Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп'ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня

бакалавра
(назва освітньо-кваліфікаційного рівня)

студента *Садовського Ярослава Олександровича*
(ПІБ)

академічної групи *122-20-4*
(шифр)

напряму підготовки *122 Комп'ютерні науки*
(код і назва напряму підготовки)

на тему: "Розробка кросплатформеного ігрового застосунку на основі технологічного стеку Unity/C#"

Керівники	**Прізвище, ініціали**	**Оцінка за шкалою**		**Підпис**
		рейтинговою	**інституційною**	
кваліфікаційної роботи	*доц. Кабак Л.В*			
розділів:				
спеціальний	*доц. Кабак Л.В*			
економічний	*доц. Касьяненко Л.В.*			
Рецензент	*доц. Каштан В.Ю.*			
Нормоконтролер	*доц. Гуліна І. Г.*			

Дніпро
2024

Міністерство освіти і науки України
НТУ «Дніпровська політехніка»

ЗАТВЕРДЖЕНО:
завідувач кафедри
програмного забезпечення комп'ютерних систем
(повна назва)

М.О. Алексєєв
(підпис) (прізвище, ініціали)

« » 2024 року

ЗАВДАННЯ
на кваліфікаційну роботу

бакалавра
(назва освітньо-кваліфікаційного рівня)

студента *122-20-4* *Садовського Я.О.*
(група) (прізвище та ініціали)

тема кваліфікаційної роботи "Розробка кросплатформеного ігрового застосунку на основі технологічного стеку Unity/C#"

затверджена наказом ректора НТУ «ДП» від *23 . 05 .2024 р.* № *470 -с*

Розділ	Зміст виконання	Термін виконання
Спеціальний	***На основі матеріалів виробничої практики та інших науково-технічних джерел провести аналіз стану рішення проблеми та постановку задачі.*** ***Обґрунтувати вибір та здійснити реалізацію методів вирішення проблеми***	***13.05.2024 р.***
Економічний	***Провести розрахунок трудомісткості розробки програмного забезпечення, витрат на створення ПЗ й тривалості його розробки***	***27.05.2024 р.***

Завдання видав _____ *доц.Кабак Л.В.*
(підпис) (посада, прізвище, ініціали)

Завдання прийняв до виконання _____ _____
(підпис) (прізвище, ініціали)

Дата видачі завдання: *14.01.2024 р.*

Термін подання кваліфікаційної роботи до ЕК: *10.06.2024 р.*

РЕФЕРАТ

Пояснювальна записка: 97 с., 38 рис., 1 табл., 3 дод., 22 джерела.

Об'єкт розробки: кросплатформена гра в жанрі платформер, розроблена з використанням ігрового рушія Unity та мови програмування C#.

Мета кваліфікаційної розробки: створення захопливої відеогри із застосуванням сучасних технологій комп'ютерної розробки.

Вступна частина розповідає про зародження платформерів, їх позитивний вплив на розвиток логічного мислення користувачів.

Перший розділ присвячено огляду предметної галузі платформерів, історії виникнення та еволюції цього жанру відеоігор, а також аналізу культових ігор цього жанру. Сформульовано мету розробки, її актуальність та конкретні завдання.

Другий розділ деталізує процес проектування і розробки створеного ігрового додатку на платформі Unity. Описує проектування ігрової механіки, графічних елементів, анімацій, звуку та інтерфейсу користувача. Наведено структуру програми, алгоритми роботи, вхідні/вихідні дані. Вказано вимоги до апаратного забезпечення та інструкції запуску.

Економічний розділ містить розрахунки трудомісткості створення інформаційної системи, перелік витрат на його розробку та визначення загального терміну виконання проєкту.

Практична цінність полягає у розробці якісного та зручного у використанні ігрового застосунку, динамічним геймплеєм та сучасною візуалізацією для отримання задоволення від ігрового процесу користувачами різних вікових категорій.

Актуальність обумовлена постійно зростаючим попитом на нові комп'ютерні ігри, що надає можливість застосувати передові технології розробки для створення конкурентоспроможного продукту.

Список ключових слів: ГРА, UNITY, ПЛАТФОРМЕР, СИСТЕМА, НАЛАШТУВАННЯ, РУШІЙ, ПЕРСОНАЖ, КОРИСТУВАЧ, ПРОГРАМА.

ABSTRACT

Explanatory note: 97 p., 38 fig., 1 table, 3 extra., 22 sources.

Object of development: a cross-platform game in the platformer genre, developed using the unity game engine and the c# programming language.

The purpose of the qualification development: creating an exciting video game using modern computer development technologies.

The introductory part describes the origin of platformers and their positive impact on the development of users' logical thinking.

The first section is devoted to an overview of the subject area of platformers, the history of the emergence and evolution of this genre of video games, as well as an analysis of cult games in this genre. The purpose of the development, its relevance, and specific tasks are formulated.

The second section details the design and development process of the created game application on the unity platform. It describes the design of game mechanics, graphic elements, animations, sound, and user interface. The structure of the program, working algorithms, input/output data are given. Hardware requirements and launch instructions are specified.

The economic section contains calculations of the labor intensity of creating the information system, a list of costs for its development, and the determination of the overall project execution time.

The practical value lies in the development of a high-quality and user-friendly game application with dynamic gameplay and modern visualization for users of different age categories to enjoy the gaming process.

The relevance is due to the constantly growing demand for new computer games, which provides an opportunity to apply advanced development technologies to create a competitive product that can become the basis for further projects in this field.

List of keywords: GAME, UNITY, PLATFORMER, APPLICATION, SETTINGS, ENGINE, CHARACTER, USER, PROGRAM.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ПК – персональний комп’ютер;

GPU – графічний процесор;

ЦП – центральний процесор;

ОП – оперативна пам’ять;

ШІ – штучний інтелект;

ОС – операційна система;

UNITY – багатоплатформовий інструмент для розробки відеоігор і застосунків;

C# – мова програмування;

EXE – розширення виконуваного файлу, що застосовується в системах Microsoft Windows;

APK – формат архівних виконуваних файлів-програм для Android;

UI – інтерфейс користувача (user interface);

ІС – інформаційна система.

ВСТУП

З розвитком технологій та популярністю ігор що росте, ринок ігрових додатків стрімко розширюється, залучаючи нових гравців та створюючи нові можливості для розробників. На цей час існує широкий спектр технічних можливостей для створення ігор, які підходять для різних ігрових пристроїв і підходять до кількох платформ. Це дозволяє виводити програмні продукти на один із найбільш швидкорослих ринків світу, що постійно розвивається.

Комп'ютерні ігри вже давно перестали бути лише джерелом розваг. Вони викликають незабутні емоції та залишають дуже приємні враження, а також дарують унікальний досвід. Навколо ігор швидко утворилися великі спільноти, професійні змагання і навіть нові види спорту.

Сьогодні ігри стали не тільки популярним видом розваги, але й потужною економічною силою. Індустрія ігор значно впливає на інші сфери, такі як фільми й музика. Комп'ютерні ігри пропонують різноманітні жанри: стратегія, будівництво, шутери, спортивні та багато інших. Ігри стали більш реалістичними та інтерактивними завдяки прогресу технологій та доступності ігрових платформ, таких як віртуальна реальність, мобільні пристрої, комп’ютери та консолі.

Особливу увагу серед різних жанрів заслуговують 2D платформери. Цей жанр ігор, який включає такі класичні приклади як Donkey Kong та Pac-Man, привертає увагу гравців своєю простотою та захоплюючим ігровим процесом. Платформери зазвичай пропонують гравцям подолати різноманітні перешкоди та виклики, використовуючи персонажа, який рухається по платформах у двовимірному просторі. Однією з ключових особливостей таких ігор є їхні невисокі технічні вимоги, що робить їх доступними для широкої аудиторії, незалежно від технічних можливостей пристроїв.

Метою даної роботи є розробка власної багаторівневої відеогри у жанрі платформер, який зможе подарувати широкий спектр емоцій. За допомогою мультиплатформеного рушію Unity, що дозволяє створити ігровий додаток, який

буде доступним для різних платформ, таких як персональні комп'ютери, мобільні пристрої та консолі. Це забезпечує широкий доступ до гри та дозволяє залучити максимальну кількість користувачів.

Таким чином, актуальність цієї роботи дуже значна, оскільки вона стосується необхідності розробки нових програмних продуктів, які дозволять ринку індустрії розваг адаптуватися до постійно мінливого середовища інформаційних технологій. Розробка мультиплатформенних ігрових додатків дозволяє задовольнити попит на нові ігрові продукти та сприяє розвитку ігрової індустрії в цілому.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКИ ЗАДАЧІ

1.1. Загальні відомості с предметної галузі

Відеоігри — це інтерактивні програми для розваг, які дозволяють користувачам керувати персонажами або об’єктами на екрані через інтерфейс. Відеоігри є важливою частиною сучасної культури та розваг, оскільки вони дозволяють гравцям зануритися в різноманітні історії та світи. Популярність платформерів, які з’явилися на початок 1980-х років, значно вплинула на розвиток індустрії відеоігор. Сьогодні існує широкий спектр технічних можливостей для створення ігор, які підходять для різних ігрових пристроїв, які підходять для багатьох платформ.

Перші відеоігри з'явилися в 1950-60-х роках як експериментальні проекти в наукових лабораторіях. Однією з перших ігор, що здобула популярність, була "Pong" (1972) від Atari, яка імітувала настільний теніс. Вона стала першою комерційно успішною відеогрою, яка призвела до вибухового зростання ринку відеоігор у 1970-х роках.

Платформери — це жанр відеоігор, в якому гравець керує персонажем, який пересувається по платформах, ухиляється від перешкод і перемагає супротивників. Можливість точно контролювати рух і стрибки персонажа - одна з головних особливостей платформерів. Гра приваблювала гравців різного віку завдяки простому дизайну та захопливому ігровому процесу.

Ця група відеоігор поставила основу для жанра платформерів, встановивши стандарти та напрямки розвитку, які досі впливають на сучасні платформери. Кожен з них привніс нові ідеї та значний внесок у формування жанру, і він залишається актуальним для гравців і розробників протягом багатьох десятиліть.

1. Donkey Kong (1981) (Рис. 1.1)

- Розробник: Nintendo
- Платформи: Аркадні автомати, пізніше випущена для консолей
- Особливості: Однією з перших і найвідоміших платформерів стала гра “Donkey Kong”, розроблена Nintendo в 1981 році. У цій грі гравець керував персонажем на ім’я Джампмен (пізніше відомим як Маріо), який повинен був рятувати принцесу від гігантської мавпи, долаючи різноманітні перешкоди. “Donkey Kong” стала першою грою, яка запропонувала багаторівневий ігровий процес, і заклала основи жанру платформерів.
- Продажі: Протягом 1980-х років Donkey Kong продалася більш ніж 132 тисячами аркадних машин, принісши близько 280 мільйонів доларів доходу. Крім того, гра була портована на різні домашні консолі та комп’ютери, що додатково збільшило її продажі. [1]

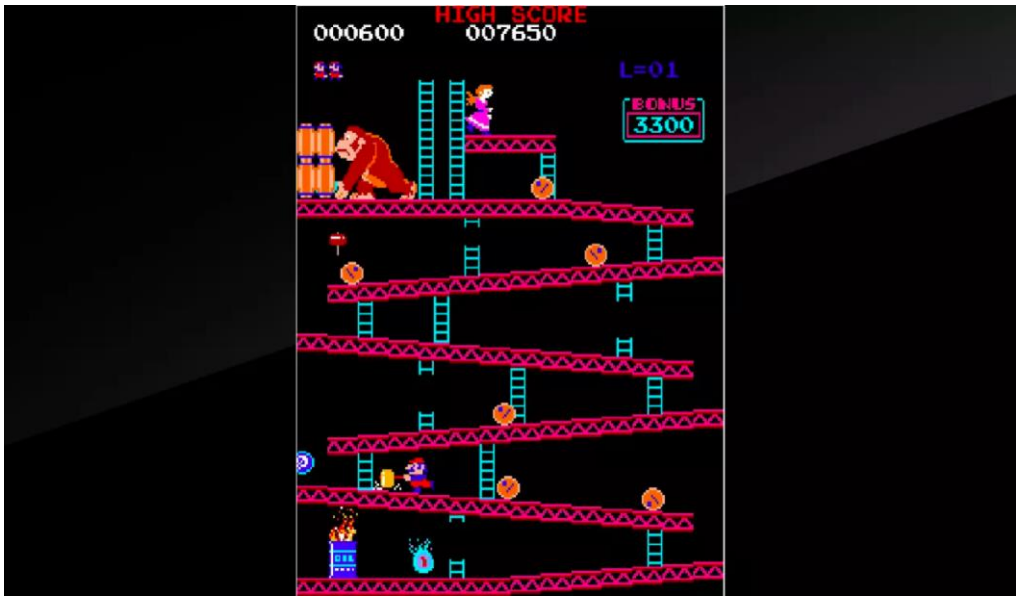

Рис. 1.1 – Інтерфейс гри Donkey Kong

2. Pitfall! (1982) (Рис. 1.2)

- Розробник: Activision
- Платформи: Atari 2600

- Особливості: Гра включає в себе горизонтальне скролювання. У цій грі гравець керував персонажем на ім'я Гаррі, який досліджував джунглі, уникаючи пасток і збираючи скарби.
- Продажі: Pitfall! продалася понад 4 мільйонами копій, ставши однією з найбільш продаваних ігор на Atari 2600. Гра значно вплинула на розвиток жанру платформерів. [2]

Рис. 1.2 – Інтерфейс гри Pitfall!

3. Super Mario Bros. (1985) (Рис. 1.3)

- Розробник: Nintendo
- Платформи: Nintendo Entertainment System (NES)
- Особливості: Це одна з найбільш знакових ігор жанру. “Super Mario Bros.” Встановила нові стандарти для жанра, запропонувавши гравцям багатий ігровий світ з різноманітними рівнями, ворогами та секретами. Гра стала символом Nintendo і закріпила Маріо як одного з найвідоміших персонажів у відеоіграх стрибаючи на ворогів і збираючи бонуси.
- Продажі: Було продано понад 40 мільйонів копій по всьому світу, що зробило її однією з найбільш продаваних відеоігор в історії. Це допомогло NES закріпитися як домінуюча консоль на ринке. [2]

Рис. 1.3 – Інтерфейс гри Super Mario Bros.

4. Prince of Persia (1989) (Рис. 1.4)

- Розробник: Broderbund
- Платформи: Apple II, потім порти на різні платформи
- Особливості: Здобула значний успіх і стала культовою класикою. Ця гра вирізнялася своїми плавними анімаціями та високою складністю. Гравець керує Принцем, який має врятувати принцесу, стрибаючи і долаючи різноманітні пастки у підземеллі.
- Продажі: до початка 1990-х років було продано близько 2 мільйонів копій по всьому світу на різних платформах. Гра була особливо успішною на Apple II, Amiga та IBM PC, і її популярність зростала з роками, коли її портували на інші системи. [1]

Рис. 1.5 – Інтерфейс гри Prince of Persia

5. Sonic the Hedgehog (1991) (Рис. 1.5)

- Розробник: Sega
- Платформи: Sega Genesis/Mega Drive
- Особливості: Sonic the Hedgehog відзначався швидкістю і динамічністю геймплею. Гравці керують Соником, який бореться з доктором Роботником, знищуючи ворогів і збираючи кільця на високій швидкості.
- Продажі: Продано понад 15 мільйонів копій, що зробило її однією з найуспішніших ігор для Sega Genesis. Успіх Sonic допоміг Sega значно збільшити свою частку на ринке консолей. [1]

Рис. 1.6 – Гра Sonic the Hedgehog

У 1990-х роках популярні франшизи, такі як «Super Mario World» і «Donkey Kong Country», сприяли поширенню 2D-відеоігор. Sonic the Hedgehog (1991) від Sega додав нові швидкості та динаміки, що зробило його надзвичайно популярним. Інші ігри, такі як "Earthworm Jim" (1994) і "Crash Bandicoot" (1996) також досягли непоганого успіху на ринке.

Проте, у 90-ті роки розробники та гравці зацікавилися більше тривимірними іграми завдяки прогресу технологій. Коли у 1996 році вийшов 3D-платформер-революціонер "Super Mario 64", він ознаменував новий етап у розвитку жанру головоломок. Ця гра перевернула ідею платформера з ніг на голову, надавши гравцям новий рівень свободи та можливості спілкуватися з ігровим світом. Це призвело до створення 3D-анімації, яка використовувалася в таких відеоіграх, як Banjo-Kazooie (1998), а також Spyro the Dragon (1998).

Поступово платформери почали втрачати популярність у відеоіграх, поступаючись іншим жанрам, таким як шутери від першої особи та рольові ігри. Згідно з дослідженнями, у 2002 році частка платформерів на ринке відеоігор впала з 15 відсотків у 1998 році до лише 2 відсотків у 2002 році. [3]

У цьому сенсі 90-ті роки стали переломним моментом для жанра платформерів, оскільки розробники та гравці почали відкривати нові

перспективи завдяки переходу на 3D-технології, одночасно зберігаючи традиційні 2D-ігри, які були дорогими для багатьох гравців.

2D платформери не зникли повністю, не зважаючи на спад. Завдяки зусиллям незалежних розробників на початку 2000-х років цей жанр знову став популярним. Ігри, такі як Braid і Super Meat Boy, продемонстрували, що 2D платформери можуть бути сучасними, захопливими та викликати у гравців спогади. Ці відеоігри пропонують унікальний ігровий досвід, використовуючи нові методи геймплею та візуального стилю.

Відродження ігор цього жанру також було підтримано великими студіями. З випуском New Super Mario Bros і Donkey Kong Country Returns розробники довели, що вони все ще пропонують одні з найдоступніших розваг для багатьох гравців, і 2D платформери все ще можуть бути комерційно успішними. [4]

1.2. Призначення розробки та галузь застосування

Відеоігри завжди піддавалися критиці за те, що вони нібито розбещують дітей, які люблять грати в них. Більшість цих скарг надходить від стурбованих батьків та організацій, які просто не хочуть, щоб їхні діти піддавалися впливу чогось небезпечного. Але всупереч цим звинуваченням, наука постійно демонструє прихильність до відеоігор. Дослідження з психології та вивчення мозку показали, що помірна кількість відеоігор має низку переваг. При правильному використанні ігри корисні для дітей і дорослих, незалежно від віку. [5]

Відеоігри розвивають мозок і покращують його здатність зберігати і запам'ятовувати інформацію, як короткострокову, так і довгострокову. Також допомагають загострити вашу увагу та зменшити імпульсивність. Оскільки протягом усієї гри ви працюєте над досягненням певної мети, ви вчитеся зосереджувати та спрямовувати всі свої дії на цю мету.

Оскільки у іграх все відбувається дуже швидко, ваш мозок також тренується швидко обробляти інформацію. Було виявлено, що люди, які часто

грають, обробляють інформацію швидше, ніж інші, і при цьому приймають розумні рішення для досягнення мети. Таким чином, можна стверджувати, що при помірному і розумному використанні відеоігри можуть стати корисним інструментом для навчання та розвитку, надаючи позитивний вплив на мозок і психічне здоров'я.

Метою цієї кваліфікаційної роботи є створення ігрового додатку, який може бути використано для відпочинку, зниження рівеню стресу та розвитку логічного мислення.

1.3. Підстава для розробки

Розробка (виконання кваліфікаційної роботи) має такі підстави:

- навчання за спеціальністю 122 Комп’ютерні науки;
- навчальний план та графік навчального процесу;
- наказ ректора НТУ «ДП» від 23.05.2024 № 469-с;
- тема кваліфікаційної роботи “Розробка кросплатформеного ігрового застосунку на основі технологічного стеку Unity/C#”

1.4. Постановка завдання

Метою кваліфікаційної роботи є створення гри в жанрі "платформер" для різних платформ, таких як Windows, MacOS та Android, з використанням двигуна Unity та мови C#.

Правила гри наступні: на кожному рівні герою потрібно подолати всіх монстрів. Після перемоги над всіма монстрами, потрібно знайти магічний камінь, натиснути клавішу “E” і рівень закінчиться. Якщо гравець отримує надто багато урону, тоді в нього закінчиться життя і він програє. При падінні с платформи рівень почнеться заново.

Для успішної реалізації ІС потрібно звернути увагу на такі етапи:

– аналіз цільової аудиторії гри;

– проєктування концепції гри;
– розробка механіки гри;
– створення різноманітних рівнів;
– створення ворогів;
– створення інтерфейсу користувача;
– проведення тесту для виявлення та виправлення помилок;
– оптимізація гри під різні платформи.

Створена інформаційна система повинен бути функціональною грою, яка реалізовує наступні функції:

- Пересування головного героя.
- Поставити гру на паузу у любий час.
- Відкрити меню налаштувань.
- Змінити гучність.
- Обрати рівень для проходження.
- Вийти з гри.

1.5. Вимоги до програми або програмного виробу

1.5.1. Вимоги до функціональних характеристик

Інформаційна система повинна відповідати певним функціональним вимогам:

– інтуїтивно зрозумілий та зручний користувацький UI інтерфейс до різних розмірів екрану;
– бути мультиплатформеним, справно працювати на таких платформах: Windows, MacOS та Android забезпечуючи стабільну та плавну роботу на кожній з них;
– зберігати налаштування користувача та прогрес між сенсами.

1.5.2. Вимоги до інформаційної безпеки

Для забезпечення безпеки ІС користувача має бути наявним такий перелік умов:

1. Забезпечення неможливості передачі даних до третьої сторони.
2. Забезпечення захисту конфіденційності даних користувача.
3. Збереження цілісності даних у випадку збою системи.
4. Локальне збереження даних для підвищення захищеності.

1.5.3. Вимоги до складу та параметрів технічних засобів

Для запуска відеогри на ОС Windows та MacOS, потрібно щоб характеристика була приближена до таблиці (табл. 1.1)

Таблиця 1.1

Рекомендована конфігурація для запуска гри на ПК

Тип	Модель	Характеристики
Процесор	Intel Pentium U5400	2 ядра по 1,07 ГГц
Відеокарта	GeForce GTX 670	GK104, GDDR5 128-bit2048mb
ОП	GR2400D464L17S	DDR4, 2048MB, 2400 MHz
Жорсткий диск (SSD)	ST1000LM048	1TB, 5400RPM, 128MB, 2.5
Материнська плата	ASUS P8B75-V	Socket 1155, Intel B75
Блок живлення	GameMax GM	600W, ATX 20+4pin

Рекомендовані характеристики для Android:

− ОС: 4.0 (API 14) і вище;

− ЦП: ARMv6 (32-bit);

− Відеоадаптер: з підтримкою OpenGL ES 2.0;

− Відеопам’ять: 64 МБ;

− Накопичувач: 100+ МБ вільного місця на диске;

− ОП: 512 МБ.

1.5.4. Вимоги до інформаційної та програмної сумісності

Для коректної праці додатку потрібен DirectX, він є важливим компонентом для розробки ігор, оскільки він надає розробникам доступ до апаратних засобів обчислень та графіки. Він забезпечує ефективне управління ресурсами системи, що дозволяє створювати складні та реалістичні візуальні ефекти. Наприклад, DirectX 12 надає можливість використовувати кілька ядер процесора для розподілу задач рендерингу, що покращує продуктивність ігрових додатків, особливо на системах з потужними GPU. Це знижує навантаження на процесор, що призводить до більш стабільних та високих частот кадрів у грі. [6]

Оскільки при розробці ігрового додатку на C# використовується рушій Unity, він сумісний майже з усіма сучасними ОС. Тим не менш, необхідно дотримуватися вимог, зазначених у розділі 1.5.3. Оновлення можна встановити за допомогою стандартних функцій ринків Windows, MacOS та Android/iOS.

РОЗДІЛ 2
ПРОЄКТУВАННЯ ТА РОЗРОБКА ІНФОРМАЦІЙНОЇ СИСТЕМИ

2.1. Функціональне призначення системи

Комп'ютерна гра "Sky Adventure" розроблена як крос-платформний проєкт для Windows, Linux, MacOS та Android на рушії Unity. Жанр гри - піксельний 2D платформер. Мова програмування для написання ігрової логіки - C#. [21]

Основна концепція гри - класичний геймплей платформера з піксельною ретро-графікою. Мета полягає у подоланні кожного рівня шляхом пересування персонажа від старту до речі за допомогою якої можна пройти рівень. Для цього потрібно долати різноманітні перешкоди, уникати пасток, знищувати ворогів та знайти магічний камінь, який активує портал і дозволяє завершити рівень. Керування персонажем відбувається за допомогою клавіатури або сенсорного екрану на мобільних пристроях.

Одną з ключових особливостей гри є ретельно налаштована фізична модель, що створює реалістичне відчуття ваги та інерції під час руху і стрибків персонажа. Це значно ускладнює управління і вимагає від гравця вправності і вміння розраховувати траєкторії стрибків. На кожному рівні грає унікальна музична композиція, вона підіймає гравцям настрій та допомагає розважатися.

Основними ігровими механіками у "Sky Adventure" є стрибки, боротьба з ворогами та подолання перешкод. Стрибки є визначальним елементом, що забезпечує просування по рівнях. Від гравця вимагається постійно слідкувати за напрямком стрибка для уникнення загроз. Вороги є перешкодами, яких потрібно обережно оминати або знешкоджувати. Крім того, рівні насичені різноманітними пастками - шипами, нерухомими чи рухомими перешкодами, обвалами платформ.

Противниками у грі виступають піксельні монстри. Деякі з них є відносно простими і рухаються по зациклених траєкторіях. Також є монстри які полюють на гравця та переслідують його. Для проходження рівня необхідно впоратися з

усіма ворогами, обережно ухилятися від їх атак, підгадати найкращий момент та зробити контратаку. Персонаж гравця має обмежену кількість життів. Якщо запас життів закінчиться, гравцеві буде зараховано поразку, тоді прийдеться проходити рівень спочатку. При падінні в прірву, гра перезапускається з його початку. Прогрес, накопичений на рівні, не зберігається.

Управління в "Sky Adventure" реалізовано лише для клавіатури та сенсорних пристроїв на базі Android. Режим сенсорного керування активується автоматично при виявленні мобільного пристрою. Підтримки геймпадів або будь-якого іншого способу введення не передбачені. Для навігації меню гри використовується лише комп’ютерна мишка або сенсорний екран. Головне меню містить пункти для вибору рівня, налаштувань гри та виходу. Важливою особливістю є блокування доступу до наступного рівня, поки не буде пройдено поточний, що змушує гравця послідовно проходити всі рівні по черзі без можливості пропуску будь-якого рівню. Під час проходження рівня, при необхідності, можна викликати меню паузи, яке дозволяє переглянути схему керування та повернутися в головне меню.

Візуальне оформлення "Sky Adventure" повністю виконане в піксельному стилі. Крім ретро-графіки, ця концепція проявляється у деяких елементах ігрового процесу та обмеженнях. Наприклад, персонаж може рухатися лише горизонтально праворуч/ліворуч та вертикально стрибати - без можливості виконувати складні маневри або застосовувати додаткові дії. Аналогічно, напрямок стрибка задається перед його початком за допомогою переміщення вліво/вправо, а під час самого стрибка траєкторію вже не можна змінити. Все це накладає суворі рамки, притаманні класичним платформерам, і вимагає від гравця чіткого планування та відпрацьованої реакції.

2.2. Опис застосованих математичних методів

Під час розробки цього ПД використання складних математичних методів не було необхідним, оскільки функціональні вимоги обмежувалися реалізацією ігрового процесу в жанрі двовимірного платформера. Відповідно, для досягнення поставлених цілей застосовувалися лише базові арифметичні операції та математичні функції зі стандартної бібліотеки Mathf, що надається середовищем розробки Unity.

У структурі SecondOrderDynamics, яка реалізує плавність і дієздатність рухомої платформи, задіяні такі математичні методи:

1. Чисельне розв'язання диференціальних рівнянь. Для моделювання плавного руху платформи з урахуванням інерції використовувалося чисельне інтегрування звичайного диференціального рівняння другого порядку, що описує коливальну систему. Це забезпечує поступову зміну швидкості платформи без різких ривків.
2. Векторні операції. Положення платформи і розмір пов'язаних візуальних елементів з нею був заданий за допомогою операцій додавання і масштабування векторів.
3. Лінійна інтерполяція застосовувалася для плавної зміни цільової довжини переміщення платформи на кожному кроці моделювання залежно від заданої швидкості переміщення і часу між кадрами.
4. Перевірка граничних умов. Поточний стан платформи (підйом або опускання) визначали шляхом порівняння її положення з допустимим діапазоном довжин переміщення.
5. Базові арифметичні дії. Додавання, віднімання, множення і ділення, використовували для обчислення допоміжних змінних і коефіцієнтів диференціального рівняння.

Таким чином, не було необхідності використовувати складні математичні методи, базові арифметичні операції та функції, доступні в бібліотеці Mathf,

дозволили реалізувати всі алгоритми, необхідні для правильної роботи ігрового застосування в жанрі платформера.

2.3. Опис використаних технологій та мов програмування

Для створення 2D гри "Sky Adventure" використовувались такі програми: рушій Unity 2022.3.26f1, Microsoft Visual Studio Code, Aseprite 1.3 та мова програмування C#.

Unity з'явився внаслідок ідеї данської компанії Over The Hedge, пізніше перейменованої на Unity Technologies. Засновники поставили перед собою амбітну мету - розробити універсальний ігровий рушій, який міг би працювати на різних платформах та операційних системах.

Після кількох років наполегливої праці, у 2005 році побачила світ перша версія Unity. Справжнім проривом для Unity стала повна кросплатформеність, досягнута у 2008 році. Відтоді рушій дозволяв створювати ігри та додатки для персональних комп'ютерів, мобільних пристроїв iOS та Android. Для Unity це відкрило нові горизонти та масштабне коло потенційних користувачів. [7]

На сьогоднішній день Unity є одним з найпопулярніших ігрових рушіїв у світі. Його використовують як великі студії для розробки масштабних комерційних проектів, так і незалежні розробники для створення ігор різного розміру та складності. Інтерфейс платформи. (рис. 2.1)

Рис. 2.1 – Інтерфейс рушія Unity

Unity 2D - це спеціальний набір інструментів в ігровому рушії Юніті, розроблений для створення двовимірних ігор та додатків. Він надає розробникам різноманітні можливості для роботи з 2D-графікою, спрайтами, анімацією, фізикою, користувацьким інтерфейсом та багатьма іншими аспектами.

Unity має спеціальні компоненти та інструменти, оптимізовані для роботи з двовимірним середовищем. Наприклад, компонент Sprite Renderer дозволяє легко відображати та маніпулювати спрайтами (двовимірними зображеннями), а інструмент Sprite Editor спрощує процес створення та редагування спрайтів. Unity підтримує різні методи рендерингу: традиційний рендеринг спрайтів, рендеринг з використанням тайлів (tilemap) (плиток для створення рівнів) і рендеринг з фізикою (для реалістичної взаємодії об'єктів).

Перемикання між 2D і 3D режимами та комбінування елементів в проектах Unity є однією з найбільш цікавих і унікальних можливостей цього ігрового рушія. Така унікальна відмінність дозволяє розробникам легко поєднувати двовимірні та тривимірні об'єкти, персонажів, середовища та інші ігрові елементи. Наприклад, можна створити основний ігровий світ у 3D, а інтерфейс

користувача, спрайти персонажів чи деякі декоративні елементи зробити у 2D. Або навпаки, світ та персонажі можуть бути 2D, а окремі об'єкти чи спецефекти - у 3D. Така гнучкість відкриває широкі творчі можливості для створення неповторних і захопливих ігор. Розробники можуть експериментувати, знаходити ідеальне поєднання 2D та 3D елементів, щоб досягти бажаного стилю, атмосфери та ігрового процесу. Це дозволяє створювати по-справжньому унікальні ігрові проекти, які виділятимуться серед інших.

Що стосується системи анімації у Unity 2D, то вона надзвичайно гнучка і потужна. Вона дозволяє створювати плавні, реалістичні та виразні рухи персонажів, об'єктів, спрайтів та інших ігрових елементів. Розробники можуть використовувати різні техніки анімації: ключові кадри та ручна прорисовка рухів, так і новітні методи, такі як кісткова анімація. Кісткова анімація - це технологія, яка імітує реальний рух кісток скелету, створюючи дуже природні та реалістичні анімації. Вона широко використовується у 3D анімації, але також підтримується і в Unity 2D. З її допомогою розробники можуть створювати надзвичайно деталізовані та виразні рухи персонажів, їхніх кінцівок, облич та інших частин тіла.

Однією з найсильніших сторін Unity є його кросплатформеність. [8] Ця особливість дає змогу розробникам створювати ігри та додатки один раз, а потім легко розгортати їх на безлічі різних платформ і пристроїв без витрати зусиль на портування.

Кросплатформеність означає, що Unity підтримує широкий спектр операційних систем, включно з ПК (Windows, macOS, Linux), мобільними пристроями (iOS, Android), ігровими консолями (PlayStation, Xbox, Nintendo Switch), веб-браузерами, віртуальною та доповненою реальністю. Незалежно від того, на якій платформі розробляється проєкт, Unity надає єдиний набір інструментів та інтерфейс для розроблення, що значно спрощує процес створення кросплатформних додатків. Завдяки цій особливості, розробникам не потрібно писати окремий код або створювати окремі версії своїх проектів для кожної платформи. Замість цього вони можуть зосередитися на створенні однієї

основної версії додатка, яку потім можна легко скомпілювати та запустити на різних платформах за допомогою вбудованих інструментів Unity.

Кросплатформеність також забезпечує економію часу і ресурсів. Замість того, щоб наймати окремі команди розробників для кожної платформи, студії можуть використовувати одну команду, що працює в єдиному робочому процесі та з єдиною кодовою базою. Це значно знижує витрати і підвищує ефективність розробки. Кросплатформеність Unity дає змогу розробникам охопити максимально широку аудиторію, не обмежуючись лише однією або кількома платформами. Їхні ігри та додатки можуть бути доступні на безлічі пристроїв, що значно розширює кількість користувачів і розміри доходу. Зі сторони розробників це дуже вигідно.

Одним із ключових понять в Unity є концепція ассетів (assets). Асети - це будь-які ресурси, наприклад: 2D-3D моделі, текстури, звуки, шрифти, сценарії, анімації, спрайти. Все що використовуються для створення ігор або застосунків усередині Unity.

Кожен проєкт має власну папку ассетів, куди розробники імпортують і організовують усі необхідні ресурси. Ця папка являє собою свого роду бібліотеку, де зберігаються всі елементи, з яких "будується" гра або застосунок. На (рис. 2.2) зображено структуру папки Assets моєї гри.

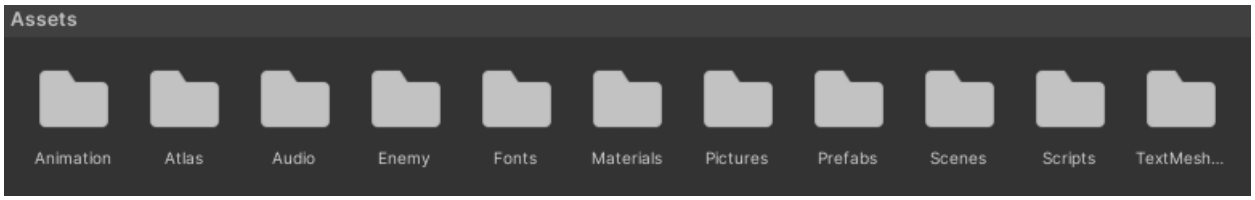

Рис. 2.2 – Файли папки Assets

Unity підтримує широкий спектр форматів ассетів, включно з найпоширенішими: .fbx, .obj, .png, .jpg, .mp3, .wav і багато інших. Це дає змогу розробникам використовувати різні зовнішні інструменти та програми для створення і редагування ассетів, а потім імпортувати їх у проєкт. До простого

імпорту ресурсів, Unity також надає інструменти для їх подальшої обробки та оптимізації всередині середовища розробки:

1. Налаштування стиснення текстур: ця функція дозволяє встановлювати різні параметри стиснення для текстур: формат стиснення (ETC, DXT, PVRTC), якість стиснення, максимальний розмір текстури. Це дає змогу знайти оптимальний баланс між якістю зображення та розміром файлу, забезпечуючи ефективніше використання пам'яті та покращення продуктивності.
2. Імпорт анімацій: Unity підтримує імпорт анімацій з різноманітних зовнішніх пакетів: Aseprite, Maya, Blender. Під час імпорту анімацій можна налаштовувати різні параметри, імпортувати лише потрібні кадри чи змінювати частоту кадрів. Також є можливість застосовувати кісткову анімацію для створення плавних та реалістичних рухів персонажів.
3. Створення навігаційних сіток для ШІ: Unity має вбудовані інструменти для створення навігаційних сіток (NavMesh), які використовуються для забезпечення штучного інтелекту та навігації ігрових істот (ворогів, союзників) в середовищі гри. Ці інструменти дозволяють автоматично генерувати навігаційні сітки на основі геометрії ігрового світу, а також налаштовувати різні параметри, такі як область навігації, перешкоди та області, заборонені для переміщення.
4. Рендеринг: Також є численні інструменти для налаштування та оптимізації процесу рендерингу, що впливає на візуальну якість та продуктивність гри. Наприклад, можна налаштовувати параметри затінення, освітлення, застосовувати пост-обробку зображень, використовувати “запікання” освітлення.

Ці інструменти дозволяють якісно налаштовувати різні аспекти ігрових ресурсів та процесів безпосередньо в середовищі Unity, без необхідності використовувати додаткове стороннє програмне забезпечення. Це забезпечує більш зручний та ефективний робочий процес, дозволяючи оптимізувати ігрові

проєкти для досягнення кращої продуктивності, візуальної якості та загального досвіду гравця. [9]

Одна з найефективніших і найбільш корисних функцій ассетів – це наявність змоги повторного використання. Дозволено легко переміщати й переносити ассети між різними сценами одного проєкту та зовсім різними проектами. Ця функція є найпростішою та в одночас дуже ефективною. Завдяки перевикористовуваності асетів, не потрібно створювати ідентичні ресурси, такі як спрайти, моделі, текстури, анімації чи скрипти, спочатку для кожної нової сцени або проєкту. Натомість, можливо імпортувати вже існуючі, раніше створені ассети і використовувати їх повторно у своїх нових розробках. Використання одних і тих самих асетів дозволяє зберегти візуальну цілісність гри.

Ще однією з найважливіших та найбільш корисних концепцій у середовищі Unity є префаби (Prefabs). Вони представляють собою заготовки для ігрових об'єктів, що складаються з кількох компонентів. Префаби проєкту продемонстровано на (рис. 2.3)

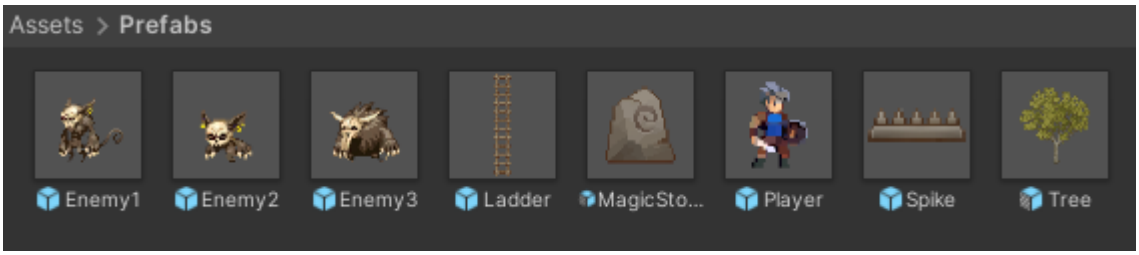

Рис. 2.3 – Заготовлені префаби

Щоб створити префаб, потрібно об'єднати різноманітні компоненти, такі як спрайти, скрипти, текстури, елементи фізики, в єдиний об'єкт усередині сцени рушія. Після цього даний об'єкт можна зберегти як префаб у теці ресурсів проєкту. Також префаби можуть бути вкладені один в одного, створюючи ієрархічну структуру. Це дозволяє створювати більш складні композитні об'єкти, що складаються з кількох взаємопов'язаних префабів. Приклад зображено на (рис. 2.4)

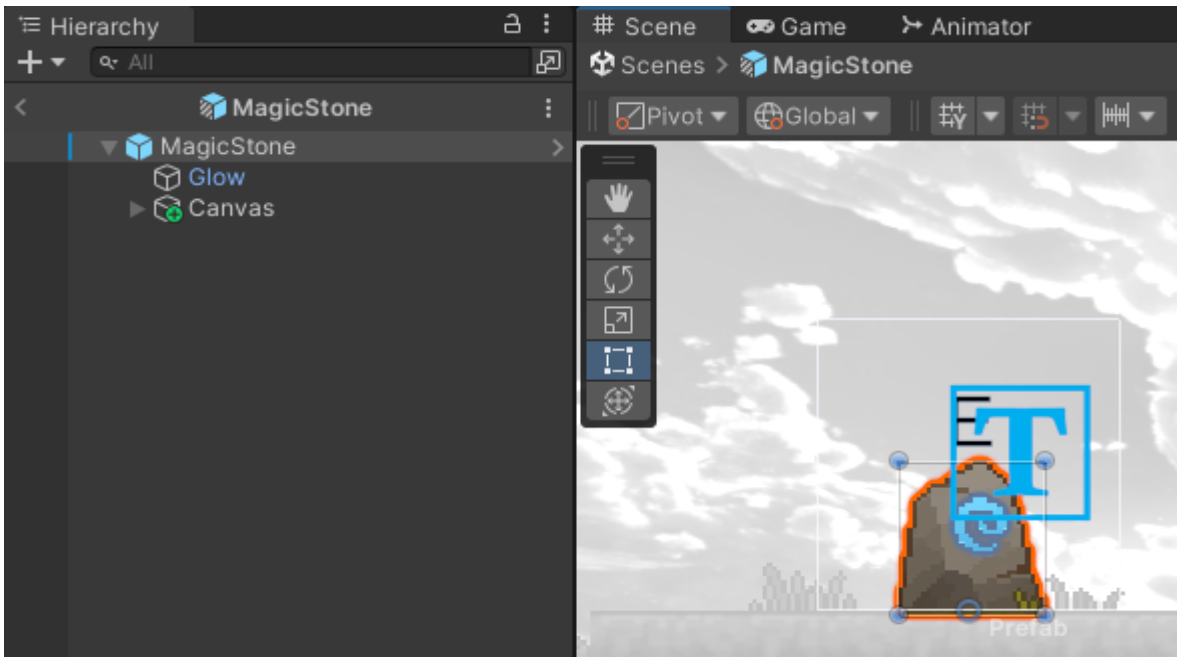

Рис. 2.4 – Ієрархічна структура префабу MagicStone

Unity пропонує розширені можливості для гнучкої роботи з префабами. [10] Існує опція переважування (overriding), яка дозволяє змінювати окремі компоненти екземпляра префабу, не змінюючи при цьому сам вихідний префаб. Таким чином, можна створювати різні варіації однакових об'єктів з індивідуальними налаштуваннями.

Основна перевага префабів полягає в тому, що після їх створення вони можуть бути багаторазово використані. При цьому всі екземпляри матимуть однакові компоненти та структуру, задані під час створення префабу. Якщо внести зміни до будь-якого компонента префабу, ці зміни автоматично застосуються до всіх його екземплярів у проекті. Ця особливість є надзвичайно корисною для об'єктів, що повторюються в грі, таких як вороги, будівлі, елементи оточення. Замість того, щоб налаштовувати кожен об'єкт з нуля, краще створити префаб після створеного та налаштованого об’єкта, а після застосовувати при потребі.

Варто зазначити, що концепція префабів не є унікальною для Unity, аналогічні рішення існують і в інших ігрових рушіях. Проте, реалізація префабів у Unity вважається однією з найбільш продуманих та зручних у використанні.

Інтерфейс Unity в моему випадку має такі компоненти: вкладка Scene, Game, Project, Console, Hierarchy, Animation, Animator, Inspector, панель інструментів, Tile Palette, Lighting.

Вкладка Scene в Unity є одним з найважливіших інструментів для розміщення та створення ігрових об'єктів. Вона представляє собою своєрідну "сцену" або робочий простір, де можна конструювати ігрові рівні, розміщуючи та маніпулюючи різноманітними об'єктами, такими як моделі, освітлення, камери, колайдери та інші компоненти.

Основні функції вкладки Scene: [11]

1. Створення та розміщення об'єктів: Створювати нові ігрові об'єкти або розміщувати існуючі префаби в межах сцени. Це формує основу ігрового світу.
2. Маніпуляція об'єктами: Об'єкти в сцені можна переміщувати, обертати, масштабувати та змінювати їх властивості.
3. Ієрархія об'єктів: Вкладка Scene дозволяє створювати ієрархічні структури об'єктів, де одні об'єкти є дочірніми по відношенню до інших, забезпечуючи організованість та зручність управління.
4. Попередній перегляд гри: Дозволяє переглядати та тестувати свої ігрові сцени в режимі реального часу безпосередньо у вікні сцени, що спрощує процес налагодження та ітерацій.
5. Робота з камерами: у вкладці Scene можна розміщувати та налаштовувати камери, які визначають поле зору гравця та відображення ігрового світу.
6. Налаштування освітлення: можна додавати та налаштовувати різні джерела світла, щоб створювати бажану атмосферу та візуальні ефекти в ігровому світі.

Вкладка Game (рис. 2.5) слугує для попереднього перегляду та тестування ігрових сцен у режимі реального часу з точки зору гравця. Під час роботи в цій вкладці, можна змінювати різні змінні, зміни не зберігаються, після припинення роботи вкладки, всі змінні повернуться до попередніх значень.

Вона має кілька ключових функцій: [12]

1. Попередній перегляд: Вікно Game показує, як виглядатиме ігровий світ с точки зору користувача. Це дозволяє побачити кінцевий результат своєї роботи та швидко оцінити зовнішній вигляд і ігровий процес.
2. Тестування геймплею: Вкладку Game можна використовувати для тестування ігрової механіки, пересування персонажів, взаємодії з об'єктами, тригерами та іншими ігровими системами в режимі реального часу.
3. Налагодження: Вікно Game надає інструменти для налагодження, такі як перегляд значень змінних, логів та продуктивності додатку в межах запущеної сцени.
4. Імітація платформи: Unity дозволяє імітувати різні цільові платформи та розділену здатність екрану у вкладці Game, щоб можна було перевірити, як виглядає гра та як працюватиме на різних пристроях.

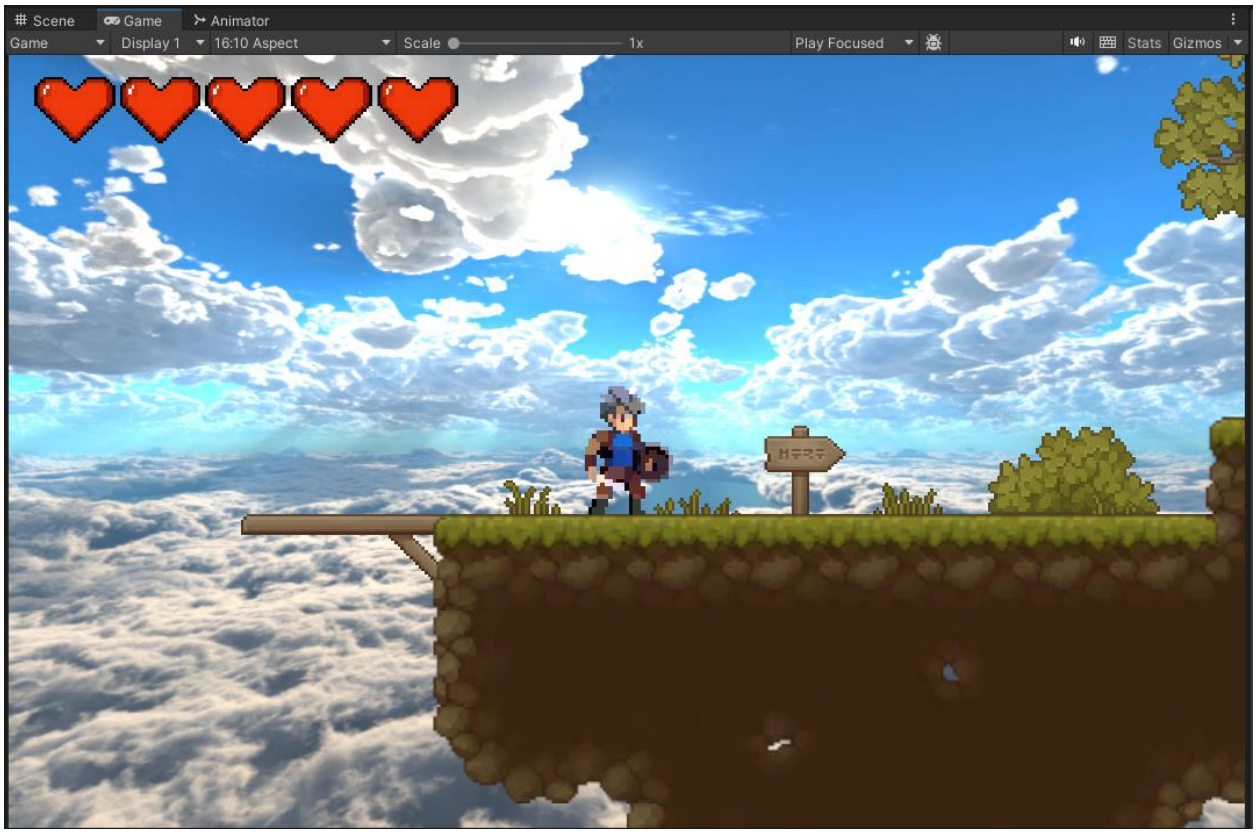

Рис. 2.5 – Вкладка Game

Вкладка Project є центральним місцем для організації та керування всіма ресурсами, або асетами, що використовуються в ігровому проекті. Основні функції цієї вкладки: [13]

1. Навігація та впорядкування ассетів: Вкладка Project відображає ієрархічну структуру папок і файлів проєкту. Це дозволяє легко знаходити, переглядати та впорядковувати спрайти, текстури, скрипти, аудіофайли та все інше.
2. Імпорт та експорт асетів: за допомогою вкладки Project можна імпортувати нові асети до проєкту з зовнішніх джерел. Також є можливість експортувати асети з проєкту.
3. Перегляд та редагування асетів: Вкладка Project надає вбудовані інструменти для їх перегляду та базового редагування безпосередньо в Unity.
4. Створення та керування префабами: Вкладка Project дозволяє створювати, зберігати та керувати префабими.
5. Доступ до Asset Store: З вкладки Project є прямий доступ до офіційного магазину асетів Unity (Asset Store), де можна шукати, переглядати та встановлювати додаткові ресурси та інструменти від сторонніх розробників.

Вкладка Console (рис. 2.6) у середовищі розробки Unity потрібна для відстеження та налагодження коду додатків. Кілька ключових функцій: [14]

1. Виведення повідомлень в консоль допомагає відстежувати стан виконання коду шляхом виведення повідомлень.
2. Виявлення та відстеження помилок. Якщо в коді існують якісь помилки або недоліки, консоль допоможе відстежити їх, виводячи повідомлення помилок.
3. Налагодження та тестування. Console є корисним інструментом для налагодження та тестування додатків.

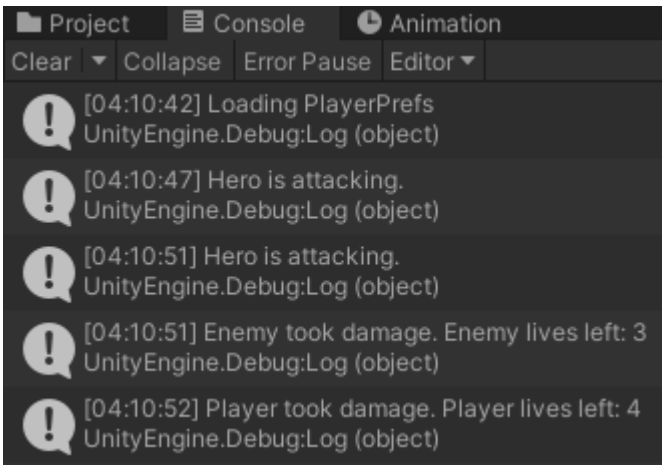

Рис. 2.6 – Вивід повідомлень у Console

Вкладка Hierarchy (рис. 2.7) в Unity відіграє ключову роль в організації та управлінні ігровими об'єктами на сцені. Основні функції цієї вкладки: [15]

1. Відображення ієрархічної структури: Вкладка Hierarchy візуалізує всі ігрові об'єкти, присутні на відкритій сцені, у вигляді ієрархічного дерева. Це допомагає легко відстежувати батьківські та дочірні відносини між об'єктами.
2. Створення та вкладення об'єктів: за допомогою вкладки Hierarchy можна створювати нові порожні ігрові об'єкти або вкладати існуючі об'єкти один в одного для побудови складних ієрархічних структур.
3. Керування об'єктами: Вкладка Hierarchy дозволяє вмикати/вимикати, переміщувати, змінювати назви та видаляти ігрові об'єкти у сцені.
4. Організація об'єктів: Правильне структурування ієрархії в цій вкладці допомагає тримати проект організованим та полегшує орієнтацію серед численних ігрових об'єктів.
5. Вибір об'єктів: Вкладка Hierarchy є зручним способом вибрати один або кілька ігрових об'єктів для редагування або налаштування їх властивостей та компонентів.
6. Вкладення префабів: Крім звичайних ігрових об'єктів, у вкладці Hierarchy також можна вкладати префаби, що полегшує створення складних ієрархічних структур з попередньо складених компонентів.

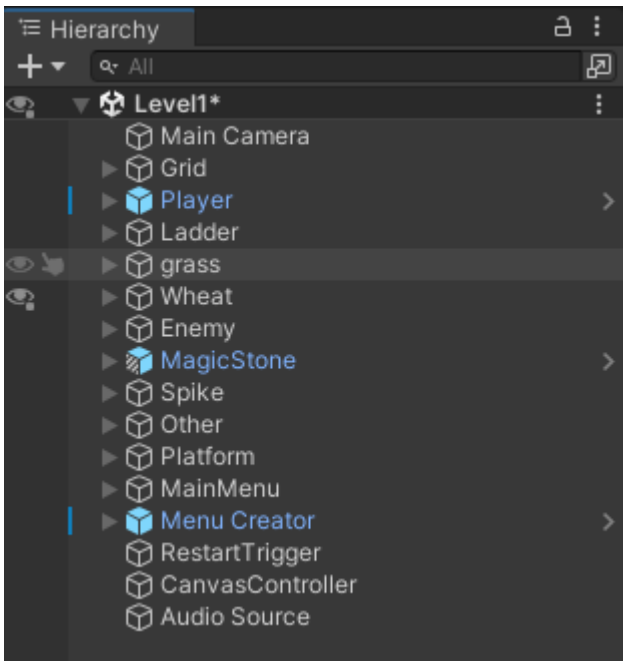

Рис. 2.7 – Структурування вкладки Hierarchy

Вкладка Animation (рис. 2.8) надає інструменти для створення й керування анімаціями ігрових об'єктів та персонажів. Основні функції цієї вкладки: [16]

1. Створення анімацій: на вкладці Animation можна створювати нові анімаційні кліпи. Це дозволяє оживляти персонажів, об'єкти та навколишнє середовище.
2. Редагування анімацій: Вкладка Animation надає зручний інтерфейс для редагування анімаційних кліпів, додавати, видаляти та змінювати ключові кадри, регулювати криві анімації.
3. Перегляд анімацій: у цій вкладці можна попередньо переглядати анімації в режимі реального часу, що полегшує налагодження.
4. Кісткова анімація: Вкладка Animation підтримує систему кісткової анімації, яка дозволяє створювати реалістичні анімації для персонажів та інших об'єктів із складною деформацією.

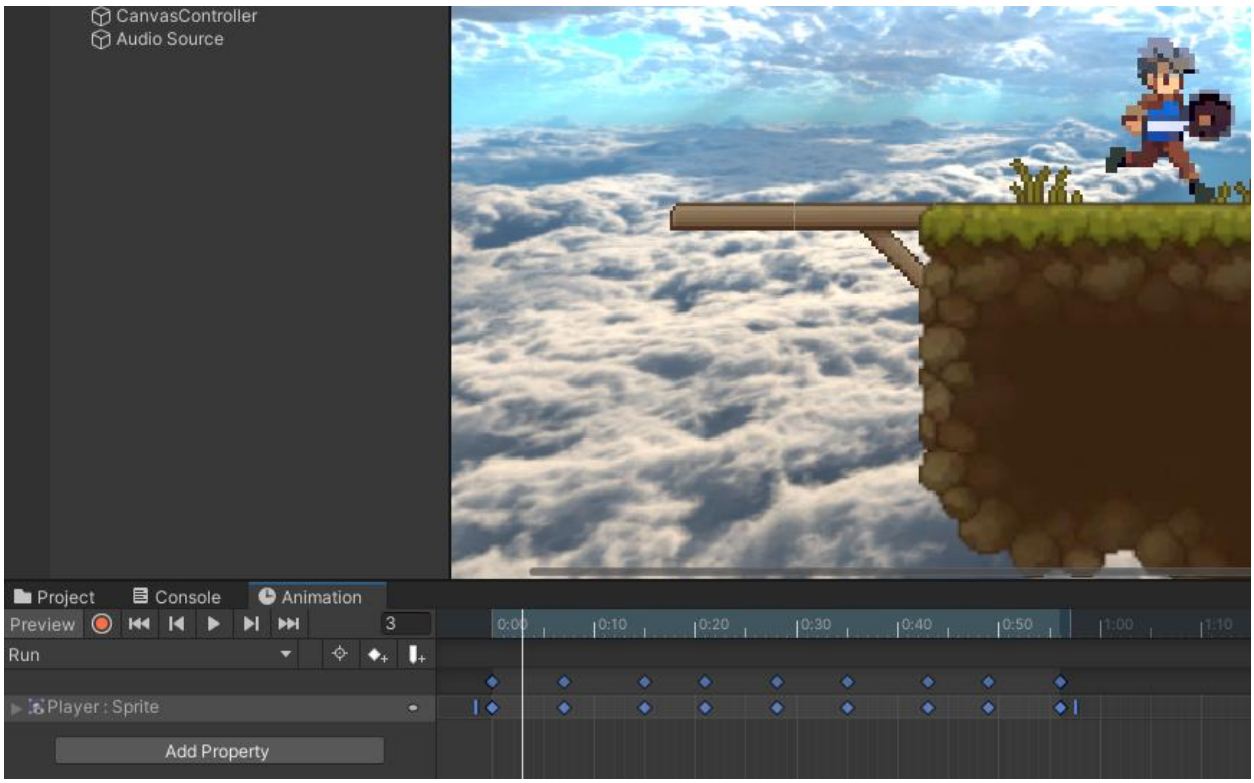

Рис. 2.8 – Демонстрація анімації бігу

Вкладка Animator (рис. 2.9) в Unity призначена для створення та керування анімаційними контролерами для ігрових персонажів та об'єктів. Ось основні функції цієї вкладки: [17]

1. Візуальне створення станів анімацій: у вкладці Animator можна створювати стани анімацій, які представляють різні поведінки або дії персонажа: “біг”, “стрибок”, “атака”.
2. Налаштування переходів між станами: за допомогою вкладки Animator можна визначати умови та параметри, за якими відбуваються переходи між різними станами анімацій.
3. Керування параметрами анімацій: Вкладка Animator надає зручний інтерфейс для керування параметрами, які впливають на анімації.

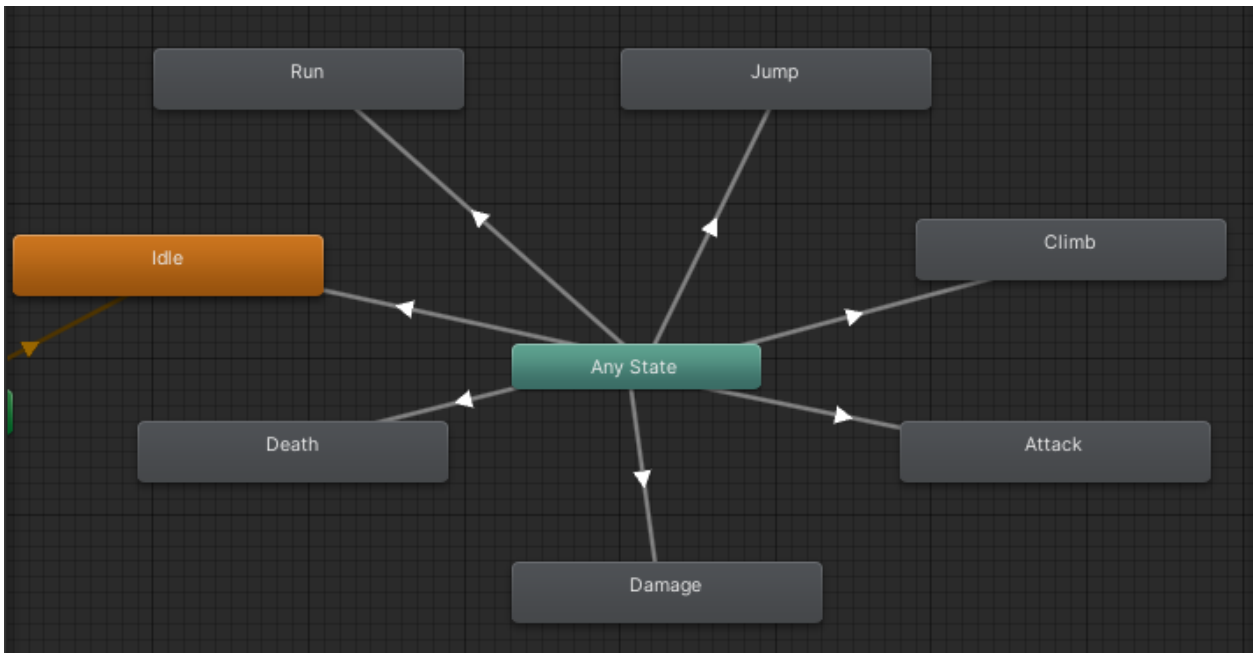

Рис. 2.9 – Контролер анімацій персонажа

Вкладка Inspector (рис. 2.10) в Unity виконує ключову роль у налаштуванні та керуванні властивостями та компонентами ігрових об'єктів. Основні функції цієї вкладки: [18]

1. Перегляд і редагування властивостей: Inspector відображає всі властивості та компоненти вибраного ігрового об'єкта. Можливо переглянути та відредагувати властивості для швидкого налаштування об'єктів.
2. Додавання та видалення компонентів: за допомогою Inspector можна додавати або видаляти компоненти.
3. Налаштування компонентів: Вкладка дозволяє налаштовувати властивості різних компонентів, прикріплених до об'єкта, таких як: Rigidbody 2D, Box Collider 2D, transform, скрипти та інші.
4. Перегляд та редагування префабів: Inspector дозволяє переглядати та редагувати властивості префабів.
5. Контекстна допомога: для багатьох компонентів у вкладці Inspector доступна контекстна допомога, яка пояснює їх призначення та налаштування.

6. Присутня функція фіксації вікна, потрібна для тривалої роботи з одним компонентом.

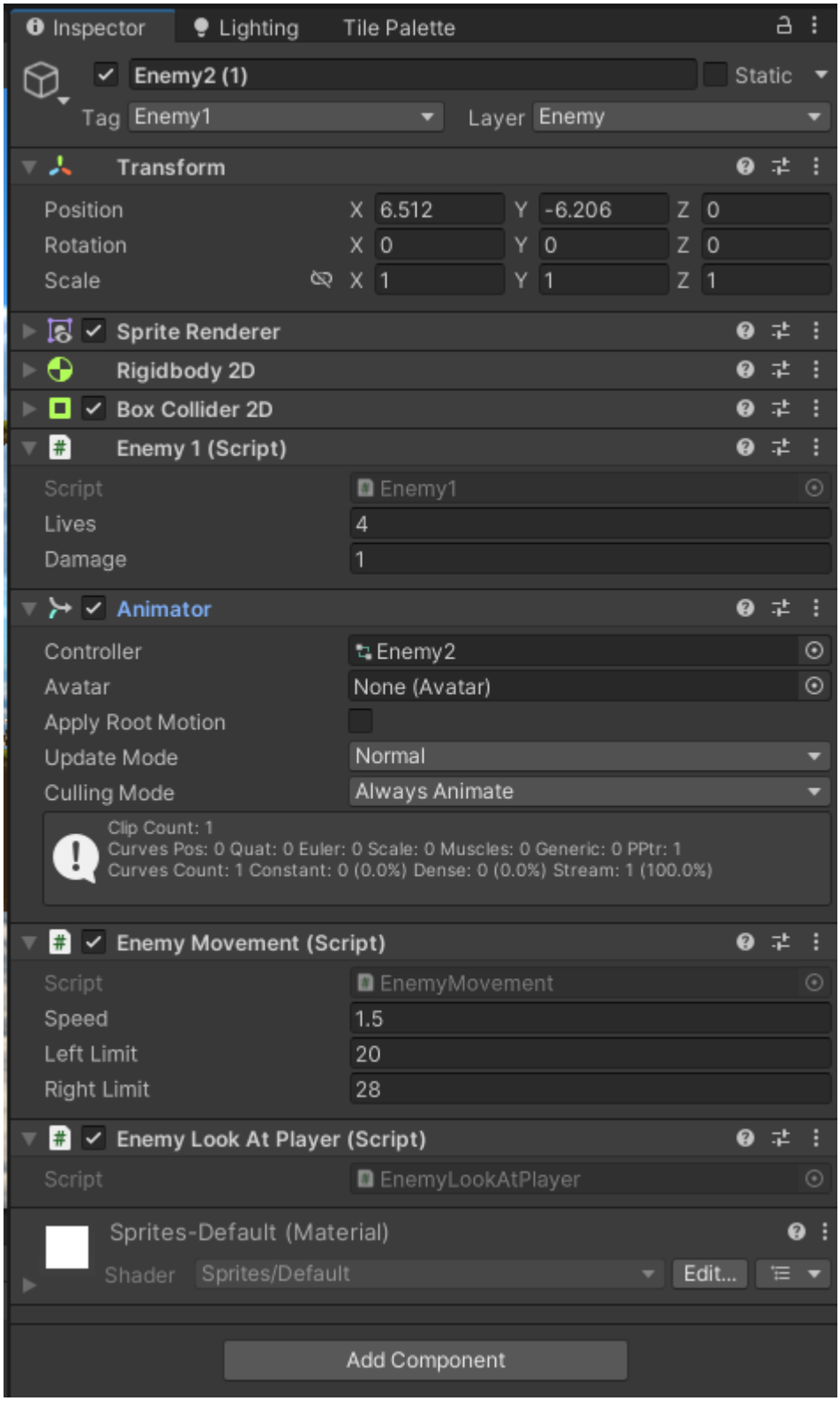

Рис. 2.10 – Демонстрація компонентів монстра

Вкладка Tile Palette (Рис. 2.11) в Unity використовується для створення рівнів та ігрових середовищ, що складаються з тайлів або плиток. Ось основні функції цієї вкладки: [19]

1. Організація тайлових наборів: Tile Palette дозволяє імпортувати, зберігати та керувати різними наборами тайлів, такими як текстури підлоги, стін, об'єктів оточення.
2. Нанесення тайлів на сцену: За допомогою Tile Palette можна будувати тайли безпосередньо на ігрову сцену в режимі перегляду сцени. Це значно прискорює процес створення рівнів та ландшафтів з повторюваних елементів.
3. Створення тайлових палітр: Unity дозволяє створювати та зберігати налаштовані палітри тайлів для повторного використання в майбутніх проектах.
4. Підтримка різних шарів: Tile Palette підтримує роботу з декількома шарами тайлів, що дозволяє організовувати різні типи елементів на окремих шарах для зручності.

Рис. 2. 11 – Рівень створений за допомогою Tile Palette

При створенні ігрового додатку "Sky Adventure" використовувалися пакети зображені на (рис. 2.12)

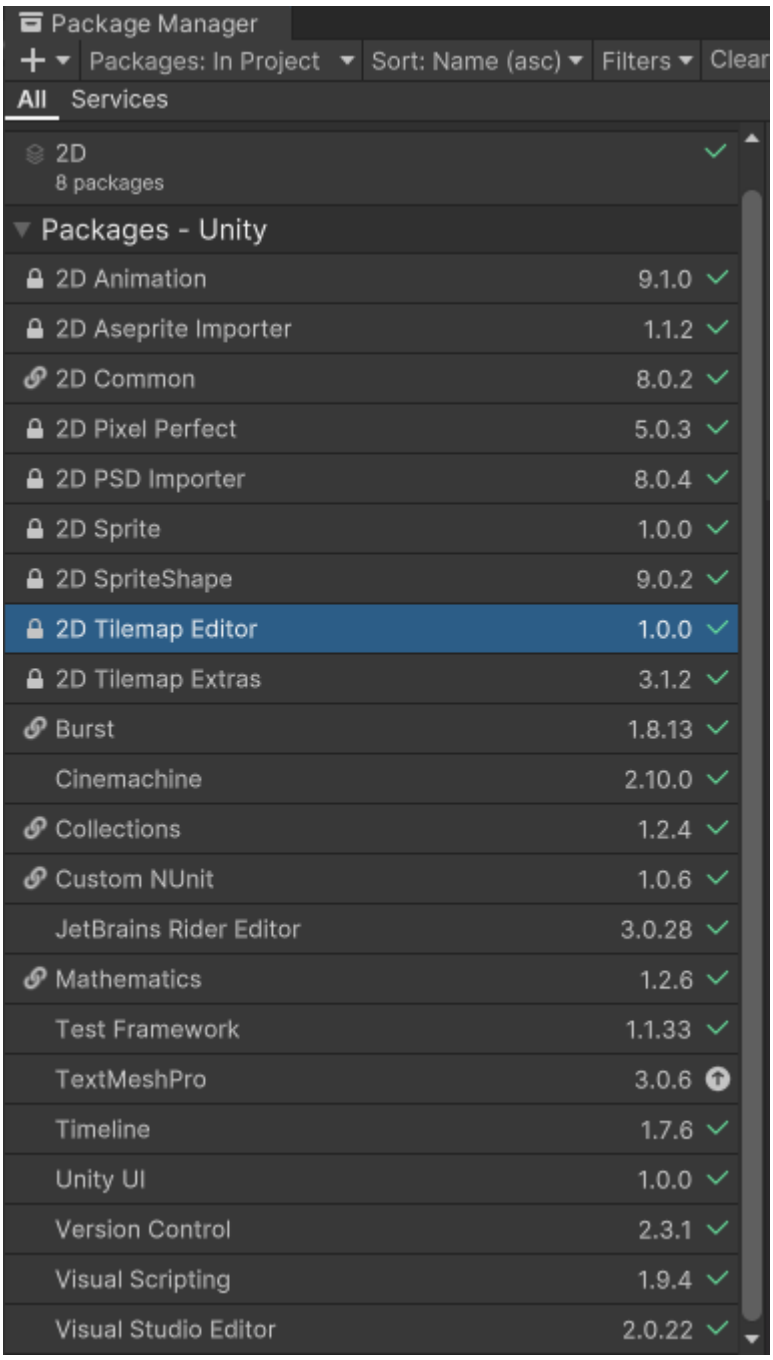

Рис. 2.12 – Перелік інстальованих пакетів

Microsoft Visual Studio Code [20] є сучасною програмною платформою для створення та редагування коду, яка виходить за рамки звичайного текстового редактора. Ця розробка від Microsoft набула широкої популярності серід професійних розробників програмного забезпечення різних галузей, зокрема тих, хто працює з ігровим рушієм Unity. Застосування Visual Studio Code в контексті розробки ігор на Unity має низку істотних переваг.

Одна з найголовніших відмінностей – це кросплатформенність цього засобу розробки, який підтримується на різних операційних системах, таких як Windows, macOS та Linux. Це забезпечує гнучкість та уніфікований досвід роботи незалежно від використовуваної платформи.

Однією з визначальних рис Visual Studio Code (рис. 2.13) є його легкість та висока продуктивність. На відміну від деяких масивних інтегрованих середовищ розробки (IDE), він демонструє швидке завантаження та стабільну роботу навіть на менш потужних апаратних ресурсах. Ця особливість робить його привабливим для розробників ігор в Unity, де часто потрібно одночасно виконувати ресурсномісткі завдання.

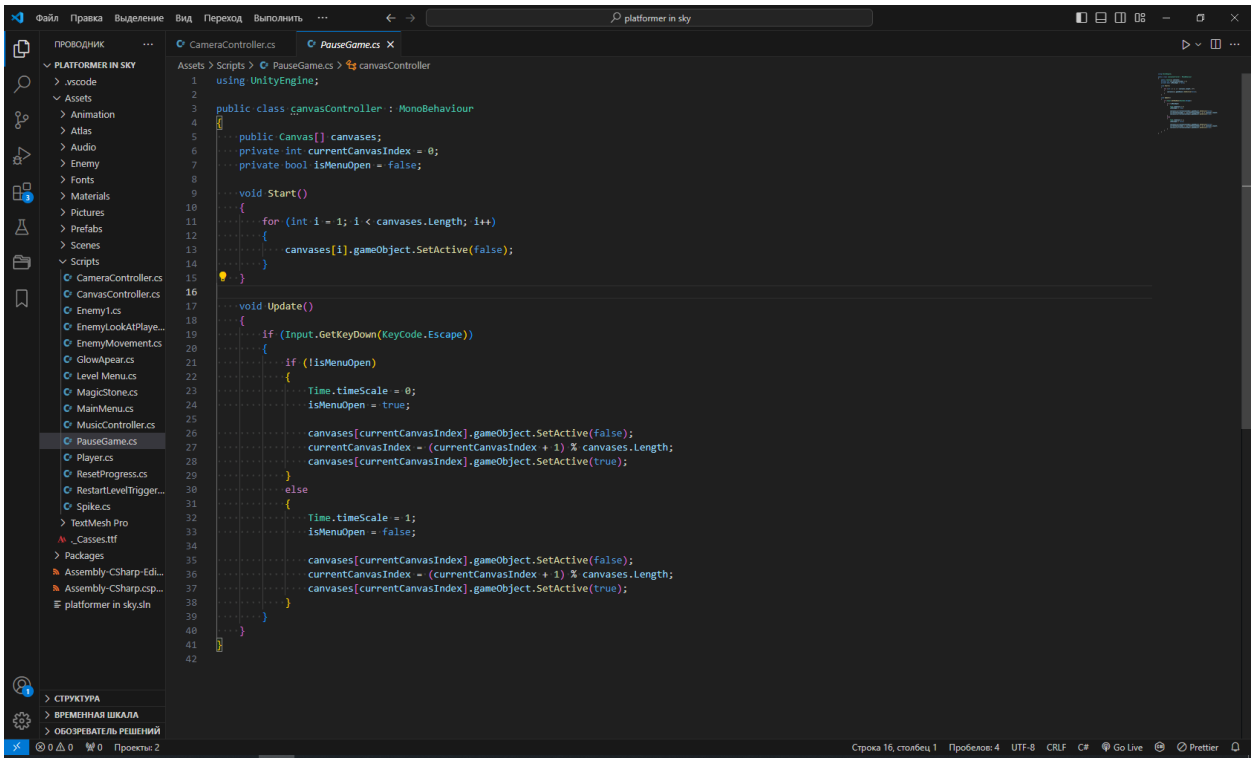

Рис. 2.13 – Середовище розробки Visual Studio Code

Важливою перевагою Visual Studio Code є його розширювана екосистема плагінів та додаткових компонентів, які дозволяють адаптувати його функціональність відповідно до специфічних потреб розробників. Зокрема, доступні спеціалізовані розширення для роботи з Unity, що забезпечують додаткову підтримку та інструменти для ефективної розробки ігор на цій платформі.

Visual Studio Code пропонує потужні можливості для налагодження коду, інтеграцію з системами контролю версій, вбудовані інструменти для роботи з термінальними вікнами та багато іншого. Ці функції є критично важливими для забезпечення високої якості та стабільності розроблюваних ігрових проєктів. Microsoft Visual Studio Code є гнучкою та високопродуктивною платформою для створення та редагування коду, яка завдяки своїй кросплатформенності, легкості, розширюваності та інтеграції з Unity стала цінним інструментом для ефективної розробки ігор на цьому ігровому рушію.

У процесі створення ігор за допомогою ігрового рушія Unity провідну роль відіграє мова програмування C#. Ця об'єктно-орієнтована мова, розроблена компанією Microsoft, тісно інтегрована з середовищем Unity та забезпечує розробникам потужний інструментарій для реалізації ігрових проектів. Однією з переваг використання C# у розробці ігор на Unity є її продуктивність та оптимізація для швидкого виконання коду. Як компільована мова, C# забезпечує високу ефективність, що є критично важливим для програмістів різних сфер. Це дозволяє реалізовувати складну ігрову логіку та обробляти великі обсяги даних без значного впливу на продуктивність.

Важливо також відзначити зручність та зрозумілість синтаксису C#, що полегшує процес навчання та написання коду для розробників. Наявність потужної бібліотеки класів .NET Framework забезпечує широкий спектр функціональних можливостей для роботи з різноманітними аспектами ігрового розвитку. Середовище розробки Visual Studio Code, створене Microsoft, пропонує тісну інтеграцію з Unity та надійну підтримку C#. Розробники можуть скористатися інструментами для налагодження, автоматичного доповнення, рефакторингу та багатьма іншими функціями, що значно спрощують процес написання та оптимізації ігрових скриптів.

Активна спільнота розробників C# та наявність численних ресурсів, бібліотек та фреймворків роблять цю мову програмування ще більш привабливою для створення ігор на Unity. Розробники можуть скористатися готовими рішеннями та інструментами, що прискорює процес розробки та

дозволяє зосередитися на унікальних аспектах свого ігрового проєкту. Завдяки своїй продуктивності, зрозумілому синтаксису, тісній інтеграції з Unity та широкій підтримці, мова програмування C# стала невід'ємною частиною процесу розробки ігор на цій платформі. Використання C# у поєднанні з можливостями Visual Studio Code дозволяє створювати високоякісні, ефективні та захопливі ігрові проекти, які відповідають вимогам сучасних гравців.

У процесі створення ігор, особливо тих, які використовують ретро-дизайн і пікселі, програма Aseprite [22] відіграє важливу роль, де користувачі можуть створювати чудові зображення та анімації, працюючи безпосередньо з окремими пікселями. Розробники Aseprite приділили значну увагу забезпеченню зручного та інтуїтивного інтерфейсу. Навіть починаючі користувачі можуть легко опанувати його та ефективно працювати з програмою. Однією з ключових функцій є підтримка анімованих спрайтів, що дозволяє створювати живі та динамічні анімації для персонажів, об'єктів та інших елементів ігрового світу. Одним з основних застосувань Aseprite є створення спрайтів для ігор. Спрайти - це невеликі графічні зображення, які часто використовуються для відображення персонажів, об'єктів чи інших елементів ігрового світу. У Aseprite доступні зручні інструменти для малювання та редагування пікселів, що дозволяє створювати яскраві та деталізовані спрайти з високою точністю. Завдяки інтуїтивному інтерфейсу та спеціалізованим інструментам, Aseprite дозволяє працювати з пікселями з високою точністю та контролем. Користувачі можуть експериментувати з різними техніками, такими як розтушовка, змішування кольорів та створення складних візерунків, що сприяє вираженню індивідуального стилю у їхніх піксельних роботах. На (рис. 2.14) зображено інтерфейс програми Aseprite з частиною анімацій головного героя.

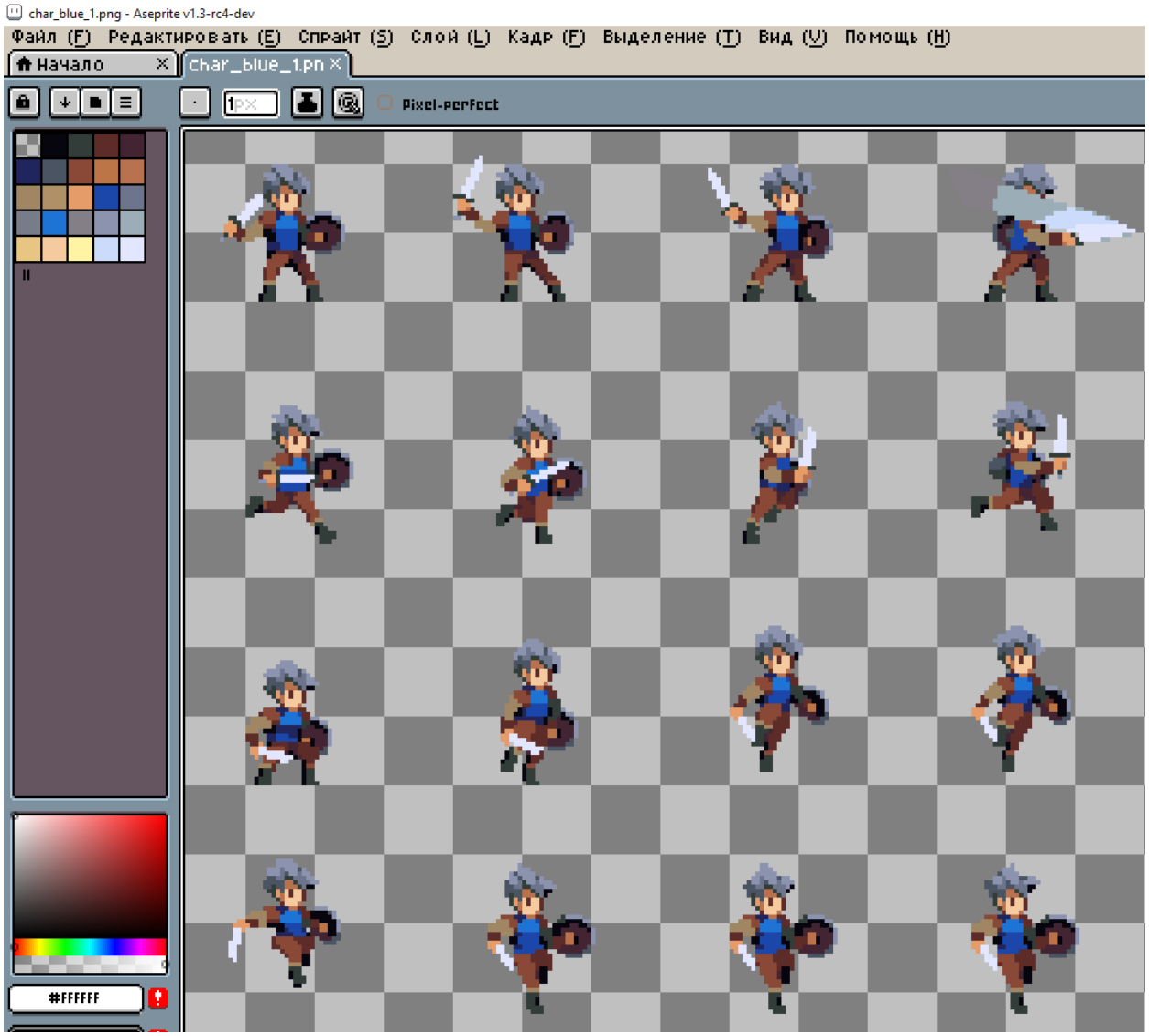

Рис. 2.14 – Анімації головного героя

Особливо корисною є можливість роботи з палітрами кольорів у Aseprite. Користувачі можуть скористатися великою кількістю вбудованих палітр або ж створити власні, налаштувавши кольори відповідно до своїх вподобань та потреб ігрового проекту. Це забезпечує гнучкість у досягненні бажаного стилю та атмосфери. Aseprite має добру інтеграцію з популярними ігровими рушіями, такими як Unity. Це дозволяє розробникам безпосередньо імпортувати створені пікселеві ресурси до своїх ігрових проектів, забезпечуючи зручний процес.

2.4. Опис структури системи та алгоритмів її функціонування

В даному проєкті ми маємо 9 сцен:

1. MainMenu
2. Level 1
3. Level 2
4. Level 3
5. Level 4
6. Level 5
7. Level 6
8. Level 7
9. Level 8

В першу чергу завантажується сцена MainMenu. (Рис. 2.15)

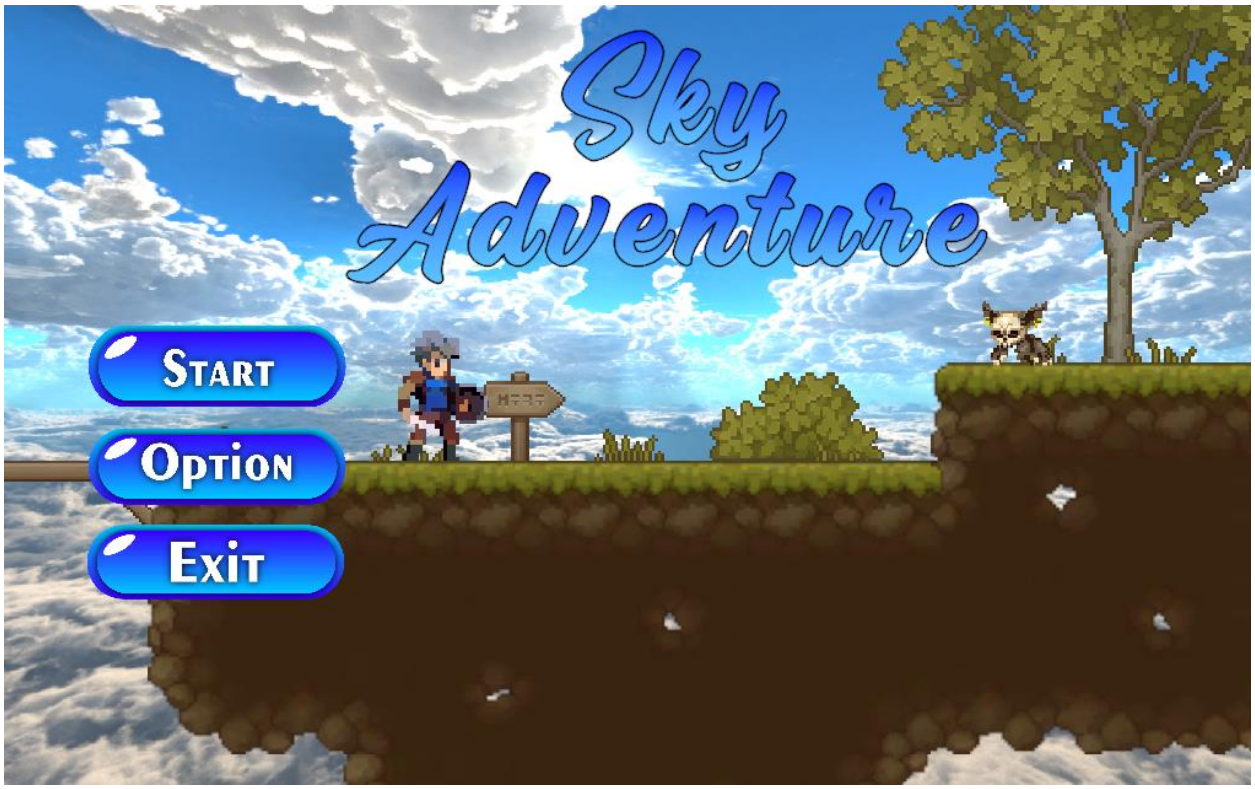

Рис. 2.15 – Головне меню гри

Для ілюстрації різноманітних способів взаємодії користувачів з програмним забезпеченням та послідовностей ігрових дій нижче представлені

спеціалізовані діаграми, відомі як діаграми прецедентів (Use Case). Ці візуальні моделі демонструють можливі сценарії експлуатації додатку та перебіг ігрового процесу.

Нижче наведена діаграма варіантів використання Main Menu. (рис. 2.16) Ця діаграма показує різні варіанти дії, які користувач може виконати у головному меню програми. Серед можливих варіантів взаємодії:

– вибрати рівень;
– обрати розмір екрану;
– перегляд налаштувань;
– зміна гучності музики;
– зміна гучності звуків;
– вийти з гри.

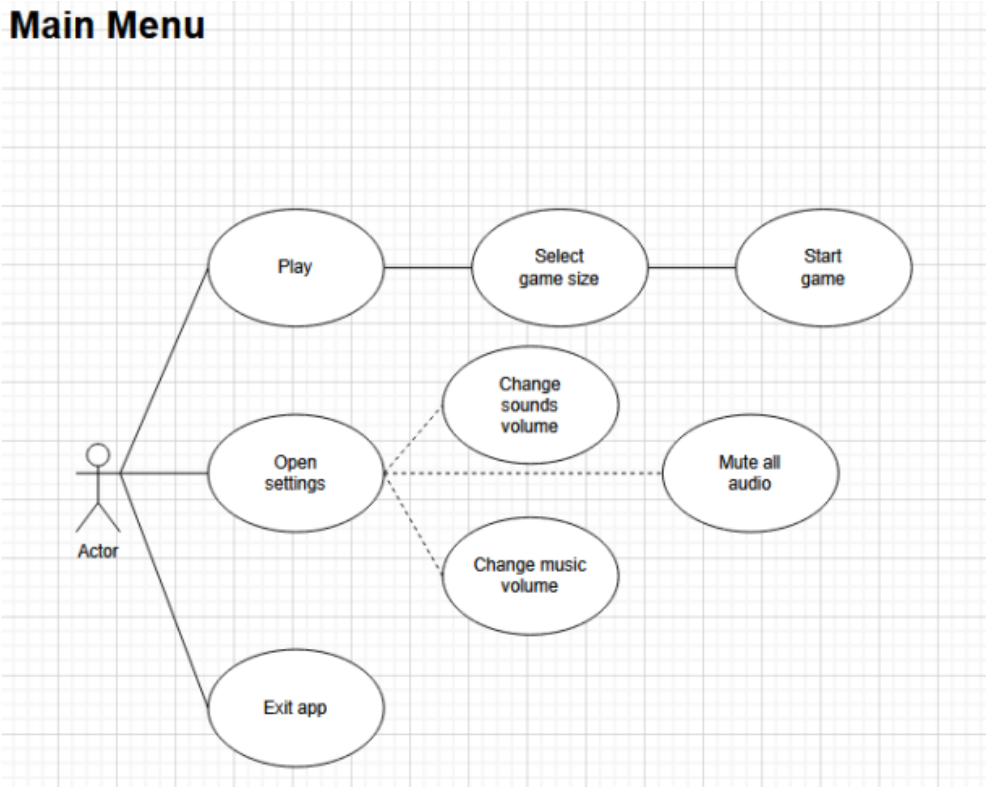

Рис. 2.16 – Діаграма головного меню

Діаграма нижче демонструє різні варіанти взаємодії користувача з меню, що відкривається під час паузи у грі. (рис. 2.17) Ця схема ілюструє опції, такі як відновлення ігрового процесу, зміна налаштувань або повернення до початкового

Main Menu. Користувач може обрати один із цих шляхів, керуючись своїми поточними потребами та бажаннями.

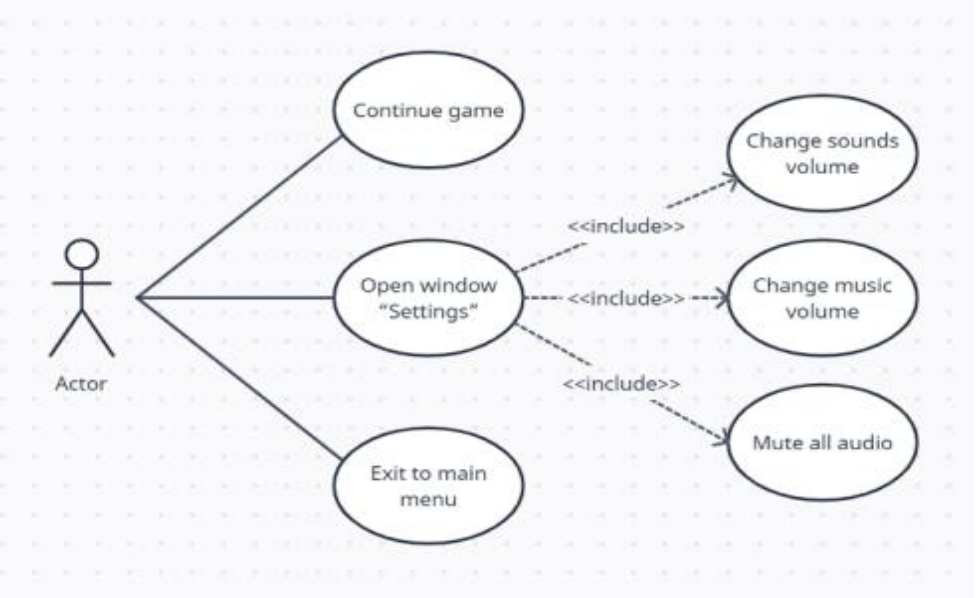

Рис. 2.17 – Діаграма паузи

Герой має 5 життів, (рис. 2.18) які прікріплені до лівого верхнього кута, коли він доторкається до колайдеру ворога, його кількість життів зменшується, червоне сердечко зникає, при цьому персонаж становиться невразливим на одну секунду. Коли кількість життів в героя буде нуль, він помре, а рівень доведеться проходити с початку.

Рис. 2.18 – Життя герою

2.5. Обгрунтування та організація вхідних та вихідних даних програми

Відповідно до вимог, сформульованих у постановці задачі, вхідними даними для роботи програми є:

– пересування персонажу;
– налаштування гри;

Вихідні дані:

- відображення положення головного героя;
- відтворення звукових та музичних ефектів;
- демонстрація анімацій різноманітних об'єктів;
- відображення елементів користувацького інтерфейсу.

2.6. Опис розробленої системи

2.6.1. Використані технічні засоби

Під час процесу створення гри використовувався персональний комп'ютер з наступною конфігурацією:

- ОС: Windows 10;
- ЦП: Intel(R) Core(TM) i3-2120 CPU @ 3.30GHz;
- Відеокарта: NVIDIA GeForce GTX 1050 Ti;
- ОП: 32 Гб;
- Жорсткий диск: 314 Гб;

2.6.2. Використані програмні засоби

Серед ключових програмних засобів, які використовувалися в процесі створення гри, можна виділити:

- Microsoft Visual Studio Code
- Нушій Unity 2022.3.26f1
- Aseprite 1.3

2.6.3. Виклик та завантаження програми

Розглянемо процес встановлення та запуску програми на ПК з операційною системою Windows: (Рис. 2.19)

1. Завантажити архів з програмою з інтернету.
2. Розпакувати архів за допомогою архіватора.
3. Знайти файл з розширенням .exe серед розпакованих файлів.
4. Запустити .exe файл подвійним кліком.

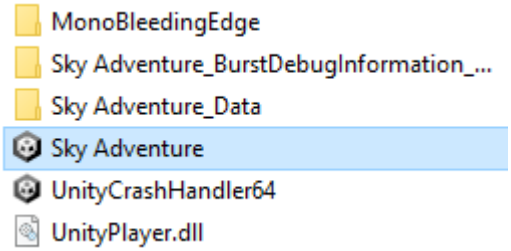

Рис. 2.19 – структура скомпільованого додатку

Розглянемо процес встановлення та запуску програми на телефон з операційною системою Android: (Рис. 2.20)

1. Завантажити .apk файл з інтернету.
2. Запустити файл.
3. З'явиться стандартне вікно встановлення.
4. Після встановлення додатку, запустіть його з головного екрана.

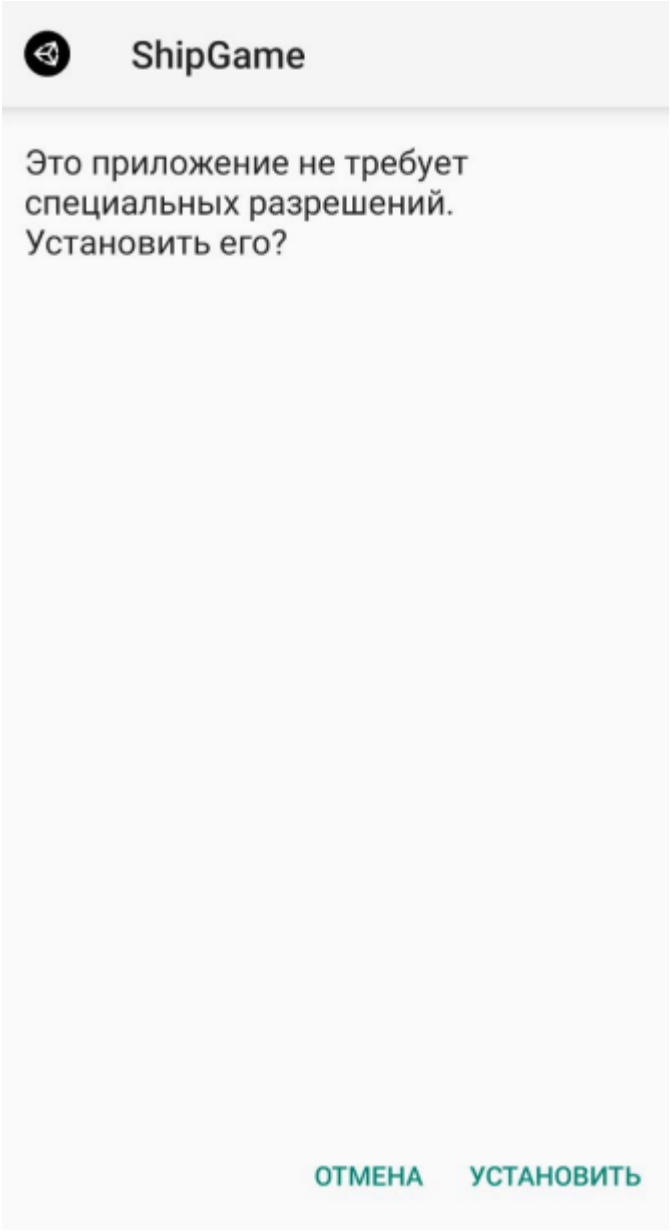

Рис. 2.20 – інтерфейс інсталяції додатку на Android

2.6.4. Опис інтерфейсу користувача

Після запуски гри, користувач потрапляє до сцени Main Menu. (Рис. 2.21)

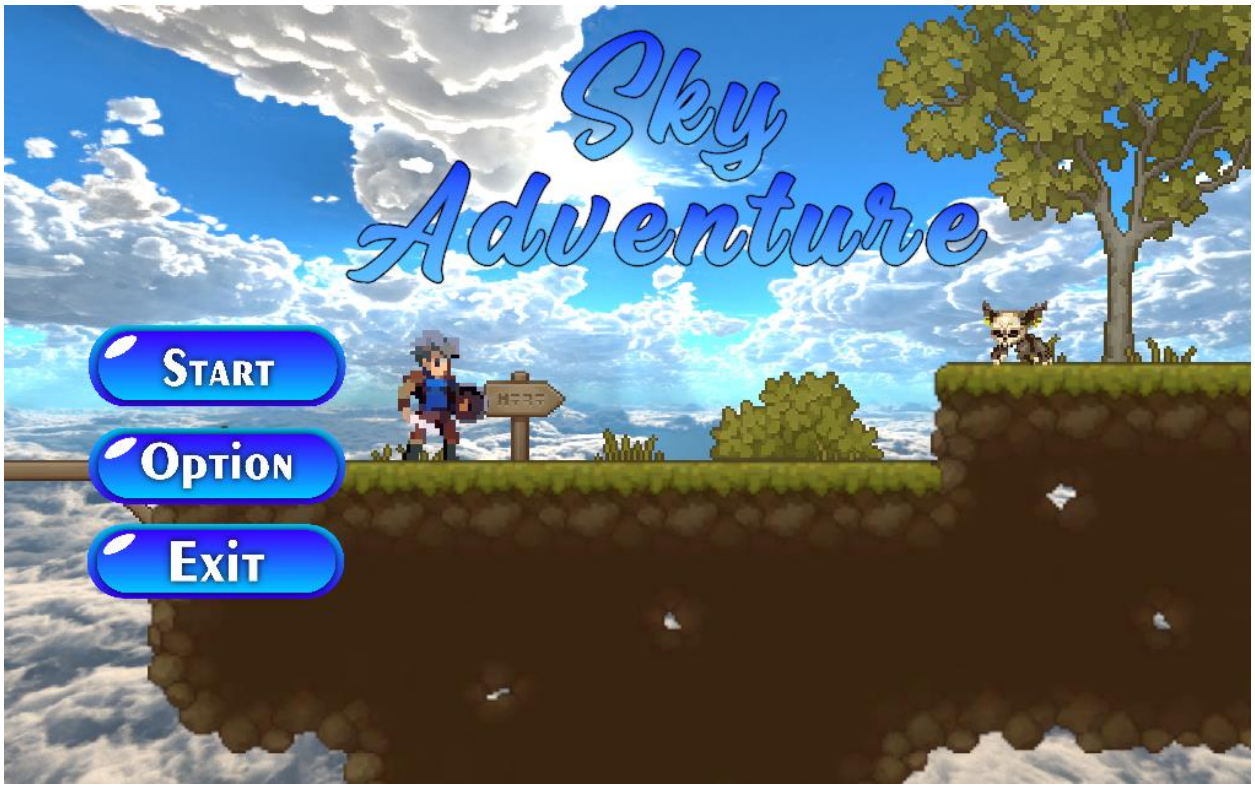

Рис. 2.21 – Scene Main Menu

При натисканні кнопки Play, в нас відкриється панель (рис. 2.22) з вибором рівня. Якщо ви до цього не проходили жодного рівня, то вам буде доступний лише перший рівень, після його проходження відкриється другий і так далі.

Рис. 2.22 – Панель для вибору рівня

При відкритті меню Option користувач отримує доступ до налаштувань гри. (Рис. 2.23)

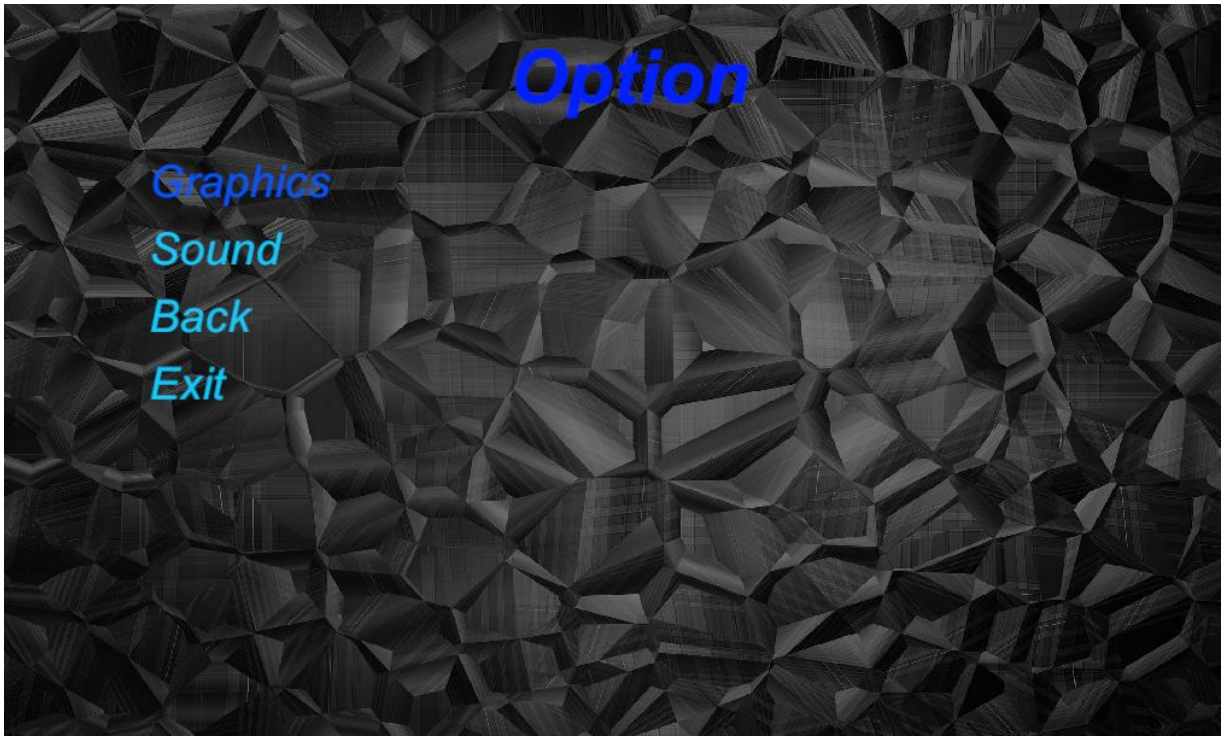

Рис. 2.23 – Меню Option

У вкладці Graphics можна покращити графіку гри та відрегулювати яскравість. (Рис. 2.24)

Рис. 2.24 – Панель графіки

Якщо перейти до панелі Sounds (Рис. 2.25) там можна відрегулювати гучність застосунку.

Рис. 2.25 – Панель гучності звуку

Після виходу з меню налаштувань, з'являється напис (рис. 2.26) про збереження змін в меню.

Рис. 2.26 – Збереження змін в налаштуванні

На (рис. 2.27) продемонстрована анімація Climbing.

Рис. 2.27 – Одна з анімацій героя

При влучанні по ворогах, вони втрачають життя, в цей момент вони спалахають червоним коліром, (рис. 2.28) це слугує сигналом влучання в ціль. Ще однією ознакою є звук. При промаху буде зовсім інший звук.

Рис. 2.28 – Ознака влучання

В знак підтвердження того, про що я згадував на самому початку щодо адаптивного користувацького інтерфейсу, можно побачити на (рис. 2.29) і

(рис. 2.30). Ці ілюстрації демонструють концепцію адаптивного UI, здатного динамічно підлаштовуватися під різні розміри екранів. Незалежно від розміру екрана, смуга життів головного героя залишається незмінною - вона не переміщується і не зникає.

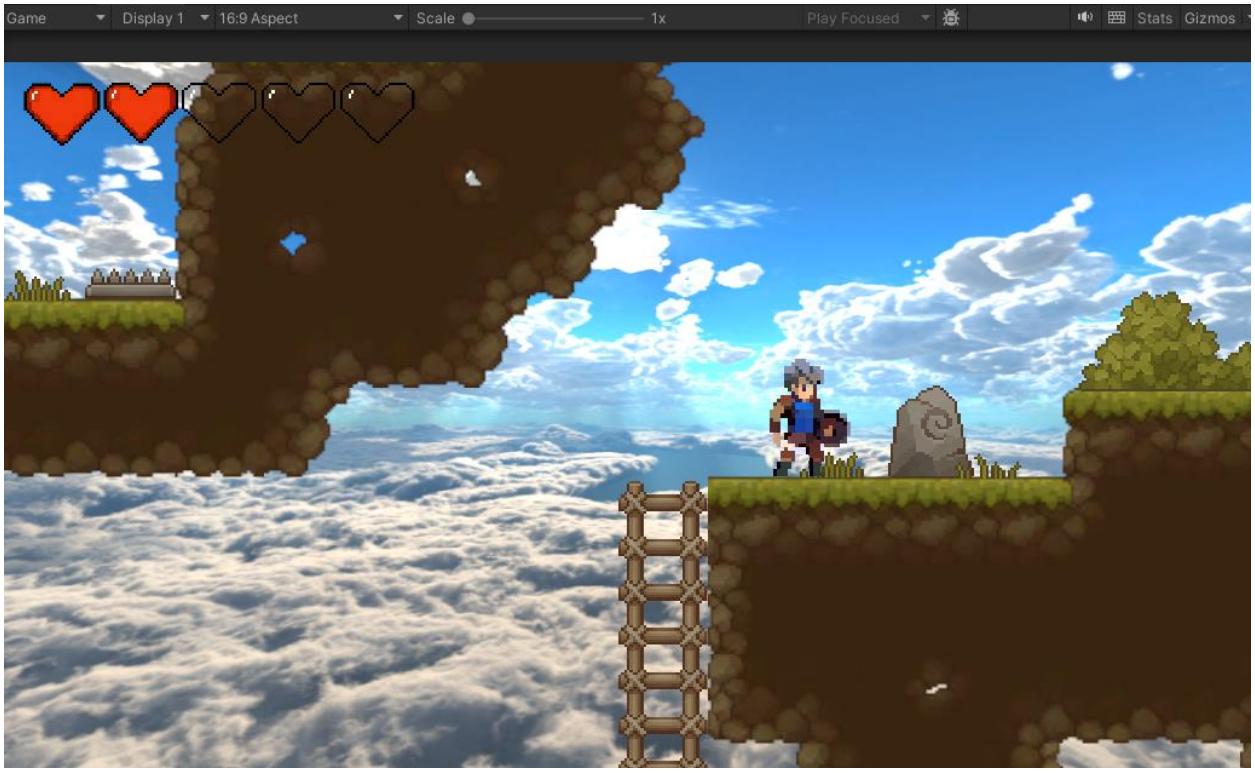

Рис. 2.29 – Формат екрану 16:9

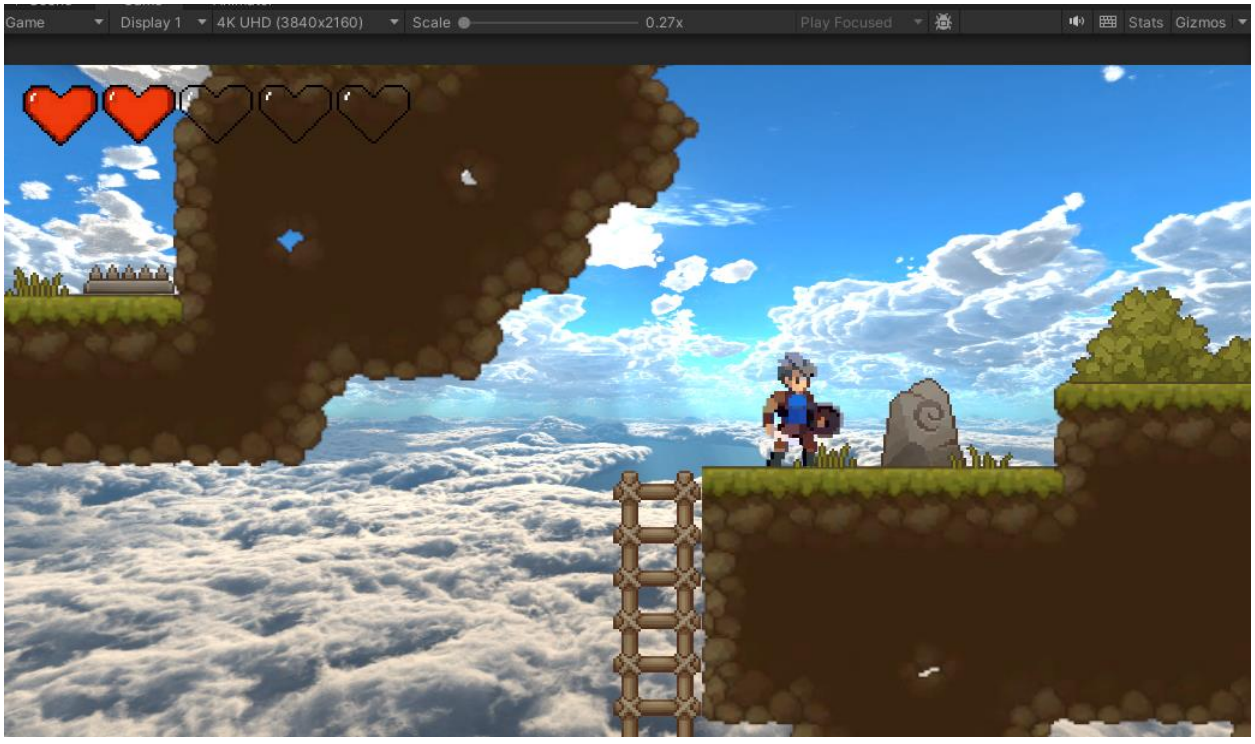

Рис. 2.30 – Формат екрану 3840*2160

Коли смуга життів героя закінчиться, почнеться анімація Death. (рис. 2.31)

Рис. 2.31 – Програш

Якщо герою не вдається пройти рівень, то йому доведеться проходити знову, тому що новий рівень не відкриється поки не буде завершений минулий. (Рис. 2.32)

Рис. 2.32 – Демонстрація панелі вибору рівня після невдалої спроби проходження рівня

Для того щоб відкрити наступний рівень, ваш персонаж повинен вижити та вбити всіх ворогів, коли ворогів не залишеться, магічний камінь (Рис. 2.32) стане інтерактивним, після взаємодії з ним, рівень буде пройдено.

Рис. 2.32 – MagicStone

РОЗДІЛ 3. ЕКОНОМІЧНИЙ РОЗДІЛ

3.1. Розрахунок трудомісткості та вартості розробки програмного продукту

Початкові дані:

1. передбачуване число операторів програми – 1450;
2. коефіцієнт складності програми – 1,3;
3. коефіцієнт корекції програми в ході її розробки – 0,1;
4. годинна заробітна плата програміста – 200 грн/год;

Згідно зі статистикою наданою сайтом DOU за посиланням https://jobs.dou.ua/salaries/?period=2023-12&position=Game%20Designer&domain=GameDev&education=5. Було встановлено що медіана заробітної плати для професії Game Designer становить 2100 долларів на місяць. При 192 годин робочого часу на місяць, визначаємо погодинну плату, яка становить 10,94 дол/год. На момент написання диплому, курс долара становить 40,55 гривень. За допомогою цих данних розраховуємо годинну заробітну плату, 443,62 грн/год.

5. коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі – 1,2;
6. коефіцієнт кваліфікації програміста, обумовлений від стажу роботи з даної спеціальності – 0,8;
7. вартість машино-години ЕОМ – 5,62 грн/год.

Трудомісткість розробки програмного забезпечення можна розрахувати за допомогою формули:

$$t = t_o + t_u + t_a + t_n + t_{отл} + t_д \text{, людино-годин} \qquad (3.1)$$

де t_o - витрати праці на підготовку й опис поставленої задачі (приймається 50);

$t_и$ – витрати праці на дослідження алгоритму рішення задачі;

$t_а$ – витрати праці на розробку блок-схеми алгоритму;

$t_п$ – витрати праці на програмування по готовій блок-схемі;

$t_{отл}$ – витрати праці на налагодження програми на ЕОМ;

$t_д$ – витрати праці на підготовку документації.

Складові витрати праці визначаються через умовну кількість операторів у програмному забезпеченні, що розробляється.

Умовне число операторів (підпрограм):

$$Q = q \cdot C \cdot (1 + p), \quad (3.2)$$

де q – передбачуване число операторів (1450);

C – коефіцієнт складності програми (1,3);

p – коефіцієнт кореляції програми в ході її розробки (0,1).

$$Q = 1450 \cdot 1{,}3 \cdot (1 + 0{,}1) = 2073;$$

Витрати праці на вивчення опису задачі визначається з урахуванням уточнення опису і кваліфікації програміста:

$$t_и = \frac{Q \cdot B}{(75..85) \cdot k}, \quad (3.3)$$

де В – коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі (1,2);

К – коефіцієнт кваліфікації програміста, обумовлений стажем роботи з даної спеціальності (0.8);

$t_u = (1450 \cdot 1{,}2) / (75 \cdot 0{,}8) = 29$ людино-годин

Витрати праці на розробку алгоритму рішення задачі:

$$t_a = \frac{Q}{(20 \ldots 25) \cdot k}, \quad (3.4)$$

де Q – умовне число операторів програми;

k – коефіцієнт кваліфікації програміста.

Після підстановки значень в формулу (3.4), отримаємо:

$t_a = 1450 / (20 \cdot 0.8) = 90{,}625$ людино-годин.

Витрати на складання програми по готовій блок-схемі визначається за формулою:

$$t_n = \frac{Q}{(20 \ldots 25) \cdot k}, \quad (3.5)$$

Після підстановки значень в формулу (3.5), отримаємо:

$t_n = 1450 / (25 \cdot 0.8) = 72{,}5$ людино-годин.

Витрати праці на налагодження програми на ЕОМ:

– за умови автономного налагодження одного завдання:

$$t_{отл} = \frac{Q}{(4..5) \cdot k}, \quad (3.5)$$

$t_{отл} = 1450 / (4 \cdot 0{,}8) = 453$ людино-годин.

– за умови комплексного налагодження завдання:

$$t^{k}_{отл} = 1{,}5 \cdot t_{отл}, \quad (3.6)$$

$t^{k}_{отл} = 1{,}5 \cdot 453 = 679{,}5$ людино-годин.

Витрати праці на підготовку документації:

$$t_{д} = t_{др} + t_{до}, \quad (3.7)$$

де $t_{др}$ – трудомісткість підготовки матеріалів і рукопису:

$$t_{др} = \frac{Q}{(15..20) \cdot k}, \quad (3.8)$$

$t_{до}$ – трудомісткість редагування, печатки й оформлення документації:

$$t_{до} = 0{,}75 \cdot t_{др}, \quad (3.9)$$

Підставивши всі значення, отримаємо:

$t_{др} = 1450 / (15 \cdot 0{,}8) = 120{,}83$ людино-годин.

$t_{до} = 0{,}75 \cdot 120{,}83 = 90{,}62$ людино-годин.

$t_{д} = 120{,}83 + 90{,}62 = 211{,}45$ людино-годин.

Повертаючись до формули (3.1), отримаємо повну оцінку трудомісткості розробки програмного забезпечення:

$t = 90{,}625 + 72{,}5 + 453 + 29 + 50 + 211{,}45 = 906{,}575$ людино-годин.

3.2. Розрахунок витрат на створення програми

Витрати на створення ПЗ $K_{ПО}$ включають витрати на заробітну плату виконавця програми $З_{ЗП}$ і витрат машинного часу, необхідного на налагодження програми на ЕОМ:

$$K_{ПО} = З_{ЗП} + З_{МВ}, \text{ грн} \qquad (3.10)$$

Заробітна плата виконавців:

$$З_{ЗП} = t \cdot C_{ПР}, \text{ грн} \qquad (3.11)$$

де t – загальна трудомісткість, людино-годин;

$C_{ПР}$ – середня годинна заробітна плата програміста, грн/година

З урахуванням того, що середня годинна зарплата програміста становить 443,62 грн / год, отримуємо:

$$З_{ЗП} = 906{,}575 \cdot 443{,}62 = 402\ 174{,}8 \text{ грн}$$

Вартість машинного часу, необхідного для налагодження програми на ЕОМ:

$$З_{мв} = t_{отл} \cdot C_{мч}, \text{ грн} \qquad (3.12)$$

де $t_{отл}$ – трудомісткість налагодження програми на ЕОМ, год;

$C_{мч}$ – вартість машино-години ЕОМ, грн/год

Необхідна вартість для налагодження машинного часу:

$$З_{мв} = 453 \cdot 5{,}62 = 2545{,}86 \text{ грн}$$

Звідси витрати на створення програмного продукту:

$$K_{ПО} = 2545{,}86 + 402\ 174{,}8 = 404\ 720{,}66 \text{ грн.}$$

Очікуваний період створення ПЗ:

$$T = \frac{t}{B_k \cdot F_p}, \text{міс.} \qquad (3.12)$$

де B_k - число виконавців (1);

F_p - місячний фонд робочого часу (при 44 годинному робочому тижні F_p=192 годин);

t – загальна трудомісткість, людино годин.

Витрати на створення програмного продукту:

$$T = 906{,}575 / 1 \cdot 192 = 4{,}72 \text{ міс.}$$

Висновок: для розробки 2D кросплатформеного ігрового застосунку за підрахунками знадобиться 906,575 людино-години. За попередніми оцінками, процес створення цього програмного забезпечення від початкової стадії до завершення триватиме приблизно 4,72 місяці. Загальна вартість розробки, враховуючи трудовитрати, витратні матеріали, програмне і апаратне забезпечення, становить – 404 720,66.

ВИСНОВКИ

У ході виконання даної кваліфікаційної роботи було проведено дослідження сучасних технологій та підходів до розробки комп'ютерних ігор. В результаті цього вивчення було успішно реалізовано кросплатформений ігровий застосунок у жанрі платформер на основі технологічного стеку Unity/C#.

Актуальність даного проєкту полягає у створенні захопливого ігрового продукту, який може використовуватися для розваги, відпочинку та отримання задоволення користувачами різних вікових категорій.

Процес розробки здійснювався на провідному ігровому рушії Unity 3D, який забезпечив зручне середовище для реалізації ігрової механіки, графіки та звуку. Для написання скриптів та програмування логіки гри було обрано мову програмування C#, що найкраще підходить для об'єктно-орієнтованого підходу, дозволяючи описувати складну ігрову логіку через систему класів. Для створення яскравої пікселевої графіки та анімацій був використаний спеціалізований графічний редактор Aseprite, який надав необхідний функціонал для роботи з палітрами кольорів, малювання спрайтів та експорту графічних ресурсів.

У створеній грі головний герой вільно пересувається у двовимірному просторі, маючи можливість рухатися вліво, вправо, вгору та вниз. На ігрових рівнях персонаж має долати перешкоди та вступати у протистояння з різноманітними ворогами.

По завершенні гри користувачеві відображається підсумкова панель з результатом (перемога чи програш). Під час ігрового процесу гравець може призупинити гру для налаштування графічних параметрів та регулювання гучності відповідно до своїх вподобань. Головне меню програми містить кнопки для виходу з гри, відкриття налаштувань та початку нової ігрової сесії.

Практична цінність розробленої програми полягає у можливості для користувачів отримати веселий, захопливий та приємний відпочинок,

поринувши у світ яскравої гри-платформера з цікавою ігровою механікою та високоякісною графікою.

За результатами економічного розділу, загальна трудомісткість розробки цього програмного забезпечення становить 906 людино-години, орієнтовний термін розробки – 4,7 місяця, а вартість створення – 404 720 гривень.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. History of platforms URL: https://professionalmoron.com/2024/04/07/history-of-platform-games/ (дата звернення: 03.05.2024).

2. Platforn games. URL: https://classicgamerevival.com/platform-games/ (дата звернення: 03.05.2024).

3. URL: https://www.gameinformer.com/b/features/archive/2011/07/14/why-the-platformer-still-matters.aspx (дата звернення: 04.05.2024).

4. Evolution of platformers. URL: https://www.redbull.com/in-en/evolution-of-platformers (дата звернення: 05.05.2024).

5. Video Games. URL: https://www.noypigeeks.com/explained/benefits-playing-video-games/ (дата звернення: 07.05.2024).

6. Techradar. URL: https://www.techradar.com/news/fortnite-gets-dx12-support-to-run-better-on-some-pcs-will-ray-tracing-come-next (дата звернення: 09.05.2024).

7. Unity. URL: https://unity.com/ru/our-company (дата звернення: 11.05.2024).

8. Джозеф Хокінг. Unity в дії, мультиплатформенна розробка на C#. "Пітер" – 2018. 512с.

9. Asset Workflow. URL: https://docs.unity3d.com/Manual/AssetWorkflow.html (дата звернення: 10.06.2024).

10. Prefabs. URL: https://docs.unity3d.com/Manual/Prefabs.html (дата звернення: 10.06.2024).

11. Scene. URL: https://docs.unity3d.com/Manual/CreatingScenes.html (дата звернення: 10.06.2024).

12. Game window. URL: https://docs.unity3d.com/Manual/GameView.html (дата звернення: 11.06.2024).

13. Project Settings. URL: https://docs.unity3d.com/Manual/comp-ManagerGroup.html (дата звернення: 11.06.2024).

14. Console. URL: https://docs.unity3d.com/Manual/Console.html (дата звернення: 11.06.2024).

15. Hierarchy. URL: https://docs.unity3d.com/Manual/Hierarchy.html (дата звернення: 11.06.2024).

16. AnimationClips. URL: https://docs.unity3d.com/Manual/AnimationClips.html (дата звернення: 12.06.2024).

17. AnimatorController. URL: https://docs.unity3d.com/Manual/class-AnimatorController.html (дата звернення:12.06.2024).

18. Inspector window. URL: https://docs.unity3d.com/Manual/UsingTheInspector.html (дата звернення:12.06.2024).

19. Tile Palette. URL: https://docs.unity3d.com/Manual/Tile-Palette-visual-elements.html (дата звернення:12.06.2024).

20. Visual Studio Code. URL: https://code.visualstudio.com (дата звернення:13.06.2024).

21. Мова C#. URL: https://learn.microsoft.com/ru-ru/dotnet/csharp/ (дата звернення:13.06.2024).

22. Програма Aseprite. URL: https://www.aseprite.org/ (дата звернення:14.06.2024).

ДОДАТОК А

ЛІСТИНГ ПРОГРАМИ

Лістинг CameraController.cs :

```
using UnityEngine;

public class CameraController : MonoBehaviour
{
  public float dumping = 1.5f;
  public Vector2 offset = new Vector2(2f, 1f);
  public bool isLeft;
  private Transform player;
  private int lastX;

  void Start()
  {
    offset = new Vector2(Mathf.Abs(offset.x), offset.y);
    FindPlayer(isLeft);
  }

  public void FindPlayer(bool playerIsLeft)
  {
    player = GameObject.FindGameObjectWithTag("Player").transform;
    lastX = Mathf.RoundToInt(player.position.x);
    if (playerIsLeft)
    {
      transform.position = new Vector3(player.position.x - offset.x, player.position.y - offset.y, transform.position.z);
    }
    else
    {
      transform.position = new Vector3(player.position.x + offset.x, player.position.y + offset.y, transform.position.z);
    }
  }

  void Update()
  {
    if (player)
    {
      int currentX = Mathf.RoundToInt(player.position.x);
      if (currentX > lastX) isLeft = false; else if (currentX < lastX) isLeft = true;
      lastX = Mathf.RoundToInt(player.position.x);

      Vector3 target;
      if (isLeft)
      {
        target = new Vector3(player.position.x - offset.x, player.position.y + offset.y, transform.position.z);
      }
      else
      {
        target = new Vector3(player.position.x + offset.x, player.position.y + offset.y, transform.position.z);
      }

      Vector3 currentPosition = Vector3.Lerp(transform.position, target, dumping * Time.deltaTime);
      transform.position = currentPosition;
    }
  }
}
```

Лістинг CanvasController.cs :

```
using UnityEngine;

public class CanvasController : MonoBehaviour
{
   private Canvas canvas;

   void Start()
   {
      canvas = GetComponent<Canvas>();
      if (canvas != null)
      {
         canvas.enabled = false;
      }
   }

   void Update()
   {
      GameObject[] enemies = GameObject.FindGameObjectsWithTag("Enemy1");
      if (enemies.Length == 0)
      {
         if (canvas != null)
         {
            canvas.enabled = true;
         }
      }
      else
      {
         if (canvas != null)
         {
            canvas.enabled = false;
         }
      }
   }
}
```

Лістинг Enemy1.cs :

```
using System.Collections;
using UnityEngine;

public class Enemy1 : MonoBehaviour
{
  public int lives = 5;
  public int damage = 1;
  private bool isDead = false;

  private bool isDamageCooldown = false;
  private float damageCooldownDuration = 1f;

  private Rigidbody2D rb;
  private SpriteRenderer sprite;
  private BoxCollider2D boxCollider;
  private Animator anim;
  private Hero player;

  private void Awake()
  {
    rb = GetComponent<Rigidbody2D>();
```

```
    sprite = GetComponentInChildren<SpriteRenderer>();
    anim = GetComponent<Animator>();
    boxCollider = GetComponent<BoxCollider2D>();

    if (rb == null)
      Debug.LogError("Rigidbody2D is not assigned!");
    if (sprite == null)
      Debug.LogError("SpriteRenderer is not assigned!");
    if (anim == null)
      Debug.LogError("Animator is not assigned!");
    if (boxCollider == null)
      Debug.LogError("BoxCollider2D is not assigned!");

    player = GameObject.FindGameObjectWithTag("Player").GetComponent<Hero>();
  }

  private void OnTriggerEnter2D(Collider2D other)
  {
    if (other.CompareTag("Player"))
    {
      Hero player = other.GetComponent<Hero>();
      if (player != null)
      {
        player.TakeDamage(damage);
        Debug.Log("Player took damage. Player lives left: " + player.health);
      }
    }
  }

  public void TakeDamage(int damage)
  {
    if (!isDead && !isDamageCooldown)
    {
      lives -= damage;
      if (lives <= 0 && !isDead)
      {
        Die();
      }
      StartCoroutine(DamageCooldown());
    }
  }

  private IEnumerator DamageCooldown()
  {
    isDamageCooldown = true;
    yield return new WaitForSeconds(damageCooldownDuration);
    isDamageCooldown = false;
  }

  public void Die()
  {
    isDead = true;
    Debug.Log("Enemy died!");
    Destroy(gameObject);
  }
}
```

Лістинг EnemyLookAtPlayer.cs :

```
using UnityEngine;

public class EnemyLookAtPlayer : MonoBehaviour
```

```
{
    private Transform player;
    private bool isFacingRight = true;

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

    private void Update()
    {
        if (player != null)
        {
            if (transform.position.x > player.position.x && !isFacingRight)
            {
                Flip();
            }
            else if (transform.position.x < player.position.x && isFacingRight)
            {
                Flip();
            }
        }
    }

    private void Flip()
    {
        isFacingRight = !isFacingRight;
        Vector3 scale = transform.localScale;
        scale.x *= -1;
        transform.localScale = scale;
    }
}
```

Лістинг EnemyMovement.cs :

```
using UnityEngine;

public class EnemyMovement : MonoBehaviour
{
    public float speed = 2f;
    public float leftLimit = -5f;
    public float rightLimit = 5f;
    private Rigidbody2D rb;
    private SpriteRenderer spriteRenderer;
    private bool movingRight = true;
    private GameObject player;

    private void Awake()
    {
        rb = GetComponent<Rigidbody2D>();
        spriteRenderer = GetComponent<SpriteRenderer>();
        player = GameObject.FindWithTag("Player");
        if (rb == null || spriteRenderer == null || player == null)
        {
            Debug.LogError("Rigidbody2D, SpriteRenderer или Player не визначені!");
        }
    }

    private void FixedUpdate()
    {
        Move();
```

```
    }

    private void Move()
    {
        if (movingRight && transform.position.x >= rightLimit)
        {
            movingRight = false;
        }
        else if (!movingRight && transform.position.x <= leftLimit)
        {
            movingRight = true;
        }

        Vector2 direction = movingRight ? Vector2.right : Vector2.left;
        rb.velocity = new Vector2(direction.x * speed, rb.velocity.y);

        if ((direction.x > 0 && !spriteRenderer.flipX) || (direction.x < 0 && spriteRenderer.flipX))
        {
            spriteRenderer.flipX = !spriteRenderer.flipX;
        }
    }

    private void MoveManually()
    {
        Vector3 position = transform.position;
        if (movingRight && position.x >= rightLimit)
        {
            movingRight = false;
        }
        else if (!movingRight && position.x <= leftLimit)
        {
            movingRight = true;
        }

        Vector3 direction = movingRight ? Vector3.right : Vector3.left;
        position += direction * speed * Time.fixedDeltaTime;
        transform.position = position;

        if ((direction.x < 0 && !spriteRenderer.flipX) || (direction.x > 0 && spriteRenderer.flipX))
        {
            spriteRenderer.flipX = !spriteRenderer.flipX;
        }
    }

    private void OnDrawGizmosSelected()
    {
        Gizmos.color = Color.yellow;
        Gizmos.DrawLine(new Vector2(leftLimit, transform.position.y), new Vector2(rightLimit, transform.position.y));
    }
}
```

Лістинг GlowApear.cs :

```
using UnityEngine;

public class GlowApear : MonoBehaviour
{
    public SpriteRenderer targetSprite;
    private bool enemiesDefeated = false;

    void Start()
    {
```

```
        if (targetSprite != null)
        {
            targetSprite.enabled = false;
        }
    }

    void Update()
    {
        if (!enemiesDefeated)
        {
            CheckEnemies();
        }
    }

    void CheckEnemies()
    {
        GameObject[] enemies = GameObject.FindGameObjectsWithTag("Enemy1");
        if (enemies.Length == 0)
        {
            enemiesDefeated = true;
            if (targetSprite != null)
            {
                targetSprite.enabled = true;
                Debug.Log("Спрайт з'явився: " + targetSprite.gameObject.name);
            }
        }
    }
}
```

Лістинг LevelMenu.cs :

```
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;

public class LevelMenu : MonoBehaviour
{
  public Button[] buttons;
  public Sprite lockedSprite;
  public Sprite unlockedSprite;

  private void Awake()
  {
    int unlockedLevel = PlayerPrefs.GetInt("UnlockedLevel", 1);

    for (int i = 0; i < buttons.Length; i++)
    {
      if (i < unlockedLevel)
      {
        buttons[i].interactable = true;
        buttons[i].GetComponent<Image>().sprite = unlockedSprite;
      }
      else
      {
        buttons[i].interactable = false;
        buttons[i].GetComponent<Image>().sprite = lockedSprite;
      }
    }
  }

  public void OpenLevel(int levelId)
  {
```

```
    string levelName = "Level" + levelId;
    SceneManager.LoadScene(levelName);
  }
}
```

Лістинг MagicStone.cs :

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class MagicStone : MonoBehaviour
{
  public SpriteRenderer targetSprite;
  private bool enemiesDefeated = false;
  private bool isUsable = false;

  void Start()
  {
    if (targetSprite != null)
    {
      targetSprite.enabled = true;
    }
  }

  void Update()
  {
    if (!enemiesDefeated)
    {
      CheckEnemies();
    }
    else if (isUsable && Input.GetKeyDown(KeyCode.E))
    {
      LoadMainMenu();
      UnlockNewLevel();
    }
  }

  void CheckEnemies()
  {
    GameObject[] enemies = GameObject.FindGameObjectsWithTag("Enemy1");
    if (enemies.Length == 0)
    {
      enemiesDefeated = true;
      if (targetSprite != null)
      {
        targetSprite.enabled = true;
        isUsable = true;
      }
    }
  }

  void LoadMainMenu()
  {
    SceneManager.LoadScene("MainMenu");
  }

  void UnlockNewLevel()
  {
    if (SceneManager.GetActiveScene().buildIndex >= PlayerPrefs.GetInt("ReachedIndex"))
    {
      PlayerPrefs.SetInt("ReachedIndex", SceneManager.GetActiveScene().buildIndex + 1);
      PlayerPrefs.SetInt("UnlockedLevel", PlayerPrefs.GetInt("UnlockedLevel", 1) + 1);
```

```
        PlayerPrefs.Save();
      }
    }
}
```

Лістинг MainMenu.cs :

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class MainMenu : MonoBehaviour
{
  public void PlayGame()
  {
    SceneManager.LoadScene("Level 1");
  }

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

Лістинг MusicController.cs :

```
using UnityEngine;

public class MusicController : MonoBehaviour
{
  private static MusicController instance;

  private void Awake()
  {
    if (instance != null)
      Destroy(gameObject);
    else
    {
      instance = this;
      DontDestroyOnLoad(transform.gameObject);
    }
  }
}
```

Лістинг PauseGame.cs :

```
using UnityEngine;

public class canvasController : MonoBehaviour
{
   public Canvas[] canvases;
   private int currentCanvasIndex = 0;
   private bool isMenuOpen = false;

   void Start()
   {
      for (int i = 1; i < canvases.Length; i++)
      {
         canvases[i].gameObject.SetActive(false);
      }
   }
```

```
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Escape))
        {
            if (!isMenuOpen)
            {
                Time.timeScale = 0;
                isMenuOpen = true;

                canvases[currentCanvasIndex].gameObject.SetActive(false);
                currentCanvasIndex = (currentCanvasIndex + 1) % canvases.Length;
                canvases[currentCanvasIndex].gameObject.SetActive(true);
            }
            else
            {
                Time.timeScale = 1;
                isMenuOpen = false;

                canvases[currentCanvasIndex].gameObject.SetActive(false);
                currentCanvasIndex = (currentCanvasIndex + 1) % canvases.Length;
                canvases[currentCanvasIndex].gameObject.SetActive(true);
            }
        }
    }
}
```

Лістинг Player.cs :

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

public class Hero : MonoBehaviour
{
    [SerializeField] private float speed = 3f;
    [SerializeField] private float jumpForce = 10f;
    [SerializeField] private float climbSpeed = 3f;
    [SerializeField] public int health = 5;

    [SerializeField] private Image[] hearts;
    [SerializeField] private Sprite aliveHeart;
    [SerializeField] private Sprite deadHeart;

    [SerializeField] private AudioSource jumpSound;
    [SerializeField] private AudioSource ladderSound;
    [SerializeField] private AudioSource deathSound;
    [SerializeField] private AudioSource damageSound;
    [SerializeField] private AudioSource attackenemy;
    [SerializeField] private AudioSource attackmiss;

    private Rigidbody2D rb;
    private SpriteRenderer sprite;
    private Animator anim;
    private BoxCollider2D boxCollider;

    private bool isGrounded;
    private bool isClimbing;
    private bool isNearLadder;
    private float originalGravityScale;

    private bool isDead = false;
```

```
private bool isDamageCooldown = false;
private float damageCooldownDuration = 1.2f;

private Vector2 touchStartPos;
private float swipeThreshold = 3f;

public int damage = 1;
public bool isAttacking = false;
public bool isRecharged = true;

public Transform attackPos;
public float attackRange;
public LayerMask Enemy;

public static Hero Instance { get; set; }

private States State
{
  get { return (States)anim.GetInteger("State"); }
  set { anim.SetInteger("State", (int)value); }
}

private void Awake()
{
  rb = GetComponent<Rigidbody2D>();
  sprite = GetComponentInChildren<SpriteRenderer>();
  anim = GetComponent<Animator>();
  boxCollider = GetComponent<BoxCollider2D>();
  originalGravityScale = rb.gravityScale;
  Instance = this;
  isRecharged = true;
}

public enum States
{
  Idle,
  Run,
  Jump,
  Climb,
  Attack,
  Damage,
  Death
}

public void TakeDamage(int damage)
{
  if (!isDead && !isDamageCooldown)
  {
    health -= damage;
    State = States.Damage;
    damageSound.Play();
    if (health <= 0)
    {
      foreach (var h in hearts)
        h.sprite = deadHeart;
      Die();
    }

    StartCoroutine(DamageCooldown());
  }
}
```

```
private IEnumerator DamageCooldown()
{
  isDamageCooldown = true;
  yield return new WaitForSeconds(damageCooldownDuration);
  isDamageCooldown = false;
}

public void Die()
{
  if (!isDead)
  {
    isDead = true;
    State = States.Death;
    deathSound.Play();
    Debug.Log("Hero died!");
    StartCoroutine(HandleDeath());
  }
}

private IEnumerator HandleDeath()
{
  yield return new WaitForSeconds(2f);
  SceneManager.LoadScene("MainMenu");
}

private void FixedUpdate()
{
  CheckGround();
}

private void Update()
{
  if (isDead) return;

  if (Application.platform == RuntimePlatform.Android || Application.platform == RuntimePlatform.IPhonePlayer)
  {
    if (Input.touchCount > 0)
    {
      Touch touch = Input.GetTouch(0);

      if (touch.phase == TouchPhase.Began)
      {
        touchStartPos = touch.position;
      }
      else if (touch.phase == TouchPhase.Moved)
      {
        Vector2 touchDelta = touch.position - touchStartPos;

        if (Mathf.Abs(touchDelta.x) > swipeThreshold && Mathf.Abs(touchDelta.y) < swipeThreshold)
        {
          if (touchDelta.x < 0)
          {
            transform.position += Vector3.left * speed * Time.deltaTime;
            sprite.flipX = true;
            State = States.Run;
          }
          else
          {
            transform.position += Vector3.right * speed * Time.deltaTime;
            sprite.flipX = false;
            State = States.Run;
          }
```

```
        }
        else if (touchDelta.y > swipeThreshold && Mathf.Abs(touchDelta.x) < swipeThreshold && isGrounded)
        {
          Jump();
        }
      }
      else if (touch.phase == TouchPhase.Ended)
      {
        State = States.Idle;
      }
    }
}


for (int i = 0; i < hearts.Length; i++)
{
  if (i < health)
    hearts[i].sprite = aliveHeart;
  else
    hearts[i].sprite = deadHeart;

  if (i < hearts.Length)
    hearts[i].enabled = true;
  else
    hearts[i].enabled = false;
}

if (!isClimbing)
{
  if (Input.GetButton("Horizontal"))
  {
    Run();
  }
  else if (isGrounded)
  {
    State = States.Idle;
  }

  if (Input.GetButtonDown("Fire1") && isRecharged)
  {
    Attack();
  }

  if (isGrounded && Input.GetButtonDown("Jump"))
  {
    Jump();
  }

  if (!isGrounded && rb.velocity.y != 0)
  {
    State = States.Jump;
  }
}

if (isNearLadder && Mathf.Abs(Input.GetAxis("Vertical")) > 0.1f)
{
  isClimbing = true;
  rb.velocity = Vector2.zero;
  rb.gravityScale = 0;
  State = States.Climb;
}
```

```
    if (isClimbing)
    {
      Climb();
      if (Mathf.Abs(Input.GetAxis("Vertical")) < 0.1f)
      {
        State = States.Idle;
      }
      else
      {
        State = States.Climb;
      }

      if (Input.GetButtonDown("Jump"))
      {
        isClimbing = false;
        rb.gravityScale = originalGravityScale;
      }
    }
  }

  private void Run()
  {
    Vector3 dir = transform.right * Input.GetAxis("Horizontal");
    transform.position = Vector3.MoveTowards(transform.position, transform.position + dir, speed * Time.deltaTime);
    sprite.flipX = dir.x < 0.0f;

    if (isGrounded) State = States.Run;
  }

  private void Jump()
  {
    rb.AddForce(transform.up * jumpForce, ForceMode2D.Impulse);
    State = States.Jump;
    isGrounded = false;
    jumpSound.Play();
  }

  private void Attack()
  {
    if (isGrounded && isRecharged)
    {
      Debug.Log("Hero is attacking.");
      State = States.Attack;
      isAttacking = true;
      isRecharged = false;

      StartCoroutine(AttackAnimation());
      StartCoroutine(AttackCoolDown());
    }
  }

  private IEnumerator AttackAnimation()
  {
    float attackAnimationDuration = 0.4f;
    State = States.Attack;
    yield return new WaitForSeconds(attackAnimationDuration);

    isAttacking = false;
    if (!isDead) State = States.Idle;
  }

  private IEnumerator AttackCoolDown()
```

```
{
  float attackCooldownDuration = 1f;
  yield return new WaitForSeconds(attackCooldownDuration);
  isRecharged = true;
}

private IEnumerator EnemyOnAttack(Collider2D enemy)
{
  SpriteRenderer enemyColor = enemy.GetComponent<SpriteRenderer>();
  if (enemyColor != null)
  {
    enemyColor.color = new Color(0.8f, 0.25f, 0.25f);
    yield return new WaitForSeconds(0.4f);
    if (enemyColor != null)
    {
      enemyColor.color = new Color(1, 1, 1);
    }
  }
}

private void OnAttack()
{
  Collider2D[] colliders = Physics2D.OverlapCircleAll(attackPos.position, attackRange, Enemy);

  if (colliders.Length == 0)
  {
    attackmiss.Play();
  }
  else
  {
    attackenemy.Play();
    for (int i = 0; i < colliders.Length; i++)
    {
      Hero hero = colliders[i].GetComponent<Hero>();
      if (hero != null)
      {
        hero.TakeDamage(damage);
      }
      StartCoroutine(EnemyOnAttack(colliders[i]));

      Enemy1 enemy = colliders[i].GetComponent<Enemy1>();
      if (enemy != null)
      {
        enemy.TakeDamage(damage);
        Debug.Log("Enemy took damage. Enemy lives left: " + enemy.lives);
      }
    }
  }
}

private void OnDrawGizmosSelected()
{
  Gizmos.color = Color.red;
  Gizmos.DrawWireSphere(attackPos.position, attackRange);
}

private void Climb()
{
  float vDirection = Input.GetAxis("Vertical");
  transform.Translate(Vector3.up * vDirection * climbSpeed * Time.deltaTime);

  if (isNearLadder && Mathf.Abs(Input.GetAxis("Vertical")) > 0.1f)
```

```
    {
      State = States.Climb;
      ladderSound.Play();
    }
    else
    {
      State = States.Idle;
    }
  }

  private void CheckGround()
  {
    Vector2 size = new Vector2(boxCollider.size.x - 0.1f, 0.1f);
    Vector2 center = (Vector2)transform.position + boxCollider.offset + Vector2.down * (boxCollider.size.y / 2 - 0.05f);
    Collider2D[] colliders = Physics2D.OverlapBoxAll(center, size, 0f);

    isGrounded = false;
    foreach (var collider in colliders)
    {
      if (collider != boxCollider)
      {
        isGrounded = true;
        break;
      }
    }

    if (isGrounded && isClimbing)
    {
      isClimbing = false;
      rb.gravityScale = originalGravityScale;
      State = States.Idle;
    }
  }

  private void OnTriggerEnter2D(Collider2D other)
  {
    if (other.CompareTag("Ladder"))
    {
      isNearLadder = true;
      if (isClimbing)
      {
        isClimbing = false;
        rb.gravityScale = originalGravityScale;
        State = States.Idle;
      }
    }

    if (other.CompareTag("Enemy1"))
    {
      Enemy1 enemy = other.GetComponent<Enemy1>();
      if (enemy != null)
      {
        enemy.TakeDamage(damage);
        Debug.Log("Enemy took damage. Enemy lives left: " + enemy.lives);
      }
    }
  }

  private void OnTriggerStay2D(Collider2D other)
  {
    if (other.CompareTag("Obstacle"))
    {
```

```
        Debug.Log("Player hit a trap.");
        Hero.Instance.TakeDamage(damage);
      }
    }

    private void OnTriggerExit2D(Collider2D other)
    {
      if (other.CompareTag("Ladder"))
      {
        isNearLadder = false;
        if (isClimbing)
        {
          isClimbing = false;
          ladderSound.Stop();
          rb.gravityScale = originalGravityScale;
          State = States.Idle;
        }
      }
    }
}
```

Лістинг ResetProgress.cs :

```
using UnityEngine;

public class ProgressResetter : MonoBehaviour
{
    void Start()
    {
        ResetLevel();
    }

    public void ResetLevel()
    {
        PlayerPrefs.SetInt("UnlockedLevel", 1);
        PlayerPrefs.Save();
        Debug.Log("Level progress has been reset.");
    }
}
```

Лістинг RestartLevelTrigger.cs :

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class RestartLevelTrigger : MonoBehaviour
{
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.gameObject.CompareTag("Player"))
        {
            SceneManager.LoadScene(SceneManager.GetActiveScene().name);
        }
    }
}
```

Лістинг Spike.cs :

```
using UnityEngine;

public class Spike : MonoBehaviour
```

```
{
  public int damage = 1;

  private void OnCollisionEnter2D(Collision2D collision)
  {
    if (collision.gameObject.CompareTag("Player"))
    {
      Debug.Log("Player hit a trap.");
      collision.gameObject.GetComponent<Hero>().TakeDamage(damage);
    }
  }
}
```

Лістинг ElevatorEditor.cs :

```
using UnityEditor;

namespace Cainos.PixelArtPlatformer_VillageProps
{
   [CustomEditor(typeof(Elevator))]
   public class ElevatorEditor : LucidEditor.LucidEditor
   {
   }
}
```

Лістинг Elevator.cs :

```
using UnityEngine;
using Cainos.LucidEditor;
using Cainos.Common;

namespace Cainos.PixelArtPlatformer_VillageProps
{
   public class Elevator : MonoBehaviour
   {
      [FoldoutGroup("Params")] public Vector2 lengthRange = new Vector2(2, 5);
      [FoldoutGroup("Params")] public float waitTime = 1.0f;
      [FoldoutGroup("Params")] public float moveSpeed = 3.0f;
      [FoldoutGroup("Params")] public State startState = State.Up;

      [FoldoutGroup("Reference")] public Rigidbody2D platform;
      [FoldoutGroup("Reference")] public SpriteRenderer chainL;
      [FoldoutGroup("Reference")] public SpriteRenderer chainR;

      [FoldoutGroup("Runtime"), ShowInInspector]
      public float Length
      {
         get { return length; }
         set
         {
            if (value < 0) value = 0.0f;
            this.length = value;

            platform.transform.localPosition = new Vector3(0.0f, -value, 0.0f);
            chainL.size = new Vector2(0.09375f, value - 8 * 0.03125f);
            chainR.size = new Vector2(0.09375f, value - 8 * 0.03125f);
         }
      }
      private float length;

      [FoldoutGroup("Runtime"), ShowInInspector]
```

```
public State CurState
{
    get { return curState; }
    set
    {
        curState = value;
    }
}
private State curState;

[FoldoutGroup("Runtime"), ShowInInspector]
public bool IsWaiting
{
    get { return isWaiting; }
    set
    {
        if (isWaiting == value) return;
        isWaiting = value;
        waitTimer = 0.0f;
    }
}
private bool isWaiting = false;

private float waitTimer;
private float curSpeed;
private float targetLength;
private SecondOrderDynamics secondOrderDynamics = new SecondOrderDynamics(4.0f, 0.3f, -0.3f);

private void Start()
{
    curState = startState;
    Length = curState == State.Up ? lengthRange.y : lengthRange.x;
    targetLength = Length;

    secondOrderDynamics.Reset(targetLength);
}

private void Update()
{
    if (IsWaiting)
    {
        waitTimer += Time.deltaTime;
        if (waitTimer > waitTime) IsWaiting = false;
        curSpeed = 0.0f;
    }
    else
    {
        if (curState == State.Up)
        {
            curSpeed = -moveSpeed;
            if (targetLength < lengthRange.x)
            {
                curState = State.Down;
                IsWaiting = true;
            }
        }
        else if (curState == State.Down)
        {
            curSpeed = moveSpeed;
            if (targetLength > lengthRange.y)
            {
                curState = State.Up;
```

```
                IsWaiting = true;
            }
        }
    }

    targetLength += curSpeed * Time.deltaTime;
}

private void FixedUpdate()
{
    Length = secondOrderDynamics.Update(targetLength, Time.fixedDeltaTime);
}

public enum State
{
    Up,
    Down
}
}
}
```

Лістинг MovingPlatform.cs :

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

namespace Cainos.PixelArtPlatformer_VillageProps
{

public class MovingPlatform : MonoBehaviour
{
    public float velocityInheritPercent = 0.8f;

    private List<Transform> onPlatformObjects;
    private Vector3 prevPos;
    private Vector2 velocity;

    private void Start()
    {
        onPlatformObjects = new List<Transform>();
        prevPos = transform.position;
    }

    private void FixedUpdate()
    {
        velocity = (transform.position - prevPos) / Time.fixedDeltaTime;
        prevPos = transform.position;

        foreach (Transform t in onPlatformObjects)
        {
            t.Translate(velocity * Time.fixedDeltaTime);
        }
    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.attachedRigidbody && collision.attachedRigidbody.bodyType == RigidbodyType2D.Dynamic)
        {
            if (onPlatformObjects.Contains(collision.transform)) return;

            onPlatformObjects.Add(collision.transform);
            if (collision.attachedRigidbody) collision.attachedRigidbody.velocity -= velocity * velocityInheritPercent;
```

```
        }
    }

    private void OnTriggerExit2D(Collider2D collision)
    {
        if (collision.attachedRigidbody && collision.attachedRigidbody.bodyType == RigidbodyType2D.Dynamic)
        {
            if (onPlatformObjects.Contains(collision.transform) == false) return;
            onPlatformObjects.Remove(collision.transform);

            if (collision.attachedRigidbody) collision.attachedRigidbody.velocity += velocity * velocityInheritPercent;
        }
    }
  }
}
```

Лістинг SecondOrderDynamics.cs :

```
using UnityEngine;

namespace Cainos.Common
{
  [System.Serializable]
  public struct SecondOrderDynamics
  {
    private Vector3 xp;
    private Vector3 y, yd;
    private float k1, k2, k3;
    private Vector3 xd;
    private float k2_stable;

    [SerializeField] private float f;
    [SerializeField] private float d;
    [SerializeField] private float r;

    public float Frequency
    {
      get { return f; }
      set
      {
        if (value <= 0.01f) value = 0.01f;

        f = value;
        UpdateInnerParams();
      }
    }

    public float Damping
    {
      get { return d; }
      set
      {
        if (value <= 0.01f) value = 0.01f;

        d = value;
        UpdateInnerParams();
      }
    }

    public float Response
    {
      get { return r; }
```

```
    set
    {
        r = value;
        UpdateInnerParams();
    }
}

public SecondOrderDynamics(float frequency, float damping, float response)
{
    this.f = 1.0f;
    this.d = 0.0f;
    this.r = 0.0f;
    xp = Vector3.zero;
    y = Vector3.zero;
    xd = Vector3.zero;
    yd = Vector3.zero;
    k1 = 0.0f;
    k2 = 0.0f;
    k3 = 0.0f;
    k2_stable = 0.0f;

    Reset(frequency, damping, response, Vector3.zero);
}

public void Reset(float frequency, float damping, float response, Vector3 x0)
{
    f = frequency;
    d = damping;
    r = response;

    xp = x0;
    y = x0;
    xd = Vector3.zero;
    yd = Vector3.zero;

    UpdateInnerParams();
}

public void Reset(Vector3 x0)
{
    Reset(f, d, r, x0);
}

public void Reset(Vector2 x0)
{
    Reset(f, d, r, x0);
}

public void Reset(float x0)
{
    Reset(f, d, r, Vector3.one * x0);
}

public Vector3 Update(Vector3 x, float t)
{
    if (t < Mathf.Epsilon) return y;

    xd = (x - xp) / t;
    xp = x;

    k2_stable = Mathf.Max(k2, 1.1f * (t * t * 0.25f + t * k1 * 0.5f));
    y += t * yd;
```

```
            yd += t * (x + k3 * xd - y - k1 * yd) / k2_stable;

            return y;
        }

        public Vector2 Update(Vector2 x, float t)
        {
            return Update(new Vector3(x.x, x.y, 0.0f), t);
        }

        public float Update(float x, float t)
        {
            return Update(Vector3.one * x, t).x;
        }

        private void UpdateInnerParams()
        {
            k1 = d / (Mathf.PI * f);
            k2 = 1.0f / ((2.0f * Mathf.PI * f) * (2.0f * Mathf.PI * f));
            k3 = r * d / (2.0f * Mathf.PI * f);
        }
    }
}
```

Лістинг OptionsCreator.cs :

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

public class OptionsCreator : MonoBehaviour
{
    [Header("Script References")]
    [SerializeField]
    private MenuInputController myMenuInputController;
    [SerializeField]
    private ColorChangerToFocusedAndUnfocused myColorChanger;
    [SerializeField]
    private DictionaryCreator myDictionary;
    [SerializeField]
    private OptionsAndSubOptionsComunicator myOptSuboptComunicator;

    [SerializeField]
    OptionsPositionConfigurator myOptionsPositionConfigurator;

    [Header("Objects")]
    [SerializeField]
    GameObject goOptionPrefab;

    [SerializeField]
    GameObject goCanvasMenu;

    [SerializeField]
    string sBackButtonText;
    [SerializeField]
    string sQuitButtonText;

    [Tooltip("These options will become the GameObject parents of the SubOptions.")]
    [SerializeField]
    string[] aMenuOptions;

    private List<GameObject> listOptions = new List<GameObject>();
```

```
int iOptionFocused = 0;

public void EnableClickInButtons(bool value)
{
    for (int i = 0; i < listOptions.Count; i++)
    {
        listOptions[i].GetComponent<Button>().interactable = value;
    }
}

public void CreateDictionaryWithOptions()
{
    myDictionary.CreateDictionary(aMenuOptions);
}

public void IncreaseOptionFocusedNumber()
{
    iOptionFocused++;

    if (iOptionFocused > (listOptions.Count - 1))
    {
        iOptionFocused = 0;
    }
}

public void DecreaseOptionFocusedNumber()
{
    iOptionFocused--;

    if (iOptionFocused < 0)
    {
        iOptionFocused = (listOptions.Count - 1);
    }
}

public void CheckOptionFocusedAndDoTheCorrectAction()
{
    if (iOptionFocused == listOptions.Count - 2)
    {
        Debug.Log("Back Button...");
        ResetAndHideMenuWithBackButton();
    }
    else if (iOptionFocused == listOptions.Count - 1)
    {
        Debug.Log("Quitting game...");
        myMenuInputController.QuitGame();
    }
    else
    {
        myMenuInputController.SetOptionsFocused(false);
        EnableClickInButtons(false);
        myOptSuboptComunicator.ActivateAllSuboptionsOfAnOption(listOptions, GetTextFromOptionFocused());
    }
}

public List<GameObject> GetList()
{
    return listOptions;
}
public int GetListCount()
{
```

```
    return listOptions.Count;
}
public GameObject GetObjectFromListWithIndex(string sValueInDictionary)
{
    return (listOptions[myDictionary.GetIndexFromValue(sValueInDictionary)]);
}
public GameObject GetObjectFromListWithIndex(int iIndex)
{
    return (listOptions[iIndex]);
}

public int GetIndexOfOptionFocused()
{
    return iOptionFocused;
}

public GameObject GetOptionPrefab()
{
    return goOptionPrefab;
}

public int GetNumberOfChildrenOfOptionFocused()
{
    return (listOptions[iOptionFocused].transform.GetChild(0).transform.childCount - 1);
}

public string GetBackButtonText()
{
    return sBackButtonText;
}

public void SetIndexOfOptionFocused(int iNewValue)
{
    iOptionFocused = iNewValue;
}

public void ResetAndHideMenuWithBackButton()
{
    ResetOptions();
    myMenuInputController.ShowMenu();
}

public void ResetOptions()
{
    int iCount = listOptions.Count;

    if (iCount > 0)
    {
        for (int i = 0; i < iCount - 2; i++)
        {
            listOptions[i].transform.GetChild(0).gameObject.SetActive(false);
            ChangeOptionTextColorToUnfocused(listOptions[i].GetComponent<Text>());
        }
        ChangeBackButtonTextColorToUnfocused(listOptions[iCount - 2].GetComponent<Text>());
        ChangeQuitButtonTextColorToUnfocused(listOptions[iCount - 1].GetComponent<Text>());
        ChangeOptionTextColorToFocused(listOptions[0].GetComponent<Text>());
    }

    iOptionFocused = 0;
    myMenuInputController.SetOptionsFocused(true);
}
```

```
    public string GetTextFromOptionFocused()
    {
        return listOptions[iOptionFocused].GetComponent<Text>().text;
    }

    public void ChangeBackButtonTextColorToFocused(Text textLabel)
    {
        myColorChanger.ChangeTextColorToFocused(textLabel);
    }

    public void ChangeBackButtonTextColorToUnfocused(Text textLabel)
    {
        myColorChanger.ChangeTextColorToUnfocused(textLabel);
    }

    public void ChangeQuitButtonTextColorToFocused(Text textLabel)
    {
        myColorChanger.ChangeTextColorToFocused(textLabel);
    }

    public void ChangeOptionTextColorToFocused(Text textLabel)
    {
        myColorChanger.ChangeTextColorToFocused(textLabel);
    }

    public void ChangeQuitButtonTextColorToUnfocused(Text textLabel)
    {
        myColorChanger.ChangeTextColorToUnfocused(textLabel);
    }

    public void ChangeOptionTextColorToUnfocused(Text textLabel)
    {
        myColorChanger.ChangeTextColorToUnfocused(textLabel);
    }

    public void BuildOptions()
    {
        for (int i = 0; i < aMenuOptions.Length; i++)
        {
            GameObject goNewOption = Instantiate(goOptionPrefab, goCanvasMenu.transform);
            int currentOptionIndex = i;

            goNewOption.transform.localPosition = myOptionsPositionConfigurator.GetPositionForIndex(i);
            goNewOption.GetComponent<Text>().text = aMenuOptions[i];
            goNewOption.GetComponent<Button>().onClick.AddListener(delegate
            {
                myMenuInputController.SetOptionsFocused(false);
                myOptSuboptComunicator.ActivateAllSuboptionsOfAnOption(listOptions,
goNewOption.GetComponent<Text>().text);
                iOptionFocused = currentOptionIndex;
            });

            if (i == 0)
            {
                ChangeOptionTextColorToFocused(goNewOption.GetComponent<Text>());
            }
            else
                ChangeOptionTextColorToUnfocused(goNewOption.GetComponent<Text>());

            listOptions.Add(goNewOption);

            myOptionsPositionConfigurator.InstantiatePanelForSuboptionsOfOptionI(goNewOption.transform, i);
```

```
        }

        GameObject goBackButton = Instantiate(goOptionPrefab, goCanvasMenu.transform);
        goBackButton.transform.localPosition = myOptionsPositionConfigurator.GetPositionForIndex(listOptions.Count);
        goBackButton.GetComponent<Text>().text = sBackButtonText;
        goBackButton.GetComponent<Button>().onClick.AddListener(delegate
        {
            ResetAndHideMenuWithBackButton();
        });

        ChangeBackButtonTextColorToUnfocused(goBackButton.GetComponent<Text>());
        listOptions.Add(goBackButton);

        GameObject goQuitButton = Instantiate(goOptionPrefab, goCanvasMenu.transform);
        goQuitButton.transform.localPosition = myOptionsPositionConfigurator.GetPositionForIndex(listOptions.Count);

        goQuitButton.GetComponent<Text>().text = sQuitButtonText;
        goQuitButton.GetComponent<Button>().onClick.AddListener(delegate { myMenuInputController.QuitGame(); });

        ChangeQuitButtonTextColorToUnfocused(goQuitButton.GetComponent<Text>());
        listOptions.Add(goQuitButton);
    }
}
```

ДОДАТОК Б

ВІДГУК
Керівника економічного розділу
на кваліфікаційну роботу бакалавра на тему:
«Розробка кросплатформеного ігрового застосунку на основі технологічного стеку Unity/C#»
Студента групи 122-20-4 Садовського Ярослава Олександровича

Керівник економічного розділу
доц. каф. ПЕП та ПУ, к.е.н **Л.В. Касьяненко**

ДОДАТОК В

ПЕРЕЛІК ФАЙЛІВ НА ДИСКУ

Ім’я файлу	**Опис**
Пояснювальні документи	
КваліфікаційнаРобота_Садовського.doxc	Пояснювальна записка до кваліфікаційної роботи. Документ Word.
КваліфікаційнаРобота _Садовського.pdf	Пояснювальна записка до кваліфікаційної роботи в форматі PDF
Програма	
diplom.rar	Архів. Містить код програми і откомпільовану програму.
Презентація	
Презентація_ Садовського.ppt	Презентація кваліфікаційної роботи