принцип роботи анімації або відеоматеріалів полягає в швидкому заміщенні одного зображення іншим, яке трохи відрізняється від попереднього. Ці зображення називаються кадрами, і чим їх більше та менш

помітні зміни між ними, тим чіткішою і плавнішою буде анімація. Важливим фактором є частота прокрутки кадрів, яка має бути достатньо високою, щоб людське око не розпізнавало перехід між ними. Кадри повинні бути розраховані на демонстрацію з конкретною частотою, щоб не виникало ефекту пришвидшення чи сповільнення анімації. Загальноприйнятим стандартом частоти кадрів, при якому людське око не розпізнає перехід між кадрами, є 24 кадри в секунду.

З часу виникнення анімації технології та техніки значно розвинулись, і сьогодні у двовимірній графіці застосовуються два основних типи анімацій: анімація з ключовими кадрами (Keyframe) та анімація за методом «прямо вперед» (Straight Ahead), започаткована аніматорами студії Disney і вперше згадана в книзі 1981 року «12 базових принципів анімації».

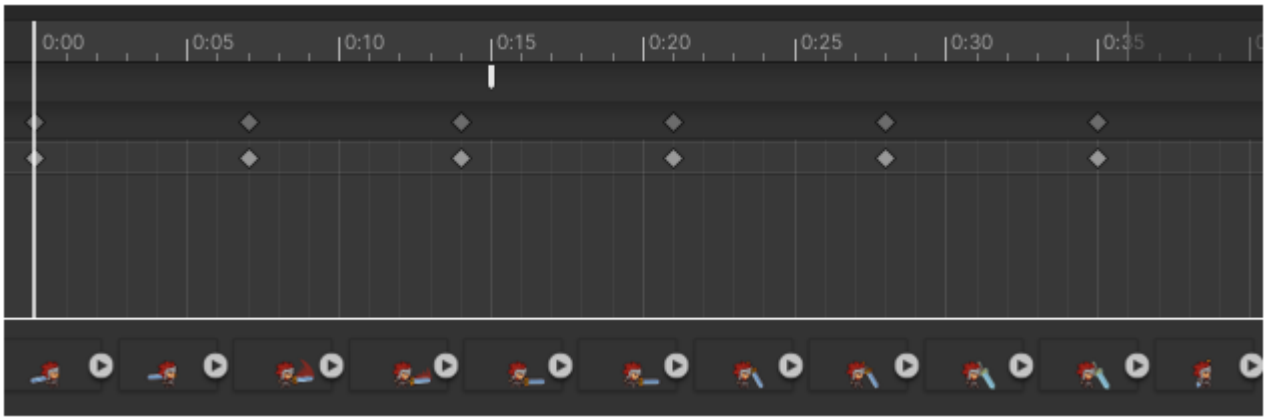

Рисунок 2.7 – Проста 2D анімація

Основна різниця між цими методами полягає в наявності ключових кадрів. У випадку анімації з ключовими кадрами процес полягає в створенні ключових положень об’єктів, між якими згодом малюються проміжні кадри для згладжування переходу. У методі «прямо вперед» є лише перший ключовий кадр, після якого аніматор малює кожен кадр окремо, не знаючи точно, куди це його приведе. Це надає певну творчу свободу, але також несе ризики, оскільки відсутні ключові кадри, які допомагають підтримувати зображення однаковими, наприклад, у розмірі або формі об’єкта.

Ключові кадри дозволяють простіше спланувати анімацію і відслідковувати її створення. Це також дає можливість не створювати ідеально однакові проміжні кадри, наприклад, під час активних сцен, де об’єкт може бути розтягнутий або розмитий для плавного переходу між ключовими кадрами, або ж навпаки, робити перехід більш різким і динамічним. Такий підхід широко використовується в японській анімації, відомій як аніме, де багато динамічних сцен і переходів. Аніме часто створюється за мотивами манги (японських коміксів) і намагається зберегти оригінальний стиль, пришвидшуючи процес анімації, використовуючи ключові кадри з коміксів і з’єднуючи їх проміжними кадрами. Це полегшує роботу та покращує плавність або динаміку анімації, оскільки проміжні кадри можуть не зберігати точні форми ключових кадрів і їх кількість може бути зменшена або взагалі відсутня.

Аніме значно вплинуло на розвиток анімації та кінематографу, запровадивши власні стилі, ракурси, способи зображення емоцій, динамічних сцен та фантастичних сюжетів, які поширилися в інші культури. Багато прийомів, що виникли в процесі розвитку аніме, зараз використовуються в фільмах та мультфільмах по всьому світу.

В сучасній 2D анімації використовуються обидва методи, які залучають ключові кадри. Перший метод, похідний від старого методу студії Disney, тепер включає ключові кадри, але вимагає детальної промальовки кожного кадру і більше характерний для західних студій. Другий метод, який застосовується в

японській анімації, використовує в основному лише ключові кадри, а проміжні кадри можуть бути менш деталізованими або взагалі відсутні.

У розробці 2D ігор часто застосовується метод з аніме, оскільки він додає картинці динаміки і спрощує процес анімації. Наприклад, для анімації різкого удару мечем можна використати ключові кадри початкового та кінцевого положення меча, а між ними – один кадр, що зображає лінію удару. Це можна застосувати й до інших моментів у грі, таких як різкий випад вперед або переміщення. Такий прийом, відомий як Dash (ривок), вже давно став окремою механікою в відеоіграх, і в деяких з них є ключовим елементом ігрового процесу.

Висновки до розділу 4

У процесі розробки гри в Unity послідовно виконано кілька важливих кроків, які значно вплинули на створення захоплюючого ігрового середовища. Спершу створено префаб гравця, додано компоненти для фізики, та зіткнень, анімації, а також написано скрипти, який забезпечують рух, атаки та управління здоров’ям. Далі розроблено головне меню з кнопками «Грати» і «Вихід», додано фон і текст з назвою гри, що надає професійний вигляд та полегшує навігацію. На ігровому рівні розміщено блоки, по яких гравець може пересуватися, і прив’язано камеру до гравця, щоб забезпечити плавний та стабільний огляд. Створено ворогів, які патрулюють між заданими точками, атакують гравця, мають систему здоров’я та анімації, і можуть бути вбиті, що додає динаміки і викликів у гру. Додано пастки, які взаємодіють з гравцем і завдають шкоди, розміщено елементи оточення, такі як дерева, трава і вказівники, щоб покращити візуальну привабливість та атмосферу рівня. Нарешті, реалізовано чекпоінти, що дозволяють зберігати прогрес гравця та відновлюватися з певної точки, забезпечуючи безперервність гри та підвищуючи комфорт гравця. Ці кроки спільно створюють основу для функціональної та привабливої гри, яку можна далі розширювати та вдосконалювати.

ВИСНОВКИ

У рамках реалізації кваліфікаційної роботи бакалавру було розроблено 2D ігровий застосунок жанру Platform. В першу чергу було проведено аналіз існуючих рішень у сфері мобільних ігор цього жанру. Було вивчено кілька популярних ігрових застосунків, з яких взято найкращі ідеї та механіки, щоб сформувати основну концепцію та геймплей для власної гри.

Окремий розділ роботи було присвячено дизайну програмного забезпечення, оскільки це один із найважливіших аспектів розробки. Дизайн дозволяє уявити, як виглядатиме кінцевий продукт, і забезпечує основу для подальшої розробки. У цьому розділі було створено макет майбутнього інтерфейсу користувача, який включав детальний опис усіх елементів інтерфейсу та взаємодій користувача з грою.

Наступним етапом стало створення основних ігрових елементів. Було розроблено ворогів, предмети та здібності гравця, які додають динаміки та викликів у гру. Також було описано основні можливості гравця, такі як рух, стрибки, атаки та взаємодія з предметами, що допомагає створити захоплюючий ігровий досвід.

Важливу частину роботи становило проєктування ігрового застосунку та вибір середовища розробки. Серед чотирьох претендентів було обрано ігровий рушій Unity 3D, оскільки він поєднує в собі легкість використання та великий функціонал. В процесі роботи з Unity 3D було здобуто значний досвід, який включає роботу з фізикою, анімаціями, скриптингом та іншими аспектами розробки ігор.

У підсумку, поставлені задачі були виконані, і мета кваліфікаційної роботи була досягнута. Розроблений 2D ігровий застосунок є результатом комплексного підходу до аналізу, дизайну, розробки та тестування, що забезпечило створення функціональної та привабливої гри.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Офіційний сайт Visual Studio: вебсайт. URL: https://www.visualstudio.com/en/vs/ (дата звернення: 01.03.2024).

2. Unity: вебсайт. URL: https://unity.com/en (дата звернення 03.04.2024).

3. Крейтон, Р.Х. Основи розробки ігор у Unity Packt Publishing. 2010. 83 с.

4. Хокінг Д. М. Unity в дії. Мультиплатформенна розробка на практиці. 2016. 336 с.

5. Apperley T. H. Genre and game studies: Toward a critical approach to video game genres Simulation & Gaming. 2006. Vol. 37. No. 1. P. 6 – 23.

6. Buckland Mat. Programming Game AI by Example – Texas, Wordware Publishing. 2004. P. 25-43.

7. Clearwater D. What Defines Videogame Genre? Thinking about Genre Study after the Great Divide. The Journal of the Canadian Game Studies Association. 2011. No. 5. P. 29-49.

8. Goldstone W. Unity Game Development Essentials. Birmingham: Packt Publishing Ltd. 2009. P. 316.

9. Gregory Jason. Game Engine Architecture. – New York, CRC Press. 2009. No. 5. P. 15-24.

10. Gregory Jason. Game Engine Architecture: Second Edition. –New York, CRC Press. 2014. No. 32. P. 23-30.

11. Lengyel Eric. Mathematics for 3D Game Programming and Computer Graphics: Third Edition. – Boston, Course Technology. 2012. P. 215.

12. McShaffry, Mike, Graham David. Game coding complete: Fourth Edition. – Boston, Cource Technology. 2013. P. 184.

13. Unity Manual, Unity Documentation: довідник. URL: https://docs.unity3d.com/Manual/ (дата звернення: 08.04.2024).

14. Інформація про гру Rayman Origins: вебсайт. URL: https://www.ubisoft.com/en-us/game/rayman/origins/ (дата звернення 05.04.2024).

15. Jared H. Developing 2D Games with Unity: Independent Game Programming with C#: ISBN – 13, 2009.

16. Gregory J. Game Engine Architecture. USA: A K Peters / CRC Press, 2009. 864 p.

17. Christopher W. T. An architectural approach to level design: George Mason University, 2014.

18. Jared H. Developing 2D Games with Unity: Independent Game Programming with C#: ISBN – 13, 2009.

19. Інформація про гру Hollow Khight Origins: вебсайт. URL: https://www.hollowknight.com/ (дата звернення 07.04.2024).

20. Інформація про гру Ori and the Blind Forest: вебсайт. URL: https://www.orithegame.com/blind-forest/ (дата звернення 08.04.2024).

ДОДАТОК А

Код програмного забезпечення

CameraController.cs

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

public class CameraController : MonoBehaviour
{

    [SerializeField] private float speed;
    private float currentPosX;
    private Vector3 velocity = Vector3.zero;


    [SerializeField] private Transform player;
    [SerializeField] private float aheadDistance;
    [SerializeField] private float cameraSpeed;
    private float lookAhead;

    private void Update()
    {

        transform.position = new Vector3(player.position.x + lookAhead, transform.position.y, transform.position.z);
        lookAhead = Mathf.Lerp(lookAhead, (aheadDistance * player.localScale.x), Time.deltaTime * cameraSpeed);
    }

    public void MoveToNewRoom(Transform _newRoom)
    {
        currentPosX = _newRoom.position.x;
    }
}
```

SelectionArrow.cs

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

public class SelectionArrow : MonoBehaviour
{
    [SerializeField] private RectTransform[] buttons;
    [SerializeField] private AudioClip changeSound;
    [SerializeField] private AudioClip interactSound;
    private RectTransform arrow;
    private int currentPosition;

    private void Awake()
    {
        arrow = GetComponent<RectTransform>();
    }
    private void OnEnable()
    {
        currentPosition = 0;
        ChangePosition(0);
    }
    private void Update()
    {

        if (Input.GetKeyDown(KeyCode.UpArrow) || Input.GetKeyDown(KeyCode.W))
```

```
            ChangePosition(-1);
        if (Input.GetKeyDown(KeyCode.DownArrow) || Input.GetKeyDown(KeyCode.S))
            ChangePosition(1);

        if (Input.GetKeyDown(KeyCode.KeypadEnter) || Input.GetKeyDown(KeyCode.E))
            Interact();
    }

    private void ChangePosition(int _change)
    {
        currentPosition += _change;

        if (_change != 0)
            SoundManager.instance.PlaySound(changeSound);

        if (currentPosition < 0)
            currentPosition = buttons.Length - 1;
        else if (currentPosition > buttons.Length - 1)
            currentPosition = 0;

        AssignPosition();
    }
    private void AssignPosition()
    {

        arrow.position = new Vector3(arrow.position.x, buttons[currentPosition].position.y);
    }
    private void Interact()
    {
        SoundManager.instance.PlaySound(interactSound);


        buttons[currentPosition].GetComponent<Button>().onClick.Invoke();
    }
}
```

SoundManager.cs

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

public class SoundManager : MonoBehaviour
{
    public static SoundManager instance { get; private set; }
    private AudioSource soundSource;
    private AudioSource musicSource;

    private void Awake()
    {
        soundSource = GetComponent<AudioSource>();
        musicSource = transform.GetChild(0).GetComponent<AudioSource>();


        if (instance == null)
        {
            instance = this;
            DontDestroyOnLoad(gameObject);
        }

        else if (instance != null && instance != this)
            Destroy(gameObject);


        ChangeMusicVolume(0);
```

```
        ChangeSoundVolume(0);
    }
    public void PlaySound(AudioClip _sound)
    {
        soundSource.PlayOneShot(_sound);
    }

    public void ChangeSoundVolume(float _change)
    {
        ChangeSourceVolume(1, "soundVolume", _change, soundSource);
    }
    public void ChangeMusicVolume(float _change)
    {
        ChangeSourceVolume(0.3f, "musicVolume", _change, musicSource);
    }

    private void ChangeSourceVolume(float baseVolume, string volumeName, float change, AudioSource source)
    {

        float currentVolume = PlayerPrefs.GetFloat(volumeName, 1);
        currentVolume += change;


        if (currentVolume > 1)
            currentVolume = 0;
        else if (currentVolume < 0)
            currentVolume = 1;


        float finalVolume = currentVolume * baseVolume;
        source.volume = finalVolume;


        PlayerPrefs.SetFloat(volumeName, currentVolume);
    }
}
```

EnemyFiballHolder.cs

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

public class EnemyFireballHolder : MonoBehaviour
{
    [SerializeField] private Transform enemy;

    private void Update()
    {
        transform.localScale = enemy.localScale;
    }
}
```

EnemyPatrol.cs

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

public class EnemyPatrol : MonoBehaviour
{
    [Header("Patrol Points")]
    [SerializeField] private Transform leftEdge;
    [SerializeField] private Transform rightEdge;

    [Header("Enemy")]
```

```
    [SerializeField] private Transform enemy;

    [Header("Movement parameters")]
    [SerializeField] private float speed;
    private Vector3 initScale;
    private bool movingLeft;

    [Header("Idle Behaviour")]
    [SerializeField] private float idleDuration;
    private float idleTimer;

    [Header("Enemy Animator")]
    [SerializeField] private Animator anim;

    private void Awake()
    {
        initScale = enemy.localScale;
    }
    private void OnDisable()
    {
        anim.SetBool("moving", false);
    }

    private void Update()
    {
        if (movingLeft)
        {
            if (enemy.position.x >= leftEdge.position.x)
                MoveInDirection(-1);
            else
                DirectionChange();
        }
        else
        {
            if (enemy.position.x <= rightEdge.position.x)
                MoveInDirection(1);
            else
                DirectionChange();
        }
    }

    private void DirectionChange()
    {
        anim.SetBool("moving", false);
        idleTimer += Time.deltaTime;

        if (idleTimer > idleDuration)
            movingLeft = !movingLeft;
    }

    private void MoveInDirection(int _direction)
    {
        idleTimer = 0;
        anim.SetBool("moving", true);


        enemy.localScale = new Vector3(Mathf.Abs(initScale.x) * _direction,
            initScale.y, initScale.z);


        enemy.position = new Vector3(enemy.position.x + Time.deltaTime * _direction * speed,
            enemy.position.y, enemy.position.z);
    }
}
```

MeleeEnemy.cs

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

public class MeleeEnemy : MonoBehaviour
{
    [Header("Attack Parameters")]
    [SerializeField] private float attackCooldown;
    [SerializeField] private float range;
    [SerializeField] private int damage;

    [Header("Collider Parameters")]
    [SerializeField] private float colliderDistance;
    [SerializeField] private BoxCollider2D boxCollider;

    [Header("Player Layer")]
    [SerializeField] private LayerMask playerLayer;
    private float cooldownTimer = Mathf.Infinity;


    private Animator anim;
    private Health playerHealth;
    private EnemyPatrol enemyPatrol;

    private void Awake()
    {
        anim = GetComponent<Animator>();
        enemyPatrol = GetComponentInParent<EnemyPatrol>();
    }

    private void Update()
    {
        cooldownTimer += Time.deltaTime;


        if (PlayerInSight())
        {
            if (cooldownTimer >= attackCooldown)
            {
                cooldownTimer = 0;
                anim.SetTrigger("meleeAttack");
            }
        }

        if (enemyPatrol != null)
            enemyPatrol.enabled = !PlayerInSight();
    }

    private bool PlayerInSight()
    {
        RaycastHit2D hit =
            Physics2D.BoxCast(boxCollider.bounds.center + transform.right * range * transform.localScale.x * colliderDistance,
            new Vector3(boxCollider.bounds.size.x * range, boxCollider.bounds.size.y, boxCollider.bounds.size.z),
            0, Vector2.left, 0, playerLayer);

        if (hit.collider != null)
            playerHealth = hit.transform.GetComponent<Health>();

        return hit.collider != null;
    }
    private void OnDrawGizmos()
    {
```

```
        Gizmos.color = Color.red;
        Gizmos.DrawWireCube(boxCollider.bounds.center + transform.right * range * transform.localScale.x *
colliderDistance,
            new Vector3(boxCollider.bounds.size.x * range, boxCollider.bounds.size.y, boxCollider.bounds.size.z));
    }

    private void DamagePlayer()
    {
        if (PlayerInSight())
            playerHealth.TakeDamage(damage);
    }
}
```

RangedEnemy.cs

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

public class RangedEnemy : MonoBehaviour
{
    [Header("Attack Parameters")]
    [SerializeField] private float attackCooldown;
    [SerializeField] private float range;
    [SerializeField] private int damage;

    [Header("Ranged Attack")]
    [SerializeField] private Transform firepoint;
    [SerializeField] private GameObject[] fireballs;

    [Header("Collider Parameters")]
    [SerializeField] private float colliderDistance;
    [SerializeField] private BoxCollider2D boxCollider;

    [Header("Player Layer")]
    [SerializeField] private LayerMask playerLayer;
    private float cooldownTimer = Mathf.Infinity;

    [Header("Fireball Sound")]
    [SerializeField] private AudioClip fireballSound;


    private Animator anim;
    private EnemyPatrol enemyPatrol;

    private void Awake()
    {
        anim = GetComponent<Animator>();
        enemyPatrol = GetComponentInParent<EnemyPatrol>();
    }

    private void Update()
    {
        cooldownTimer += Time.deltaTime;


        if (PlayerInSight())
        {
            if (cooldownTimer >= attackCooldown)
            {
                cooldownTimer = 0;
                anim.SetTrigger("rangedAttack");
            }
        }
```

```
        if (enemyPatrol != null)
            enemyPatrol.enabled = !PlayerInSight();
    }

    private void RangedAttack()
    {
        SoundManager.instance.PlaySound(fireballSound);
        cooldownTimer = 0;
        fireballs[FindFireball()].transform.position = firepoint.position;
        fireballs[FindFireball()].GetComponent<EnemyProjectile>().ActivateProjectile();
    }
    private int FindFireball()
    {
        for (int i = 0; i < fireballs.Length; i++)
        {
            if (!fireballs[i].activeInHierarchy)
                return i;
        }
        return 0;
    }

    private bool PlayerInSight()
    {
        RaycastHit2D hit =
            Physics2D.BoxCast(boxCollider.bounds.center + transform.right * range * transform.localScale.x * colliderDistance,
            new Vector3(boxCollider.bounds.size.x * range, boxCollider.bounds.size.y, boxCollider.bounds.size.z),
            0, Vector2.left, 0, playerLayer);

        return hit.collider != null;
    }
    private void OnDrawGizmos()
    {
        Gizmos.color = Color.red;
        Gizmos.DrawWireCube(boxCollider.bounds.center + transform.right * range * transform.localScale.x *
colliderDistance,
            new Vector3(boxCollider.bounds.size.x * range, boxCollider.bounds.size.y, boxCollider.bounds.size.z));
    }
}
```

Health.cs

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

public class Health : MonoBehaviour
{
    [Header("Health")]
    [SerializeField] private float startingHealth;
    public float currentHealth { get; private set; }
    private Animator anim;
    private bool dead;

    [Header("iFrames")]
    [SerializeField] private float iFramesDuration;
    [SerializeField] private int numberOfFlashes;
    private SpriteRenderer spriteRend;

    [Header("Components")]
    [SerializeField] private Behaviour[] components;
    private bool invulnerable;

    [Header("Death Sound")]
    [SerializeField] private AudioClip deathSound;
    [SerializeField] private AudioClip hurtSound;

    private void Awake()
    {
        currentHealth = startingHealth;
        anim = GetComponent<Animator>();
        spriteRend = GetComponent<SpriteRenderer>();
    }
    public void TakeDamage(float _damage)
    {
        if (invulnerable) return;
        currentHealth = Mathf.Clamp(currentHealth - _damage, 0, startingHealth);

        if (currentHealth > 0)
        {
            anim.SetTrigger("hurt");
            StartCoroutine(Invunerability());
            SoundManager.instance.PlaySound(hurtSound);
        }
        else
        {
            if (!dead)
            {

                foreach (Behaviour component in components)
                    component.enabled = false;

                anim.SetBool("grounded", true);
                anim.SetTrigger("die");

                dead = true;
                SoundManager.instance.PlaySound(deathSound);
            }
        }
    }
    public void AddHealth(float _value)
    {
        currentHealth = Mathf.Clamp(currentHealth + _value, 0, startingHealth);
    }
```

```
    private IEnumerator Invunerability()
    {
        invulnerable = true;
        Physics2D.IgnoreLayerCollision(10, 11, true);
        for (int i = 0; i < numberOfFlashes; i++)
        {
            spriteRend.color = new Color(1, 0, 0, 0.5f);
            yield return new WaitForSeconds(iFramesDuration / (numberOfFlashes * 2));
            spriteRend.color = Color.white;
            yield return new WaitForSeconds(iFramesDuration / (numberOfFlashes * 2));
        }
        Physics2D.IgnoreLayerCollision(10, 11, false);
        invulnerable = false;
    }
    private void Deactivate()
    {
        gameObject.SetActive(false);
    }

    public void Respawn()
    {
        AddHealth(startingHealth);
        anim.ResetTrigger("die");
        anim.Play("Idle");
        StartCoroutine(Invunerability());
        dead = false;


        foreach (Behaviour component in components)
            component.enabled = true;
    }
}
```

Healthbar.cs

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

public class Healthbar : MonoBehaviour
{
    [SerializeField] private Health playerHealth;
    [SerializeField] private Image totalhealthBar;
    [SerializeField] private Image currenthealthBar;

    private void Start()
    {
        totalhealthBar.fillAmount = playerHealth.currentHealth / 10;
    }
    private void Update()
    {
        currenthealthBar.fillAmount = playerHealth.currentHealth / 10;
    }
}
```

HealthCollectible.cs

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

public class HealthCollectible : MonoBehaviour
{
    [SerializeField] private float healthValue;
    [SerializeField] private AudioClip pickupSound;
```

```
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.tag == "Player")
        {
            SoundManager.instance.PlaySound(pickupSound);
            collision.GetComponent<Health>().AddHealth(healthValue);
            gameObject.SetActive(false);
        }
    }
}
```

PlayerAttack.cs

```
using UnityEngine;

public class PlayerAttack : MonoBehaviour
{
    [SerializeField] private float attackCooldown;
    [SerializeField] private Transform firePoint;
    [SerializeField] private GameObject[] fireballs;
    [SerializeField] private AudioClip fireballSound;

    private Animator anim;
    private PlayerMovement playerMovement;
    private float cooldownTimer = Mathf.Infinity;

    private void Awake()
    {
        anim = GetComponent<Animator>();
        playerMovement = GetComponent<PlayerMovement>();
    }

    private void Update()
    {
        if (Input.GetMouseButton(0) && cooldownTimer > attackCooldown && playerMovement.canAttack())
            Attack();

        cooldownTimer += Time.deltaTime;
    }

    private void Attack()
    {
        SoundManager.instance.PlaySound(fireballSound);

        anim.SetTrigger("attack");
        cooldownTimer = 0;

        fireballs[FindFireball()].transform.position = firePoint.position;
        fireballs[FindFireball()].GetComponent<Projectile>().SetDirection(Mathf.Sign(transform.localScale.x));
    }
    private int FindFireball()
    {
        for (int i = 0; i < fireballs.Length; i++)
        {
            if (!fireballs[i].activeInHierarchy)
                return i;
        }
        return 0;
    }
}
```

PlayerMovement.cs

```
using UnityEngine;
```

```
public class PlayerMovement : MonoBehaviour
{
    [Header("Movement Parameters")]
    [SerializeField] private float speed;
    [SerializeField] private float jumpPower;

    [Header("Coyote Time")]
    [SerializeField] private float coyoteTime;
    private float coyoteCounter;

    [Header("Multiple Jumps")]
    [SerializeField] private int extraJumps;
    private int jumpCounter;

    [Header("Wall Jumping")]
    [SerializeField] private float wallJumpX;
    [SerializeField] private float wallJumpY;

    [Header("Layers")]
    [SerializeField] private LayerMask groundLayer;
    [SerializeField] private LayerMask wallLayer;

    [Header("Sounds")]
    [SerializeField] private AudioClip jumpSound;

    private Rigidbody2D body;
    private Animator anim;
    private BoxCollider2D boxCollider;
    private float wallJumpCooldown;
    private float horizontalInput;

    private void Awake()
    {

        body = GetComponent<Rigidbody2D>();
        anim = GetComponent<Animator>();
        boxCollider = GetComponent<BoxCollider2D>();
    }

    private void Update()
    {
        horizontalInput = Input.GetAxis("Horizontal");

        if (horizontalInput > 0.01f)
            transform.localScale = Vector3.one;
        else if (horizontalInput < -0.01f)
            transform.localScale = new Vector3(-1, 1, 1);

        anim.SetBool("run", horizontalInput != 0);
        anim.SetBool("grounded", isGrounded());

        if (Input.GetKeyDown(KeyCode.Space))
            Jump();

        if (Input.GetKeyUp(KeyCode.Space) && body.velocity.y > 0)
            body.velocity = new Vector2(body.velocity.x, body.velocity.y / 2);

        if (onWall())
        {
            body.gravityScale = 0;
            body.velocity = Vector2.zero;
        }
        else
        {
```

```
        body.gravityScale = 7;
        body.velocity = new Vector2(horizontalInput * speed, body.velocity.y);

        if (isGrounded())
        {
            coyoteCounter = coyoteTime;
            jumpCounter = extraJumps;
        }
        else
            coyoteCounter -= Time.deltaTime;
    }
}

private void Jump()
{
    if (coyoteCounter <= 0 && !onWall() && jumpCounter <= 0) return;

    SoundManager.instance.PlaySound(jumpSound);

    if (onWall())
        WallJump();
    else
    {
        if (isGrounded())
            body.velocity = new Vector2(body.velocity.x, jumpPower);
        else
        {

            if (coyoteCounter > 0)
                body.velocity = new Vector2(body.velocity.x, jumpPower);
            else
            {
                if (jumpCounter > 0)
                {
                    body.velocity = new Vector2(body.velocity.x, jumpPower);
                    jumpCounter--;
                }
            }
        }
        coyoteCounter = 0;
    }
}

private void WallJump()
{
    body.AddForce(new Vector2(-Mathf.Sign(transform.localScale.x) * wallJumpX, wallJumpY));
    wallJumpCooldown = 0;
}


private bool isGrounded()
{
    RaycastHit2D raycastHit = Physics2D.BoxCast(boxCollider.bounds.center, boxCollider.bounds.size, 0, Vector2.down,
0.1f, groundLayer);
    return raycastHit.collider != null;
}
private bool onWall()
{
    RaycastHit2D raycastHit = Physics2D.BoxCast(boxCollider.bounds.center, boxCollider.bounds.size, 0, new
Vector2(transform.localScale.x, 0), 0.1f, wallLayer);
    return raycastHit.collider != null;
}
public bool canAttack()
{
    return horizontalInput == 0 && isGrounded() && !onWall();
```

```
    }
}
```

PlayerRespawn.cs

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


public class PlayerRespawn : MonoBehaviour
{
    [SerializeField] private AudioClip checkpoint;
    private Transform currentCheckpoint;
    private Health playerHealth;
    private UIManager uiManager;

    private void Awake()
    {
        playerHealth = GetComponent<Health>();
        uiManager = FindObjectOfType<UIManager>();
    }

    public void RespawnCheck()
    {
        if (currentCheckpoint == null)
        {
            uiManager.GameOver();
            return;
        }

        playerHealth.Respawn();
        transform.position = currentCheckpoint.position;
        Camera.main.GetComponent<CameraController>().MoveToNewRoom(currentCheckpoint.parent);
    }
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.gameObject.tag == "Checkpoint")
        {
            currentCheckpoint = collision.transform;
            SoundManager.instance.PlaySound(checkpoint);
            collision.GetComponent<Collider2D>().enabled = false;
            collision.GetComponent<Animator>().SetTrigger("appear");
        }
    }
}
```

Projectile.cs

```
using UnityEngine;

public class Projectile : MonoBehaviour
{
    [SerializeField] private float speed;
    private float direction;
    private bool hit;
    private float lifetime;

    private Animator anim;
    private BoxCollider2D boxCollider;

    private void Awake()
    {
        anim = GetComponent<Animator>();
        boxCollider = GetComponent<BoxCollider2D>();
```

```
    }
    private void Update()
    {
        if (hit) return;
        float movementSpeed = speed * Time.deltaTime * direction;
        transform.Translate(movementSpeed, 0, 0);

        lifetime += Time.deltaTime;
        if (lifetime > 5) gameObject.SetActive(false);
    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
        hit = true;
        boxCollider.enabled = false;
        anim.SetTrigger("explode");

        if (collision.tag == "Enemy")
            collision.GetComponent<Health>().TakeDamage(1);
    }
    public void SetDirection(float _direction)
    {
        lifetime = 0;
        direction = _direction;
        gameObject.SetActive(true);
        hit = false;
        boxCollider.enabled = true;

        float localScaleX = transform.localScale.x;
        if (Mathf.Sign(localScaleX) != _direction)
            localScaleX = -localScaleX;

        transform.localScale = new Vector3(localScaleX, transform.localScale.y, transform.localScale.z);
    }
    private void Deactivate()
    {
        gameObject.SetActive(false);
    }
}
```

MenuManager.cs

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

public class MenuManager : MonoBehaviour
{
  public void PlayGame()
    {
        Application.LoadLevel("Level1");
    }
    public void Exit()
    {
        Application.Quit();
    }
}
```

UIManager.cs

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

```
public class UIManager : MonoBehaviour
{
    [Header("Game Over")]
    [SerializeField] private GameObject gameOverScreen;
    [SerializeField] private AudioClip gameOverSound;

    [Header("Pause")]
    [SerializeField] private GameObject pauseScreen;

    private void Awake()
    {
        gameOverScreen.SetActive(false);
        pauseScreen.SetActive(false);
    }
    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.Escape))
        {
            PauseGame(!pauseScreen.activeInHierarchy);
        }
    }

    #region Game Over
    public void GameOver()
    {
        gameOverScreen.SetActive(true);
        SoundManager.instance.PlaySound(gameOverSound);
    }

    public void Restart()
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
    }

    public void MainMenu()
    {
        SceneManager.LoadScene(0);
    }
    public void Quit()
    {
        Application.Quit(); //Quits the game (only works in build)

#if UNITY_EDITOR
        UnityEditor.EditorApplication.isPlaying = false; //Exits play mode (will only be executed in the editor)
#endif
    }
    #endregion

    #region Pause
    public void PauseGame(bool status)
    {
        //If status == true pause | if status == false unpause
        pauseScreen.SetActive(status);

        //When pause status is true change timescale to 0 (time stops)
        //when it's false change it back to 1 (time goes by normally)
        if (status)
            Time.timeScale = 0;
        else
            Time.timeScale = 1;
    }
    public void SoundVolume()
    {
        SoundManager.instance.ChangeSoundVolume(0.2f);
```

```
    }
    public void MusicVolume()
    {
        SoundManager.instance.ChangeMusicVolume(0.2f);
    }
    #endregion
}
```

VolumeText.cs

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

public class VolumeText : MonoBehaviour
{
    [SerializeField] private string volumeName;
    [SerializeField] private string textIntro;
    private TextMeshProUGUI txt;

    private void Awake()
    {
        txt = GetComponent<TextMeshProUGUI>();
    }
    private void Update()
    {
        UpdateVolume();
    }
    private void UpdateVolume()
    {
        float volumeValue = PlayerPrefs.GetFloat(volumeName) * 100;
        txt.text = textIntro + volumeValue.ToString();
    }
}
```