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 *Лісовця Яна Ігоровича*
(ПІБ)

академічної групи *122-20-4*
(шифр)

спеціальності *122 Комп'ютерні науки*
(код і назва спеціальності)

освітньої програми Комп'ютерні науки
(назва освітньої програми)

на тему: *Розробка ігрового додатку на рушії Unity з використанням мови програмування C#*

Керівники	**Прізвище, ініціали**	**Оцінка за шкалою**		**Підпис**
		рейтинговою	**інституційною**	
кваліфікаційної роботи	*проф. Мороз Б.І.*			
розділів:				
спеціальний	*проф. Мороз Б.І.*			
економічний	*доц. Касьяненко Л.В.*			
Рецензент				
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* *Лісовця Яна Ігоровича*
(група) (прізвище та ініціали)
тема кваліфікаційної роботи *Розробка ігрового додатку на рушії Unity з використанням мови програмування C#*

затверджена наказом ректора НТУ «ДП» від *23.05.2024 р.* № *469-с*

Розділ	Зміст виконання	Термін виконання
Спеціальний	***На основі матеріалів проєктно-технологічної практики та інших науково-технічних джерел провести аналіз стану рішення проблеми та постановку задачі. Обґрунтувати вибір та здійснити реалізацію методів вирішення проблеми***	***04.06.2024.***
Економічний	***Провести розрахунок трудомісткості розробки програмного забезпечення, витрат на створення ПЗ й тривалості його розробки***	***18.06.2024.***

Завдання видав _____ *проф. Мороз Б.І.*
(підпис) (посада, прізвище,ініціали)
Завдання прийняв до виконання _____ *Лісовець Я.І.*
(підпис) (прізвище, ініціали)

Дата видачі завдання: *14.01.2024 р.*

Термін подання кваліфікаційної роботи до ЕК: *21.06.2024р.*

РЕФЕРАТ

Пояснювальна записка: 85 с., 32 рис., 3 дод., 22 джерел.

Об'єкт розробки: ігровий застосунок в жанрі 2D платформер, у візуальному стилі «Pixel Art»

Мета кваліфікаційної роботи: створення ігрового застосунку на платформі ігрового рушія Unity.

У вступі розглядається аналіз та сучасний стан проблеми, конкретизується мета кваліфікаційної роботи та галузь її застосування, наведено обґрунтування актуальності теми та уточнюється постановка завдання.

У першому розділі проаналізовано предметну галузь, визначено актуальність завдання та призначення розробки, сформульовано постановку завдання, зазначено вимоги до програмної реалізації, технологій та програмних засобів.

У другому розділі проаналізовані наявні рішення, обрано платформу для розробки, виконано проєктування і розробка програми, описана робота системи, алгоритм і структура його функціонування, а також виклик та завантаження програми, визначено вхідні і вихідні дані, охарактеризовано склад параметрів технічних засобів.

В економічному розділі визначено трудомісткість розроблення програмного забезпечення, проведений підрахунок вартості роботи по створенню програми та розраховано час на його створення.

Практичне значення полягає у створенні ігрового застосунку, який буде виконувати розважальну функцію і також розвивати логічне мислення у гравця. Додаток має певну кількість ігрових механік, звукових та візуальних ефектів.

Актуальність даного програмного продукту полягає в попиті на 2D платформери, в яких гравці можуть отримати унікальний ігровий досвід.

Список ключових слів: ПРОГРАМА, ДВОМІРНИЙ, ІГРОВИЙ РУШІЙ, UNITY, МЕХАНІКА, ІГРОВИЙ ДОДАТОК.

ABSTRACT

Explanatory note: 85 p., 32 figures, 3 appendices, 22 sources.

Object of development: game application in the genre of 2D platformer, in the visual style of "Pixel Art"

The purpose of the qualification work: creation of a game application on the Unity game engine platform.

In the introduction, the analysis and current state of the problem is considered, the purpose of the qualification work and the field of its application are specified, the justification of the relevance of the topic is given, and the statement of the task is clarified.

In the first section, the subject area is analyzed, the relevance of the task and the purpose of the development is determined, the task statement is formulated, and the requirements for software implementation, technologies and software tools are specified.

In the second section, available solutions are analyzed, a platform for development is chosen, the design and development of the program is performed, the operation of the system, the algorithm and the structure of its functioning are described, as well as the call and download of the program, the input and output data are determined, and the composition of the parameters of the technical means is characterized.

In the economic section, the labor intensity of software development is determined, the cost of work on creating the program is calculated, and the time for its creation is calculated.

The practical meaning is to create a game application that will perform an entertaining function and also develop logical thinking in the player. The application has a certain number of game mechanics, sound and visual effects.

The relevance of this software product lies in the demand for 2D platformers in which players can get a unique gaming experience.

List of keywords: SOFTWARE, 2D, GAME ENGINE, UNITY, MECHANICS, GAME APP.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ПК – персональний комп’ютер;

ЦП – центральний процесор;

ОЗП – оперативна пам’ять;

ОС – операційна система;

2D – двомірний простір;

3D – тривимірний простір;

ААА – термін, що позначає клас високобюджетних комп'ютерних ігор;

ЗМІСТ

ВСТУП

Індустрія комп'ютерних ігор є однією з найбільш динамічно зростаючих галузей сучасної економіки, що поєднує в собі новітні технології, креативність та інновації. За останні роки ігри пройшли великий шлях, від простих розваг, до доволі серйозних інструментів розвитку та навчання. Завдяки розвиненим цифровим технологіям, зокрема у сфері графіки, штучного інтелекту та мережевих комунікацій, були створені складні і багатогранні ігрові світи, які приваблюють мільйони гравців по всьому світу.

Ігрові рушії відіграють ключову роль у створенні цих віртуальних світів, забезпечуючи платформу для розробки та тестування. Серед великої кількості ігрових рушіїв можно виділити Unity, який зарекомендував себе як ідеальний інструмент для створення як простих інді-проектів, так і великих комерційних ігор, що охоплюють різні платформи, від мобільних пристроїв до консолей і ПК. Unity вражає своєю універсальністю, адже є потужним, гнучким та доступним інструментом для розробки ігор.

У рамках цієї дипломної роботи буде розглянуто процес розробки 2D платформера, що створюється з метою розважити та випробувати гравців, використовуючи Unity та мову програмування C#. 2D платформери є одним з найпопулярніших жанрів в ігровій індустрії, завдяки своїй простоті та захоплюючому геймплею. Вони надають гравцям можливість зануритися у різноманітні світи, де вони можуть досліджувати локації, вирішувати головоломки та долати перешкоди.

Ця робота розпочнеться з аналізу сучасних тенденцій в розробці комп'ютерних ігор та значення ігрових рушіїв у цьому процесі. Далі буде

представлено огляд Unity, його можливостей та переваг для інді-розробників. Особлива увага буде приділена мові програмування C# як основному інструменту для створення ігрової логіки в Unity.

У практичній частині буде розглянуто етапи розробки 2D платформера: від концептуалізації та проектування до реалізації та тестування.

Метою даної дипломної роботи є не лише створення функціональної гри, але й демонстрація можливостей Unity та C# для розробки інді-проектів. Крім того, в рамках роботи буде проаналізовано, як використання сучасних інструментів та методів розробки може вплинути на кінцевий продукт, його якість та привабливість для гравців. Це дозволить не тільки здобути практичний досвід у створенні ігор, але й підкреслити важливість інновацій та креативного підходу в сучасній ігровій індустрії.

РОЗДІЛ 1

АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1. Загальні відомості з предметної галузі

1.1.1. Історія розвитку ігрової індустрії

Комп'ютерні ігри є однією з найдинамічніших та найприбутковіших галузей індустрії розваг. Вони охоплюють широкий спектр жанрів, від простих аркадних ігор до складних симуляторів та масових багатокористувацьких онлайн-ігор. З появою електронних комп’ютерів, починається історія комп'ютерних ігор 1940 - 1950 років. Однією з найвідоміших ігор тих часів є "Tennis for Two," розроблена в 1958 році фізиком Вільямом Гігінботамом.

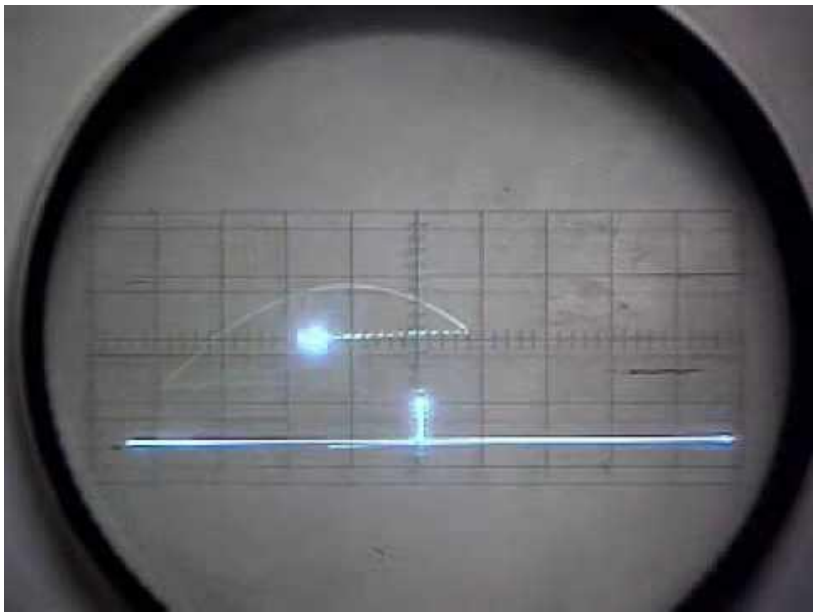

Рис. 1.1. Фотографія гри «Tennis for Two»

У 1962 році Стів Рассел створив "Spacewar!" для комп'ютера ПДП-1. Це одна з перших комп'ютерних ігор, де гравцю потрібно керувати космічним кораблем та брати участь у космічних боях. Хоча ці ігри не були комерційно доступними, вони заклали основу для майбутніх розробок.

Рис. 1.2. Фотографія гри «Spacewar!»

1970-ті роки ознаменували комерційне народження відеоігор. У 1972 році Нолан Бушнелл і Тед Дабні створили аркадну відеогру «Понг», яка стала масовим хітом і закріпила Atari як головного гравця в ігровій індустрії. Кінець 1970-х і початок 1980-х часто називають Золотим століттям аркадних ігор. Такі культові ігри, як «Space Invaders», «Pac-Man» і «Donkey Kong», домінували в аркадах і стали феноменом культури.

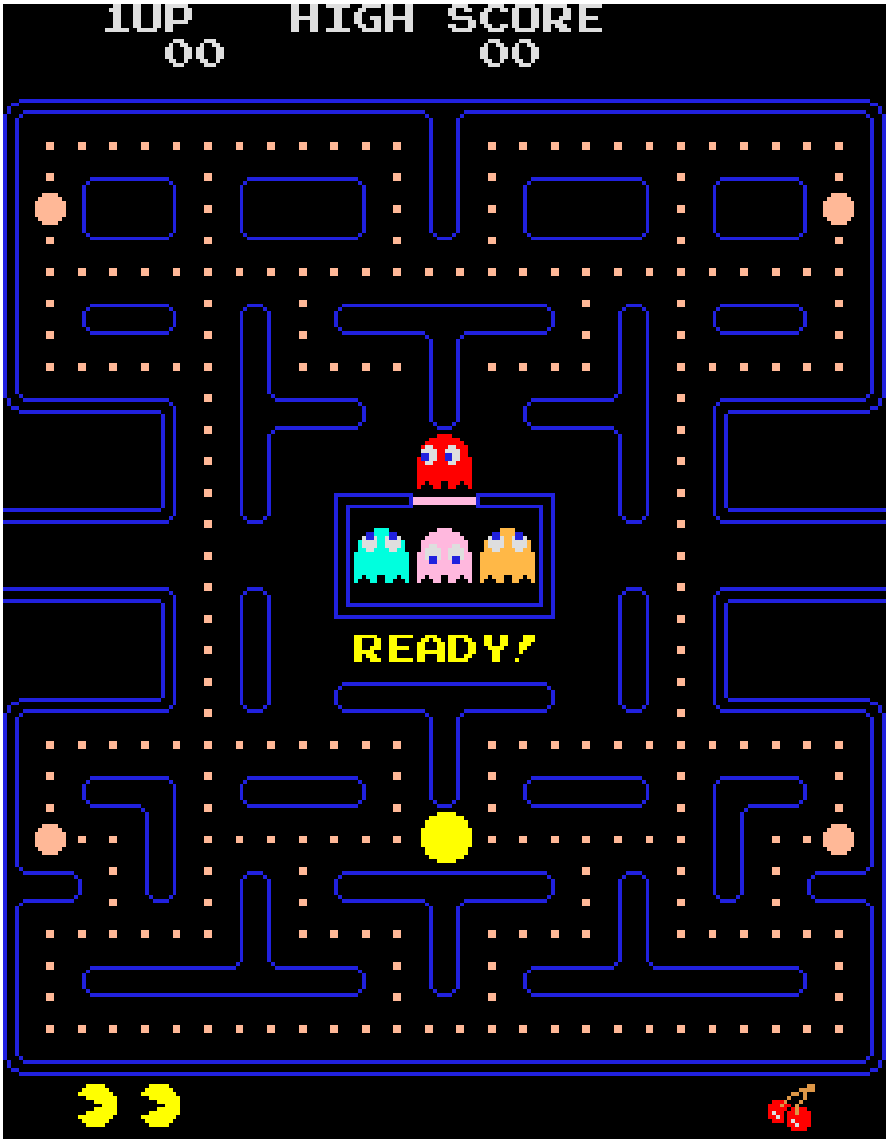

Рис. 1.3. Скріншот гри «Pac-Man»

Індустрія комп’ютерних ігор отримала значний розвиток 1990-ті роки, з появою потужних відеокарт. Такі проєкти, як «Doom» і «Half-Life», розширили межі ігрового дизайну та технологій. У 2000-х роках з’явилися багатокористувацькі онлайн-ігри, як «World of Warcraft», в які грали мільйони людей у всьому світі.

Рис. 1.4. Скріншот гри «Half-Life»

У 2010-х роках, ігрова індустрія пережила величезну еволюцію з погляду технологій та інновацій. Поява нових консолей, передових графічних процесорів і штучного інтелекту вплинула на розробку та сферу застосування комп'ютерних ігор.

Поява таких ігрових приставок, як PlayStation 4 та Xbox One відображала значний стрибок у графічних можливостях і потужності нових комплектуючих на той час. При цьому обидві приставки, мали можливість хмарних обчислень та інтеграції з іншими медіа-сервісами.

У 2016 році на смартфони випустили гру «Pokémon GO». Розробники інтегрували у гру технологію доповненої реальності, дозволяючи гравцям ловити віртуальних покемонів у реальних місцях, за допомогою своїх смартфонів. Геймплей передбачає, що гравець буде фізично активним протягом всієї гри. Поєднання доповненої реальності, ігрового процесу на

основі визначення місця розташування та соціальної взаємодії зробило «Pokémon GO» новаторським і культурно значущим явищем у світі ігор.

Рис. 1.5. Скріншоти гри «Pokémon GO»

Виникнення платформ та магазинів ігор, такі як Steam, Epic Games і т.д., зробило революцію в розповсюдженні ігор та інді-розробці ігор. Steam був однією з перших великих платформ, яка запропонувала комплексний цифровий ринок ігор для ПК. Це позбавило розробників потреби продавати фізичні носії та зробило ігри більш доступними для покупки. Розповсюдження ігор означало, що навіть невеликі інді-розробники могли охопити глобальну аудиторію, якщо їхні ігри мали попит серед гравців. Steam включає такі функції, як автоматичні оновлення, форуми спільноти та підтримка модів, завдяки чому створив успішну екосистему.

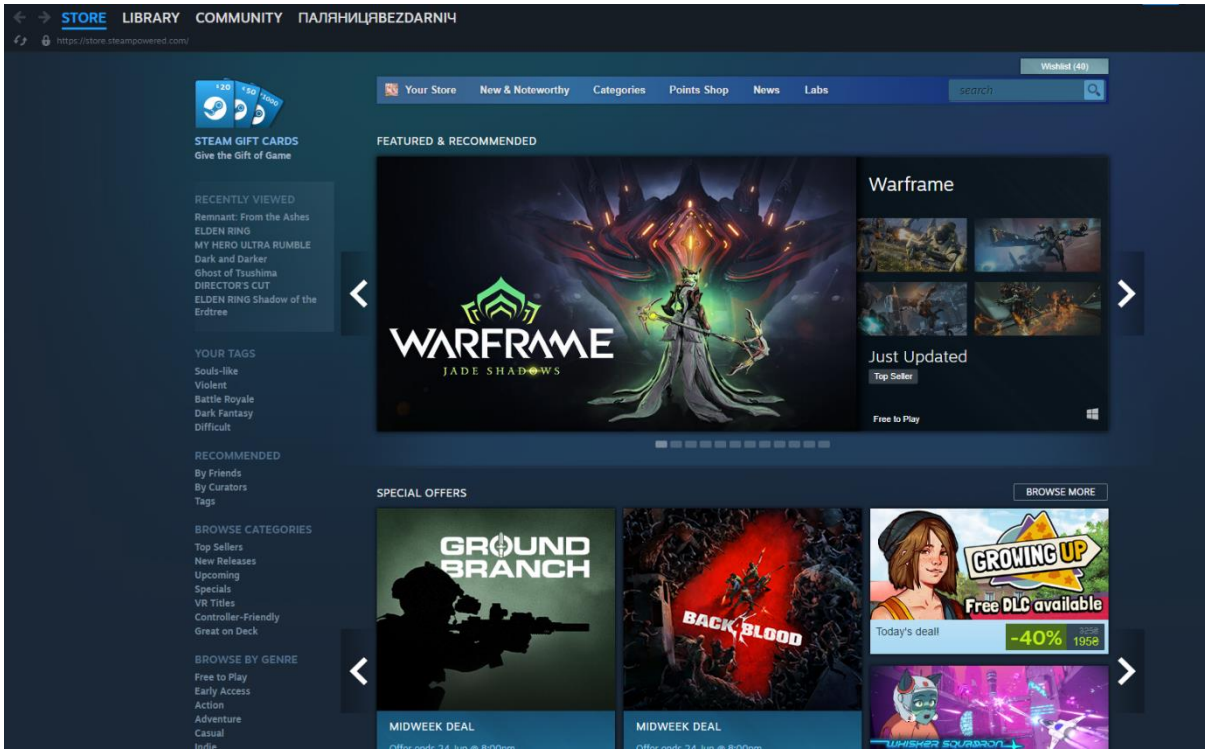

Рис. 1.6. Скріншот головної сторінки Steam

Раніше створення гри означало переговори з видавцями, виготовлення фізичних копій і забезпечення місця на полицях для торгівлі фізичними копіями. Усе це вимагало значних фінансових інвестицій і зв’язків з галуззю. Цифрові магазини усунули багато перешкод, дозволивши розробникам більше зосередитися на творчості та розробці. Такі платформи, як Steam і Epic Games Store, зазвичай отримують відсоток від продажів, але зазвичай це вигідніше порівняно з традиційними угодами видавців.

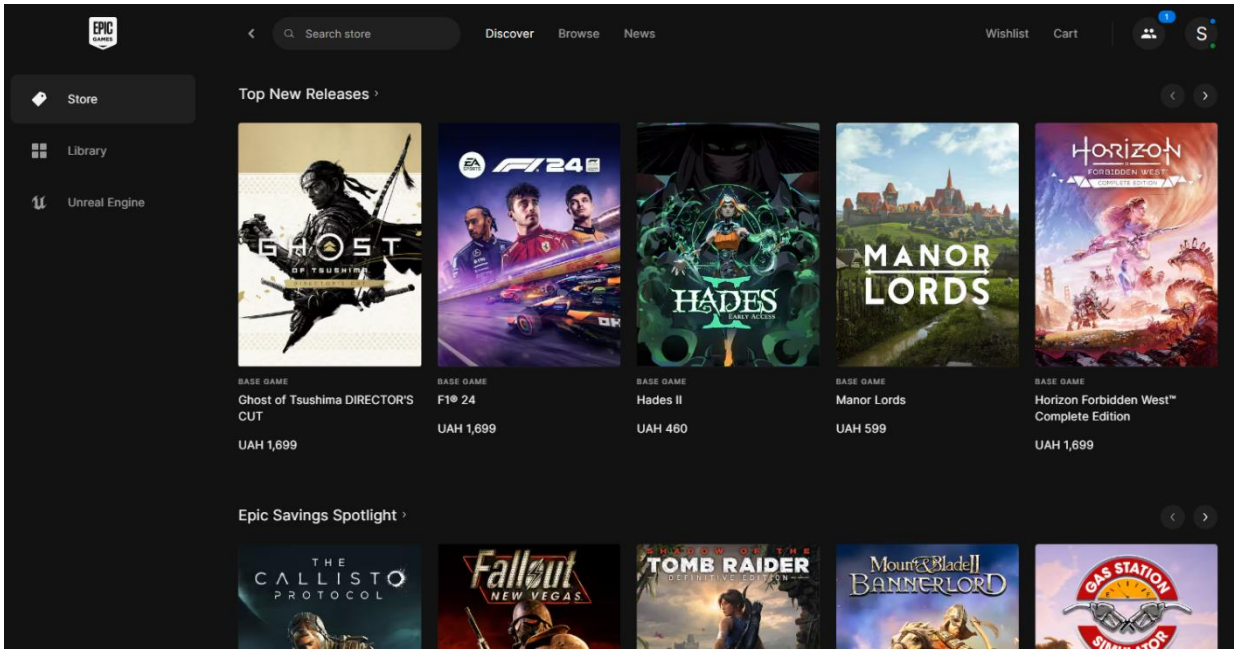

Рис. 1.7. Скріншот головної сторінки Epic Games

Така доступність призвела до поширення ігор, від блокбастерів AAA до нішевих інді-ігор. Це створило яскравий і різноманітний ігровий ландшафт, де гравці мають широкий вибір. Крім того, конкуренція між платформами спонукає до їх розвитку та винайдення нових функцій. Такі назви, як «Hollow Knight» і «Undertale», продемонстрували креативність і різноманітність інді-розробників.

Рис. 1.8. Інді-гра «Undertale»

Наразі ігрова індустрія продовжує розвиватися та використовувати нові технології. У наші дні віртуальна реальність і доповнена реальність, стають все більш поширеними серед людей. Сервіси хмарних ігор, як Google Stadia та Xbox Cloud Gaming, мають на меті зробити високоякісні ігри доступнішими. З кожним роком з'являються нові технології з можливостями інтеграції штучного інтелекту, процедурної генерації і т.д. Оскільки ігри все більше інтегруються в основну культуру, їхній вплив на розваги, освіту та соціальну взаємодію продовжує зростати.

1.1.2. Аналіз та порівняння ігрових рушіїв

Ігровий рушій є фундаментальним програмним забезпеченням, яке забезпечує базову функціональність для розробки ігор. Вони надають

набір інструментів і функцій, які оптимізують різні аспекти розробки ігор, дозволяючи розробникам зосереджуватися на творчих і дизайнерських елементах гри, а не створювати все з нуля.

Деякі розробники створюють власні ігрові рушії для основи своїх ігор. Наприклад, польська ігрова студія CD Projekt Red використовує власний рушій REDengine. На цьому рушії були побудовані такі ігри, як The Witcher та Cyberpunk 2077. Використання власного ігрового рушія дає змогу створювати унікальні ігрові механіки і більш гнучкі ігрові світи, які будуть влаштовувати розробників. На жаль такий підхід до створення ігор має свої недоліки. Ігровий рушій складно оптимізувати, та налаштувати його стабільну роботу, через що у грі може бути багато багів та помилок.

Рис. 1.9. Скріншот гри «The Witcher 3», що розроблена на базі рушія REDengine

Найбільш відомі такі рушії як CryEngine, Unreal Engine, та Unity. Розробники обирають рушій, на основі вимог до своєї гри [6]. Наприклад на основі рушію Unreal Engine створюють ігри з більш реалістичною графікою.

Рис. 1.10. Скріншот з фотореалістичної гри «Unrecord», створена на рушії Unreal Engine 5.

Unreal Engine – один з найпопулярніших ігрових рушіїв для розробки AAA-ігор. Unreal Engine використовує C++, так що при належному знанні цієї мови, можно розробити гру на цьому рушію. В Unreal вбудовано практично все, що може знадобиться розробнику, включаючи 3D-моделювання та роботу з ландшафтом. Через таке різноманіття інструментів, освоїти Unreal Engine складніше, проте можно створювати по-справжньому вражаючі ігри [12].

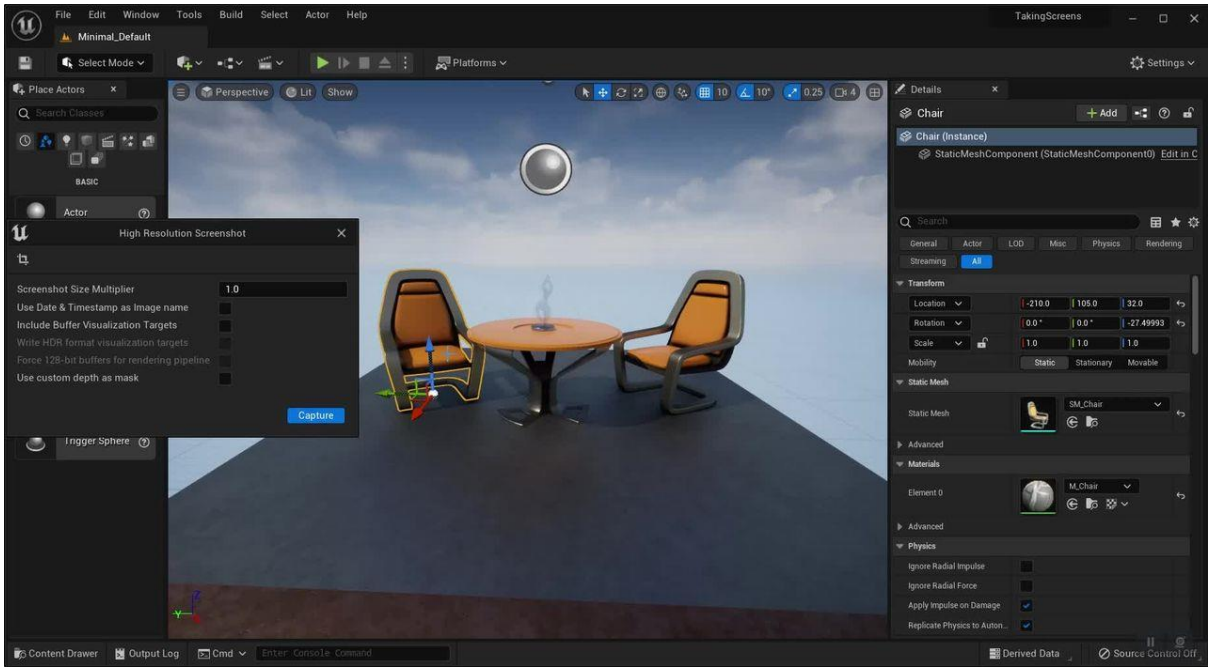

Рис. 1.11. Інтерфейс ігрового рушія Unreal Engine

Unity - це один з найпопулярніших ігрових рушіїв, який часто використовується для розробки як 2D, так і 3D ігор. Він був створений компанією Unity Technologies і вперше випущений у 2005 році. Unity підтримує різні платформи, включаючи Windows, macOS, Android, iOS, та інші, що є перевагою цього рушія.

Unity часто використовуються для розробки інді ігор. Unity пропонує інді-розробникам потужний набір інструментів та ресурсів, які дозволяють створювати високоякісні ігри з мінімальними витратами та зусиллями. Це робить Unity ідеальним інструментом розробки ігор для новачків, а також для досвідчених розробників, які шукають гнучкість та продуктивність.

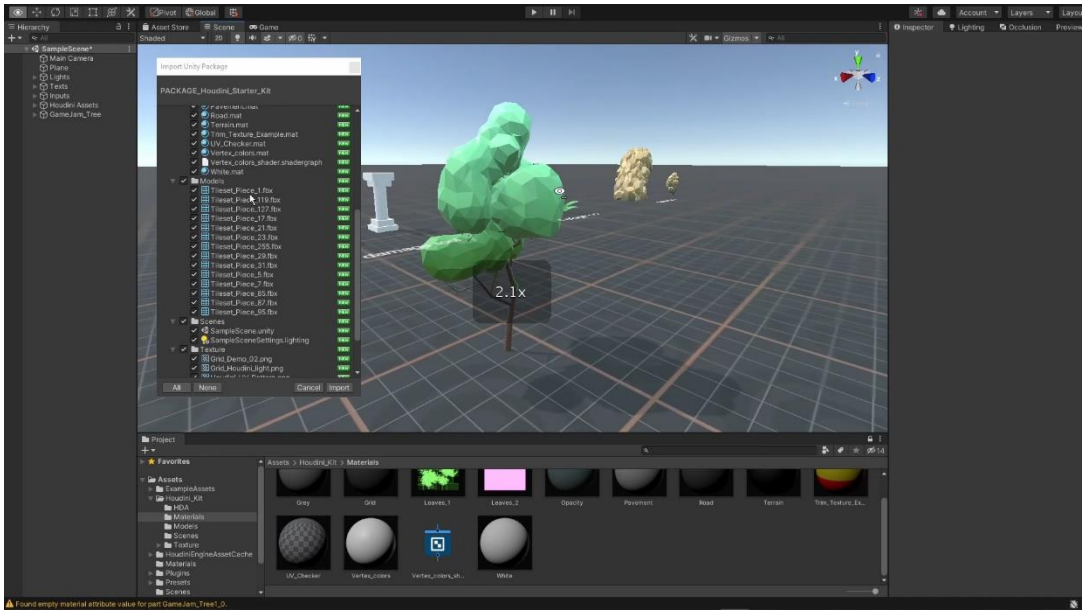

Рис. 1.12. Інтерфейс ігрового рушія Unity

Unity надає зручні інструменти для роботи з асетами, що робить процес розробки зручним і ефективним. Імпорт і експорт ассетів в Unity відбувається практично автоматично, завдяки доброзичливим інтерфейсам і підтримці великої кількості форматів. Файли зображень, моделі, анімації, звуки і багато інших типів даних можуть бути легко додані до проєкту. При цьому оптимізація для різних платформ відіграє важливу роль, оскільки кожна платформа має свої вимоги до розміру і якості ассетів. Unity надає інструменти для оптимізації текстур, моделей та інших ресурсів, щоб розроблений застосунок працював максимально ефективно [3].

Unity чудово взаємодіє з різними сторонніми інструментами і програмами. Наприклад, під час інтеграції з Photoshop, Blender та іншими програмами, розробники можуть легко експортувати графічні та 3D ресурси прямо в Unity без втрати якості або додаткових конвертацій. Це прискорює процес розробки і забезпечує безшовну співпрацю між дизайнерами і розробниками. З іншого боку, Unity також пропонує

використання хмарних сервісів для спільної роботи. Такі сервіси, як Unity Collaborate, дають змогу командам працювати над одним проєктом у реальному часі, забезпечуючи синхронізацію змін і зручне управління версіями [2].

1.1.3. Аналіз ігор розроблених на Unity

Hollow Knight — це 2D метроідванія, у якій є багато можливостей для відкриття та дослідження. Гра випущена у 2017 році, її хвалять за прекрасний художній стиль, складний геймплей і насичений захоплюючий світ. Мапа гри має багато секретів та розгалужених шляхів [1]. Гра має вражаючу мапу з великою кількістю секретів, які потрібно знайти. Гравць також буде битись з великою кількістю різноманітних та складних ворогів.

Ключові особливості ігрового процесу:

- Гравці подорожують світом гри, знаходячи предмети колекціонування та відкриваючи нові території. Карта є нелінійною, що заохочує гравців досліджувати та відвідувати території з новими здібностями.

- Бій швидкий і заснований на навичках, які гравець отримує протягом гри. Лицар володіє цвяхом (мечем), щоб атакувати ворогів.

- Гравці можуть використовувати артефакти, щоб проходити рівні та перемогати ворогів. Таким чином кожен гравець може обрати свій власний стиль проходження, та створити свій "білд" [4].

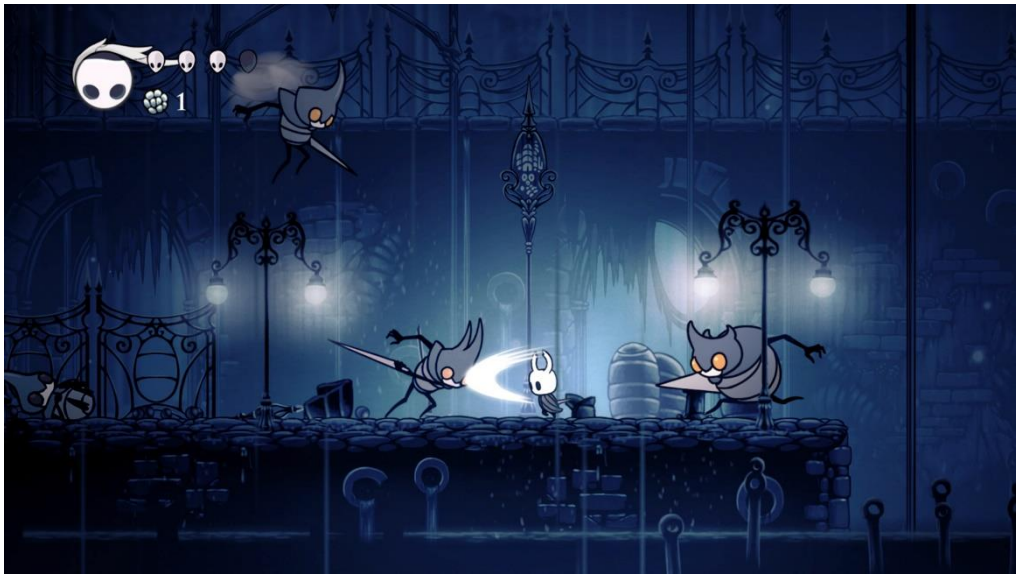

Рис. 1.13. Скріншот з гри «Hollow Knight»

Cuphead - комп'ютерна гра, яка поєднує в собі жанри «біжи та стріляй» із напруженими битвами з босами. Її головною особливістю є візуальна складова. Працюючи над стилістикою Cuphead, канадські інді-розробники надихалися роботами аніматорів 1930-х років [5]. Cuphead анімована повністю на папері як справжній мультфільм тих часів. Однак Cuphead сподобалася гравцям не лише завдяки мальованій анімації. За мультяшною естетикою ховається шалений хардкор. Невелика кількість життів та різноманітність випробувань робить гру дуже складною.

Геймплей гри складається з:

- Складних та багатофазних битв з босами. Кожен бос має унікальні шаблони, атаки та трансформації, які гравці повинні вивчити та адаптуватися до них.

- Рівнів «Біжи та стріляй»: окрім битв із босами, де гравці мають долати перешкоди, перемагати ворогів і збирати монети.

- Гравці можуть використовувати різну зброю, кожна з якої має унікальні властивості та стилі атаки. Пошук правильної комбінації зброї є ключовим для подолання конкретних викликів і босів.

- Гра підтримує локальну кооперативну гру, дозволяючи двом гравцям об’єднатися для співпраці та спільних завдань.

Рис. 1.14. Скріншот з гри «Cuphead»

Ori and the Blind Forest — це двовимірна пригодницька гра, яка поєднує дослідницьку роботу з тяжкими випробуваннями. Гравець має відкривати нові здібності, які відкривають раніше недоступні території.

Ori має наступні геймплейні особливості:

- Головний персонаж рухається з неймовірною спритністю, чіпляючись за стіни та виконуючи повітряні маневри.

- Приховані секрети: карта гри наповнена колекційними предметами та секретами, які покращують здібності персонажа.
- Взаємодія з гравцем: неймовірний світ гри розкриває свою історію через навколишнє середовище та вражаючі візуальні ефекти, та всілякі деталі на фоні.

Рис. 1.15. Скріншот з гри «Ori and the Blind Forest»

Метою цього дослідження є розробка гри в жанрі 2D платформеру, запозичення найкращих механік та вдосконалення ігрового досвіду гравця:

- Підтримка інтересу гравців: Завдяки багатьом факторам загрози гравцю, та гарному балансу складності, гра завжди буде тримати гравця на по готові і не набридне.

- Приваблива атмосфера та дизайн: Важливо зробити гру візуально привабливою для гравця. Завдяки візуальному та звуковому оформленню можна створити атмосферу, що запам’ятовується та виділяється на фоні інших проєктів.
- Унікальність геймплею: Розроблений ігровий додаток має виділятись на фоні усіх існуючих платформерів. Планується поєднати особливості таких ігор, як Hollow Knight і Cuphead. А саме повністю лінійне проходження та відносно відкритий світ. Я планую поєднати лінійність рівнів, та вибір шляху їх проходження. Гравцю потрібно обрати оптимальний та максимально ефективний варіант проходження рівня.

1.2 Призначення розробки та галузь застосування

Основне призначення розробки 2D платформеру полягає у створенні захоплюючого та розважального досвіду для гравців. Метою є занурення користувача в цікавий ігровий світ, де він може взаємодіяти з оточенням, вирішувати головоломки, долати перешкоди та насолоджуватися геймплеєм. В процесі гри користувачі отримують можливість:

- Розважатися: Головне завдання гри - розважити гравця. Ігровий процес має бути цікавим та захоплюючим, забезпечуючи гравцям задоволення від проведеного часу.
- Зануритися в ігровий світ: Створення атмосферного ігрового середовища, яке може включати унікальний візуальний стиль, та привабливих персонажів.

- Розвивати навички: Платформери часто вимагають від гравців точності, координації та швидкої реакції, що сприяє розвитку цих навичок.

2D платформери, як жанр комп'ютерних ігор, знаходять широке застосування в різних галузях, виходячи за межі суто розважальної сфери:

- Освітні ігри: Платформери можуть використовуватися для створення навчальних ігор, які допомагають дітям та дорослим засвоювати нові знання через інтерактивний досвід.
- Когнітивна терапія: Ігри можуть використовуватися в когнітивній терапії для пацієнтів з різними розладами, допомагаючи їм покращувати пам'ять, увагу та інші когнітивні функції.
- Соціальні меседжі: Ігри можуть бути інструментом для поширення важливих соціальних повідомлень та підвищення обізнаності про певні проблеми.
- Промо-ігри: Компанії можуть використовувати платформери як частину своїх маркетингових кампаній, створюючи ігри для просування продуктів або брендів.

1.3. Підстава для розробки

Розробка ігрового застосунку на базі рушія Unity та мови програмування C# є частиною моєї дипломної роботи, яку я виконую на завершальному етапі навчання в університеті. Дипломна робота є важливим елементом освітнього процесу, що дозволяє студентам

продемонструвати здобуті знання, навички та здатність до самостійної наукової або практичної діяльності.

Основною підставою для розробки цього проекту є необхідність виконання вимог навчального плану та досягнення поставлених освітніх цілей. Дипломна робота надає можливість узагальнити та застосувати теоретичні знання, отримані під час навчання, на практиці, вирішуючи конкретні завдання та проблеми.

1.4. Постановка завдання

Основним завданням цієї дипломної роботи є розробка 2D платформеру на базі рушія Unity з використанням мови програмування C#. Цей проект має на меті створення інтерактивного, захоплюючого та візуально привабливого ігрового застосунку, який забезпечить гравцям цікавий та розважальний досвід.

Першим кроком у виконанні завдання є розробка концепції гри, яка включає визначення жанру, основних механік геймплею та візуального стилю.

Після завершення етапу проектування розпочинається безпосередня розробка гри на базі рушія Unity. Це включає написання коду на мові C# для реалізації ігрової логіки, анімації, фізики, взаємодії з користувачем та інших аспектів гри.

Виконання цього завдання вимагає всебічного підходу, поєднання теоретичних знань та практичних навичок, а також ефективного управління часом та ресурсами.

1.5. Вимоги до програми або програмного виробу

1.5.1. Вимоги до функціональних характеристик

Для реалізації поставленого завдання потрібно реалізувати:

- Рух персонажа: Гравець повинен мати можливість керувати персонажем за допомогою клавіатури або контролера, забезпечуючи плавні рухи, стрибки та інші дії.
- Фізика: Реалізація фізичних властивостей, таких як гравітація, інерція та зіткнення, що додає реалістичності геймплею.
- Візуальний стиль: Високоякісні 2D графічні елементи, які створюють атмосферу гри.
- Фоновий саундтрек: Атмосферна музика, яка підтримує настрій гри.
- Аудіо ефекти: Звуки є важливою складовою більшості ігор, це потрібно для того, щоб інформувати гравця про події у грі.
- Анімації: Анімації необхідні для підтримки зацікавленості гравців, покращення візуального стилю та загального вигляду гри. Також це важливо для інформування гравців про атаки ворогів.

- Зручний інтерфейс: Інтуїтивно зрозумілий інтерфейс користувача, що включає меню, показники здоров'я, очок та інші елементи, необхідні для геймплею.
- Налаштування гри: Можливість налаштування основних параметрів гри, таких як гучність звуку, та інші опції.

1.5.2. Вимоги до інформаційної безпеки

Основні вимоги до інформаційної безпеки включають:

- Конфіденційність: Мінімізація збору особистої інформації та забезпечення її конфіденційності відповідно до законодавства про захист даних.
- Регулярні оновлення: Впровадження регулярних оновлень та патчів для усунення вразливостей та поліпшення безпеки гри.

1.5.3. Вимоги до складу та параметрів технічних засобів

Мінімальні вимоги для запуску гри:

- персональний комп’ютер
- миша
- клавіатура
- процесор Intel Core i3
- відеокарта GeForce GTX 560
- 4 гб операційної пам’яті
- 1 гб вільного місця на диску

1.5.4. Вимоги до інформаційної та програмної сумісності

Гра буде сумісна з Windows 8, 10 та 11.

Гра буде розроблена за допомогою мови програмування C#.

РОЗДІЛ 2

ПРОЕКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОГО ПРОДУКТУ

2.1 Функціональне призначення програми

Метою створення даної 2D гри-платформера на базі рушію Unity є розробка інтерактивного розважального продукту, що поєднує в собі елементи класичних платформерів та цікаві і складні рівні. Гра призначена для широкої аудиторії гравців, які цінують динамічні та складні ігрові процеси, які потребують від гравця швидкої реакції, точності та стратегічного мислення.

Основним функціональним призначенням цієї гри є забезпечення гравця захоплюючим ігровим досвідом, який складається з наступних ключових елементів:

- **Проходження складних рівнів**: Гра складається з численних рівнів, кожен з яких має унікальний дизайн та підвищену складність. Гравець повинен подолати різноманітні перешкоди, такі як шипи, пили та вогняні капкани. Ці елементи створюють складне ігрове середовище, що вимагає від гравця максимальної зосередженості та точності у діях.
- **Взаємодія з ворогами**: На кожному рівні гравець зустрічає різних ворогів, яких необхідно знищити або обійти. Це додає грі елементи екшену та підвищує інтенсивність ігрового процесу. Кожен тип ворога має свої особливості та поведінкові патерни, що вимагає від гравця розробки різних стратегій для їх подолання.

- **Візуальний та звуковий дизайн**: Гра має привабливий візуальний стиль, який створює атмосферу середньовічних пригод. Супровідний звуковий дизайн підкреслює напругу та динаміку ігрового процесу [8].

Дана 2D гра-платформер є захоплюючим продуктом, який пропонує гравцеві цікавий ігровий досвід. Вона підходить для широкого кола гравців і може використовуватися як для розваги, так і для розвитку певних навичок та якостей.

2.2 Опис застосованих математичних методів

При розробці 2D гри-платформера на базі рушію Unity використовується низка математичних методів, які забезпечують коректне функціонування ігрових механік, фізики, та анімацій. Застосування цих методів дозволяє створити цікавий ігровий процес, що відповідає вимогам до сучасних ігор:

Векторна математика: Вектори є основою для багатьох аспектів 2D гри. Вони використовуються для визначення положення об'єктів на ігровому полі, напрямку їх руху та швидкості. Наприклад, для обчислення траєкторії стрибка персонажа враховується вектор початкової швидкості та вплив гравітації.

Колізійні алгоритми: Важливим аспектом є виявлення зіткнень між об'єктами, такими як персонаж, вороги та пастки. Для цього використовуються алгоритми колізії, які базуються на геометричних методах. Зокрема, для простих об'єктів можуть використовуватися перевірки зіткнень за допомогою прямокутних або круглих

колізійних областей. Більш складні форми можуть вимагати застосування алгоритмів виявлення колізій для багатокутників.

Інтерполяція та анімація: Для плавного переходу між різними положеннями та станами об'єктів використовується інтерполяція. Наприклад, лінійна інтерполяція допомагає плавно змінювати координати об'єкта від однієї точки до іншої за певний час. Цей метод застосовується для анімації руху персонажа, зміни його стану, а також для створення ефектів, таких як поступове зникнення або з'явлення об'єктів.

Загалом, застосування цих математичних методів дозволяє створити фізику та точні анімації. Використання векторної математики, фізичних моделей, колізійних алгоритмів, інтерполяції є невід'ємною частиною розробки 2D гри-платформера на рушію Unity.

2.3 Опис використаної архітектури та шаблонів проектування

Архітектура гри базується на принципах компонентно-орієнтованого програмування, що є основою для рушію Unity. Основний елемент архітектури Unity — це GameObject, до якого можуть бути прикріплені різні компоненти, що надають об'єкту певну функціональність.

Такий підхід дозволяє легко розширювати можливості об'єктів шляхом додавання або видалення компонентів.

Основні компоненти архітектури:

- GameObjects та компоненти: Всі об'єкти в грі представлені як GameObjects. До них прикріплені компоненти, які визначають їхню поведінку, зовнішній вигляд та інші властивості.

Наприклад, для персонажа гравця можуть бути додані компоненти для руху, стрибків, взаємодії з оточенням та анімації.

- Сцени: Гра складається з кількох сцен, кожна з яких відповідає певному рівню гри. Сцени містять GameObjects та їхні компоненти, а також налаштування освітлення, камери та інші параметри.
- Скрипти: Логіка гри реалізована за допомогою скриптів на мові C#. Скрипти прикріплюються до GameObjects як компоненти, що дозволяє легко організувати та керувати поведінкою об'єктів.

У процесі розробки гри використовуються різні шаблони проектування, які допомагають покращити структуру та підтримку коду:

Шаблон Observer та State

У цій роботі присутні елементи шаблону Observer та State. Аніматор (anim) можна розглядати як спостерігача, який реагує на зміни стану об'єкта, встановлені за допомогою методів SetBool та SetTrigger. Це дозволяє легко керувати різними станами персонажа, такими як біг, стрибок та перебування на землі.

Інкапсуляція

Інкапсуляція — це один з ключових принципів об'єктно-орієнтованого програмування (ООП), який передбачає обмеження доступу до певних частин об'єкта, зокрема до його внутрішніх даних, і надання доступу до цих даних лише через визначені методи. Інкапсуляція дозволяє захистити внутрішній стан об'єкта від небажаного втручання та

модифікації ззовні. Інкапсуляція сприяє розробці модульного коду, де кожен клас виконує окрему роль і має чітко визначену відповідальність. Зміни у внутрішній реалізації об'єкта не впливають на код, який використовує цей об'єкт, що полегшує підтримку і розширення коду.

У моїй грі є клас PlayerMovement, який контролює рухи персонажа. Інкапсуляція допомагає приховати внутрішні деталі реалізації, такі як змінні для управління станами анімацій, і надає лише необхідні методи для взаємодії.

2.4 Опис використаних технологій та мов програмування

Основні технології та мови програмування, які застосовуються в цьому проекті, включають Unity, C#, а також інструменти та бібліотеки, інтегровані в Unity для роботи з анімаціями, фізикою і управлінням грою. Unity є одним з найпопулярніших ігрових рушіїв, що надає широкий набір інструментів для створення як 2D, так і 3D ігор. Завдяки Unity можна швидко розробляти прототипи та створювати повноцінні ігрові проекти. Основні можливості Unity, які використовуються у моєму проекті, включають:

- Редактор сцен: З його допомогою можна розміщувати об'єкти, налаштовувати рівні, додавати колізії та тригери для взаємодії об'єктів.
- Фізичний рушій: Unity використовує фізичний рушій Box2D для 2D ігор, який забезпечує реалістичні взаємодії між об'єктами, такі як гравітація, зіткнення та відскоки.

- Анімаційний інструментарій: Unity надає можливості для створення та налаштування анімацій за допомогою Animator та анімаційних кліпів, що дозволяє оживити персонажів та ворогів гри.
- Пакетні менеджери та асети: Unity Asset Store та вбудовані пакетні менеджери дозволяють легко інтегрувати додаткові інструменти та ресурси, такі як шрифти, звуки, графіка та скрипти, що прискорює процес розробки.

C# є основною мовою програмування для створення ігрової логіки в Unity. Завдяки своїй строго типізованій природі та підтримці об'єктно-орієнтованого програмування, C# дозволяє розробляти ефективний, організований та легкий для підтримки код.

Unity надає потужні інструменти для тестування та налагодження, включаючи вбудований дебаггер, що дозволяє відслідковувати значення змінних у реальному часі, а також Unity Test Runner для автоматизованого тестування компонентів гри. Це дозволяє швидко виявляти та виправляти помилки, забезпечуючи високу якість кінцевого продукту.

Використання Unity та мови програмування C# у розробці 2D платформерної гри забезпечує широкий набір інструментів та можливостей для створення складних, цікавих та інтерактивних ігрових світів. Завдяки інкапсуляції, шаблонам проектування та потужним інструментам для тестування та налагодження, можна створювати високоякісні ігрові проекти, які легко підтримувати та розширювати.

2.5 Опис структури програми та алгоритмів її функціонування

Цикл гри виглядає наступним чином:

Рис. 2.1. Цикл роботи додатка

Структура та правильна організація папок зі скриптами, спрайтами та звуками в Unity, є важливою для оптимізації процесу розробки та зручної навігації. Таким чином папка Scripts містить усі написані скрипти, папка Sprites містить усі зображення, що використовувались для створення графіки та візуалу. У папці Animation знаходяться усі створені анімації, за допомогою аніматора Unity [7].

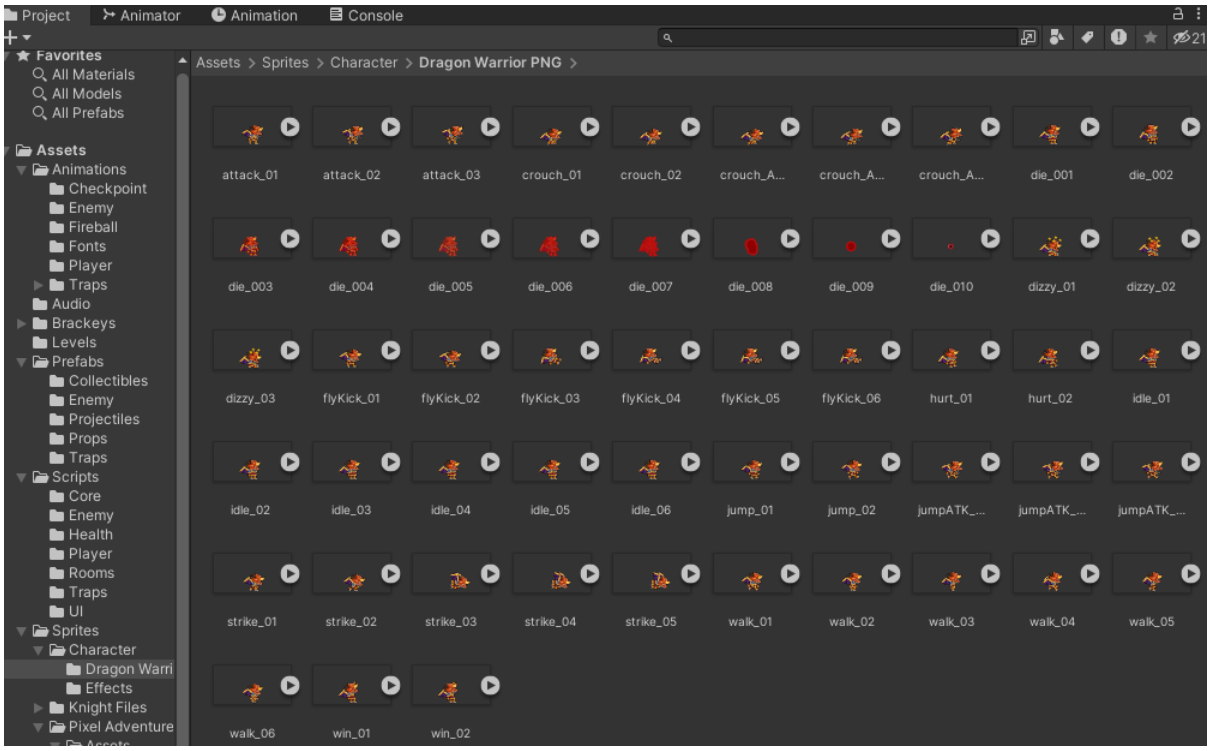

Рис. 2.2. Структура папок з файлами в Unity

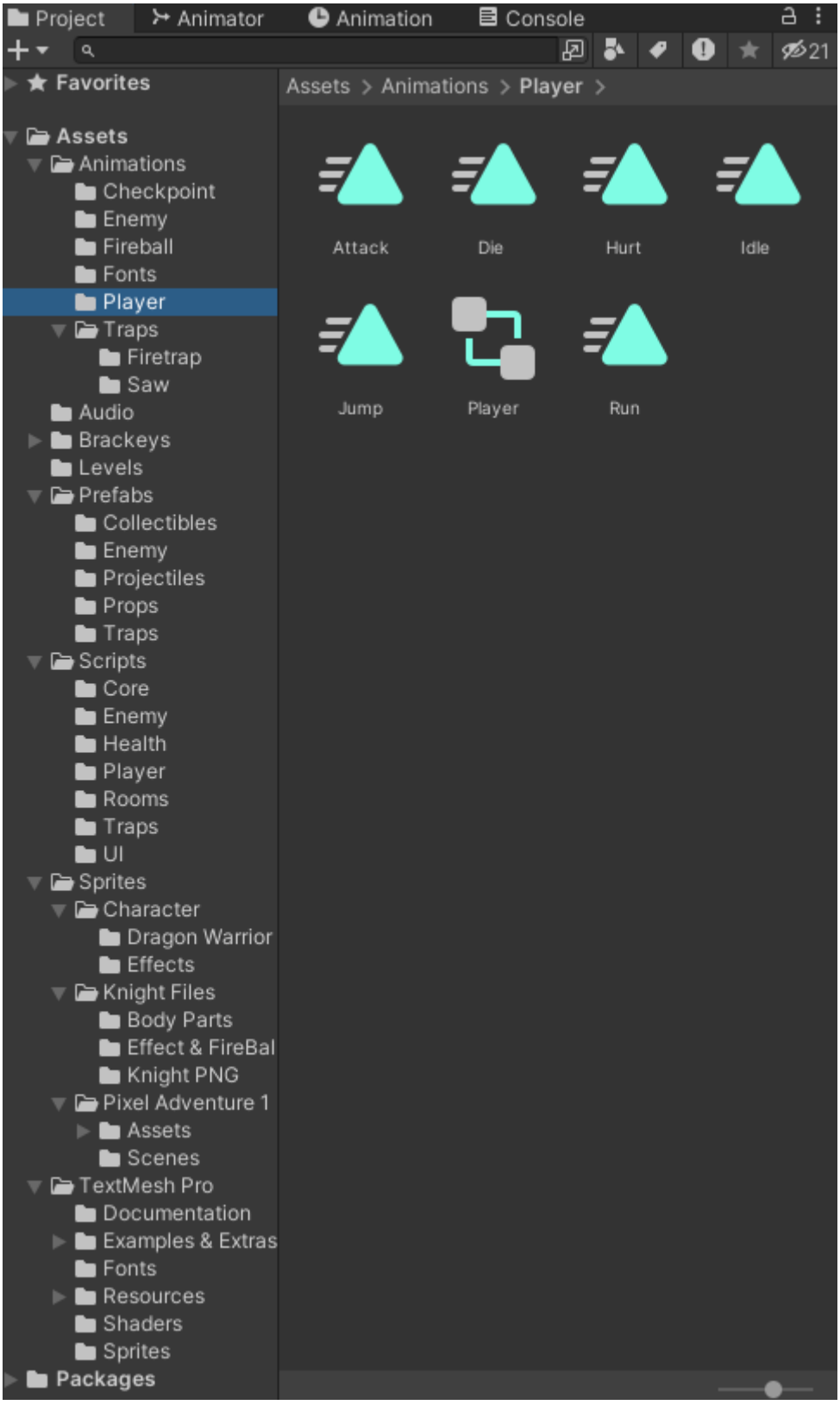

Рис. 2.3. Набір анімацій головного персонажа

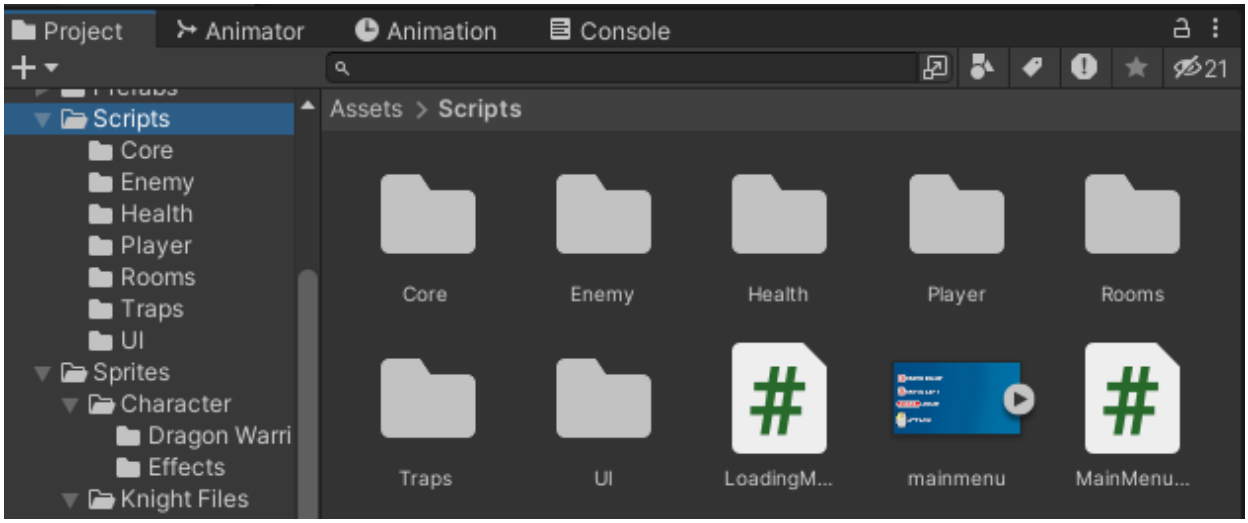

Рис. 2.4. Пака що містить усі скрипти

Розроблений ігровий додаток складається з декількох сцен, що також важливо для структури програми.

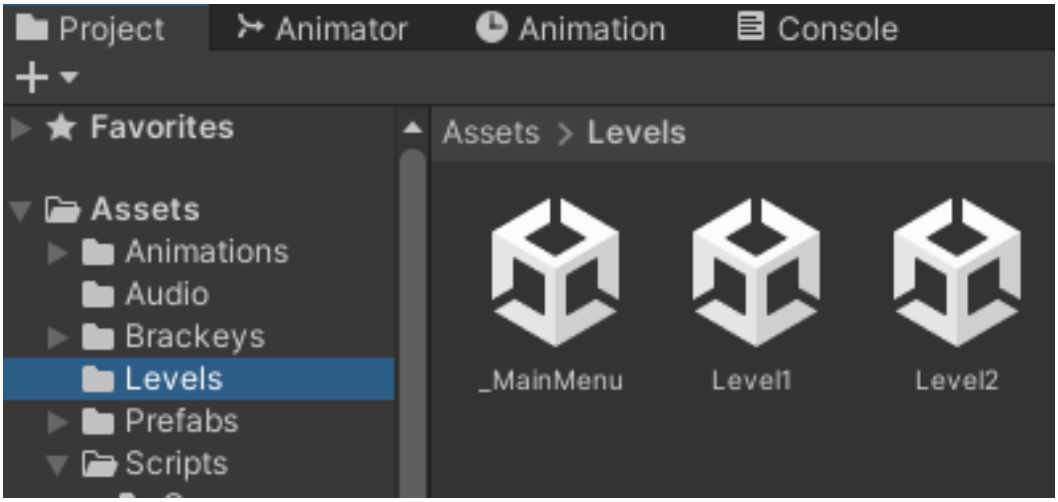

Рис. 2.5. Папка що містить усі сцени

В Unity ієрархія ігрових об’єктів має вирішальне значення для організації та керування структурою гри, дозволяючи розробникам встановлювати чіткі взаємозв’язки між об’єктами. Система ієрархії дозволяє створити батьківські або дочірні зв’язки, які визначають, як об’єкти взаємодіють і поводяться. Батьківські об’єкти служать

контейнерами або контролерами для дочірніх об’єктів, тобто будь-яке перетворення (положення, обертання, масштаб), застосоване до батьківського об’єкта, автоматично поширюватиметься на його дочірні об’єкти. Ієрархічне розташування має важливе значення для структурування складних сцен, спрощення керування об’єктами та оптимізації продуктивності [14].

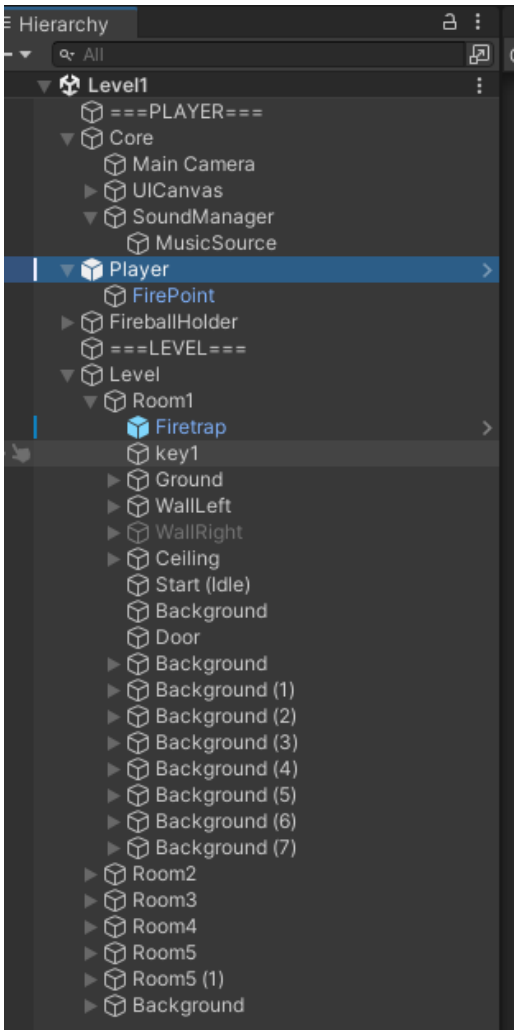

Рис. 2.6. Ієрархія ігрових об’єктів

Ще один корисний інструмент для організації та оптимізації структури проєкту - функція Prefab. Розробники використовують Prefab

для створення багаторазових шаблонів для ігрових об’єктів, спрощуючи процес розробки. Префаби дозволяють модифікувати ігровий об’єкт з усіма його компонентами, властивостями та дочірніми об’єктами, які потім можуть бути створені кілька разів протягом гри. Усі зміни внесені до префабу, автоматично поширюються на всі його екземпляри, спрощуючи оновлення та модифікації. Використання префабів важливе для підтримки організованості та ефективності розробки.

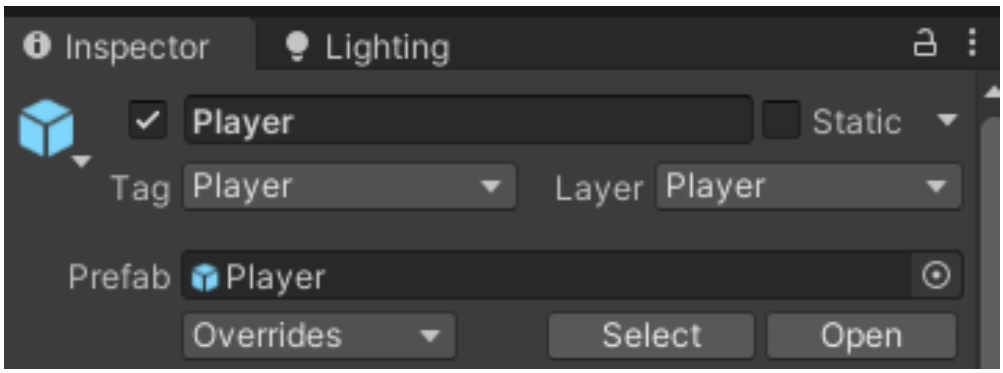

Рис. 2.7. Ігровий об’єкт у формі Prefab

2.6 Обґрунтування та організація вхідних та вихідних даних програми

Вхідні дані у моєму проєкті це натиснуті користувачем клавіші на клавіатурі і натиснута кнопка миші. Також вхідні дані забезпечуються за допомогою інтерактивних елементів в грі, з якими гравець буде взаємодіяти. Периферійні пристрої виконують наступні функції:

- клавіатура відповідає за рух персонажа. Кнопка `A` - рух ліворуч, `D`- рух праворуч, `SPACE` - стрибок
- використання мишки необхідне для здійснення атаки. Для цього потрібно натиснути ліву кнопку мишки.

Вихідними даними є візуальна та аудіо інформація, яку гравець отримує під час проходження гри:

- анімації персонажів та різних ігрових об’єктів, які активуються при натисканні на кнопки клавіатури або мишки
- звукові ефекти, що спрацьовують при отриманні пошкоджень, стрибках, смерті персонажа та активації інших тригерів [9].

2.7 Опис роботи програмного продукту

2.7.1 Використані технічні засоби

Програма розроблялась на персональному комп’ютері з такими характеристиками:

- ЦП AMD Ryzen 5 2400G 3.6GHz;
- NVIDIA Geforce GTX 1660 Super 6Gb;
- ОЗП 32Гб;
- SSD Patriot 1Tb;
- дисплей FullHD (1920x1080) 170Гц;
- механічна клавіатура
- миша
- Windows 10 64 разрядна

2.7.2 Використані програмні засоби

Для розробки гри застосовувались наступні програмні засоби:

- Unity editor;
- Visual Studio;
- Adobe After Effects;

Редактор Unity - редактор призначений для створення інтерактивних 2D і 3D додатків та керування ними, починаючи від ігор і закінчуючи симуляціями та віртуальною реальністю. Я використовував цей редактор для розробки та побудови ігрових рівнів і керування ними.

Visual Studio - це середовище розробки, для оптимізації розробки різних типів програмних додатків. Він підтримує широкий спектр мов програмування, включаючи C#, C++, Python, JavaScript тощо, що робить його універсальним інструментом для розробників у різних галузях. Я використовував цей застосунок для написання та редагування коду мовою C#.

Adobe After Effects - програма для цифрових візуальних ефектів та анімацій. Я використав цю програму для створення дизайну банерів, з кнопками керування, які гравець побачить на початку гри.

2.7.3 Виклик та завантаження програми

Для запуску програми потрібно виконати файл KnightSlayer.exe, який знаходиться в папці з усіма іншими програмними файлами. Програму можна запустити тільки на пристроях з ОС Windows.

2.7.4 Опис інтерфейсу користувача

У головному меню користувач може побачити назву гри – Knight Slayer. Також користувач має можливість одразу ознайомитись з керуванням у грі.

Play: ця опція запустить ігровий рівень, та користувач зможе почати гру.

Quit: ця опція здійснить вихід з ігрового додатка.

Рис. 2.8. Головне меню ігрового додатка

На початку гри, користувач може помітити такий елемент інтерфейсу, як три червоних серця, що знаходяться у лівому верхньому куту екрану. Це показник здоров’я персонажа, який може змінюватись залежно від того, отримує персонаж пошкодження або відновлює здоров'я. У випадку втрати усіх трьох сердець, гра закінчиться. Також користувач має можливість ще раз ознайомитись з кнопками керування персонажем.

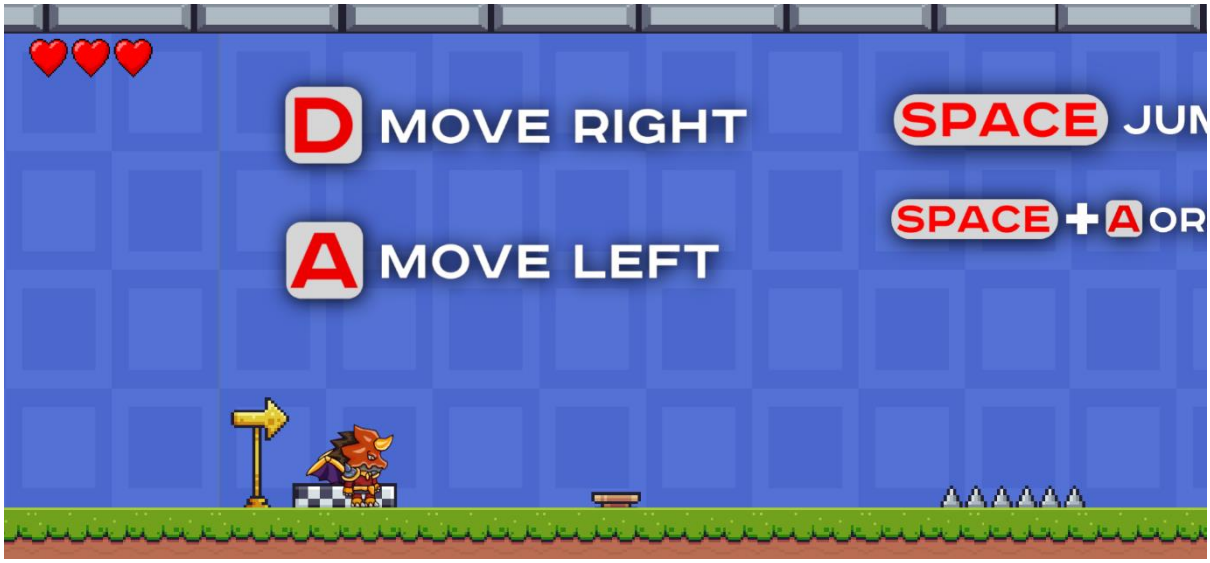

Рис. 2.9. Скріншот з початку гри

Вогняна пастка. Якщо гравець наступить на вогняну пастку, то через 0,3 секунди вона активується і випустить вогонь. У тому разі, якщо гравець не встигне відійти і залишиться на тому ж місці, йому буде нанесене пошкодження вогнем і відніме одне очко здоров’я.

Рис. 2.10. Вогняна пастка

Шипи. Гострі шипи можуть бути розташовані на землі або платформі. Якщо гравець наступить на шипи, то у нього віднімуть одне очко здоров’я.

Рис. 2.11. Гострі шипи

Пила. Одна з пасток в розробленій грі – пила, що швидко обертається та наносить пошкодження персонажу, віднімаючи одне очко здоров’я. Також пили рухаються по горизонталі з різною швидкістю та відстанню.

Рис. 2.12. Пила

Лицарі-мечники. Лицарі одні з найсильніших супротивників в розробленому додатку. Деякі лицарі патрулюють заданий периметр, поки інші стоять не рухаючись, в очікувані гравця. Лицарі можуть атакувати гравця своїм мечем. Така атака відніме 2 очка здоров’я, що становить велику загрозу для гравця. Гравець може уникнути даної атаки, якщо діяти акуратно та правильно вибрати час, щоб самому атакувати ворога. Ще один зі способів перемогти лицаря – перестрибнути його. Навіть під час ворожої атаки, гравець не отримає пошкодження у повітрі, та зможе успішно переміститись за спину ворога. Проте треба також враховувати, що просто при зіткненні з моделькою лицаря, у гравця віднімуть одне очко здоров’я. Це зроблено для того, щоб гравець на мав змогу просто пробігати цей тип ворогів.

Рис. 2.13. Лицар атакує гравця

Чекпоінт. У розробленій грі присутні місця, в яких гравець може закріпити свій прогрес. Дійшовши до такого місця, що має характерний вигляд, гравець активує прапорець. У випадку смерті, це дозволить починати гру з цього прапорцю, а не проходити увесь

рівень з самого початку. Безумовно це є полегшенням гри, проте важливо для правильного балансу, враховуючи складність платформенгу в розробленій грі.

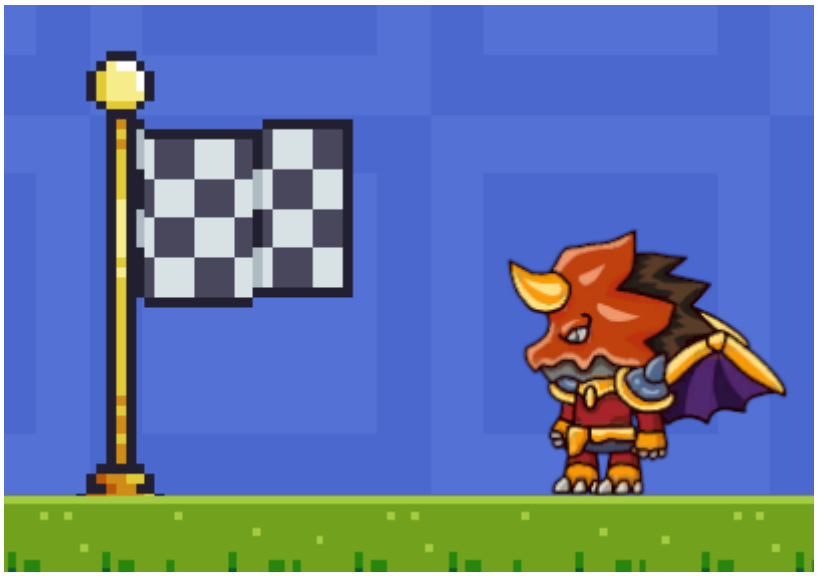

Рис. 2.14. Місце збереження прогресу гравця

Шматок м’яса. Гравець має можливість відновити частину здоров’я персонажу. Для цього потрібно знайти шматок м’яса та з’їсти його. Це додасть персонажу одне очко здоров’я.

Рис. 2.15. М’ясо для відновлення здоров’я

Гравець може викликати меню паузи, натиснувши кнопку Escape.

Resume: ця опція продовжить гру, та вийде з меню паузи.

Sounds: ця опція регулює гучність звукових ефектів.

Music: ця опція регулює гучність фонової музики у грі.

Go to menu: ця опція дозволяє користувачу перейти до головного меню.

Exit: ця опція здійснить вихід з гри.

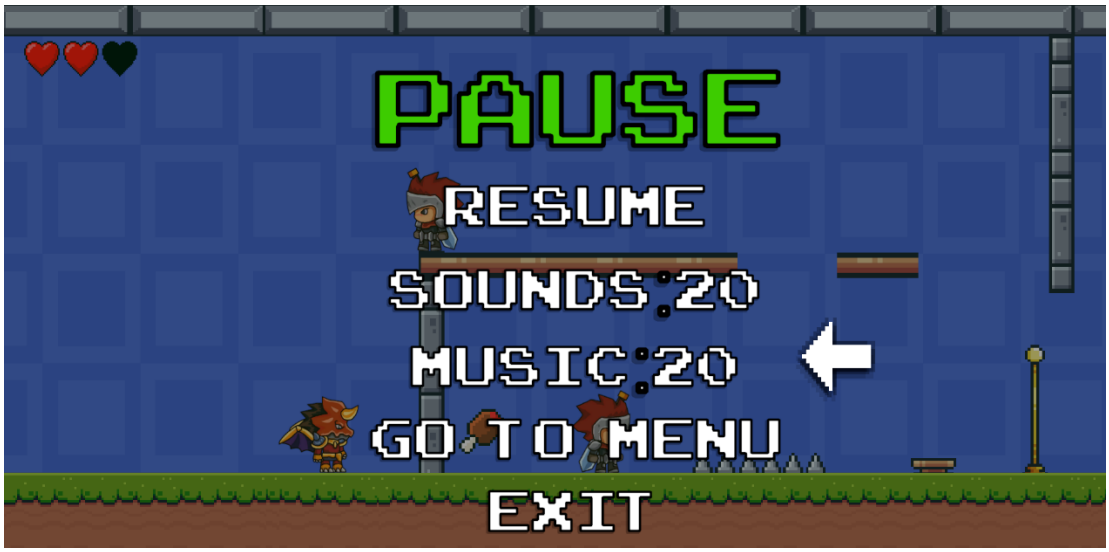

Рис. 2.16. Меню паузи гри

У випадку смерті персонажу, гравець побачить це меню.

Restart: ця опція дозволяє почати гру зі старту.

Main menu: ця опція дозволяє користувачу перейти до головного меню.

Exit: ця опція здійснить вихід з гри.

Рис. 2.17. Меню смерті персонажа

РОЗДІЛ 3
ЕКОНОМІЧНИЙ РОЗДІЛ

3.1. Визначення трудомісткості та вартості розробки програмного продукту

Початкові дані:

1. передбачуване число операторів програми - 1200;
2. коефіцієнт складності програми - 1,3;
3. коефіцієнт корекції програми в ході її розробки - 0,1;
4. годинна заробітна плата програміста - 221,42 грн/год; згідно з інформацією з сайту Work.ua (https://www.work.ua/salary-unity+developer/), заробітна позиції Unity Developer, станом на 2024 рік, становить 31000 грн [13]. Середня кількість робочих годин в місяць становить 140 годин [11]. Таким чином можна визначити годинну заробітну плату.
5. коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі - 1,3;
6. коефіцієнт кваліфікації програміста, обумовлений від стажу роботи з даної спеціальності - 0,8;
7. вартість машино-години ЕОМ – 17,64 грн/год(2,64 грн – кВт/год [10], 10 грн – ПЗ, 5 грн - амортизація).

Нормування праці в процесі створення ПЗ істотно ускладнено в силу творчого характеру праці програміста. Тому трудомісткість розробки ПЗ може бути розрахована на основі системи моделей з різною точністю оцінки.

Трудомісткість розробки ПЗ можна розрахувати за формулою:

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

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

t_u - витрати праці на дослідження алгоритму рішення задачі;

t_a - витрати праці на розробку блок-схеми алгоритму;

t_n - 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 умовне число операторів у програмному забезпеченні, яке розробляється. Умовне число операторів (підпрограм):

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

де q - передбачуване число операторів (1200);

C - коефіцієнт складності програми (1,3);

p - коефіцієнт корекції програми в ході її розробки (0,1).

Звідси умовне число операторів в програмі:

$$Q = 1{,}3 \cdot 1200 \cdot (1 + 0{,}1) = 1716$$

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

$$t_u = \frac{Q \cdot B}{(75..85) \cdot k}, \text{ людино-годин,} \qquad (3.3)$$

де B - коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі(1,3);

k - коефіцієнт кваліфікації програміста, обумовлений від стажу роботи з даної спеціальності. При стажі роботи до 2 років він складає 0,8.

$$t_u = (1716 \cdot 1.3) / (85 \cdot 0{,}8) = 32{,}8 \text{ людино-годин}$$

Витрати праці на розробку алгоритму рішення задачі визначаються за формулою:

$$t_a = \frac{Q}{(20\ldots25) \cdot k}, \text{ людино-годин,} \qquad (3.4)$$

де Q – умовне число операторів програми;

k – коефіцієнт кваліфікації програміста.

Підставивши відповідні значення в формулу (3.4), отримаємо:

t_a = 1716 / (20 · 0,8) = 107,25 людино-годин.

Витрати на складання програми по готовій блок-схемі:

$$t_n = \frac{Q}{(20\ldots25)\cdot k}, \text{ людино-годин,} \qquad (3.5)$$

t_n = 1716 / (25 · 0,8) = 85,8 людино-годин.

Витрати праці на налагодження програми на ЕОМ:

– за умови автономного налагодження одного завдання:

$$t_{отл} = \frac{Q}{(4..5)\cdot k}, \text{ людино-годин,} \qquad (3.6)$$

$t_{отл}$ = 1716 / (5 · 0,8) = 429 людино-годин.

– за умови комплексного налагодження завдання:

$$t^k_{отл} = 1{,}5\cdot t_{отл}, \text{ людино-годин,} \qquad (3.7)$$

$t^k_{отл}$ = 1,5 · 429 = 643,5 людино-годин.

Витрати праці на підготовку документації визначаються за формулою:

$$t_{д} = t_{др} + t_{до}, \text{ людино-годин,} \quad (3.8)$$

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

$$t_{др} = \frac{Q}{(15..20) \cdot k}, \text{ людино-годин,} \quad (3.9)$$

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

$$t_{до} = 0{,}75 \cdot t_{др}, \text{ людино-годин,} \quad (3.10)$$

Підставляючи відповідні значення, отримаємо:

$t_{др}$ = 1716 / (20 · 0,8) = 107,25 людино-годин.

$t_{до}$ = 0,75 · 107,25 = 80,43 людино-годин.

$t_{д}$ = 107,25 + 80,43 = 187,68 людино-годин.

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

t = 50 + 32,8 + 107,25 + 85,8 + 429 + 187,68 = 806,73 людино-годин.

3.2. Розрахунок витрат на створення програми

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

$$К_{ПО} = З_{ЗП} + З_{МВ}, \text{ грн,} \quad (3.11)$$

Заробітна плата виконавців визначається за формулою:

$$З_{ЗП} = t \cdot С_{ПР}, \text{ грн,} \quad (3.12)$$

де: t - загальна трудомісткість, людино-годин;

$С_{ПР}$ - середня годинна заробітна плата програміста, грн/година

$З_{ЗП}$ = 806,73 · 221,42 = 178 626,156 грн.

Вартість машинного часу, необхідного для налагодження програми на ЕОМ:

$$З_{мв} = t_{отл} \cdot С_{мч}, \text{ грн,} \quad (3.13)$$

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

$C_{мч}$ - вартість машино-години ЕОМ, грн/год (0,68).

$$З_{мв} = 429 \cdot 17{,}64 = 7\ 567{,}56 \text{ грн.}$$

Звідси витрати на створення програмного продукту:

$$К_{ПО} = 7\ 567{,}56 + 178\ 626{,}156 = 1\ 351\ 764\ 153{,}09936 \text{ грн.}$$

Очікуваний період створення ПЗ:

$$T = \frac{t}{B_k \cdot F_p}, \text{ міс.,} \qquad (3.14)$$

де B_k - число виконавців (дорівнює 1);

F_p - місячний фонд робочого часу (при 40 годинному робочому тижні F_p=176 годин);

t – загальна трудомісткість, людино годин()

Звідси витрати на створення програмного продукту:

$$T = 806{,}73 / 1 \cdot 176 \approx 4{,}58 \text{ міс.}$$

Висновок: час затрачений на розробку даного ігрового додатку складає 806,73 людино-годин. Приблизний час розробки складе 4,58

місяців при 40 годинному робочому тижні. На розробку ігрового додатку сумарно буде витрачено 1 351 764 153,09936 грн.

ВИСНОВКИ

Моя кваліфікаційна робота описує той шлях, що пройшла ігрова індустрія за час свого існування. Завдяки стрімкому розвитку технологій, розробники мають безліч можливостей для створення дивовижних ігрових світів. Ігри вже давно є не просто розвагою, а потужним інструментом для навчання, розвитку та донесення важливих ідей до суспільства. Відеоігри є невідємною частиною сучасної культури та мають великий попит. Це означає, що сфера розробки ігор має велике майбутнє і перспективи розвитку. У сучасні дні розробка ігрових додатків є більш простою та доступною, у порівнянні з минулими десятиліттями. Тож будь-яка людина може випробувати себе у ролі розробника та створити власний інді-проєкт. Враховуючи масштаби ігрової індустрії, такі спеціалісти завжди мають попит і зможуть знайти роботу у сфері розробки відеоігор.

У кваліфікаційній роботі було розглянуто ігрові рушії, їх призначення та сфери застосування. Були згадані такі рушії, як CyEngine, Unreal Engine, Unity і REDengine. Проте з усіх перерахованих середовищ розробки, для новачків та інді-розробників, Unity найкращий варіант. Unity Engine надає широкий набір інструментів, які дозволяють створювати високоякісні ігри з мінімальними витратами. Це є прекрасною можливістю почати свій шлях в розробці відеоігор.

Жанр платформера з’явився відразу з появою перших відеоігор і завжди був актуальним серед гравців. Навіть враховуючи сучасні графічні можливості та велику кількість ігор з реалістичною графікою, візуальний стиль під назвою «Pixel Art» досі має популярність. Розглянуті ігри в жанрі платформер, такі як Hollow Knight і Cuphead мають повністю

лінійне проходження або відносно відкритий світ. В своїй розробці я поєднав лінійність рівнів, та вибір шляху їх проходження. Гравцю потрібно обрати оптимальний та максимально ефективний варіант проходження рівня.

Практичною частиною кваліфікаційної роботи є створення ігрового додатка в жанрі 2D платформера і візуального стилю Pixel Art. Гравець має можливість керувати рухом персонажа, вбивати ворогів, оминати пастки та проходити рівень обраним шляхом. Гравець може орієнтуватись на анімації ворогів та пасток, щоб уникати пошкоджень та успішно проходити рівні.

Економічний розділ описує витрати на створення ігрового додатка, що становить 1 351 764 153,09936 грн. Час затрачений на розробку буде становити 4,58 місяця.

Розроблений ігровий додаток має великий потенціал для розвитку та покращень, таких як нові рівні, пастки та нові типи ворогів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Hollow Knight / URL: https://store.steampowered.com/app/367520/Hollow_Knight/
2. Techy Mau / URL: https://techymau.games/blog/why-unity-software-is-a-great-video-game-development-platform
3. Lemon School / URL: https://lemon.school/blog/shho-take-unity
4. IGN / URL: https://www.ign.com/articles/2018/06/22/hollow-knight-review
5. Itvdn / URL: https://itvdn.com/ua/blog/article/7best-unity-games
6. Senfil / URL: https://senfil.net/index.php?newsid=321
7. AssetStore / URL: https://assetstore.unity.com/packages/2d/characters/pixel-adventure-1-155360
8. Mixkit / URL: https://mixkit.co/free-sound-effects/game/?page=2
9. Itch / URL: https://itch.io/game-assets/tag-menu/tag-sound-effects
10. Yasno / URL: https://yasno.com.ua/b2c-tariffs
11. Planeta-inform / URL: https://planeta-inform.com.ua/norma-rabochego-vremeni/
12. Wikipedia / URL: https://en.wikipedia.org/wiki/Unreal_Engine
13. WorkUA / URL: https://www.work.ua/salary-dnipro-it/
14. Medium / URL: https://medium.com/oke-software/creating-2d-platformer-game-in-unity3d-eecfeac06a58

ДОДАТОК А
ЛІСТИНГ ПРОГРАМИ

CameraController.cs

```
using UnityEngine;

public class CameraController : MonoBehaviour
{
    //Room camera
    [SerializeField] private float speed;
    private float currentPosX;
    private Vector3 velocity = Vector3.zero;

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

    private void Update()
    {

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

    }

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

SoundManager.cs

```
using UnityEngine;
```

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

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

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

        ChangeMusicVolume(0);
        ChangeSoundVolume(0);

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

    public void ChangeSoundVolume(float _change)
    {
        ChangeSourceVolume(1, "soundVolume", _change, soundSource);
    }

    public void ChangeMusicVolume(float _change)
    {
        ChangeSourceVolume(0.3f, "musicVolume", _change, musicSource);
    }
```

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

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



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

        //Assign final value
        float finalVolume = currentVolume * baseVolume;
        source.volume = finalVolume;



        PlayerPrefs.SetFloat(volumeName, currentVolume);
    }



}
```

PlayerMovement.cs

```
using UnityEngine;

public class PlayerMovement : MonoBehaviour
{
    [SerializeField] private float speed;
    [SerializeField] private float jumpPower;

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

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

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

[SerializeField] private LayerMask groundLayer;
[SerializeField] private LayerMask wallLayer;

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

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



private void Awake()
{
    body = GetComponent<Rigidbody2D>();
    anim = GetComponent<Animator>();
    boxCollider = GetComponent<BoxCollider2D>();
}

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

    //Flip player when facing left/right.
    if (horizontalInput > 0.01f)
        transform.localScale = Vector3.one;
    else if (horizontalInput < -0.01f)
        transform.localScale = new Vector3(-1, 1, 1);

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

    //Jump
```

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

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

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

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

}

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

    SoundManager.instance.PlaySound(jumpSound);

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

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

        coyoteCounter = 0;
    }
}

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

private bool isGrounded()
{
    RaycastHit2D raycastHit = Physics2D.BoxCast(boxCollider.bounds.center, boxCollider.bounds.size, 0,
Vector2.down, 0.1f, groundLayer);
    return raycastHit.collider != null;
}

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

```
        }
```

PlayerAttack.cs

```
using UnityEngine;

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

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

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

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

        cooldownTimer += Time.deltaTime;
    }

    private void Attack()
    {
        SoundManager.instance.PlaySound(fireballSound);
        anim.SetTrigger("attack");
        cooldownTimer = 0;

        fireballs[FindFireball()].transform.position = firePoint.position;
        fireballs[FindFireball()].GetComponent<Projectile>().SetDirection(Mathf.Sign(transform.localScale.x));
```

```
    }
    private int FindFireball()
    {
        for (int i = 0; i < fireballs.Length; i++)
        {
            if (!fireballs[i].activeInHierarchy)
                return i;
        }
        return 0;
    }
            }
```

Projectile.cs

```
using UnityEngine;

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

    private Animator anim;
    private BoxCollider2D boxCollider;

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

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

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

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

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

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

Health.cs

```
using UnityEngine;
using System.Collections;

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

```
private Animator anim;
private bool dead;

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

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

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

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

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

            //Player
            if(GetComponent<PlayerMovement>() != null)
```

```
            GetComponent<PlayerMovement>().enabled = false;

        //Enemy
        if(GetComponentInParent<EnemyPatrol>() != null)
            GetComponentInParent<EnemyPatrol>().enabled = false;

        if(GetComponent<MeleeEnemy>() != null)
            GetComponent<MeleeEnemy>().enabled = false;

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

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

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

    //Player
    if (GetComponent<PlayerMovement>() != null)
        GetComponent<PlayerMovement>().enabled = true;

    //Enemy
    if (GetComponentInParent<EnemyPatrol>() != null)
        GetComponentInParent<EnemyPatrol>().enabled = true;

    if (GetComponent<MeleeEnemy>() != null)
        GetComponent<MeleeEnemy>().enabled = true;
```

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

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

    private void Deactivate()
    {
        gameObject.SetActive(false);
    }
            }
```

MeleeEnemy.cs

```
using UnityEngine;

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

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

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

[Header("Attack Sound")]
[SerializeField] private AudioClip attackSound;

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

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

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

    if (PlayerInSight())
    {
        if (cooldownTimer >= attackCooldown && playerHealth.currentHealth > 0)
        {
            cooldownTimer = 0;
            anim.SetTrigger("meleeAttack");
            SoundManager.instance.PlaySound(attackSound);
        }
    }

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

private bool PlayerInSight()
{
    RaycastHit2D hit =
```

```
        Physics2D.BoxCast(boxCollider.bounds.center + transform.right * range * transform.localScale.x *
colliderDistance,
        new Vector3(boxCollider.bounds.size.x * range, boxCollider.bounds.size.y, boxCollider.bounds.size.z),
        0, Vector2.left, 0, playerLayer);

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

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

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

Firetrap.cs

```
using UnityEngine;
using System.Collections;

public class Firetrap : MonoBehaviour
{
  [SerializeField] private float damage;

  [Header("Firetrap Timers")]
  [SerializeField] private float activationDelay;
  [SerializeField] private float activeTime;
  private Animator anim;
  private SpriteRenderer spriteRend;
```

```
[Header("SFX")]
[SerializeField] private AudioClip firetrapSound;

private bool triggered;
private bool active;

private Health playerHealth;

private void Awake()
{
    anim = GetComponent<Animator>();
    spriteRend = GetComponent<SpriteRenderer>();
}

private void Update()
{
    if(playerHealth != null && active)
        playerHealth.TakeDamage(damage);
}

private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.tag == "Player")
    {
        playerHealth = collision.GetComponent<Health>();

        if (!triggered)
            StartCoroutine(ActivateFiretrap());

        if (active)
            collision.GetComponent<Health>().TakeDamage(damage);
    }
}
private void OnTriggerExit2D(Collider2D collision)
{
    if (collision.tag == "Player")
        playerHealth = null;
}

private IEnumerator ActivateFiretrap()
```

```
    {

        triggered = true;
        spriteRend.color = Color.red;



        SoundManager.instance.PlaySound(firetrapSound);
        spriteRend.color = Color.white;
        active = true;
        anim.SetBool("activated", true);



        yield return new WaitForSeconds(activeTime);
        active = false;
        triggered = false;
        anim.SetBool("activated", false);
    }
            }
```

UIManager.cs

```
using UnityEngine;
using UnityEngine.SceneManagement;

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

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

    private void Awake()
    {
        gameOverScreen.SetActive(false);
        pauseScreen.SetActive(false);
    }
    private void Update()
    {
```

```
    if (Input.GetKeyDown(KeyCode.Escape))
    {
        if(pauseScreen.activeInHierarchy)
            PauseGame(false);
        else
            PauseGame(true);
    }
}

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

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

//Activate game over screen
public void MainMenu()
{
    SceneManager.LoadScene(0);
}

//Quit game/exit play mode if in Editor
public void Quit()
{
    Application.Quit(); //Quits the game (only works in build)

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

#region Pause
```

```
    public void PauseGame(bool status)
    {

        pauseScreen.SetActive(status);



        if (status)
            Time.timeScale = 0;
        else
            Time.timeScale = 1;
    }
    public void SoundVolume()
    {
        SoundManager.instance.ChangeSoundVolume(0.2f);
    }
    public void MusicVolume()
    {
        SoundManager.instance.ChangeMusicVolume(0.2f);
    }

    #endregion
}
```

ДОДАТОК Б

ВІДГУК

Керівника економічного розділу

на кваліфікаційну роботу бакалавра на тему:

«Розробка ігрового додатку на рушії Unity з використанням мови програмування C#»

Студента групи 122-20-4 Лісовця Яна Ігоровича

Керівник економічного розділу

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

ДОДАТОК В

ПЕРЕЛІК ФАЙЛІВ НА ДИСКУ

Ім’я файла	Опис
Пояснювальні документи	
Диплом.doc	Пояснювальна записка до кваліфікаційної роботи. Документ Word.
Диплом.pdf	Пояснювальна записка до кваліфікаційної роботи в форматі PDF
Програма	
Diplom.zip	Архів. Містить коди програми і файли гри
Презентація	
Презентація.ppt	Презентація кваліфікаційної роботи