Національний лісотехнічний університет України
(повне найменування вищого навчального закладу)
Навчально-науковий інститут комп'ютерних наук
та інформаційних технологій
(повне найменування інституту, назва факультету (відділення))
Кафедра комп'ютерних наук
(повна назва кафедри (предметної, циклової комісії))

Магістерська кваліфікаційна робота

другий (магістерський)
(рівень вищої освіти)

на тему: *"2D платформер на ігровому рушії Unity"*

Виконав: студент 6 курсу групи КН-61м
спеціальності
122 "Комп'ютерні науки"
(шифр і назва спеціальності)

Скульбеда О. С.
(прізвище та ініціали)

Керівник *Процик Ю. С.*
(прізвище та ініціали)

Рецензент Флуд Л.О.
(прізвище та ініціали)

Львів – 2025

Національний лісотехнічний університет України
(повне найменування вищого навчального закладу)

ННІ комп'ютерних наук та інформаційних технологій
Кафедра комп'ютерних наук
Рівень вищої освіти другий (магістерський)
Спеціальність 122 "Комп'ютерні науки"
(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри КН

_____ Борецька І. Б.

"10" грудня 2025 року

З А В Д А Н Н Я
НА МАГІСТЕРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Скульбеді Олександру Сергійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи *"2D платформер на ігровому рушії Unity"*

керівник роботи *Процик Юрій Степанович, к.ф-м.н., доцент.*
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від *"29" квітня 2025 року* № *С-288*

2. Термін подання студентом роботи *10 грудня 2025 р.*

3. Вихідні дані до роботи *Аналіз шляхів вирішення задачі, організаційна структура застосунку*

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)
Перелік скорочень та умовних позначень. Вступ.
Розділ 1. Стан проблемної області.
Розділ 2. Інформаційне забезпечення.
Розділ 3. Математичне забезпечення.
Розділ 4. Програмне забезпечення.
Розділ 5. Розроблення стартап-проєкту.
Висновки. Список використаних джерел. Додатки.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
Слайди для доповіді (підготовка матеріалу для доповіді загальним обсягом 10-12 слайдів)

6. Дата видачі завдання *1 травня 2025 року*

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1.	Аналіз науково-технічної літератури та огляд існуючих рішень за темою дослідження	*01.05.2025 р.* *21.05.2025 р.*	виконано
2.	Обгрунтування концепції розробки та порівняльний аналіз методів і засобів розв'язання задачі	*22.05.2025 р.* *23.06.2025 р.*	виконано
3.	Об'єктна постановка задачі, визначення вимог та математична формалізація ігрових механік	*24.06.2025 р.* *28.07.2025 р.*	виконано
4.	Програмна реалізація завдання	*29.07.2025 р.* *13.10.2025 р.*	виконано
5.	Тестування програмного продукту та отриманих результатів	*14.10.2025 р.* *05.11.2025 р.*	виконано
6.	Оформлення пояснювальної записки та здача на рецензування	*06.11.2025 р.* *10.12.2025 р.*	виконано

Студент _____ Скульбеда О. С.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Процик Ю. С.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Магістерська робота містить 54 сторінки пояснювальної записки, 13 рисунків, 2 додатки, 20 джерел.

У даній роботі розроблено та реалізовано 2D платформер для персональних комп'ютерів на базі ігрового рушія Unity. Проведено аналіз сучасних тенденцій та механік у жанрі 2D платформерів, що дозволило визначити оптимальну архітектуру та ігровий дизайн для створення захоплюючого ігрового досвіду.

Розроблено модульну програмну архітектуру гри, реалізовано ключові ігрові механіки, включаючи рух персонажа (стрибки, біг, взаємодія з оточенням), систему колізій, інтерактивні елементи рівнів та систему управління ігровими станами.

Ключові слова: 2D платформер, ігровий рушій Unity, ігрова механіка, контролер персонажа, розробка ігор, ігровий дизайн.

ABSTRACT

The master's thesis consists of 54 pages of the explanatory note, 13 figures, 2 appendices, and 20 references.

This work develops and implements a 2D platformer game for personal computers based on the Unity game engine. An analysis of modern trends and mechanics in the 2D platformer genre has been conducted, which allowed for the determination of the optimal architecture and game design to create an engaging gaming experience.

A modular software architecture for the game has been developed, with key gameplay mechanics implemented, including character movement (jumping, running, interaction with the environment), collision system, interactive level elements, and game state management system.

Keywords: 2D platformer, Unity game engine, game mechanics, character controller, game development, game design.

ТЕХНІЧНЕ ЗАВДАННЯ

Необхідно розробити повністю функціональну гру, з як мінімум одним готовим рівнем, що передбачає виконання таких завдань:

1. Рух гравця: Створення скриптів для горизонтального руху, стрибків, (опціонально) присідання/ковзання. Плавна та чутлива взаємодія з фізикою.
2. Колізії та взаємодія: Налаштування колізійних шарів та тригерів для взаємодії з платформами, стінами, ворогами, бонусами.
3. Базові механіки: Реалізація базових ворогів з простими патернами, системи «життя» гравця.
4. Дизайн рівнів: Створення одного-двох цікавих та логічно продуманих рівнів з платформами, перешкодами, ворогами, бонусами.
5. Користувацький інтерфейс (UI): Базовий UI для відображення інформації (життя, пауза/рестарт).
6. Звукові ефекти та музика: Додавання звукових ефектів та фонової музики для атмосфери.
7. Тестова збірка: Збірка проекту у виконуваний файл для тестування на інших системах.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ

2D – Двовимірний простір (Two-Dimensional).

ПК – Персональний комп'ютер (Personal Computer, PC).

ООП – Об'єктно-орієнтоване програмування (Object-Oriented Programming).

API – Інтерфейс прикладного програмування (Application Programming Interface).

SDK – Комплект засобів розробки (Software Development Kit).

IDE – Інтегроване середовище розробки (Integrated Development Environment).

GUI – Графічний інтерфейс користувача (Graphical User Interface, GUI).

UI – Користувацький інтерфейс (User Interface, UI).

UX – Досвід користувача (User Experience, UX).

ВСТУП

Сучасні цифрові технології суттєво змінюють способи створення інтерактивних продуктів, і відеоігри в цьому процесі відіграють особливу роль. Вони поєднують інженерні рішення, алгоритмічне мислення та творчі підходи до проєктування. Попри стрімкий розвиток 3D-графіки, 2D-платформери залишаються популярним жанром, оскільки дають змогу зосередитися на точності механік і якості геймплею. Розробка такого типу ігор потребує глибокого розуміння математичних моделей руху, систем колізій, правил взаємодії об’єктів та принципів архітектури програмних рішень.

Ігровий рушій Unity надає гнучке середовище для всіх цих аспектів, поєднуючи інструменти для роботи з графікою, фізикою, анімацією та логікою. Завдяки підтримці мови C# він дозволяє створювати складні ігрові системи, не втрачаючи контролю над їх внутрішньою структурою. Тому вибір Unity для навчального дослідження є цілком обґрунтованим і забезпечує можливість відтворити повний цикл проєктування 2D-гри.

Об’єкт дослідження — процес розробки інтерактивних програмних систем на основі ігрових рушіїв.

Предмет дослідження — методи та засоби створення 2D платформера з використанням Unity і мови програмування C#.

Мета роботи — розробити повноцінний 2D платформер, який демонструє застосування математичних моделей, програмних шаблонів та інженерних підходів у побудові ігрової логіки.

Для досягнення поставленої мети визначено такі **завдання**:

- проаналізувати сучасний стан проблемної області та існуючі підходи до створення 2D-ігор;
- сформувати інформаційну модель системи та описати взаємодію ігрових об’єктів;
- побудувати математичні моделі руху й колізій, що необхідні для платформера;
- спроєктувати архітектуру програмного забезпечення та реалізувати ключові механіки гри;

- провести тестування роботи системи та оцінити її функціональність.

Наукова новизна одержаних результатів полягає у поєднанні математичних моделей фізики руху з адаптованою архітектурою ігрових модулів, що дозволяє сформувати узагальнений підхід до створення 2D платформера в умовах навчального дослідження. У роботі систематизовано принципи побудови ігрових механік у Unity та продемонстровано спосіб їх реалізації через скінченні автомати станів і структурні шаблони програмування.

Практичне значення одержаних результатів полягає в тому, що створена гра може бути використана як навчальний приклад для студентів спеціальності «Комп’ютерні науки», а запропоновані моделі та архітектурні рішення — у якості основи для подальших проєктів або розширення ігрового функціоналу. Розроблений платформер підтверджує можливість застосування теоретичних знань у реальному програмному продукті та демонструє цілісний процес створення інтерактивної системи.

РОЗДІЛ 1. СТАН ПРОБЛЕМНОЇ ОБЛАСТІ

Платформери належать до найстаріших жанрів комп’ютерних відеоігор і фактично сформували уявлення про ранні можливості інтерактивних розваг. Класичні назви на кшталт Super Mario, Contra, Prince of Persia, Castlevania, Megaman чи Sonic the Hedgehog давно стали частиною історії ігрової індустрії. Ці проєкти часто згадують як взірці жанру, адже саме вони визначили його ключові риси й задали напрям розвитку на багато років вперед [1].

У більшості таких ігор можна помітити спільні риси:

- ігровий процес подається зі сторони, що дозволяє чітко бачити простір і перешкоди;
- рівні побудовані так, щоб гравець постійно долав платформи та перешкоди стрибками;
- управління орієнтоване на точність руху й своєчасність дій.

З плином часу жанр змінювався разом із розвитком технологій. Коли графічні можливості комп’ютерів дозволили працювати у тривимірному просторі, платформери поступово почали переходити від 2D до 3D-подачі. Так з’явилися серії на кшталт Spyro, Crash Bandicoot чи Shrek. Деякі класичні франшизи також намагалися переосмислити себе у 3D, однак не всі експерименти були вдалими. Наприклад, ранні спроби перенести Prince of Persia у тривимірність отримали значну критику через незручне керування та суперечливі технічні рішення.

Розвиток 3D-технологій відкрив розробникам можливість впроваджувати нові механіки: вертикальний і горизонтальний біг по стінах, складні акробатичні рухи, розширену взаємодію з оточенням. Такі елементи значно урізноманітнили геймплей і розширили рамки жанру.

Однак попри активний розвиток 3D-ігор, двовимірні проєкти не втратили актуальності. Деякі з них не лише успішно виходять на сучасні платформи, а й конкурують з масштабними тривимірними хітами. До таких можна віднести Hollow Knight, Ori and the Blind Forest, Dead Cells, Katana Zero та інші. Частину цих ігор підтримували й видавали великі компанії — зокрема Microsoft, що свідчить про

високу якість та комерційний потенціал жанру. Значна частина сучасних 2D-проєктів зберігає стиль піксель-арту, що відсилає до естетики 90-х і підкреслює ностальгійний характер жанру, поєднуючи його з сучасними технічними можливостями.

1.1 Технічні особливості та обмеження рушія

Створення 2D-платформера часто сприймається як не надто складний процес, однак у реальній розробці виникає багато технічних нюансів, які впливають на стабільність та якість готового продукту. Unity дозволяє створювати подібні ігри досить швидко, проте за можливостями рушія стоїть чимало обмежень, які потрібно враховувати на всіх етапах роботи .

Попри те, що двовимірні проєкти зазвичай споживають менше ресурсів, ніж тривимірні, вони все одно потребують продуманої оптимізації. До прикладу, велика кількість дрібних об’єктів на сцені може призвести до зниження продуктивності. Кожен колайдер, скрипт, анімація або партикл-система додає до загального навантаження, і якщо їх використовувати без системного підходу, навіть простий рівень може почати гальмувати.

Суттєву роль відіграє також правильне налаштування фізики. Unity пропонує компоненти RigidBody2D і Collider2D, які значно полегшують роботу, проте за неправильного використання вони можуть створювати додаткові проблеми. Наприклад, надто часті перевірки колізій або використання складних колайдерів там, де можна обійтися простими примітивами, погіршують загальну продуктивність. У платформерах, де важлива точність руху та зіткнень, такі деталі мають велике значення.

Окрема тема — управління персонажем. Гравці дуже чутливо реагують на затримки, надмірну інерцію або "ватність" виконання команд. У платформерах саме відчуття контролю визначає, наскільки комфортною буде гра. Якщо персонаж реагує із затримкою або має неприродну фізику, це майже гарантовано зіпсує враження, навіть якщо візуально гра виглядає добре. Тому розробнику доводиться багато експериментувати з Input System, вручну налаштовувати параметри руху, задавати межі прискорення, висоту стрибка, взаємодію з поверхнями.

Таким чином, технічна частина розробки 2D-платформера потребує не лише знань можливостей Unity, а й розуміння того, як працюють внутрішні системи рушія — фізика, оновлення кадру, оптимізація та обробка введення[2].

1.2 Дизайн гри та сучасні ринкові умови

Дизайн 2D-платформера — це значно більше, ніж просто створення рівнів зі стрибками. Сучасні гравці мають високу вимогливість, а ринок інді-ігор перенасичений схожими проєктами, тож створити щось конкурентоспроможне стає дедалі важчою задачею. Для того, щоб гра не загубилася серед інших, розробнику доводиться продумувати не лише механіки, а й цілісну концепцію, стиль та структуру подачі.

Одне з основних дизайнерських завдань — зробити гру унікальною. Гравці сьогодні бачили величезну кількість платформерів: з космічною тематикою, музичними рівнями, піксель-артом, мультяшними персонажами, акробатичними механіками тощо. Тому нова гра повинна мати деталь, патерн, механіку чи сюжетний елемент, який запам’ятається. Це може бути незвична фізика, інтерактивні об’єкти, особлива система бою або нестандартний стиль подачі.

Паралельно з цим виникає питання балансу складності. Успішний платформер поступово навчає гравця новим елементам і робить це непомітно. Простий рівень може стати тренуванням, а складні перешкоди вводяться лише після того, як гравець засвоїв базові рухи. Якщо гра порушує цю логіку — наприклад, різко ускладнює рівні або робить їх занадто механічними — гравець швидко втрачає інтерес.

Тут важливо створити так званий "ігротік" або ігровий потік (flow). Це стан, у якому гравець відчуває, що завдання достатньо складні, щоб бути цікавими, але не настільки важкі, щоб викликати роздратування. Досягти цього часто нелегко: іноді доводиться переробляти рівні, змінювати механіки ворогів, тестувати кілька варіантів їхньої поведінки, перевіряти розташування платформ, пасток і бонусів.

Свою роль відіграють і ринкові реалії. Успішні проєкти, такі як Hollow Knight, Celeste або Dead Cells, задали високу планку якості. Вони поєднують цікавий сюжет, стильну анімацію, складність і плавність керування. На фоні таких проєктів

аматорська гра повинна мати власну "родзинку", навіть якщо вона невелика за масштабом.

1.3 Користувацький досвід, візуальна подача та художня складова

Користувацький досвід (UX) — це те, що визначає, як гравець сприймає гру в цілому. Навіть у простих платформерах він складається з безлічі дрібних деталей: насиченості кольорів, роботи анімацій, плавності переходів між сценами, стилю звуків, оформлення меню та зручності управління й своєчасність дій [4].

Візуальна частина також є важливим аспектом гри. Найперша причина полягає в тому, що якщо гра виглядає красиво то на неї вже звернуть увагу, що збільшить шанси її придбання потенційним клієнтом. Вона визначає характер гри: комедійну, атмосферну, похмуру, емоційну. Піксель-арт може викликати ностальгію, мультяшний стиль — легкість і динамічність, а мінімалістичний підхід — зосередженість на геймплеї . Важливо, щоб всі елементи відповідали один одному: фон, персонажі, платформи, ефекти й навіть шрифт у меню.

Роль анімації також важко переоцінити. Плавний рух, чіткі переходи, візуальний відгук на кожну дію роблять гру відчутною. Якщо стрибок виглядає "невагомим" або анімації ворогів занадто різкі, гравець відчуває дискомфорт на інтуїтивному рівні.

Звук створює атмосферу не менше, ніж графіка. Правильно підібрані звукові ефекти, кроки, удари, взаємодія з об’єктами, а також музичний супровід можуть значно збагатити сприйняття світу. Unity надає інструменти для роботи зі звуком, але щоб отримати якісний результат, потрібно розуміти основи аудіодизайну — уникати накладення зайвих звуків, підбирати частоти, регулювати гучність та баланс.

Інтерфейс, в свою чергу, відповідає за взаємодію гри і гравця. Він має бути логічним, зручним і чітко зрозумілим. Елементи інтерфейсу не повинні відволікати від гри або закривати корисну інформацію.

Сучасні тенденції також підкреслюють важливість доступності: можливість налаштувати розмір тексту, регулювати тип керування тощо. Це збільшує потенційну аудиторію та робить гру зручнішою для всіх.

Висновки до розділу

Розробка 2D-платформера — це комплексна робота, яка охоплює технічні, дизайнерські та художні аспекти. Для створення стабільної гри необхідно враховувати обмеження рушія та особливості роботи фізики, оптимізувати систему та забезпечити точне реагування на дії гравця.

З точки зору дизайну важливо знайти унікальність, збалансувати складність і забезпечити плавний розвиток гравця від простих завдань до складніших. Світ гри має бути цікавим, а рівні — такими, що утримують увагу.

Художня і аудіовізуальна складова формує загальне враження від гри. Стиль, анімації, звук, інтерфейс — усе це визначає атмосферу та впливає на комфорт користувача.

Таким чином, створення платформера — це багатогранний процес, який вимагає продуманості, тестування, уваги до деталей і вміння поєднувати творчість із технічними навичками.

РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ

Успішне проєктування та реалізація 2D платформера в середовищі Unity значною мірою ґрунтується на ретельно підібраній системі інформаційних ресурсів. Цей розділ розкриває комплекс джерел, що формують інформаційне підґрунтя проєкту, поєднуючи теоретичні матеріали, прикладні інструменти та практичні рекомендації, необхідні для повноцінного створення ігрового продукту.

2.1 Офіційна документація та освітні матеріали Unity

Одним із ключових інформаційних сегментів є офіційна документація Unity, яка посідає статус основного та найбільш достовірного джерела. У ній подано систематизований опис компонентів рушія, їхніх параметрів та взаємодії, а також наведено повноцінний довідник API (Application Programming Interface). До документів цього типу розробник звертається як для отримання базових пояснень, так і для уточнення специфічних деталей, що впливають на функціональність гри.
Зокрема, документація надає вичерпні відомості щодо таких підсистем:

● Input System – модуль, що відповідає за обробку користувацького вводу. Його правильне налаштування забезпечує стабільне і передбачуване керування персонажем, підтримку різних пристроїв та гнучкість адаптації під потреби гри.

● Physics 2D – набір інструментів і компонентів, що реалізує двовимірну фізику: зіткнення, взаємодію об’єктів, гравітацію, силу тертя тощо. Основними складовими є Rigidbody2D, Collider2D (BoxCollider2D, CircleCollider2D, PolygonCollider2D), а також налаштування фізичної сцени та шарів взаємодії.

● Animator та система анімацій – комплекс засобів, що дозволяє створювати, управляти та комбінувати анімаційні стани. Він використовується для реалізації поведінки персонажа, ворогів, інтерактивних елементів рівня.

● UI Canvas – інструментарій для побудови інтерфейсу користувача: панелі стану героя, лічильники ресурсів, меню паузи, інтерфейс завершення рівня.

● Particle System – можливість створювати та налаштовувати візуальні ефекти, які підсилюють враження від ігрового процесу: іскри від ударів, пил, ефекти стрибка, анімації магічних або механічних подій.

Окремо варто відзначити навчальну платформу Unity Learn, де зібрані інтерактивні курси, покрокові проєкти, приклади вихідного коду та практичні вправи. Ці матеріали сприяють швидкому засвоєнню принципів роботи з рушієм і надають практичні приклади реалізації типових ігрових механік[5].

Поряд із офіційними ресурсами важливу роль відіграють тематичні книги, фахові статті та навчальні посібники, що стосуються геймдизайну та програмної інженерії.

2.2 Онлайн-спільноти, репозиторії вихідного коду

У процесі розробки ігор на Unity важливу роль відіграють не лише офіційні джерела інформації, а й активні онлайн-спільноти, у яких обговорюються практичні аспекти створення ігрових проєктів. Такі платформи виконують функцію неформальної підтримки, дозволяючи отримувати оперативні поради, обмінюватися досвідом та знаходити нестандартні рішення. У багатьох випадках саме дискусії в спільнотах допомагають розв’язати проблему швидше, ніж самостійний пошук або аналіз документації.

Одними з найбільш інформативних майданчиків є офіційні форуми Unity, Stack Overflow, Reddit (розділи, присвячені інді-розробці та Unity), а також численні репозиторії GitHub і GitLab. Їх значення для проєкту полягає в декількох аспектах:

● Оперативне вирішення прикладних задач. Кожний технічний виклик, з яким стикається розробник, з великою ймовірністю вже був розглянутий кимось раніше. Це стосується оптимізації фізики, налаштування анімаційних станів, роботи зі спрайт-атласами, реалізації руху або обробки колізій. Форумні обговорення часто містять конкретні фрагменти коду, поради щодо виправлення помилок або детальні пояснення причин некоректної поведінки систем.

● Пошук ефективніших підходів до реалізації функціоналу. Досвідчені розробники пропонують альтернативні алгоритми або структури проєкту, що дозволяє покращити продуктивність, знизити споживання ресурсів та скоротити

дублювання коду. Це особливо цінно у 2D платформерах, де на сцені одночасно може бути значна кількість рухомих об’єктів.

● Доступ до відкритих прикладів і готових рішень. У репозиторіях відкритого коду можна знайти зразки інвентарних систем, контролерів руху, шейдерів, анімаційних FSM-машин, UI-компонентів та інших елементів. Аналіз коду інших розробників дає можливість переймати кращі архітектурні практики та уникати типових помилок на початковому етапі[6].

● Ознайомлення з інструментами та трендами. Спільноти активно обговорюють нові версії рушія, можливості пакетів Package Manager, сторонні плагіни, системи анімації, оптимізаційні техніки та підходи до побудови рівнів.

Також високу цінність становить відеоконтент. Канали досвідчених практиків, таких як Code Monkey, Blackthornprod, Sebastiaan Lague та інші, містять покрокові пояснення складних механік, аналіз архітектури популярних ігор та розбір оптимізаційних підходів. Візуальна подача матеріалу значно спрощує засвоєння складних концепцій і допомагає застосовувати їх у власному проєкті.

Важливою частиною інформаційного забезпечення є аналіз існуючих комерційних і незалежних 2D платформерів. Вивчення подібних проєктів дає розуміння того, як окремі дизайнерські рішення впливають на поведінку гравця, що робить рівні цікавими, а управління — інтуїтивним. Ретельний огляд таких ігор, як “Celeste”, “Hollow Knight”, “Dead Cells”, “Shovel Knight”, а також класичних платформерів, дозволяє отримати цінні практичні висновки:

● Оцінка структури ігрової механіки. Успішні ігри містять низку рішень щодо руху, колізій, логіки ворогів, платформ, пасток та інтерактивних об’єктів. Їх аналіз дає можливість зрозуміти, які механіки зручні для гравців, а які потребують додаткової адаптації.

● Вивчення дизайну рівнів. Аналіз траєкторій руху, розміщення перешкод, темпу ігрового процесу дозволяє визначити структуру рівнів, що створюють природний “потік” проходження та утримують увагу користувача.

● Оцінка художньої та звукової складової. Візуальний стиль, робота зі світлом, палітра кольорів, музичний супровід та звукові ефекти формують загальну емоційну

картину. Порівняння різних підходів дає можливість підібрати стилістику, що підходить саме цій грі.

● Дослідження інтерфейсних рішень. Добре продуманий UI забезпечує комфорт гравця, не привертаючи зайвої уваги до себе, але надаючи всю необхідну інформацію. Важливо розуміти, як інші проєкти реалізують навігацію, відображення стану героя, меню та системи підказок.

Подібний аналіз дозволяє виявити успішні підходи та водночас зрозуміти, яких рішень варто уникати, щоб гра була не копією, а конкурентоспроможним і впізнаваним продуктом[7].

2.3 Графічні ресурси

Спрайти головних персонажів і ворожих істот формують основу візуального стилю гри. Кожен спрайт створюється з урахуванням силуету, масштабу та читабельності, щоб гравець легко розпізнавав дії та наміри об’єкта. Для персонажа важливо передати не лише форму, а й характер — наприклад, через позу, вираз обличчя або пропорції. Вороги зазвичай проєктуються з більш виразними акцентами, щоб одразу зрозуміти їхню небезпеку чи агресивність. Усі ці елементи поєднуються в загальну художню стилістику, що підсилює атмосферу гри.

Анімаційні спрайт-листи містять набір окремих кадрів, з яких складається рух персонажів, ворогів або будь-яких інших об’єктів (див. рис. 2.1). Кожний кадр зображає певну фазу руху — крок, стрибок, замах чи атаку. Якість анімації залежить не лише від кількості кадрів, а й від плавності переходів між ними. У спрайт-листах зручно групувати окремі набори анімацій: біг, стояння, ушкодження, смерть тощо. Це дозволяє коректно налаштовувати поведінку об’єктів у Unity та забезпечує природність їхньої взаємодії з гравцем.

Рисунок 2.1 — Анімаційний спрайт-лист

Текстури застосовуються для візуального оформлення предметів оточення: платформ, дерев, каміння, будівель, декоративних елементів. Вони додають рівням деталізації та роблять світ гри переконливим. Текстури часто створюють у єдиній палітрі кольорів, щоб уникнути візуального “шуму” та зберегти цілісність стилю. Деякі об’єкти потребують додаткових варіацій текстур — наприклад, зношеність, тріщини або тіні, що допомагає передати глибину та підсилити атмосферу.

Фон рівня задає загальний настрій сцени та допомагає передати простір за межами ігрової зони (див. рис. 2.2). Це можуть бути гори, міські пейзажі, лісові масиви чи інтер’єри споруд. Часто фони складаються з декількох шарів, що дозволяє використовувати ефект паралаксного руху, створюючи ілюзію глибини. Якісний фон не відволікає, але гармонійно доповнює передній план, підтримує атмосферу та стилістику гри.

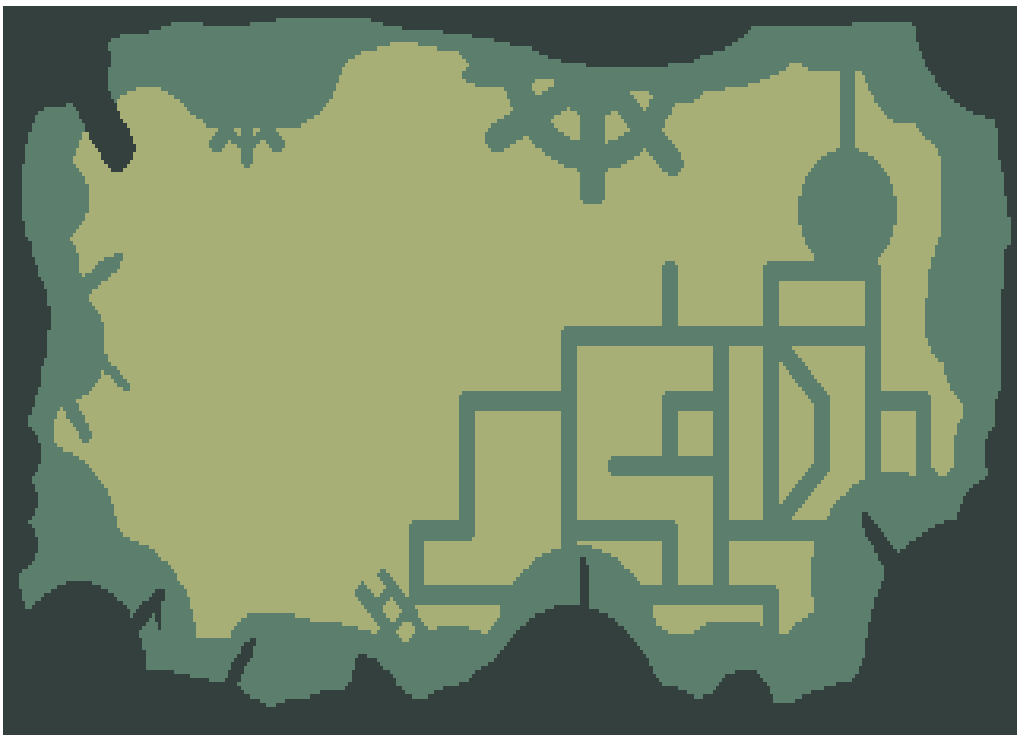

Рисунок 2.2 — Фон локації

Плиткові набори — це колекції дрібних графічних елементів, з яких складаються ігрові рівні. Вони дозволяють швидко будувати платформи, стіни, схили, декоративні об’єкти та інші структури. Кожна плитка має фіксований розмір і логічно поєднується з іншими елементами набору. Завдяки цьому розробник може створювати складні локації без необхідності малювати кожен сегмент вручну. Tilemaps також спрощують оптимізацію: повторне використання плиток зменшує навантаження на пам’ять і покращує продуктивність гри.

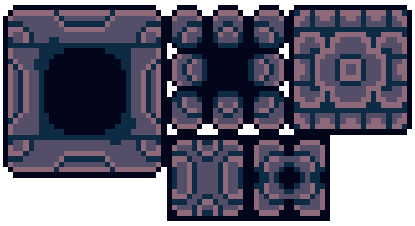

Рисунок 2.3 — Набір плиток

2.4 Аудіоінформація

Звукові ефекти відіграють важливу роль у сприйнятті дій персонажа та загальної динаміки гри. Короткі аудіосемпли для кроків, стрибків, ударів чи взаємодій із предметами допомагають гравцю краще орієнтуватися в ситуації та відчувати фізичність процесів. Правильно підібрані звуки підсилюють візуальні анімації, роблять їх переконливішими та забезпечують швидший зворотний зв’язок на дії користувача. Важливо, щоб ефекти були збалансованими за гучністю та не перекривали один одного — це дозволяє уникнути «шумового перевантаження» і створює більш комфортне ігрове середовище.

Фонова музика формує емоційний тон рівнів і задає гравцю настрій під час проходження. Саундтрек може створювати атмосферу напруженості, спокою, пригод або небезпеки залежно від того, які події відбуваються в грі. Композиції зазвичай підбираються так, щоб не відволікати від ігрового процесу, але водночас підтримувати загальний ритм і стилістику проєкту. Музика, що органічно поєднується з візуальним оформленням і механіками, покращує занурення гравця у світ гри та робить загальний досвід більш цілісним[8].

2.5 Конфігураційні дані

Фізичні параметри є основою реалістичної поведінки всіх об’єктів у сцені та визначають характер руху персонажа, ворогів і елементів довкілля.

Гравітація задає вертикальне прискорення, яке впливає на висоту стрибка, швидкість падіння та відчуття “ваги” персонажа. Її значення підбирається експериментально: якщо гравітація занадто слабка, персонаж зависає в повітрі та втрачає відчуття інерції; якщо надмірна — усі рухи стають ривковими, що негативно впливає на ігрову динаміку.

$$y(t) = y_0 + v_{y_0} \cdot t + \frac{1}{2} \cdot g \cdot t^2 \tag{2.1}$$

Маса об’єктів визначає, як вони реагують на фізичні взаємодії. Легкі об’єкти швидко змінюють швидкість при зіткненнях, можуть відскакувати або переміщуватися при мінімальному впливі. Масивні об’єкти залишаються більш стабільними й вимагають більшої сили для зміни стану. У платформері це важливо

для правильного опрацювання рухомих платформ, керованих важелів чи об’єктів, які гравець може штовхати або активувати.

Тригери використовуються для запуску певних подій у конкретних точках сцени. Це можуть бути переходи між рівнями, поява нових ворогів, перемикання анімацій, увімкнення пасток або інтерактивних об’єктів. Тригери не мають фізичних зіткнень, але реагують на потрапляння в них персонажа чи іншого об’єкта. Завдяки тригерам забезпечується логічна структура рівня та зв’язність ігрових механік.

В свою чергу **система введення** визначає, як саме гравець керує персонажем та взаємодіє з інтерфейсом гри. У Unity доступні дві основні системи: класичний **Input Manager** і сучасна **Input System**. Новіший варіант дозволяє створювати більш гнучкі та масштабовані схеми керування, особливо якщо гра підтримує декілька типів пристроїв — клавіатуру, геймпад або мобільні елементи керування.

Налаштування **Input System** зазвичай передбачає створення окремих дій, таких як “рух”, “стрибок”, “атака” чи “взаємодія”, після чого їм прив’язуються відповідні клавіші або кнопки контролера. Крім того, система дозволяє калібрувати чутливість осей, встановлювати мінімальний поріг спрацювання або налаштувати окремі профілі для різних платформ. Це робить керування більш відточеним та інтуїтивним, що безпосередньо впливає на комфорт гравця.

Навіть при використанні більш простого Input Manager важливо дотримуватися структурованості: кожна дія повинна мати власну клавішу, а реакція на натискання має бути моментальною. Неправильні налаштування системи введення можуть призвести до затримки керування, подвійних спрацювань або некоректних дій персонажа, що негативно позначиться на ігровому процесі.

2.6 Сцени

Сцена в Unity є ключовою структурною одиницею проєкту, у межах якої розміщується все, що бачить і з чим взаємодіє гравець(див. рис. 2.4). Вона об’єднує ігрові об’єкти, логіку рівня, елементи керування та графічне оформлення. Фактично сцена виступає окремою частиною гри, наприклад, рівнем, меню або спеціальною

службовою локацією. Кожний компонент сцени має своє призначення та впливає на загальну взаємодію між об’єктами.

Камера визначає область, яку бачить гравець під час гри. Вона відтворює зображення на екрані та задає перспективу, глибину і рамки видимості. Її налаштування впливають на сприйняття простору, плавність руху, а також на загальну візуальну якість сцени.

Платформи формують геометрію рівня, визначають шляхи пересування персонажа та створюють перешкоди. Їхні фізичні властивості — колізії, розміри, матеріали — забезпечують коректну взаємодію з головним героєм і ворогами. Платформи є основою геймплею в платформерах, оскільки саме від них залежить складність проходження та структура карти.

Головний персонаж – центральний об’єкт сцени, через який гравець взаємодіє зі світом гри. До персонажа прикріплюються скрипти руху, анімації, фізичні компоненти та системи контролю. Від його поведінки залежить загальна керованість і комфорт гри. У сцені він повинен бути правильно розташований і мати доступ до всіх необхідних об’єктів та тригерів.

Вороги додають рівню динаміки й викликів. Кожен ворог має власну логіку пересування, атаки та взаємодії з гравцем. Їх розміщення впливає на темп проходження й складність рівня. Важливо, щоб вороги гармонійно поєднувалися з дизайном сцени та взаємодіяли з платформами та межами рівня.

Інтерфейс користувача в сцені містить такі елементи, як лічильники здоров’я, кількість зібраних ресурсів, меню паузи чи підказки. UI забезпечує гравцю доступ до важливої інформації та підсилює інтуїтивність взаємодії з грою. Елементи інтерфейсу існують у власному просторі — Canvas — але логічно належать до сцени як її невід’ємна частина.

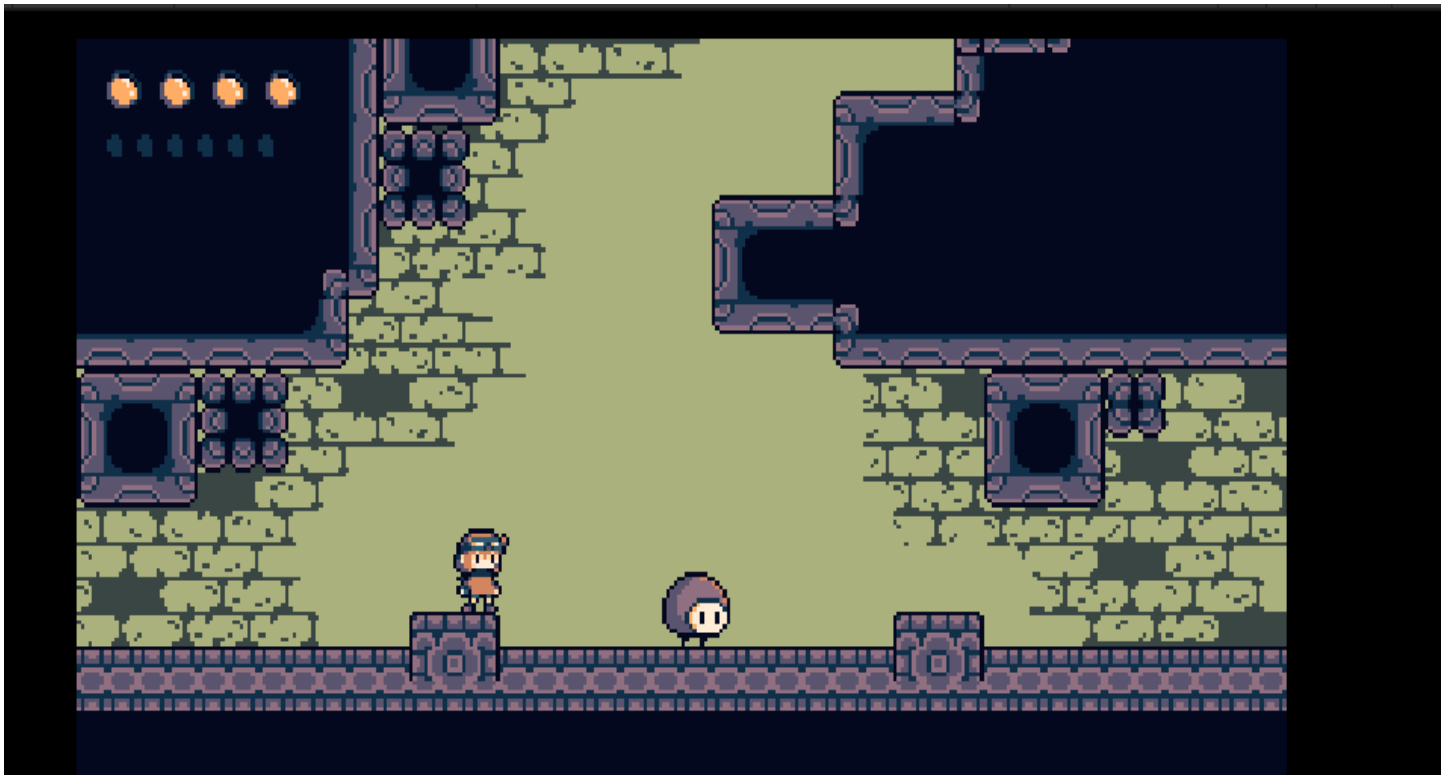

Рисунок 2.4 – Сцена

Висновки до розділу

Узагальнюючи результати розділу, можна зробити висновок, що ефективна розробка 2D платформера в середовищі Unity є неможливою без системного та комплексного підходу до роботи з інформаційними ресурсами. Усі розглянуті складові не існують ізольовано, а формують єдину інформаційну екосистему, яка підтримує процес проєктування, реалізації та подальшого вдосконалення ігрового продукту.

Поєднання офіційної технічної документації, навчальних матеріалів та практичних прикладів забезпечує методологічну основу розробки й мінімізує ризики критичних помилок на етапі реалізації. Залучення досвіду професійної спільноти та відкритих репозиторіїв дозволяє оптимізувати архітектурні рішення, підвищити якість коду та адаптувати проєкт до сучасних вимог і тенденцій ігрової індустрії.

Графічні, аудіо- та конфігураційні дані виступають не лише як допоміжні ресурси, а як інструменти формування цілісного користувацького досвіду. Їх узгоджене використання безпосередньо впливає на сприйняття гри, рівень занурення гравця та стабільність ігрового процесу. Структурна організація сцен, коректне

налаштування фізики й системи введення створюють передумови для керованого, передбачуваного та комфортного геймплею.

Таким чином, інформаційне забезпечення в межах даного проєкту виконує стратегічну функцію: воно не лише підтримує технічну реалізацію 2D платформера, а й визначає якість кінцевого продукту, його масштабованість і потенціал для подальшого розвитку.

РОЗДІЛ 3. МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

Математичне забезпечення програмного продукту є основою для побудови формалізованих моделей, що описують динаміку ігрових об’єктів, їхню поведінку та взаємодію в межах віртуального середовища. Для спеціальності галузі інформаційних технологій особливе значення має не лише використання готових фізичних моделей рушія, а й їх алгоритмічне осмислення, параметризація та адаптація до конкретних вимог геймплею.

У межах розробки 2D платформера математичне забезпечення охоплює моделі руху персонажа, механізми колізій, автомати станів, алгоритми штучного інтелекту та формалізовані залежності геймплейних параметрів. Застосування таких моделей дозволяє забезпечити відтворюваність результатів, контрольованість ігрового процесу та можливість подальшої оптимізації програмної реалізації.

3.1 Математична модель руху персонажа

Рух персонажа у двовимірному просторі описується з використанням базових положень кінематики, адаптованих до дискретної часової моделі. Координатний простір гри визначається декартовою системою координат, у якій положення об’єкта в момент часу t задається вектором:

$$\vec{p}(t) = (x(t), y(t)) \tag{3.1}$$

Швидкість руху персонажа визначається як похідна координат за часом:

$$\vec{v}(t) = \frac{d\vec{p}(t)}{dt} \tag{3.2}$$

Прискорення, у свою чергу, є похідною від швидкості:

$$\vec{a}(t) = \frac{d\vec{v}(t)}{dt} \tag{3.3}$$

У середовищі Unity обчислення здійснюються в дискретному вигляді з урахуванням фіксованого або змінного кроку часу Δt. Тоді рівняння оновлення швидкості та положення набувають вигляду:

$$\vec{v}_{t+1} = \vec{v}_t + \vec{a} \cdot \Delta t \tag{3.4}$$

$$\vec{p}_{t+1} = \vec{p}_t + \vec{v}_{t+1} \cdot \Delta t \tag{3.5}$$

Гравітаційне прискорення розглядається як стала величина, спрямована вздовж осі Y у від'ємному напрямку. Його значення визначає динаміку падіння та загальне сприйняття фізики гри.

Стрибок реалізується шляхом задання початкової вертикальної швидкості v_0 у момент відриву персонажа від поверхні. Подальший рух описується рівнянням рівноприскореного руху:

$$y(t) = y_0 + v_0 t - \frac{gt^2}{2} \tag{3.6}$$

Максимальна висота стрибка визначається з умови нульової вертикальної швидкості:

$$t_{max} = \frac{v_0}{g} \tag{3.7}$$

$$h_{max} = \frac{v_0^2}{2g} \tag{3.8}$$

Такий підхід дозволяє точно керувати параметрами стрибка та адаптувати поведінку персонажа до вимог ігрового дизайну[10].

3.2 Модель взаємодії з середовищем

Взаємодія персонажа з об'єктами сцени формалізується через математичні моделі колізій. Для оптимізації обчислень застосовуються спрощені геометричні примітиви, що забезпечують достатню точність при мінімальних витратах ресурсів.

Однією з базових моделей є осьово-орієнтований обмежувальний прямокутник AABB. Для двох об'єктів A та B зіткнення фіксується за умови:

$$A_{minX} < B_{maxX} \wedge A_{maxX} > B_{minX} \tag{3.9}$$

$$A_{minY} < B_{maxY} \wedge A_{maxY} > B_{minY} \tag{3.10}$$

Променеве трасування (Raycast) використовується для перевірки контакту з поверхнею або перешкодами. Промінь задається параметричним рівнянням:

$$\vec{r}(t) = \vec{o} + \vec{d} \cdot t, \quad t \geq 0 \tag{3.11}$$

де o — початкова точка, d — напрям вектора. Перетин променя з колайдером визначає наявність фізичної взаємодії[11].

3.3 Автомати станів

Для формалізації логіки поведінки ігрових об’єктів застосовується модель скінченного автомата станів. FSM описується кортежем:

$$FSM = \langle S, E, T, s_0 \rangle \tag{3.12}$$

де S — множина станів, E — множина подій, T — функція переходів, s_0 — початковий стан.

Для головного персонажа такими станами можуть бути: спокій, рух, стрибок, атака, отримання шкоди. Перехід між станами відбувається при виконанні заданих логічних умов, що забезпечує структурованість і передбачуваність поведінки.

3.4 Алгоритми штучного інтелекту

Штучний інтелект ворогів реалізується на основі простих математичних і логічних моделей, що відповідають вимогам реального часу. Патрулювання описується циклічним переміщенням між граничними координатами:

$$x(t+1) = x(t) + v \cdot dir \cdot \Delta t \tag{3.13}$$

Режим переслідування активується на основі обчислення евклідової відстані між ворогом і персонажем:

$$d = \sqrt{(x_p - x_e)^2 + (y_p - y_e)^2} \tag{3.14}$$

Якщо d менше за порогове значення, ворог змінює стан та коригує напрям руху в бік гравця.

3.5 Математичні моделі геймплею

Механізм нанесення шкоди формалізується як функція, що враховує атакувальні та захисні параметри:

$$D = A \cdot K - R \tag{3.15}$$

де A — базова сила атаки, K — коефіцієнт модифікації, R — рівень захисту.

Таймінги атак, відновлення та анімацій ґрунтуються на порівнянні поточного часу з контрольними значеннями:

$$t_{now} - t_{last} \geq t_{cooldown} \tag{3.16}$$

Це дозволяє регулювати темп ігрового процесу та уникати некоректних повторень дій.

3.6 Алгоритмічні схеми логіки гри

Для узагальнення математичних і логічних залежностей використовуються діаграми станів та алгоритмічні схеми. Вони подають поведінку ігрових об’єктів у вигляді графів, що дозволяє формалізувати переходи, спростити тестування та підвищити масштабованість програмної системи.

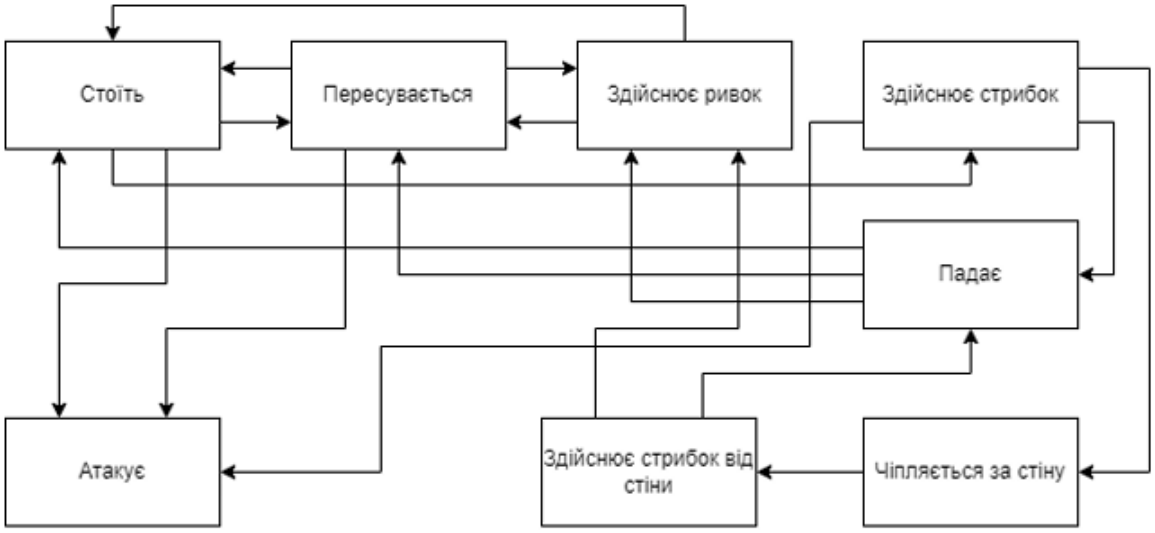

Рисунок 3.1 – Діаграма станів гравця

З точки зору інформаційних технологій, такий підхід забезпечує чітке відокремлення логіки, спрощує супровід програмного коду та створює основу для подальшого розвитку проєкту[12].

Висновки до розділу

У третьому розділі було сформовано математичне підґрунтя, необхідне для реалізації 2D платформера в середовищі Unity з позицій інформаційних технологій та алгоритмічного моделювання. Розглянуті математичні моделі дозволяють формалізувати ключові процеси ігрової системи та забезпечити керованість її поведінки в реальному часі.

Побудова моделі руху персонажа на основі рівнянь кінематики дала змогу описати зміну положення, швидкості та прискорення у дискретному часовому просторі, що відповідає принципам роботи ігрового рушія. Формалізація стрибка через аналітичні залежності між початковою швидкістю та гравітаційним прискоренням створює можливість точно налаштовувати динаміку руху й досягати прогнозованої поведінки персонажа.

Математичний опис взаємодії з ігровим середовищем, зокрема використання моделей AABB та променевого трасування, дозволяє ефективно визначати зіткнення та контакти з поверхнями при мінімальних обчислювальних витратах. Це є важливим з точки зору оптимізації продуктивності, що має особливе значення для інтерактивних систем реального часу.

Застосування скінченних автоматів станів забезпечує структурований підхід до опису логіки поведінки персонажа та ворогів. Така формалізація спрощує проєктування складних сценаріїв взаємодії, підвищує читабельність алгоритмів і створює основу для подальшого масштабування ігрової логіки.

Розглянуті алгоритми штучного інтелекту демонструють можливість використання простих математичних моделей для реалізації базових форм поведінки, таких як патрулювання та переслідування. Використання відстаней, порогових значень і часових обмежень дозволяє створити переконливу поведінку ворогів без надмірного ускладнення обчислень.

Окрему увагу приділено математичному опису геймплейних механік, зокрема нанесенню шкоди та керуванню таймінгами дій. Формалізація цих процесів забезпечує баланс ігрового процесу та дозволяє точно регулювати складність і темп проходження.

Таким чином, математичне забезпечення, розроблене в межах даного розділу, виступає фундаментом для подальшої програмної реалізації проєкту. Воно поєднує теоретичні моделі з практичними алгоритмами, забезпечує логічну цілісність ігрової системи та створює умови для ефективної реалізації 2D платформера з використанням сучасних підходів інформаційних технологій.

РОЗДІЛ 4. ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

Ігровий рушій є базовим програмним компонентом, на якому будується будь-який сучасний ігровий застосунок. Саме він забезпечує реалізацію технічної частини гри та виступає проміжною ланкою між апаратними ресурсами комп’ютера і прикладною логікою, створеною розробником. Використання готового рушія суттєво спрощує процес розробки, оскільки надає набір уже реалізованих механізмів і бібліотек, а також зручний графічний інтерфейс для роботи зі сценами, об’єктами та ресурсами.

Однією з ключових переваг ігрових рушіїв є підтримка мультиплатформенності. Завдяки цьому один і той самий програмний код може бути адаптований для запуску на різних платформах, зокрема персональних комп’ютерах, ігрових консолях та мобільних пристроях. Такий підхід зменшує витрати часу й ресурсів на розробку окремих версій програмного продукту та підвищує його доступність для кінцевих користувачів.

Ігровий рушій охоплює більшість ключових підсистем застосунку, що відповідає класичним підходам до побудови архітектури ігрових рушіїв [13]. До них належать модулі рендерингу графіки, системи анімації, обробки звуку, фізичного моделювання, керування ресурсами, а також засоби підключення та виконання програмних скриптів. Окрему роль відіграють інструменти для реалізації ігрового штучного інтелекту, що дозволяють описувати поведінку персонажів і об’єктів на основі алгоритмічних моделей та автоматів станів.

Важливою особливістю сучасних рушіїв є можливість їх повторного використання в різних проєктах. Один і той самий програмний каркас може застосовуватись для створення ігор різних жанрів або навіть програм з іншою функціональною спрямованістю, що робить рушій універсальним програмним інструментом.

Поняття ігрового рушія сформувалося на початку 1990-х років, коли зростання популярності тривимірних ігор, зокрема шутерів від першої особи, призвело до ускладнення програмної частини ігрових проєктів. У цей період розробники почали

усвідомлювати доцільність відокремлення базових технологічних компонентів від конкретного ігрового контенту. Це дало змогу повторно використовувати вже створені програмні рішення, ліцензувати їх та застосовувати в нових проєктах, скорочуючи витрати на розробку і терміни виходу продукту на ринок.

З часом такі програмні рішення еволюціонували у повноцінні комерційні ігрові рушії, які стали доступними для сторонніх розробників. До найвідоміших прикладів належать Unity, Unreal Engine, CryEngine, Frostbite та Godot. Компанії-розробники рушіїв отримують прибуток за рахунок ліцензування своїх технологій або відрахувань з продажів ігор, створених на їхній основі, що стимулює подальший розвиток і вдосконалення цих програмних платформ.

З технічної точки зору більшість ігрових рушіїв побудовані за компонентним принципом. Такий підхід дозволяє замінювати або розширювати окремі підсистеми без необхідності повної переробки архітектури. Наприклад, стандартні фізичні або звукові модулі можуть бути доповнені сторонніми бібліотеками, такими як Havok, PhysX або FMOD, залежно від вимог конкретного проєкту.

Висока гнучкість ігрових рушіїв зумовила їх використання не лише в ігровій індустрії. Завдяки можливості обробки тривимірної графіки в режимі реального часу, такі системи активно застосовуються у сфері архітектурної візуалізації, створення рекламних матеріалів, інтерактивних презентацій та навіть у кіновиробництві.

Загальною технічною основою ігрових рушіїв є використання графічних API, таких як Direct3D або OpenGL, а також низькорівневих бібліотек, що забезпечують ефективний та апаратно незалежний доступ до ресурсів комп’ютерної системи. Це дозволяє досягати високої продуктивності та стабільної роботи програмного продукту на різних конфігураціях обладнання.

Серед найбільш поширених ігрових рушіїв, які використовуються як у навчальних, так і в комерційних проєктах, можна виділити GameMaker: Studio, Unreal Engine та Unity, кожен з яких має власні особливості, переваги та сферу застосування.

Вибір програмного забезпечення здійснювався не лише з позиції зручності, а передусім з огляду на технічні характеристики, архітектурні можливості та відповідність сучасним підходам до розробки інтерактивних систем.

Особливу увагу було приділено можливості чіткої реалізації моделей руху, кінцевих автоматів та алгоритмів поведінки, а також підтримці масштабування й модифікації програмної архітектури без суттєвого перепроєктування.

4.1 Технічне обґрунтування вибору та опис ігрового рушія Unity

Unity обрано як базовий рушій з огляду на його архітектурну універсальність та оптимальну підтримку двовимірної фізики. На відміну від вузькоспеціалізованих 2D-фреймворків, Unity надає повноцінну систему управління сценами, компонентами та ресурсами, що дозволяє розглядати розробку гри як інженерний процес побудови складної програмної системи.

Ключовим технічним аргументом є компонентно-орієнтована модель, яка фактично реалізує принципи слабкої зв’язаності та повторного використання коду. Логіка руху, обробка введення, фізична взаємодія та анімація реалізуються у вигляді незалежних компонентів, що дозволяє локалізувати зміни та зменшити кількість побічних ефектів при модифікації функціоналу.

Фізичний рушій Unity для 2D базується на бібліотеці Box2D, яка використовує дискретну інтеграцію рівнянь руху та оптимізовані алгоритми виявлення зіткнень.

Наявність величезної кількості офіційних та спільнотних навчальних матеріалів, форумів, туторіалів та онлайн-курсів значно спрощує вирішення проблем та доступ до готових рішень. **Unity Asset Store** є цінним джерелом готових 2D ассетів (графіка, анімації, звуки, скрипти) та плагінів, що може значно прискорити прототипування та розробку.

Окремою перевагою є підтримка різних типів колізійних перевірок, зокрема AABB та трасування променів (Raycast), які застосовуються для визначення контакту з поверхнею, виявлення перешкод та контролю логіки поведінки персонажа. Це забезпечує ефективне поєднання математичних алгоритмів і практичної реалізації ігрової механіки.

Графічний інтерфейс ігрового рушія Unity побудований за модульним принципом і складається з набору взаємопов’язаних вікон, кожне з яких виконує окрему функцію в процесі розробки програмного продукту. Такий підхід дозволяє

організувати робочий простір максимально зручно та адаптувати його під конкретні потреби розробника.

У стандартній конфігурації інтерфейсу центральне місце займає вікно сцени, яке використовується для безпосередньої роботи з ігровим світом. У цьому вікні розробник може розміщувати об’єкти, змінювати їх положення, масштаб та орієнтацію, а також переглядати сцену в різних режимах відображення. Вікно сцени забезпечує візуальний контроль над просторовою організацією рівнів і дозволяє оперативно оцінювати результати змін.

Вікно ієрархії проєкту відображає структуру поточної сцени у вигляді дерева об’єктів. Воно демонструє взаємозв’язки між елементами ігрового світу, зокрема вкладеність об’єктів та їх батьківсько-дочірні залежності. Така структура значно спрощує керування складними сценами, що містять велику кількість компонентів, та дозволяє швидко знаходити необхідні елементи.

Окрему роль у роботі з рушієм відіграє вікно інспектора. Воно призначене для перегляду та редагування властивостей вибраного об’єкта або ресурсу. Саме через інспектор здійснюється налаштування компонентів, таких як трансформація, колайдери, фізичні параметри, аніматори та користувацькі скрипти. Завдяки цьому вікну розробник може детально керувати поведінкою об’єктів без необхідності постійного редагування програмного коду.

Вікно огляду ресурсів (Project) використовується для управління всіма файлами, що входять до складу проєкту. У ньому зберігаються скрипти, текстури, аудіофайли, анімації, префаби та інші ресурси. Дане вікно дозволяє впорядковувати матеріали проєкту за папками, що позитивно впливає на читабельність структури та зручність подальшого супроводу програмного продукту.

Окрім основних вікон, інтерфейс Unity може бути доповнений іншими допоміжними панелями, такими як вікно перегляду гри (Game), консоль для відображення повідомлень і помилок, а також інструменти для роботи з анімаціями, освітленням та профілюванням продуктивності (див. рис. 4.1). Гнучка система налаштування дозволяє змінювати розташування вікон або створювати власні робочі макети, що підвищує ефективність процесу розробки.

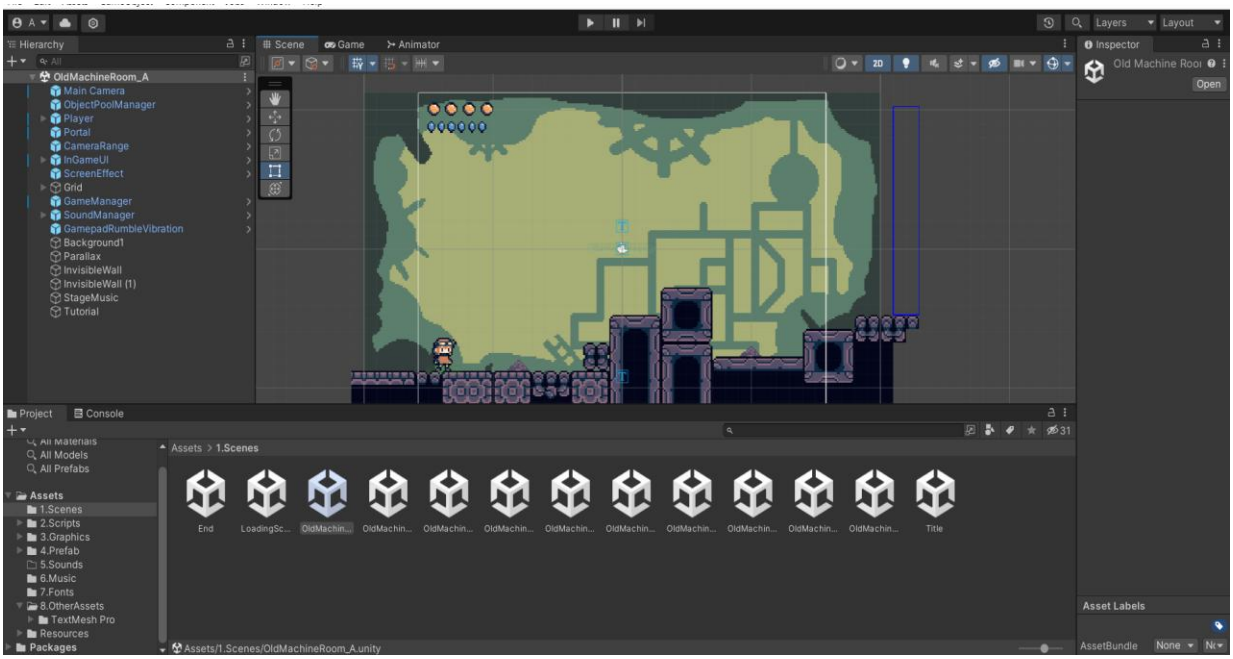

Рисунок 4.1 – Інтерфейс ігрового рушія Unity

Завдяки такій організації інтерфейсу Unity забезпечує зрозумілий і логічно структурований робочий простір, який поєднує візуальні та програмні інструменти в єдине середовище розробки.

4.2 Обґрунтування використання мови програмування C#

Мова програмування C# була обрана з огляду на її чітку об’єктно-орієнтовану модель та строгий контроль типів, що є критично важливим при реалізації складних систем з великою кількістю взаємопов’язаних об’єктів. У контексті дипломної роботи C# дозволяє формалізувати ігрову логіку у вигляді класів, що відповідають математичним моделям і автоматам станів.

З технічної точки зору важливою є підтримка подій та делегатів, які дозволяють реалізовувати реактивну модель поведінки ігрових об’єктів. Це спрощує синхронізацію між різними підсистемами гри, такими як керування, фізика та анімація, без створення жорстких залежностей.

Також C# забезпечує достатній рівень продуктивності для 2D ігор, використовуючи керовану пам’ять і механізм збирання сміття, що знижує ймовірність критичних помилок, пов’язаних з некоректним управлінням ресурсами.

Навігація по коду дозволяє розробнику легко переходити до визначень методів, класів, змінних або інтерфейсів одним кліком. Функції "Go to Definition" та "Find All References" є незамінними для розуміння великих кодових баз та швидкої модифікації функціоналу, особливо при роботі з архітектурою Unity, де скрипти взаємодіють з численними компонентами.

4.3 Технічні причини використання Visual Studio Code

Visual Studio Code використовується як основне середовище роботи з кодом з огляду на його модульну архітектуру та ефективну інтеграцію з екосистемою .NET. На відміну від повнофункціональних IDE, VS Code не нав’язує жорсткої структури проєкту, що дозволяє гнучко адаптувати процес розробки до особливостей Unity-проєкту.

Його популярність серед розробників Unity зростає завдяки ефективній інтеграції та широким можливостям розширення. VS Code інтегрується з Unity не просто як текстовий редактор, а як повноцінний інструмент розробника завдяки спеціальним розширенням, зокрема "C# Dev Kit" та "Unity Tools". Ці розширення трансформують VS Code у високоефективне середовище для розробки ігор, надаючи низку критично важливих можливостей

Значною технічною перевагою є наявність статичного аналізу коду та системи автодоповнення, які допомагають підтримувати коректність викликів Unity API та зменшують кількість синтаксичних і логічних помилок. Механізми відлагодження дозволяють виконувати покроковий аналіз виконання ігрових сценаріїв, що є особливо важливим при перевірці роботи кінцевих автоматів і складних умов переходів між станами.

4.4 Технічна доцільність використання графічних і звукових редакторів

Використання GIMP та Aseprite зумовлене необхідністю контролю якості графічних ресурсів та їх відповідності вимогам рушія Unity. З технічної точки зору важливою є можливість точного керування розмірами спрайтів, прозорістю та

структурою спрайт-листів, оскільки це безпосередньо впливає на коректність анімацій і продуктивність гри.

Audacity використовується для підготовки аудіоконтенту з урахуванням технічних обмежень ігрового рушія. Оптимізація тривалості, частоти дискретизації та формату файлів дозволяє зменшити навантаження на пам’ять і забезпечити стабільну роботу аудіопідсистеми.

4.5 Розробка механік гравця

Розробка ігрового застосунку на початковому етапі була зосереджена на проєктуванні та реалізації базових механік керування персонажем, оскільки саме вони формують основу ігрового процесу та безпосередньо впливають на комфорт і динаміку взаємодії гравця з ігровим середовищем. Коректна робота цих механік є критично важливою для подальшого розвитку проєкту та впровадження складніших ігрових систем.

У межах цього етапу необхідно було реалізувати комплекс функціональних можливостей, що забезпечують повноцінну керованість персонажа, а саме: стандартне горизонтальне переміщення (біг), виконання стрибків, взаємодію зі стінами у вигляді захоплення, можливість виконання відштовхування від стін, короткочасний ривок для швидкої зміни позиції, а також систему атаки для взаємодії з ворогами або об’єктами навколишнього середовища. Кожна з перелічених механік вимагала точного налаштування фізичних параметрів, обробки вводу користувача та синхронізації з анімаційними станами персонажа.

З метою підвищення читабельності коду, спрощення тестування та забезпечення можливості подальшого масштабування було прийнято рішення розділити функціональність керування гравцем на кілька логічно незалежних скриптів. Такий підхід відповідає принципам об’єктно-орієнтованого програмування та сприяє зменшенню зв’язності між компонентами системи[14].

У результаті архітектура керування персонажем була реалізована у вигляді трьох основних скриптів:

Player – клас, що відповідає за високорівневу логіку поведінки гравця. У ньому зосереджені методи, які описують можливі дії персонажа, такі як рух, стрибки, ривок і атака, а також керування станами анімацій. Цей клас виступає центральною ланкою між вводом користувача та фізичним переміщенням об’єкта в ігровому просторі.

GameInput (InputHandler) – статичний клас, призначений для обробки користувацького вводу. Він відповідає за зчитування сигналів з клавіатури або інших пристроїв керування та перетворення їх у зрозумілі для ігрової логіки команди[16]. Використання окремого класу для обробки вводу дозволяє легко змінювати схему керування або адаптувати її під інші платформи без внесення змін у код, що відповідає за поведінку персонажа.

ActorController – клас, який реалізує низькорівневу логіку переміщення та взаємодії з ігровим середовищем. Він відповідає за фізичні розрахунки, обробку зіткнень, перевірку контакту з поверхнями, а також коректну зміну координат персонажа у просторі сцени. Саме в цьому компоненті зосереджена робота з фізичним рушієм Unity, зокрема з системою колізій та гравітації.

Такий поділ відповідальностей між окремими скриптами дозволив створити гнучку та зрозумілу архітектуру керування гравцем, що полегшує подальший розвиток проєкту, впровадження нових механік і оптимізацію вже реалізованого функціоналу. Крім того, обрана структура значно знижує ризик виникнення помилок та спрощує процес налагодження.

4.5.1 Впровадження переміщення

У процесі проєктування ігрової механіки було обрано підхід, орієнтований на реалізацію фізично обґрунтованої системи руху персонажа. Такий вибір зумовлений необхідністю досягнення передбачуваної та інтуїтивно зрозумілої поведінки гравця у просторі ігрового рівня, що безпосередньо впливає на якість геймплею та відчуття керування.

Ігровий рушій Unity надає кілька способів реалізації переміщення об’єктів у сцені. Найпростішим з них є безпосередня зміна параметрів компонента Transform, однак такий підхід не враховує дії сил, інерції та зіткнень відповідно до законів

фізики. Альтернативним і більш коректним з точки зору моделювання руху є використання фізичного компонента Rigidbody2D, який інтегрує об’єкт у систему двовимірної фізики рушія та дозволяє застосовувати вбудовані механізми обробки гравітації, прискорення та взаємодії з іншими об’єктами.

З огляду на ці особливості, для ігрового об’єкта, що представляє персонажа, було додано компонент Rigidbody2D. Це дало змогу реалізувати рух, заснований на прикладенні сил і зміні швидкості, а не на прямій зміні координат. Для забезпечення коректної взаємодії з ігровим середовищем також було використано компонент Capsule Collider 2D, який надає об’єкту фізичну форму у вигляді капсули. Така форма є оптимальною для персонажів платформерів, оскільки вона зменшує кількість небажаних застрягань на краях платформ та забезпечує стабільні колізії під час руху і стрибків.

Після додавання фізичних компонентів стало можливим перейти до розробки користувацьких скриптів, що керують поведінкою гравця. У середовищі Unity такі скрипти також виступають як компоненти та можуть бути безпосередньо прив’язані до ігрового об’єкта, що дозволяє гнучко налаштовувати параметри руху без змін у програмному коді.

Для реалізації механіки бігу було визначено набір основних параметрів, які описують фізичні властивості руху персонажа. До них належать: максимальна швидкість пересування (Move Speed), коефіцієнти прискорення та гальмування під час руху по землі (Ground Move Acceleration, Ground Move Deceleration), аналогічні параметри для руху в повітрі (Air Move Acceleration, Air Move Deceleration), а також параметр затримки відображення візуального ефекту пилу під час бігу (Run Dust Effect Delay).

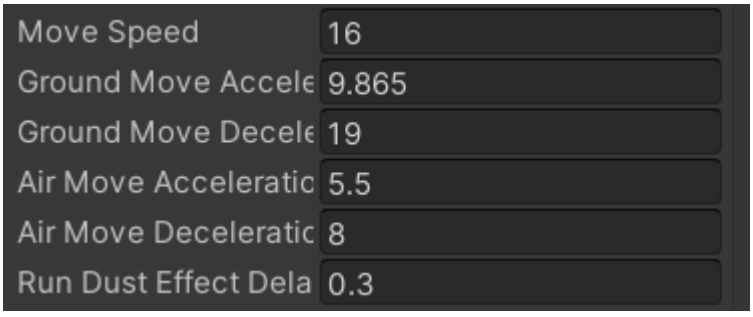

Рисунок 4.2 – Властивості переміщення компонента Player

Параметр максимальної швидкості визначає верхню межу горизонтального руху персонажа, тоді як коефіцієнти прискорення та уповільнення впливають на плавність розгону та зупинки. Розділення цих значень для станів «на землі» та «у повітрі» дозволяє досягти більш реалістичної поведінки, за якої керування персонажем у стрибку є менш чутливим, ніж під час руху по поверхні. Додатковий параметр, пов’язаний із візуальним ефектом, не впливає на фізику безпосередньо, проте підсилює відчуття руху та робить ігровий процес більш виразним.

У сукупності використання фізичних компонентів Unity та гнучко налаштованих параметрів руху дозволило сформувати достовірну і стабільну систему переміщення гравця, яка відповідає жанровим особливостям 2D платформера та створює надійну основу для подальшого розширення ігрових механік.

4.5.2 Впровадження стрибків

Механіка стрибка є однією з базових складових ігрового процесу в 2D платформері, оскільки саме вона визначає, наскільки зручно гравцеві переміщатися рівнями та взаємодіяти з навколишнім середовищем. Під час розробки даного проєкту основна увага приділялася не лише фізичній коректності стрибка, а й його відчуттю з боку користувача.

Стрибок реалізовано на основі фізичного компонента **Rigidbody2D**, що дозволяє використовувати стандартні механізми рушія Unity для роботи з гравітацією та силами. У момент натискання кнопки стрибка до персонажа прикладається вертикальний імпульс, величина якого визначається параметром **Jump Force**. Саме цей параметр відповідає за початкову різкість і «силу» стрибка, тобто за те, наскільки швидко персонаж починає рух угору.

Для контролю максимальної висоти основного стрибка використовується параметр **Jump Height**. Його значення підбиралося експериментальним шляхом з урахуванням геометрії рівнів, висоти платформ та загальної динаміки гри. Це дозволило уникнути ситуацій, коли персонаж стрибає надто високо або, навпаки, не може подолати необхідні перешкоди.

З метою урізноманітнення геймплею та підвищення мобільності персонажа було реалізовано подвійний стрибок. За його поведінку відповідає параметр **Double Jump Height**, який задає додаткову висоту під час другого стрибка в повітрі. Значення цього параметра є меншим за основну висоту стрибка, що дозволяє зберегти баланс та запобігти надмірному спрощенню проходження рівнів.

Окрему увагу приділено покращенню «відчуття» керування за рахунок використання так званого **coyote time**. Параметр **Max Coyote Time** задає максимальний проміжок часу після втрати контакту з землею, протягом якого гравець все ще може ініціювати стрибок. Такий підхід компенсує неточності реакції користувача та особливості фізичного оновлення кадрів, роблячи керування більш прощаючим і комфортним, особливо під час швидкого переміщення по платформах.Крім того, реалізовано буферизацію введення стрибка за допомогою параметра **Max Time Buffer**. Якщо гравець натискає кнопку стрибка трохи раніше моменту приземлення, команда зберігається і виконується автоматично одразу після контакту з поверхнею. Такий підхід робить ігровий процес плавнішим і зменшує кількість випадкових помилок під час швидкого проходження рівнів.

Рисунок 4.3 – Властивості стрибка компонента Player

У результаті поєднання зазначених параметрів вдалося реалізувати систему стрибка, яка виглядає природно, добре піддається налаштуванню та позитивно впливає на загальне сприйняття гри. Механіка не перевантажена зайвою складністю, але водночас забезпечує достатню глибину для побудови цікавих і різноманітних ігрових ситуацій.

4.5.3 Впровадження механік стрибків по стінах

Реалізація стрибків по стінах дозволяє гравцеві долати складні вертикальні ділянки рівнів, уникати перешкод і комбінувати різні типи руху, що позитивно впливає на загальну варіативність геймплею.

Для впровадження даного функціоналу було розроблено окрему логіку визначення контакту персонажа зі стіною. За допомогою перевірки колізій у горизонтальному напрямку визначається момент, коли гравець торкається вертикальної поверхні під час падіння або руху. У такому стані активується режим ковзання по стіні, який обмежує вертикальну швидкість персонажа.

Параметр **Wall Sliding Speed** відповідає за швидкість вертикального спуску вздовж стіни. Зменшення цього значення дозволяє зробити падіння контрольованим і передбачуваним, надаючи гравцеві додатковий час для прийняття рішень. Це особливо важливо на рівнях з підвищеною складністю та великою кількістю вертикальних перешкод.

Механіка стрибка від стіни активується у випадку натискання кнопки стрибка під час ковзання. Висота такого стрибка регулюється параметром **Wall Jump Height**, який визначає вертикальну складову імпульсу. Це значення підбирається таким чином, щоб персонаж міг досягати наступних платформ, але водночас не порушував баланс рівня.

Горизонтальний напрямок та дальність відштовхування від стіни задаються параметром **Wall Jump X Force**. Він визначає інтенсивність руху по осі X під час стрибка і забезпечує чітке віддалення персонажа від поверхні, з якої виконується відштовхування. Завдяки цьому усувається ризик повторного миттєвого зіткнення зі стіною та створюється відчуття контрольованого і плавного руху.

Для покращення візуального сприйняття механіки було додано параметр **Wall Slide Dust Effect**, який відповідає за відображення спеціального ефекту під час ковзання по стіні. Анімація частинок або пилу активується лише у відповідному стані, що підсилює зворотний зв’язок для гравця та робить взаємодію з середовищем більш наочною і виразною.

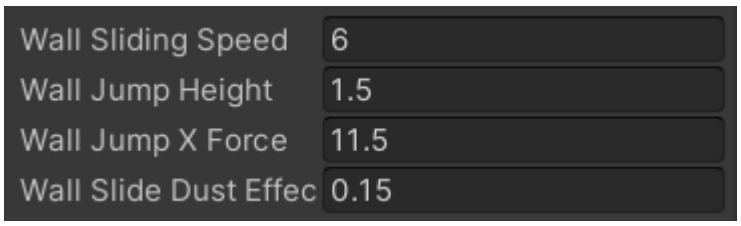

Рисунок 4.4 – Властивості стрибка по стінах компонента Player

У підсумку, реалізація стрибків та ковзання по стінах дозволила значно розширити можливості переміщення персонажа, зробити ігровий процес більш динамічним і технічно насиченим. Дана механіка органічно інтегрується в загальну систему руху та відкриває додаткові можливості для проєктування складних і цікавих рівнів.

4.6 Розробка та підбір графіки персонажів

Робота над візуальною складовою персонажів розпочалася з етапу формування загального художнього стилю гри та визначення вимог до зовнішнього вигляду ігрових об’єктів. Основною метою було забезпечити чітку візуальну ідентифікацію персонажів на фоні ігрового середовища, а також узгодженість графіки з динамічним характером геймплею 2D платформера.

На відміну від повністю авторського створення графіки, у межах даної роботи було прийнято рішення використовувати готові графічні ресурси. Основними джерелами ассетів стали **Unity Asset Store**, а також відкриті бібліотеки графіки з дозволеною ліцензією для некомерційного та навчального використання. Такий підхід дозволив суттєво скоротити час розробки, зосередивши увагу на програмній реалізації механік та логіці гри.

Обрані спрайти персонажів попередньо аналізувалися за такими критеріями:

- відповідність загальному стилю гри;
- читабельність силуету персонажа;
- наявність базового набору анімацій;
- можливість подальшої адаптації та масштабування.

Отримані графічні матеріали використовувалися як основа для подальшої інтеграції анімацій. Більшість спрайтів уже містили готові спрайт-листи з

покадровими анімаціями, що значно спростило їх використання в ігровому рушії Unity. За необхідності окремі елементи коригувалися або доповнювалися за допомогою графічного редактора Aseprite, який використовувався для перегляду, мінімального редагування та аналізу покадрових анімацій.

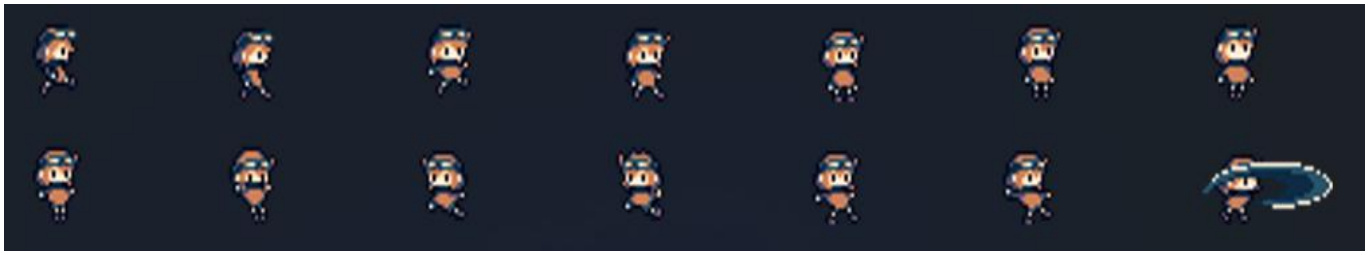

Рисунок 4.5 – Спрайтовий лист анімації персонажа

Для персонажів гри використовувався стандартний набір анімацій, необхідний для реалізації основних ігрових механік:

- стан спокою (idle);
- рух або біг;
- стрибок;
- взаємодія зі стіною;
- ривок;
- атака;
- допоміжні анімації та візуальні ефекти (частинки приземлення, деформація персонажа під час контакту із землею).

Після підбору відповідних ассетів усі спрайтові листи були імпортовані до проєкту Unity з попереднім налаштуванням параметрів. Оскільки спрайт-лист являє собою єдине зображення, що містить декілька кадрів анімації, необхідно було виконати його коректну нарізку. Для цього у властивостях текстури задавався режим **Sprite Mode – Multiple**, після чого за допомогою вбудованого Sprite Editor виконувалося автоматичне або ручне розбиття на кадри.

Додатково налаштовувалися параметри фільтрації, стиснення та максимального розміру текстури. Це дозволило уникнути появи графічних артефактів, розмиття зображень та небажаних спотворень під час масштабування спрайтів у сцені.

За аналогічним принципом було підібрано та інтегровано графіку для інших ігрових персонажів, такі як наземні та повітряні вороги. Для кожного з них використовувалися окремі спрайтові набори з відповідними анімаціями, що підкреслюють їх функціональну роль у грі.

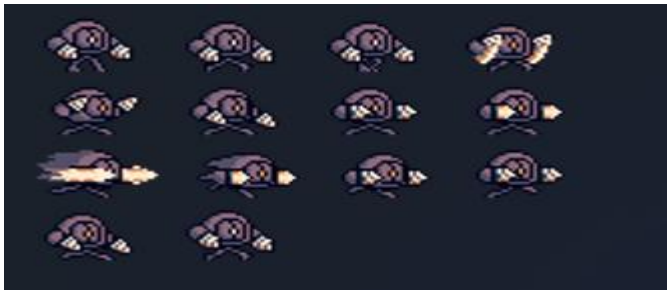

Рисунок 4.6 – Спрайт-лист ворога

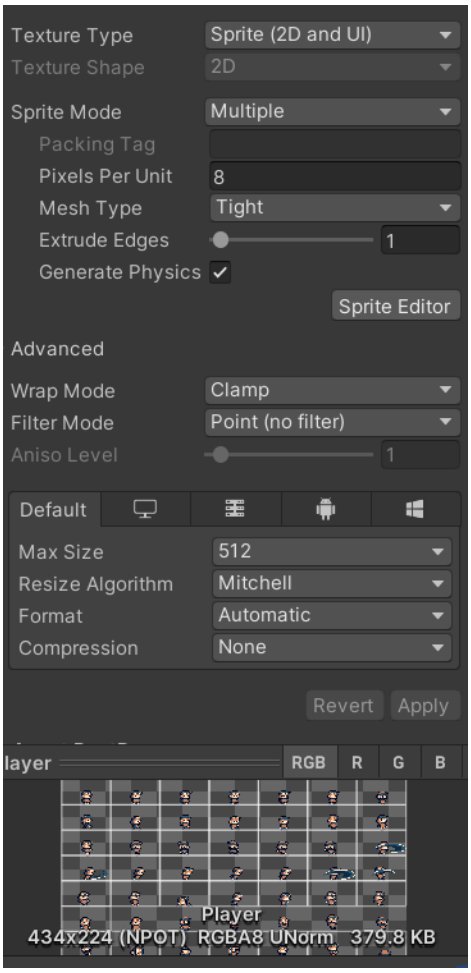

Рисунок 4.7 – Налаштування параметрів спрайтового листа в Unity

Таким чином, використання готових графічних ресурсів у поєднанні з можливостями ігрового рушія Unity дозволило створити візуально цілісну та технічно коректну систему персонажів без втрати якості та керованості проєкту.

4.7 Інтеграція аудіо

Звукове оформлення відіграє важливу роль у сприйнятті ігрового процесу, адже саме звук допомагає гравцеві глибше зануритися в атмосферу гри. Навіть за наявності якісної графіки відсутність аудіосупроводу робить ігровий світ «порожнім» та менш переконливим, що відповідає базовим принципам звукового дизайну інтерактивних систем [18].

У розробленому ігровому застосунку реалізовано систему звуків навколишнього середовища, яка формує загальний настрій локацій. Зокрема, використано ефекти, що імітують шум вітру, звуки печерного простору, потріскування багаття та нічні фонові звуки. Окрім атмосферних елементів, додано звуковий супровід ключових дій персонажа: стрибків, атак, отримання ушкоджень і моменту загибелі. Такий підхід забезпечує чіткий зворотний зв’язок і робить керування персонажем більш відчутним.

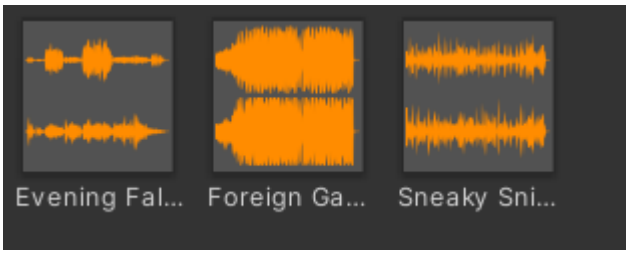

Рисунок 4.8 – Аудіо файли

Для зручності користувача в грі передбачено можливість регулювання гучності. У меню налаштувань реалізовано повзунок, за допомогою якого гравець може змінювати рівень звуку в режимі реального часу. Обране значення передається до AudioMixer — стандартного інструмента Unity, що дозволяє гнучко керувати звуковими каналами та підтримувати збалансоване звучання.

Реалізована аудіосистема не лише підсилює атмосферу гри, а й підвищує комфорт взаємодії з нею, роблячи ігровий процес більш емоційним та цілісним.

Висновки до розділу

У даному розділі було детально розглянуто технічні та програмні аспекти розробки ігрового застосунку жанру 2D платформер. Особливу увагу приділено обґрунтуванню вибору основних інструментів розробки, зокрема ігрового рушія

Unity, мови програмування C# та допоміжних середовищ і редакторів, що забезпечують повноцінний цикл створення інтерактивного продукту.

Проаналізовано архітектурні можливості Unity та показано, що компонентно-орієнтований підхід рушія дозволяє будувати масштабовану й гнучку структуру ігрової логіки. Використання фізичного рушія Box2D у поєднанні з налаштовуваними параметрами руху дало змогу реалізувати достовірну та передбачувану систему керування персонажем, що відповідає жанровим вимогам платформерів і забезпечує комфортне сприйняття геймплею.

У межах розділу було реалізовано та описано ключові механіки гравця, включно з бігом, стрибками, подвійним стрибком, ковзанням по стінах і стрибками від стін. Запропоновані рішення ґрунтуються на поєднанні фізичних моделей і логічних обмежень, що дозволяє досягти балансу між реалістичністю та ігровою динамікою. Використання таких підходів, як coyote time та буферизація введення, суттєво підвищує зручність керування і зменшує вплив дрібних помилок гравця.

Також у розділі розглянуто питання інтеграції графічних і звукових ресурсів. Використання готових ассетів у поєднанні з можливостями Unity дозволило створити візуально цілісне середовище без перевантаження процесу розробки. Реалізована аудіосистема забезпечує атмосферність ігрового процесу та надає користувачу засоби персоналізації звучання через меню налаштувань.

У підсумку можна зазначити, що результати, отримані в цьому розділі, формують міцну технічну основу для подальшого розвитку ігрового проєкту. Реалізовані рішення є гнучкими, зрозумілими з точки зору архітектури та придатними для розширення функціональності, що підтверджує доцільність обраних технологій і підходів до розробки.

РОЗДІЛ 5. РОЗРОБЛЕННЯ СТАРТАП-ПРОЄКТУ

5.1 Опис ідеї проєкту

Стартап-проєкт, що розробляється в межах даної дипломної роботи, базується на створенні 2D платформера для персональних комп'ютерів з подальшою можливістю масштабування на інші платформи. Основною ідеєю проєкту є поєднання динамічного ігрового процесу, доступного керування та впізнаваного візуального стилю, що орієнтований на сучасні тенденції інді-ігор.

Концепція гри передбачає проходження послідовності рівнів, кожен з яких побудований навколо набору механік пересування, взаємодії з оточенням та протидії ворогам. Ключовий акцент зроблено на плавності керування персонажем, чіткому фізичному відгуку та поступовому ускладненні ігрових ситуацій. Це дозволяє зберегти баланс між доступністю для нових гравців і достатнім рівнем складності для досвідченої аудиторії.

Проєкт орієнтований на сегмент незалежних ігор, де вирішальну роль відіграють не надвисокі бюджети, а якість реалізації механік, цілісність дизайну та емоційний досвід гравця. Саме тому технологічні рішення, використані у грі, поєднують у собі ефективність, гнучкість та можливість подальшого розвитку продукту.

5.2 Розроблення ринкової стратегії

Ринок інді-ігор у жанрі 2D платформерів залишається стабільно активним, незважаючи на загальну тенденцію до розвитку високобюджетних 3D-проєктів. Це пояснюється тим, що значна частина гравців цінує компактні ігри з чіткою механікою, швидким входженням у процес та художньо виразним стилем.

Цільовою аудиторією стартап-проєкту є гравці віком від 16 до 35 років, які регулярно користуються платформами цифрової дистрибуції ігор, такими як Steam або itch.io. Окрему групу становлять прихильники інді-проєктів, які активно шукають нові ігрові враження та підтримують молодих розробників.

Ринкова стратегія проєкту передбачає поступовий вихід на ринок. На початковому етапі гра може бути представлена у форматі демо-версії або раннього доступу, що дозволить отримати зворотний зв’язок від користувачів та скоригувати баланс, складність і окремі ігрові механіки. Такий підхід зменшує ризики та підвищує ймовірність позитивного сприйняття фінального продукту [19].

Додатковою складовою стратегії є формування впізнаваного образу гри через візуальний стиль, атмосферу та унікальні ігрові рішення, що дозволяє вирізнити продукт серед конкурентів у жанрі.

5.3 Аналіз технологічних можливостей і реалізації ідей проєкту

Технологічною основою стартап-проєкту обрано ігровий рушій Unity, який на сьогодні є одним із найбільш універсальних інструментів для розробки 2D-ігор. Його використання дозволяє поєднати високу продуктивність із доступністю для невеликих команд розробників або індивідуальних авторів.

Мова програмування C# забезпечує чітку структуризацію коду, підтримку об’єктно-орієнтованого підходу та зручну реалізацію складних ігрових систем, таких як автомати станів, обробка вводу, фізична взаємодія та логіка штучного інтелекту. Завдяки цьому більшість ідей, закладених у концепцію гри, можуть бути реалізовані без використання сторонніх або комерційних бібліотек.

Використання компонентної архітектури Unity дозволяє розділити функціональність на незалежні модулі, що значно спрощує тестування, налагодження та подальше розширення проєкту. Крім того, рушій підтримує інтеграцію сторонніх ассетів, що дає змогу скоротити час розробки графіки та зосередитися на геймплеї.

З технологічної точки зору стартап-проєкт має високий потенціал масштабування. На основі існуючої архітектури можлива реалізація нових рівнів, механік, режимів гри або портів на інші платформи без суттєвої перебудови програмної частини.

5.4 Маркетингова програма стартап-проєкту

Маркетингова стратегія проєкту орієнтована насамперед на цифрові канали просування, що є найбільш ефективними для інді-ігор з обмеженим бюджетом. Основний акцент робиться на формування спільноти навколо гри ще на етапі розробки.

Планується використання соціальних мереж, тематичних форумів та ігрових платформ для демонстрації процесу створення гри, публікації відео з геймплеєм та обговорення майбутніх оновлень. Такий підхід дозволяє залучити потенційних гравців і сформувати лояльну аудиторію до моменту релізу.

Важливу роль відіграє сторінка гри на платформі цифрової дистрибуції, де користувачі можуть ознайомитися з описом, скриншотами та відеоматеріалами. Якісно оформлена презентація продукту підвищує довіру до проєкту та сприяє зростанню кількості завантажень, що дозволяє вибудувати цілісну модель взаємодії між продуктом, аудиторією та каналами просування [20].

Окрім цього, можливе співробітництво з невеликими стримерами та оглядачами інді-ігор, що дозволяє отримати додаткове охоплення без значних фінансових витрат. Така маркетингова програма є гнучкою та може адаптуватися залежно від реакції аудиторії і результатів просування.

Висновки до розділу

У п’ятому розділі роботи увагу було зосереджено на розгляді розробленого ігрового застосунку не лише як програмного продукту, а як основи для потенційного стартап-проєкту. Такий підхід дозволив поєднати технічні результати роботи з практичними аспектами їх подальшого використання та комерційного розвитку.

У процесі роботи над розділом сформовано ідею стартапу, яка базується на створенні доступного та динамічного ігрового продукту з можливістю поступового розширення функціоналу. Було визначено, що проєкт має потенціал для розвитку за рахунок оновлень, додаткового контенту та адаптації під різні платформи, що є важливим чинником для сучасних цифрових продуктів.

Аналіз ринкового середовища показав, що сегмент 2D ігор залишається актуальним і конкурентним, однак водночас відкритим для нових рішень. Визначення цільової аудиторії та можливих каналів поширення дозволило окреслити реальні шляхи виходу продукту на ринок без необхідності значних початкових інвестицій. Запропоновані підходи до монетизації орієнтовані на збереження балансу між комерційною складовою та комфортом користувача.

Розгляд технологічних можливостей реалізації стартапу підтвердив, що обрані інструменти розробки є достатньо гнучкими та придатними для подальшого масштабування проєкту. Архітектура ігрового застосунку дозволяє вносити зміни й доповнення без кардинальної перебудови системи, що є важливою перевагою на етапах розвитку стартапу.

Запропонована маркетингова програма ґрунтується на використанні цифрових каналів просування та поступовому формуванні спільноти користувачів. Такий підхід відповідає сучасним тенденціям розвитку інді-проєктів і дає змогу перевіряти життєздатність ідеї на практиці.

Загалом результати, отримані в цьому розділі, свідчать про те, що розроблений ігровий застосунок може розглядатися не лише як навчальний проєкт, а й як основа для реального стартапу з перспективами подальшого розвитку та комерційної реалізації.

ВИСНОВКИ

Проведена робота дозволила повністю пройти шлях створення 2D платформера – від ідеї та математичного опрацювання механік до програмної реалізації та оформлення графіки й звуку. В ході роботи було створено реалістичну систему руху персонажа, включно зі стрибками, рухом по стінах і взаємодією з ворогами через штучний інтелект.

Для розробки обрано Unity як основний рушій і C# для програмування. Це забезпечило гнучкість у налаштуванні фізики, управлінні анімаціями та інтеграції звукових ефектів. Графічні та аудіо-ассети були підібрані та адаптовані для проекту, що створило приємне візуальне та аудіальне середовище, здатне занурити гравця в ігровий світ.

Результат роботи демонструє, що обраний підхід до розробки є ефективним і дозволяє швидко реалізовувати ідеї, тестувати ітеративно та створювати цікавий продукт. Отриманий прототип може слугувати базою для подальшого розвитку, масштабування або навіть комерційної реалізації.

Таким чином, робота підтвердила важливість поєднання теоретичних знань, сучасних технологій і креативного підходу, дозволивши створити функціональний та інтерактивний 2D платформер.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Wolf, M. J. P. *The Video Game Explosion: A History from PONG to PlayStation and Beyond*. — Westport: Greenwood Press, 2008.

2. Salen, K., Zimmerman, E. *Rules of Play: Game Design Fundamentals*. — Cambridge: MIT Press, 2004.

3. Unity Technologies. *Unity Manual: 2D Game Development, Physics2D, Input System*. — Official documentation.

4. Schell, J. *The Art of Game Design: A Book of Lenses*. — 3rd ed. — Boca Raton: CRC Press, 2019.

5. Unity Technologies. *Unity Learn Platform: Game Development Pathways*. — Official educational resources.

6. Goldstone, W. *Unity Game Development Essentials*. — Birmingham: Packt Publishing, 2019.

7. Novak, J. *Game Development Essentials: An Introduction*. — 4th ed. — Clifton Park: Delmar Cengage Learning, 2021.

8. Collins, K. *Game Sound: An Introduction to the History, Theory, and Practice of Video Game Music and Sound Design*. — Cambridge: MIT Press, 2008.

9. Millington, I., Funge, J. *Artificial Intelligence for Games*. — 2nd ed. — Boca Raton: CRC Press, 2016.

10. Bourg, D. M., Seemann, G. *Physics for Game Developers*. — Sebastopol: O'Reilly Media, 2004.

11. Eberly, D. H. *Game Physics*. — 2nd ed. — Boca Raton: CRC Press, 2010.

12. Rabin, S. (Ed.). *Game AI Pro: Collected Wisdom of Game AI Professionals*. — Boca Raton: CRC Press, 2014.

13. Gregory, J. *Game Engine Architecture*. — 3rd ed. — Boca Raton: CRC Press, 2018.

14. Nystrom, R. *Game Programming Patterns*. — Genever Benning, 2014.

15. Microsoft. *C# Programming Guide*. — Official documentation.

16. Unity Technologies. *Unity Input System Package Documentation*. — Official documentation.

17. Munro, K. *Beginning Unity 2D Game Development*. — Apress, 2018.

18. Farnell, A. *Designing Sound*. — Cambridge: MIT Press, 2010.

19. Blank, S., Dorf, B. *The Startup Owner's Manual: The Step-By-Step Guide for Building a Great Company*. — Pescadero: K&S Ranch Press, 2020.

20. Osterwalder, A., Pigneur, Y. *Business Model Generation*. — Hoboken: John Wiley & Sons, 2010.

ДОДАТКИ

ДОДАТОК А

Player.cs

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

[RequireComponent(typeof(ActorController), typeof(PlayerDamage))]

public class Player : Actor
{
    [SerializeField] float _moveSpeed = 16.5f;
    [SerializeField] float _groundMoveAcceleration = 9.865f;
    [SerializeField] float _groundMoveDeceleration = 19f;
    [SerializeField] float _airMoveAcceleration = 5.5f;
    [SerializeField] float _airMoveDeceleration = 8f;
    [SerializeField] float _runDustEffectDelay = 0.2f;

    [Space(10)]
    [SerializeField] float _jumpForce = 24f;
    [SerializeField] float _jumpHeight = 4.5f;
    [SerializeField] float _doubleJumpHeight = 2.0f;
    [SerializeField] float _maxCoyoteTime = 0.06f;
    [SerializeField] float _maxJumpBuffer = 0.25f;

    [Space(10)]
    [SerializeField] float _slidingForce = 24f;
    [SerializeField] float _dodgeDuration = 0.25f;
    [SerializeField] float _dodgeInvinsibleTime = 0.15f;
    [SerializeField] float _dodgeCooldown = 0.15f;
    [SerializeField] float _dodgeDustEffectDelay = 0.1f;

    [Space(10)]
    [SerializeField] float _wallSlidingSpeed = 8.5f;
    [SerializeField] float _wallJumpHeight = 1.5f;
    [SerializeField] float _wallJumpXForce = 8f;
    [SerializeField] float _wallSlideDustEffectDelay = 0.15f;

    [Space(10)]
    [SerializeField] int _power = 3;
    [SerializeField] float _basicAttackKnockBackForce = 8.0f;
    [SerializeField] float _fireBallKnockBackForce = 7.5f;

    [Space(10)]
    [SerializeField] Image _healingEffect;

    [Space(10)]
    [SerializeField] AudioClip _swingSound;
    [SerializeField] AudioClip _fireBallSound;
    [SerializeField] AudioClip _attackHitSound;
    [SerializeField] AudioClip _stepSound;
    [SerializeField] AudioClip _jumpSound;

    float _coyoteTime;
    float _maxJumpHeight;
```

```
float _moveX;
float _nextWallSlideDustEffectTime = 0;
float _nextRunDustEffectTime = 0;
bool _canMove = true;
bool _canWallSliding = true;
bool _hasDoubleJumped;
bool _canDodge = true;
int _xAxis;
bool _leftMoveInput;
bool _rightMoveInput;
float _timeHeldBackInput;
bool _jumpInput;
bool _jumpDownInput;
float _jumpBuffer;
bool _attackInput;
bool _specialInput;
bool _dodgeInput;
bool _isJumped;
bool _isFalling;
bool _isAttacking;
bool _isDodging;
bool _isWallSliding;
bool _isResting;
bool isBeingKnockedBack;

KnockBack _basicAttackKnockBack = new KnockBack();
KnockBack _fireBallKnockBack = new KnockBack();

Transform _backCliffChecked;
Coroutine _knockedBackCoroutine = null;
PlayerAttack _attack;
PlayerDrivingForce _drivingForce;
PlayerDamage _damage;

#region MonoBehaviour

protected override void Awake()
{
   base.Awake();
   _backCliffChecked = transform.Find("BackCliffChecked").GetComponent<Transform>();
   _damage = GetComponent<PlayerDamage>();
   _drivingForce = GetComponent<PlayerDrivingForce>();
   _attack = GetComponent<PlayerAttack>();

   _basicAttackKnockBack.force = _basicAttackKnockBackForce;
   _fireBallKnockBack.force = _fireBallKnockBackForce;

   _damage.KnockBack += OnKnockedBack;
   _damage.Died += OnDied;
}

void Start()
{
   if (!GameManager.instance.IsStarted())
   {
      actorTransform.position = GameManager.instance.playerStartPos;
      actorTransform.localScale = new Vector3(GameManager.instance.playerStartlocalScaleX, 1, 1);

      _damage.CurrentHealth = GameManager.instance.playerCurrentHealth;
      _drivingForce.CurrentDrivingForce = GameManager.instance.playerCurrentDrivingForce;

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

```
        else
        {
            GameManager.instance.playerCurrentHealth = _damage.CurrentHealth;
            GameManager.instance.playerCurrentDrivingForce = _drivingForce.CurrentDrivingForce;

            if (GameManager.instance.firstStart)
            {
                GameManager.instance.playerResurrectionPos = actorTransform.position;
                GameManager.instance.resurrectionScene = SceneManager.GetActiveScene().name;
                GameManager.instance.firstStart = false;
            }
            else
            {
                if (GameManager.instance.resurrectionScene == SceneManager.GetActiveScene().name)
                {
                    actorTransform.position = GameManager.instance.playerStartPos;
                }
            }
        }
    }

    void Update()
    {
        if (isDead || _isResting) return;

        deltaTime = Time.deltaTime;

        HandleInput();
        HandleMove();
        HandleJump();
        HandleWallSlide();
        HandleFallingAndLanding();
        HandleAttack();
        HandleDodge();
        AnimationUpdate();
    }

    #endregion

    #region Input

    void HandleInput()
    {
        if (GameManager.instance.currentGameState != GameManager.GameState.Play) return;

        _leftMoveInput = GameInputManager.PlayerInput(GameInputManager.PlayerActions.MoveLeft);
        _rightMoveInput = GameInputManager.PlayerInput(GameInputManager.PlayerActions.MoveRight);
        _xAxis = _leftMoveInput ? -1 : _rightMoveInput ? 1 : 0;

        _jumpInput = GameInputManager.PlayerInput(GameInputManager.PlayerActions.Jump);
        _jumpDownInput = GameInputManager.PlayerInputDown(GameInputManager.PlayerActions.Jump);

        _attackInput = GameInputManager.PlayerInputDown(GameInputManager.PlayerActions.Attack);
        _specialInput = GameInputManager.PlayerInputDown(GameInputManager.PlayerActions.SpecialAttack);
        _dodgeInput = GameInputManager.PlayerInputDown(GameInputManager.PlayerActions.Dodge);
    }

    #endregion
}
```

ДОДАТОК Б

Enemy.cs

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

[RequireComponent(typeof(EnemyDamage))]
public abstract class Enemy : Actor
{
   public string keyName;

   protected EnemyDamage enemyDamage;
   protected EnemyBodyTackle bodyTackle = null;
   protected EnemyData enemyData;
   protected LayerMask playerLayer;

   protected override void Awake()
   {
      if (DeadEnemyManager.IsDeadEnemy(gameObject.name))
      {
         isDead = true;
         Destroy(gameObject);
         return;
      }

      base.Awake();
      enemyDamage = GetComponent<EnemyDamage>();

      enemyData = EnemyDataManager.GetEnemyData(keyName);
      if (enemyData.isBodyTackled && TryGetComponent(out EnemyBodyTackle bodyTackle))
      {
         this.bodyTackle = bodyTackle;
         this.bodyTackle.damage = enemyData.bodyTackleDamage;
      }
      enemyDamage.MaxHealth = enemyData.health;
      enemyDamage.SuperArmor = enemyData.superArmor;
      enemyDamage.BlinkMaterial = enemyData.blinkMaterial;

      enemyDamage.KnockBack += OnKnockedBack;
      enemyDamage.Died += OnDied;

      playerLayer = LayerMask.GetMask("Player");
   }

   protected virtual void OnKnockedBack(KnockBack knockBack)
   {
      if(controller == null) return;
      controller.SlideMove(knockBack.force, knockBack.direction);
   }

   protected virtual IEnumerator OnDied()
   {
      isDead = true;
      DeadEnemyManager.AddDeadEnemy(gameObject.name);

      if(bodyTackle != null)
      {
         bodyTackle.isBodyTackled = false;
      }
```

```
        Vector2 position = new Vector2(transform.position.x, transform.position.y + 2);
        ObjectPoolManager.instance.GetPoolObject("HealingPiece", position);

        yield return null;

        Destroy(gameObject);
    }
}
```

SelectProfile.cs

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

[System.Serializable]
public class SelectProfile
{
    public Image profileImage;
    public TextMeshProUGUI playTimeText;
    public TextMeshProUGUI newText;
    public Image saveDeleteGuage;
}

public class SelectProfileScreen : MonoBehaviour
{
    public delegate void PrevScreenReturnEventHandler();
    public PrevScreenReturnEventHandler PrevScreenReturn;

    public Sprite selectImage;
    public Sprite notSelectImage;
    public TextMeshProUGUI selectProfileText;
    public TextMeshProUGUI manualText;
    public SelectProfile[] profile;

    int _currentSaveFileIndex;

    void Awake()
    {
        SelectProfileInit();
        SelectProfileRefresh();
    }

    void OnEnable()
    {
        selectProfileText.text = LanguageManager.GetText("SelectProfile");

        string backKey = LanguageManager.GetText("Back");
        string selectKey = LanguageManager.GetText("Confirm");
        string deleteKey = LanguageManager.GetText("Delete");

        string backInput, selectInput, deleteInput;

        if (GameInputManager.usingController)
        {
            backInput = GameInputManager.MenuControlToButtonText(GameInputManager.MenuControl.Cancle);
            selectInput = GameInputManager.MenuControlToButtonText(GameInputManager.MenuControl.Select);
            deleteInput = GameInputManager.MenuControlToButtonText(GameInputManager.MenuControl.Delete);
        }
        else
        {
```

```
        backInput = GameInputManager.MenuControlToKeyText(GameInputManager.MenuControl.Cancle);
        selectInput = GameInputManager.MenuControlToKeyText(GameInputManager.MenuControl.Select);
        deleteInput = GameInputManager.MenuControlToKeyText(GameInputManager.MenuControl.Delete);
    }

    manualText.text = string.Format(
        "{0} [ <color=#ffaa5e>{1}</color> ] {2} [ <color=#ffaa5e>{3}</color> ] {4} [ <color=#ffaa5e>{5}</color> ]",
        backKey, backInput,
        selectKey, selectInput,
        deleteKey, deleteInput
    );
}

void Update()
{
    bool upInput = GameInputManager.MenuInputDown(GameInputManager.MenuControl.Up);
    bool downInput = GameInputManager.MenuInputDown(GameInputManager.MenuControl.Down);
    bool backInput = GameInputManager.MenuInputDown(GameInputManager.MenuControl.Cancle);
    bool selectInput = GameInputManager.MenuInputDown(GameInputManager.MenuControl.Select);
    bool deleteInput = GameInputManager.MenuInput(GameInputManager.MenuControl.Delete);

    if (upInput)
    {
        _currentSaveFileIndex--;
        SelectProfileRefresh();
    }
    else if (downInput)
    {
        _currentSaveFileIndex++;
        SelectProfileRefresh();
    }

    if (selectInput)
    {
        GameManager.instance.GameLoad(_currentSaveFileIndex + 1);
        GameManager.instance.SetGameState(GameManager.GameState.Play);
    }
    else if (backInput)
    {
        gameObject.SetActive(false);
        PrevScreenReturn();
    }

    if (deleteInput)
    {
        SelectProfileDelete();
    }

    DeleteGaugeValueToDefault(deleteInput);
}

void SelectProfileInit()
{
    for (int i = 0; i < profile.Length; i++)
    {
        GameSaveData saveData = SaveSystem.GameLoad(i + 1);

        if (saveData != null)
        {
            profile[i].playTimeText.text =
                GameManager.instance.IntToTimeText(saveData.playTime);
            profile[i].newText.text = "";
        }
```

```
            else
            {
                profile[i].playTimeText.text = "";
                profile[i].newText.text = "NEW";
            }

            profile[i].saveDeleteGuage.fillAmount = 0f;
        }
    }

    void SelectProfileRefresh()
    {
        if (_currentSaveFileIndex >= profile.Length)
            _currentSaveFileIndex = 0;
        else if (_currentSaveFileIndex < 0)
            _currentSaveFileIndex = profile.Length - 1;

        for (int i = 0; i < profile.Length; i++)
        {
            profile[i].profileImage.sprite = notSelectImage;
        }

        profile[_currentSaveFileIndex].profileImage.sprite = selectImage;
    }

    void SelectProfileDelete()
    {
        if (SaveSystem.GameLoad(_currentSaveFileIndex + 1) == null)
            return;

        float fillAmount =
            profile[_currentSaveFileIndex].saveDeleteGuage.fillAmount +
            (0.3f * Time.deltaTime);

        fillAmount = Mathf.Clamp(fillAmount, 0f, 1.0f);
        profile[_currentSaveFileIndex].saveDeleteGuage.fillAmount = fillAmount;

        if (GameInputManager.usingController)
        {
            GamepadVibrationManager.instance
                .GamepadRumbleStart(fillAmount * 0.5f, 0.02f);
        }

        if (fillAmount >= 0.5f && fillAmount < 0.75f)
        {
            profile[_currentSaveFileIndex].saveDeleteGuage.color =
                new Color32(255, 170, 94, 255);
        }
        else if (fillAmount >= 0.75f && fillAmount < 1.0f)
        {
            profile[_currentSaveFileIndex].saveDeleteGuage.color =
                new Color32(142, 67, 91, 255);
        }
        else if (fillAmount >= 1.0f)
        {
            profile[_currentSaveFileIndex].saveDeleteGuage.fillAmount = 0;
            SaveSystem.GameDelete(_currentSaveFileIndex + 1);
            SelectProfileInit();
        }
    }

    void DeleteGaugeValueToDefault(bool deleteInput)
    {
```

```
        for (int i = 0; i < profile.Length; i++)
        {
            if (profile[i].saveDeleteGuage.fillAmount == 0 ||
                (deleteInput && i == _currentSaveFileIndex))
                continue;

            float fillAmount =
                profile[i].saveDeleteGuage.fillAmount - Time.deltaTime;

            fillAmount = Mathf.Clamp(fillAmount, 0f, 1.0f);
            profile[i].saveDeleteGuage.fillAmount = fillAmount;

            if (fillAmount >= 0.5f && fillAmount < 0.75f)
            {
                profile[i].saveDeleteGuage.color =
                    new Color32(255, 170, 94, 255);
            }
            else if (fillAmount < 0.5f)
            {
                profile[i].saveDeleteGuage.color =
                    new Color32(255, 236, 214, 255);
            }
        }
    }
}
```

SoundManager.cs

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

public class SoundManager : MonoBehaviour
{
    public static SoundManager instance = null;

    [SerializeField] AudioSource _music;
    [SerializeField] AudioSource _effect;

    float currentMusicVolume;
    AudioClip _musicClip;
    Coroutine _musicChange = null;

    void Awake()
    {
        if (instance != null)
        {
            Destroy(gameObject);
            return;
        }
        else
        {
            instance = this;
            DontDestroyOnLoad(gameObject);
        }

        if (_music == null)
        {
            _music = transform.Find("Music").GetComponent<AudioSource>();
        }
        if (_effect == null)
        {
            _effect = transform.Find("Effect").GetComponent<AudioSource>();
```

```
    }

    _music.loop = true;
}

public void MusicPlay(AudioClip audioClip, float volume = 1.0f)
{
    if (audioClip == null || _musicClip == audioClip) return;
    _musicClip = audioClip;

    if (_musicChange != null)
    {
        StopCoroutine(_musicChange);
        _musicChange = null;
    }

    _musicChange = StartCoroutine(MusicChange(volume));
}

public void MusicStop()
{
    if (_musicChange != null)
    {
        StopCoroutine(_musicChange);
        _musicChange = null;
    }

    _musicClip = null;
    _musicChange = StartCoroutine(MusicChange(0));
}

public AudioClip GetCurrentMusic() => _music.clip;

IEnumerator MusicChange(float setVolume)
{
    currentMusicVolume = SoundSettingsManager.musicVolume;

    float volume = _music.volume;
    float volumeReduction = volume * 0.015f;

    while (volume > 0f)
    {
        volume -= volumeReduction;
        _music.volume = Mathf.Clamp(volume, 0f, 1.0f);
        yield return YieldInstructionCache.WaitForSecondsRealtime(0.01f);
    }

    _music.Stop();

    _music.clip = _musicClip;
    _music.Play();

    float volumeIncrease = setVolume * 0.015f;
    while (volume < setVolume * currentMusicVolume)
    {
        volume += volumeIncrease;
        _music.volume = Mathf.Clamp(volume, 0f, setVolume * currentMusicVolume);
        yield return YieldInstructionCache.WaitForSecondsRealtime(0.01f);
    }

    if (currentMusicVolume != SoundSettingsManager.musicVolume)
    {
        _music.volume = setVolume * SoundSettingsManager.musicVolume;
```

```
        }

        _musicChange = null;
    }

    public void SoundEffectPlay(AudioClip audioClip)
    {
        _effect.volume = SoundSettingsManager.effectsVolume;
        _effect.PlayOneShot(audioClip);
    }

    public void MusicVolumeRefresh()
    {
        if (_musicChange != null)
        {
            StopCoroutine(_musicChange);
            _musicChange = null;
        }

        _music.volume = currentMusicVolume * SoundSettingsManager.musicVolume;
    }
}
```

GameManager.cs

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

public class GameManager : MonoBehaviour
{
    public static GameManager instance = null;

    public delegate void PlayerDieEventHandler();
    public event PlayerDieEventHandler PlayerDieEvent = null;

    public enum GameState
    {
        Title,
        Play,
        MenuOpen
    }

    public GameState currentGameState = GameState.Play;

    [HideInInspector] public bool firstStart = true;
    [HideInInspector] public Vector2 playerStartPos;
    [HideInInspector] public float playerStartlocalScaleX;

    [HideInInspector] public int playerCurrentHealth;
    [HideInInspector] public int playerCurrentDrivingForce;

    [HideInInspector] public string resurrectionScene;
    [HideInInspector] public Vector2 playerResurrectionPos;

    int gameDataNum = 1;

    int _prevPlayTime;
    float _startTime;

    bool _isStarted = true;
```

```
    void Awake()
    {
        if (instance != null)
        {
            Destroy(gameObject);
            return;
        }

        instance = this;
        DontDestroyOnLoad(gameObject);

        Cursor.visible = false;

        EnemyDataManager.Init();
        SceneManager.sceneLoaded += OnSceneLoaded;

        OptionsData.Init();

#if UNITY_EDITOR
        if (currentGameState == GameState.Play)
        {
            GameLoad(1, false);
        }
#endif
    }

    public bool IsStarted()
    {
        if (_isStarted)
        {
            _startTime = (float)DateTime.Now.TimeOfDay.TotalSeconds;
            _isStarted = false;
            return true;
        }
        return false;
    }

    public void SetGameState(GameState newGameState)
    {
        switch (newGameState)
        {
            case GameState.Play:
                ScreenEffect.instance.TimeStopCancle();
                break;
            case GameState.MenuOpen:
                ScreenEffect.instance.TimeStopStart();
                break;
            case GameState.Title:
                break;
        }

        currentGameState = newGameState;
    }

    public string IntToTimeText(int time)
    {
        int sec = (time % 60);
        int min = ((time / 60) % 60);
        int hour = (time / 3600);

        string secToStr = sec < 10 ? "0" + sec.ToString() : sec.ToString();
        string minToStr = min < 10 ? "0" + min.ToString() : min.ToString();
```

```
    string hourToStr = hour.ToString();

    return string.Format("{0}:{1}:{2}", hourToStr, minToStr, secToStr);
}

public void HandlePlayerDeath()
{
    playerStartPos = playerResurrectionPos;
    DeadEnemyManager.ClearDeadEnemies();
    PlayerDieEvent?.Invoke();
}

public void GameSave()
{
    string sceneName = resurrectionScene;
    List<string> seenTutorials = TutorialManager.GetSeenTutorialsToString();
    List<string> deadBosses = DeadEnemyManager.GetDeadBosses();
    List<Vector2> foundMaps = MapManager.GetDiscoveredMaps();
    int playTime = _prevPlayTime + Mathf.CeilToInt((float)DateTime.Now.TimeOfDay.TotalSeconds - _startTime);

    var saveData = new GameSaveData
    (
        playTime,
        sceneName,
        playerResurrectionPos,
        PlayerLearnedSkills.hasLearnedClimbingWall,
        PlayerLearnedSkills.hasLearnedDoubleJump,
        foundMaps,
        deadBosses,
        seenTutorials
    );

    SaveSystem.GameSave(saveData, gameDataNum);
}

public void GameLoad(int gameDataNum, bool sceneLoaded = true)
{
    var saveData = SaveSystem.GameLoad(gameDataNum);
    this.gameDataNum = gameDataNum;

    if (saveData == null)
    {
        SceneTransition.instance.LoadScene("OldMachineRoom_A");
        return;
    }

    _prevPlayTime = saveData.playTime;

    resurrectionScene = saveData.sceneName;
    playerResurrectionPos = saveData.playerPos;

    MapManager.AddDiscoveredMaps(saveData.foundMaps);

    PlayerLearnedSkills.hasLearnedClimbingWall = saveData.hasLearnedClimbingWall;
    PlayerLearnedSkills.hasLearnedDoubleJump = saveData.hasLearnedDoubleJump;

    for (int i = 0; i < saveData.deadBosses.Count; i++)
    {
        DeadEnemyManager.AddDeadBoss(saveData.deadBosses[i]);
    }

    for (int i = 0; i < saveData.seenTutorials.Count; i++)
    {
```

```
            TutorialManager.AddSeenTutorial(saveData.seenTutorials[i]);
        }

        firstStart = false;

        if (sceneLoaded)
        {
            playerStartPos = saveData.playerPos;
            SceneTransition.instance.LoadScene(saveData.sceneName);
        }
    }

    void OnSceneLoaded(Scene scene, LoadSceneMode mode)
    {
        if (currentGameState == GameState.Title)
        {
            gameDataNum = 1;

            firstStart = true;

            playerStartPos = Vector2.zero;
            playerStartlocalScaleX = 0;

            playerCurrentHealth = 0;
            playerCurrentDrivingForce = 0;

            resurrectionScene = string.Empty;
            playerResurrectionPos = Vector2.zero;

            _prevPlayTime = 0;
            _startTime = 0;

            _isStarted = true;
        }

        PlayerDieEvent = null;
    }

    void OnApplicationQuit()
    {
        if (instance.currentGameState != GameState.Title)
        {
            instance.GameSave();
        }

        OptionsData.OptionsSave();
    }
}
```

SaveSystem.cs

```
using System.Collections;
using System.Collections.Generic;
using System.IO;
using System.Security.Cryptography;
using UnityEngine;

[System.Serializable]
public class OptionsSaveData
{
    public bool fullScreenMode = true;
    public Resolution? resolution = null;
    public bool vSync = true;
```

```
    public List<int> keyMapping = new List<int>();
    public List<int> buttonMapping = new List<int>();

    public float masterVolume = 1.0f;
    public float musicVolume = 1.0f;
    public float effectsVolume = 1.0f;

    public float gamepadVibration = 1.0f;
    public bool screenShake = true;
    public bool screenFlashes = true;

    public int language = (int)LanguageManager.Language.Last;
}

[System.Serializable]
public class GameSaveData
{
    public int playTime;
    public string sceneName;
    public Vector2 playerPos;
    public bool hasLearnedClimbingWall;
    public bool hasLearnedDoubleJump;
    public List<Vector2> foundMaps = new List<Vector2>();
    public List<string> deadBosses = new List<string>();
    public List<string> seenTutorials = new List<string>();

    public GameSaveData(
        int playTime,
        string sceneName,
        Vector2 playerPos,
        bool hasLearnedClimbingWall,
        bool hasLearnedDoubleJump,
        List<Vector2> foundMaps,
        List<string> deadBosses,
        List<string> seenTutorials)
    {
        this.playTime = playTime;
        this.sceneName = sceneName;
        this.playerPos = playerPos;
        this.hasLearnedClimbingWall = hasLearnedClimbingWall;
        this.hasLearnedDoubleJump = hasLearnedDoubleJump;
        this.foundMaps = foundMaps;
        this.deadBosses = deadBosses;
        this.seenTutorials = seenTutorials;
    }
}

public static class SaveSystem
{
    private static readonly string privateKey =
        "EKDe$BMqvxvVf6z^ovKrhuf%JIJUg01CgCnadXYcOeGT%Iu5kS0xrj^09%^N";

    public static void OptionSave(OptionsSaveData optionsSaveData)
    {
        string jsonString = JsonUtility.ToJson(optionsSaveData);
        string encryptString = Encrypt(jsonString);

        using (FileStream fs = new FileStream(OptionsGetPath(), FileMode.Create, FileAccess.Write))
        {
            byte[] bytes = System.Text.Encoding.UTF8.GetBytes(encryptString);
            fs.Write(bytes, 0, bytes.Length);
        }
```

```
}

public static OptionsSaveData OptionLoad()
{
    if (!File.Exists(OptionsGetPath()))
    {
        return null;
    }

    string encryptData;
    using (FileStream fs = new FileStream(OptionsGetPath(), FileMode.Open, FileAccess.Read))
    {
        byte[] bytes = new byte[(int)fs.Length];
        fs.Read(bytes, 0, (int)fs.Length);
        encryptData = System.Text.Encoding.UTF8.GetString(bytes);
    }

    string decryptData = Decrypt(encryptData);
    OptionsSaveData saveData = JsonUtility.FromJson<OptionsSaveData>(decryptData);
    return saveData;
}

public static void GameSave(GameSaveData saveData, int dataNum)
{
    string jsonString = JsonUtility.ToJson(saveData);
    string encryptString = Encrypt(jsonString);

    using (FileStream fs = new FileStream(GetPath(dataNum), FileMode.Create, FileAccess.Write))
    {
        byte[] bytes = System.Text.Encoding.UTF8.GetBytes(encryptString);
        fs.Write(bytes, 0, bytes.Length);
    }
}

public static GameSaveData GameLoad(int dataNum)
{
    if (!File.Exists(GetPath(dataNum)))
    {
        return null;
    }

    string encryptData;
    using (FileStream fs = new FileStream(GetPath(dataNum), FileMode.Open, FileAccess.Read))
    {
        byte[] bytes = new byte[(int)fs.Length];
        fs.Read(bytes, 0, (int)fs.Length);
        encryptData = System.Text.Encoding.UTF8.GetString(bytes);
    }

    string decryptData = Decrypt(encryptData);
    GameSaveData saveData = JsonUtility.FromJson<GameSaveData>(decryptData);
    return saveData;
}

public static void GameDelete(int dataNum)
{
    if (!File.Exists(GetPath(dataNum)))
    {
        return;
    }
    File.Delete(GetPath(dataNum));
}
```

```
    static string GetPath(int dataNum) =>
        Path.Combine(Application.persistentDataPath + "/save0" + dataNum + ".save");

    static string OptionsGetPath() =>
        Path.Combine(Application.persistentDataPath + "/options.save");

    static string Encrypt(string data)
    {
        byte[] bytes = System.Text.Encoding.UTF8.GetBytes(data);
        RijndaelManaged rm = CreateRijndaelManaged();
        ICryptoTransform ct = rm.CreateEncryptor();
        byte[] results = ct.TransformFinalBlock(bytes, 0, bytes.Length);
        return System.Convert.ToBase64String(results, 0, results.Length);
    }

    static string Decrypt(string data)
    {
        byte[] bytes = System.Convert.FromBase64String(data);
        RijndaelManaged rm = CreateRijndaelManaged();
        ICryptoTransform ct = rm.CreateDecryptor();
        byte[] resultArray = ct.TransformFinalBlock(bytes, 0, bytes.Length);
        return System.Text.Encoding.UTF8.GetString(resultArray);
    }

    static RijndaelManaged CreateRijndaelManaged()
    {
        byte[] keyArray = System.Text.Encoding.UTF8.GetBytes(privateKey);
        RijndaelManaged result = new RijndaelManaged();

        byte[] newKeysArray = new byte[16];
        System.Array.Copy(keyArray, 0, newKeysArray, 0, 16);

        result.Key = newKeysArray;
        result.Mode = CipherMode.ECB;
        result.Padding = PaddingMode.PKCS7;
        return result;
    }
}
```

ControlsScreen.cs

```
using System.Collections;
using System.Collections.Generic;
using TMPro;
using MenuUI;
using UnityEngine;
using System;
using UnityEngine.InputSystem;
using UnityEngine.InputSystem.LowLevel;

public class ControlsScreen : MonoBehaviour
{
    public GameObject optionsMenuScreen;
    public TextMeshProUGUI controlsText;
    public TextMeshProUGUI actionsText;
    public TextMeshProUGUI keyboardText;
    public TextMeshProUGUI controllerText;
    public TextMeshProUGUI manualText;
    public Menu[] menu;

    int _currentMenuIndex;
    bool _isMapping;
```

```
    void Awake()
    {
        if (manualText == null)
        {
            manualText = GameObject.Find("ManualText").GetComponent<TextMeshProUGUI>();
        }
        KeyRefresh();
        ButtonRefresh();
    }

    void OnEnable()
    {
        ControlsOptionsRefresh();
        MenuUIController.MenuRefresh(menu, ref _currentMenuIndex, manualText);
        SetNonSelectedDeviceTextColor();
    }

    void Update()
    {
        if (_isMapping) return;

        bool upInput = GameInputManager.MenuInputDown(GameInputManager.MenuControl.Up);
        bool downInput = GameInputManager.MenuInputDown(GameInputManager.MenuControl.Down);
        bool selectInput = GameInputManager.MenuInputDown(GameInputManager.MenuControl.Select);
        bool backInput = GameInputManager.MenuInputDown(GameInputManager.MenuControl.Cancle);

        if (upInput)
        {
            _currentMenuIndex--;
            MenuUIController.MenuRefresh(menu, ref _currentMenuIndex, manualText);
            SetNonSelectedDeviceTextColor();
        }
        else if (downInput)
        {
            _currentMenuIndex++;
            MenuUIController.MenuRefresh(menu, ref _currentMenuIndex, manualText);
            SetNonSelectedDeviceTextColor();
        }

        if (selectInput)
        {
            menu[_currentMenuIndex].menuSelectEvent.Invoke();
            SetNonSelectedDeviceTextColor();
        }

        if (backInput)
        {
            _currentMenuIndex = 0;
            MenuUIController.MenuRefresh(menu, ref _currentMenuIndex, manualText);
            ReturnToOptionsMenuScreen();
        }
    }

    public void SetButtonMapping(string action)
    {
        var stringToAction = (GameInputManager.PlayerActions)Enum.Parse(typeof(GameInputManager.PlayerActions),
action);
        StartCoroutine(ChooseButtonToMap(stringToAction));
    }

    IEnumerator ChooseButtonToMap(GameInputManager.PlayerActions action)
    {
        _isMapping = true;
```

```
    TextMeshProUGUI buttonText;
    if (GameInputManager.usingController)
    {
        buttonText = menu[_currentMenuIndex].text[2];
    }
    else
    {
        buttonText = menu[_currentMenuIndex].text[1];
    }
    buttonText.color = new Color32(141, 105, 122, 255);
    manualText.text = LanguageManager.GetText("Empty");

    yield return null;

    while (true)
    {
        if (GameInputManager.usingController)
        {
            GamepadButton inputButton = GameInputManager.GetCurrentInputButton();

            if (inputButton != GamepadButton.Start)
            {
                GameInputManager.GamepadMapping(action, inputButton);
                ButtonRefresh();
                break;
            }
        }
        else
        {
            Key inputKey = GameInputManager.GetCurrentInputKey();

            if (inputKey != Key.None)
            {
                GameInputManager.KeyboardMapping(action, inputKey);
                KeyRefresh();
                break;
            }
        }
        yield return null;
    }

    MenuUIController.SelectMenuTextColorChange(menu[_currentMenuIndex]);
    _isMapping = false;

    MenuUIController.MenuRefresh(menu, ref _currentMenuIndex, manualText);
    SetNonSelectedDeviceTextColor();
}

public void ReturnToDefault()
{
    if (GameInputManager.usingController)
    {
        GameInputManager.ControllerInputSetDefaults();
        ButtonRefresh();
    }
    else
    {
        GameInputManager.KeyboardInputSetDefaults();
        KeyRefresh();
    }
}
```

```
void SetNonSelectedDeviceTextColor()
{
    if (menu[_currentMenuIndex].text.Length <= 1) return;

    if (GameInputManager.usingController)
    {
        menu[_currentMenuIndex].text[1].color = MenuUIController.notSelectColor;
    }
    else
    {
        menu[_currentMenuIndex].text[2].color = MenuUIController.notSelectColor;
    }
}

void ControlsOptionsRefresh()
{
    controlsText.text = LanguageManager.GetText("Controls");
    actionsText.text = LanguageManager.GetText("Actions");
    keyboardText.text = LanguageManager.GetText("Keyboard");
    controllerText.text = LanguageManager.GetText("Controller");

    for (int i = 0; i < menu.Length; i++)
    {
        switch (menu[i].text[0].name)
        {
            case "MoveLeftText":
                menu[i].text[0].text = LanguageManager.GetText("MoveLeft");
                break;
            case "MoveRightText":
                menu[i].text[0].text = LanguageManager.GetText("MoveRight");
                break;
            case "MoveUpText":
                menu[i].text[0].text = LanguageManager.GetText("MoveUp");
                break;
            case "MoveDownText":
                menu[i].text[0].text = LanguageManager.GetText("MoveDown");
                break;
            case "JumpText":
                menu[i].text[0].text = LanguageManager.GetText("Jump");
                break;
            case "AttackText":
                menu[i].text[0].text = LanguageManager.GetText("Attack");
                break;
            case "SpecialAttackText":
                menu[i].text[0].text = LanguageManager.GetText("SpecialAttack");
                break;
            case "DodgeText":
                menu[i].text[0].text = LanguageManager.GetText("Dodge");
                break;
            case "ResetToDefaultText":
                menu[i].text[0].text = LanguageManager.GetText("ResetToDefault");
                break;
        }
    }
}

void KeyRefresh()
{
    for (int i = 0; i < menu.Length; i++)
    {
        switch (menu[i].text[0].name)
        {
            case "MoveLeftText":
```

```
                menu[i].text[1].text = GameInputManager.ActionToKeyText(GameInputManager.PlayerActions.MoveLeft);
                break;
            case "MoveRightText":
                menu[i].text[1].text = GameInputManager.ActionToKeyText(GameInputManager.PlayerActions.MoveRight);
                break;
            case "MoveUpText":
                menu[i].text[1].text = GameInputManager.ActionToKeyText(GameInputManager.PlayerActions.MoveUp);
                break;
            case "MoveDownText":
                menu[i].text[1].text = GameInputManager.ActionToKeyText(GameInputManager.PlayerActions.MoveDown);
                break;
            case "JumpText":
                menu[i].text[1].text = GameInputManager.ActionToKeyText(GameInputManager.PlayerActions.Jump);
                break;
            case "AttackText":
                menu[i].text[1].text = GameInputManager.ActionToKeyText(GameInputManager.PlayerActions.Attack);
                break;
            case "SpecialAttackText":
                menu[i].text[1].text = GameInputManager.ActionToKeyText(GameInputManager.PlayerActions.SpecialAttack);
                break;
            case "DodgeText":
                menu[i].text[1].text = GameInputManager.ActionToKeyText(GameInputManager.PlayerActions.Dodge);
                break;
        }
    }
}

void ButtonRefresh()
{
    for (int i = 0; i < menu.Length; i++)
    {
        switch (menu[i].text[0].name)
        {
            case "MoveLeftText":
                menu[i].text[2].text = GameInputManager.ActionToButtonText(GameInputManager.PlayerActions.MoveLeft);
                break;
            case "MoveRightText":
                menu[i].text[2].text = GameInputManager.ActionToButtonText(GameInputManager.PlayerActions.MoveRight);
                break;
            case "MoveUpText":
                menu[i].text[2].text = GameInputManager.ActionToButtonText(GameInputManager.PlayerActions.MoveUp);
                break;
            case "MoveDownText":
                menu[i].text[2].text = GameInputManager.ActionToButtonText(GameInputManager.PlayerActions.MoveDown);
                break;
            case "JumpText":
                menu[i].text[2].text = GameInputManager.ActionToButtonText(GameInputManager.PlayerActions.Jump);
                break;
            case "AttackText":
                menu[i].text[2].text = GameInputManager.ActionToButtonText(GameInputManager.PlayerActions.Attack);
                break;
            case "SpecialAttackText":
                menu[i].text[2].text =
GameInputManager.ActionToButtonText(GameInputManager.PlayerActions.SpecialAttack);
                break;
            case "DodgeText":
                menu[i].text[2].text = GameInputManager.ActionToButtonText(GameInputManager.PlayerActions.Dodge);
                break;
        }
    }
}

void ReturnToOptionsMenuScreen()
```

```
    {
        gameObject.SetActive(false);
        optionsMenuScreen.SetActive(true);
    }
}
```

MenuUI.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using TMPro;
using UnityEngine.Events;
using System.Text;

namespace MenuUI
{
    [System.Serializable]
    public class Manual
    {
        public string key;
        public GameInputManager.MenuControl menuControl;
    }

    [System.Serializable]
    public class Menu
    {
        public TextMeshProUGUI[] text;
        public Manual[] manual;
        public UnityEvent menuSelectEvent;
    }

    public static class MenuUIController
    {
        public static readonly Color selectColor = new Color32(255, 236, 214, 255);
        public static readonly Color notSelectColor = new Color32(255, 170, 94, 255);

        public static void MenuRefresh(Menu[] menu, ref int currentMenuIndex, TextMeshProUGUI manualText = null)
        {
            if (currentMenuIndex >= menu.Length)
            {
                currentMenuIndex = 0;
            }
            else if (currentMenuIndex < 0)
            {
                currentMenuIndex = menu.Length - 1;
            }

            AllMenuTextColorInit(menu);
            SelectMenuTextColorChange(menu[currentMenuIndex]);

            if (manualText != null)
            {
                SetMenualText(menu[currentMenuIndex], manualText);
            }
        }

        public static void MenuRefresh(List<Menu> menu, ref int currentMenuIndex, TextMeshProUGUI manualText = null)
        {
            if (currentMenuIndex >= menu.Count)
            {
                currentMenuIndex = 0;
            }
```

```
    else if (currentMenuIndex < 0)
    {
        currentMenuIndex = menu.Count - 1;
    }

    AllMenuTextColorInit(menu);
    SelectMenuTextColorChange(menu[currentMenuIndex]);

    if (manualText != null)
    {
        SetMenualText(menu[currentMenuIndex], manualText);
    }
}

public static void AllMenuTextColorInit(Menu[] menu)
{
    for (int i = 0; i < menu.Length; i++)
    {
        for (int j = 0; j < menu[i].text.Length; j++)
        {
            menu[i].text[j].color = notSelectColor;
        }
    }
}

public static void AllMenuTextColorInit(List<Menu> menu)
{
    for (int i = 0; i < menu.Count; i++)
    {
        for (int j = 0; j < menu[i].text.Length; j++)
        {
            menu[i].text[j].color = notSelectColor;
        }
    }
}

public static void AllMenuTextHide(Menu[] menu)
{
    for (int i = 0; i < menu.Length; i++)
    {
        for (int j = 0; j < menu[i].text.Length; j++)
        {
            menu[i].text[j].color = new Color32(0, 0, 0, 0);
        }
    }
}

public static void AllMenuTextHide(List<Menu> menu)
{
    for (int i = 0; i < menu.Count; i++)
    {
        for (int j = 0; j < menu[i].text.Length; j++)
        {
            menu[i].text[j].color = new Color32(0, 0, 0, 0);
        }
    }
}

public static void SelectMenuTextColorChange(Menu menu)
{
    for (int j = 0; j < menu.text.Length; j++)
    {
        menu.text[j].color = selectColor;
```

```
        }
    }

    public static void SelectMenuTextHide(Menu menu)
    {
        for (int j = 0; j < menu.text.Length; j++)
        {
            menu.text[j].color = new Color(0, 0, 0, 0);
        }
    }

    public static void SetMenualText(Menu menu, TextMeshProUGUI manualText)
    {
        StringBuilder manual = new StringBuilder();
        for (int i = 0; i < menu.manual.Length; i++)
        {
            string key = LanguageManager.GetText(menu.manual[i].key);
            string input = GameInputManager.usingController
                ? GameInputManager.MenuControlToButtonText(menu.manual[i].menuControl)
                : GameInputManager.MenuControlToKeyText(menu.manual[i].menuControl);

            manual.AppendFormat("{0} [ <color=#ffaa5e>{1}</color> ]", key, input);

            if (i < menu.manual.Length - 1)
            {
                manual.Append(" ");
            }
        }
        manualText.text = manual.ToString();
    }
  }
}
```

Map.cs

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

public static class MapManager
{
    static List<Vector2> discoveredMapPositions = new List<Vector2>();

    public static void ClearDiscoveredMaps()
    {
        discoveredMapPositions.Clear();
    }

    public static void AddDiscoveredMap(Image mapImage)
    {
        discoveredMapPositions.Add(mapImage.rectTransform.anchoredPosition);
    }

    public static void AddDiscoveredMaps(List<Vector2> mapPositions)
    {
        discoveredMapPositions.AddRange(mapPositions);
    }

    public static List<Vector2> GetDiscoveredMaps() => discoveredMapPositions;
}
```

```
[System.Serializable]
public class MenuControlAndKey
{
    public GameInputManager.MenuControl[] menuControl;
    public string menuKey;
}

public class Map : MonoBehaviour
{
    const int MapTileSpacing = 8;
    const int MaxZoomLevel = 3;
    const int MinZoomLevel = 1;

    [SerializeField] TextMeshProUGUI _mapTitleText;
    [SerializeField] TextMeshProUGUI _manualText;
    [SerializeField] RectTransform _mapTileContainer;
    [SerializeField] RectTransform _playerIcon;
    [SerializeField] List<Image> _mapTiles;
    [SerializeField] MenuControlAndKey[] _menuControlAndKey;

    int _mapZoomLevel = 1;

    Vector2 _mapOriginPos = Vector2.zero;
    Camera _camera;
    Transform _cameraTransform;

    void Awake()
    {
        transform.GetComponent<PauseScreen>().MapOpend += OnMapOpend;

        _camera = Camera.main;
        _cameraTransform = _camera.GetComponent<Transform>();

        foreach (var mapTile in _mapTiles)
        {
            mapTile.gameObject.SetActive(false);
        }

        foreach (Vector2 mapPos in MapManager.GetDiscoveredMaps())
        {
            Image mapImage = _mapTiles.Find(x => x.rectTransform.anchoredPosition == mapPos);
            mapImage?.gameObject.SetActive(true);
        }

        _mapOriginPos = _mapTileContainer.anchoredPosition;
    }

    void Update()
    {
        UpdatePlayerIcon();
        DiscoverMap();
        MoveMap();
        ZoomMap();
    }

    void UpdatePlayerIcon()
    {
        float cameraHalfSizeY = _camera.orthographicSize;
        float cameraHalfSizeX = cameraHalfSizeY * _camera.aspect;

        float xPos = (_cameraTransform.position.x + cameraHalfSizeX) / (cameraHalfSizeX * 2);
        float yPos = (_cameraTransform.position.y + cameraHalfSizeY) / (cameraHalfSizeY * 2);
```

```
    int xPosIndex = Mathf.FloorToInt(xPos) * MapTileSpacing;
    int yPosIndex = Mathf.FloorToInt(yPos) * MapTileSpacing;

    _playerIcon.anchoredPosition = new Vector2(xPosIndex, yPosIndex);
}

void DiscoverMap()
{
    Image mapImage = _mapTiles.Find(x => x.rectTransform.anchoredPosition == _playerIcon.anchoredPosition);
    if (!mapImage.gameObject.activeSelf)
    {
        mapImage.gameObject.SetActive(true);
        MapManager.AddDiscoveredMap(mapImage);
    }
}

void MoveMap()
{
    float yMove = GameInputManager.MenuInput(GameInputManager.MenuControl.Up) ? 1 :
                  GameInputManager.MenuInput(GameInputManager.MenuControl.Down) ? -1 : 0;

    float xMove = GameInputManager.MenuInput(GameInputManager.MenuControl.Right) ? 1 :
                  GameInputManager.MenuInput(GameInputManager.MenuControl.Left) ? -1 : 0;

    _mapTileContainer.Translate(new Vector2(xMove, yMove) * 10f * Time.unscaledDeltaTime);

    _mapTileContainer.anchoredPosition = new Vector2(
        Mathf.Clamp(_mapTileContainer.anchoredPosition.x, -50f, 20f),
        Mathf.Clamp(_mapTileContainer.anchoredPosition.y, -20f, 20f)
    );
}

void ZoomMap()
{
    if (GameInputManager.MenuInputDown(GameInputManager.MenuControl.Select))
    {
        if (_mapZoomLevel < MaxZoomLevel)
        {
            _mapTileContainer.localScale += Vector3.one;
            _mapZoomLevel++;
        }
        else
        {
            _mapTileContainer.localScale = Vector3.one;
            _mapZoomLevel = MinZoomLevel;
        }
    }
}

public void OnMapOpend()
{
    _mapTitleText.text = LanguageManager.GetText("Map");
    SetManualText();

    _mapZoomLevel = 1;
    _mapTileContainer.localScale = Vector3.one;

    _mapTileContainer.anchoredPosition = _mapOriginPos - _playerIcon.anchoredPosition;
}

void SetManualText()
{
```

```
        var sb = new StringBuilder();

        for (int i = 0; i < _menuControlAndKey.Length; i++)
        {
            for (int j = 0; j < _menuControlAndKey[i].menuControl.Length; j++)
            {
                string menuControlToText = GameInputManager.usingController
                    ? GameInputManager.MenuControlToButtonText(_menuControlAndKey[i].menuControl[j])
                    : GameInputManager.MenuControlToKeyText(_menuControlAndKey[i].menuControl[j]);

                sb.AppendFormat("[ <color=#ffaa5e>{0}</color> ] ", menuControlToText);
            }

            string keyToName = LanguageManager.GetText(_menuControlAndKey[i].menuKey);
            sb.Append(keyToName);

            if (i < _menuControlAndKey.Length - 1)
            {
                sb.Append("  ");
            }
        }

        _manualText.text = sb.ToString();
    }
}
```

Player.cs

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

[RequireComponent(typeof(ActorController), typeof(PlayerDamage))]
public class Player : Actor
{
    [SerializeField] float _moveSpeed = 16.5f;
    [SerializeField] float _groundMoveAcceleration = 9.865f;
    [SerializeField] float _groundMoveDeceleration = 19f;
    [SerializeField] float _airMoveAcceleration = 5.5f;
    [SerializeField] float _airMoveDeceleration = 8f;
    [SerializeField] float _runDustEffectDelay = 0.2f;

    [Space(10)]
    [SerializeField] float _jumpForce = 24f;
    [SerializeField] float _jumpHeight = 4.5f;
    [SerializeField] float _doubleJumpHeight = 2.0f;
    [SerializeField] float _maxCoyoteTime = 0.06f;
    [SerializeField] float _maxJumpBuffer = 0.25f;

    [Space(10)]
    [SerializeField] float _slidingForce = 24f;
    [SerializeField] float _dodgeDuration = 0.25f;
    [SerializeField] float _dodgeInvinsibleTime = 0.15f;
    [SerializeField] float _dodgeCooldown = 0.15f;
    [SerializeField] float _dodgeDustEffectDelay = 0.1f;

    [Space(10)]
    [SerializeField] float _wallSlidingSpeed = 8.5f;
    [SerializeField] float _wallJumpHeight = 1.5f;
    [SerializeField] float _wallJumpXForce = 8f;
```

```
[SerializeField] float _wallSlideDustEffectDelay = 0.15f;

[Space(10)]
[SerializeField] int _power = 3;
[SerializeField] float _basicAttackKnockBackForce = 8.0f;
[SerializeField] float _fireBallKnockBackForce = 7.5f;

[Space(10)]
[SerializeField] Image _healingEffect;

[Space(10)]
[SerializeField] AudioClip _swingSound;
[SerializeField] AudioClip _fireBallSound;
[SerializeField] AudioClip _attackHitSound;
[SerializeField] AudioClip _stepSound;
[SerializeField] AudioClip _jumpSound;

float _coyoteTime;
float _maxJumpHeight;
float _moveX;

float _nextWallSlideDustEffectTime;
float _nextRunDustEffectTime;

bool _canMove = true;
bool _canWallSliding = true;
bool _hasDoubleJumped;
bool _canDodge = true;

int _xAxis;
bool _leftMoveInput;
bool _rightMoveInput;
float _timeHeldBackInput;

bool _jumpInput;
bool _jumpDownInput;
float _jumpBuffer;

bool _attackInput;
bool _specialInput;
bool _dodgeInput;

bool _isJumped;
bool _isFalling;
bool _isAttacking;
bool _isDodging;
bool _isWallSliding;
bool _isResting;
bool isBeingKnockedBack;

KnockBack _basicAttackKnockBack = new KnockBack();
KnockBack _fireBallKnockBack = new KnockBack();

Transform _backCliffChecked;
Coroutine _knockedBackCoroutine;
PlayerAttack _attack;
PlayerDrivingForce _drivingForce;
PlayerDamage _damage;

protected override void Awake()
{
   base.Awake();
```

```
    _backCliffChecked = transform.Find("BackCliffChecked").GetComponent<Transform>();
    _damage = GetComponent<PlayerDamage>();
    _drivingForce = GetComponent<PlayerDrivingForce>();
    _attack = GetComponent<PlayerAttack>();

    _basicAttackKnockBack.force = _basicAttackKnockBackForce;
    _fireBallKnockBack.force = _fireBallKnockBackForce;

    _damage.KnockBack += OnKnockedBack;
    _damage.Died += OnDied;
}

void Start()
{
    if (!GameManager.instance.IsStarted())
    {
        actorTransform.position = GameManager.instance.playerStartPos;
        actorTransform.localScale = new Vector3(GameManager.instance.playerStartlocalScaleX, 1, 1);

        _damage.CurrentHealth = GameManager.instance.playerCurrentHealth;
        _drivingForce.CurrentDrivingForce = GameManager.instance.playerCurrentDrivingForce;

        GameManager.instance.GameSave();
    }
    else
    {
        GameManager.instance.playerCurrentHealth = _damage.CurrentHealth;
        GameManager.instance.playerCurrentDrivingForce = _drivingForce.CurrentDrivingForce;

        if (GameManager.instance.firstStart)
        {
            GameManager.instance.playerResurrectionPos = actorTransform.position;
            GameManager.instance.resurrectionScene = SceneManager.GetActiveScene().name;
            GameManager.instance.firstStart = false;
        }
        else
        {
            if (GameManager.instance.resurrectionScene == SceneManager.GetActiveScene().name)
            {
                actorTransform.position = GameManager.instance.playerStartPos;
            }
        }
    }
}

void Update()
{
    if (isDead || _isResting) return;

    deltaTime = Time.deltaTime;

    HandleInput();
    HandleMove();
    HandleJump();
    HandleWallSlide();
    HandleFallingAndLanding();
    HandleAttack();
    HandleDodge();
    AnimationUpdate();
}

void HandleInput()
{
```

```
    if (GameManager.instance.currentGameState != GameManager.GameState.Play) return;

    _leftMoveInput = GameInputManager.PlayerInput(GameInputManager.PlayerActions.MoveLeft);
    _rightMoveInput = GameInputManager.PlayerInput(GameInputManager.PlayerActions.MoveRight);
    _xAxis = _leftMoveInput ? -1 : _rightMoveInput ? 1 : 0;

    _jumpInput = GameInputManager.PlayerInput(GameInputManager.PlayerActions.Jump);
    _jumpDownInput = GameInputManager.PlayerInputDown(GameInputManager.PlayerActions.Jump);

    _attackInput = GameInputManager.PlayerInputDown(GameInputManager.PlayerActions.Attack);
    _specialInput = GameInputManager.PlayerInputDown(GameInputManager.PlayerActions.SpecialAttack);
    _dodgeInput = GameInputManager.PlayerInputDown(GameInputManager.PlayerActions.Dodge);
}

void HandleMove()
{
    if (!_canMove) return;

    if (_xAxis != 0)
    {
        _moveX += (controller.IsGrounded ? _groundMoveAcceleration : _airMoveAcceleration) * deltaTime;

        if (actorTransform.localScale.x != _xAxis && !_isAttacking)
        {
            Flip();
            _moveX = 0;
            if (controller.IsGrounded) _nextRunDustEffectTime = 0;
        }

        if (controller.IsGrounded)
        {
            _nextRunDustEffectTime -= deltaTime;
            if (_nextRunDustEffectTime <= 0)
            {
                ObjectPoolManager.instance.GetPoolObject("RunDust", actorTransform.position, actorTransform.localScale.x);
                _nextRunDustEffectTime = _runDustEffectDelay;
                SoundManager.instance.SoundEffectPlay(_stepSound);
                GamepadVibrationManager.instance.GamepadRumbleStart(0.02f, 0.017f);
            }
        }
    }
    else
    {
        _moveX -= (controller.IsGrounded ? _groundMoveDeceleration : _airMoveDeceleration) * deltaTime;
        _nextRunDustEffectTime = 0;
    }

    _moveX = Mathf.Clamp(_moveX, 0f, 1f);
    controller.VelocityX = _xAxis * _moveSpeed * _moveX;
}

void HandleJump()
{
    if (controller.IsGrounded || controller.IsWalled) _hasDoubleJumped = false;

    if (!_isJumped)
    {
        if (JumpInputBuffer() && CoyoteTime())
        {
            if (!_isAttacking && !_isDodging && !isBeingKnockedBack)
            {
                StartJump();
            }
```

```
        }
        else if (PlayerLearnedSkills.hasLearnedDoubleJump)
        {
            if (!_hasDoubleJumped && !controller.IsGrounded && !_isWallSliding && _jumpDownInput)
            {
                StartDoubleJump();
            }
        }
    }
    else
    {
        ContinueJumping();
    }
}

void StartJump()
{
    _isJumped = true;
    _coyoteTime = _maxCoyoteTime;
    _jumpBuffer = _maxJumpBuffer;
    _maxJumpHeight = actorTransform.position.y + _jumpHeight;

    ObjectPoolManager.instance.GetPoolObject("JumpDust", actorTransform.position);
    GamepadVibrationManager.instance.GamepadRumbleStart(0.5f, 0.05f);
    SoundManager.instance.SoundEffectPlay(_stepSound);
    SoundManager.instance.SoundEffectPlay(_jumpSound);
}

void StartDoubleJump()
{
    _isJumped = true;
    _hasDoubleJumped = true;
    _maxJumpHeight = actorTransform.position.y + _doubleJumpHeight;
    animator.SetTrigger(GetAnimationHash("DoubleJump"));
    ObjectPoolManager.instance.GetPoolObject("JumpDust", actorTransform.position);
    GamepadVibrationManager.instance.GamepadRumbleStart(0.5f, 0.05f);
    SoundManager.instance.SoundEffectPlay(_jumpSound);
}

void ContinueJumping()
{
    controller.VelocityY = _jumpForce;

    if (_maxJumpHeight <= actorTransform.position.y)
    {
        _isJumped = false;
        controller.VelocityY = _jumpForce * 0.75f;
    }
    else if (!_jumpInput || controller.IsRoofed)
    {
        _isJumped = false;
        controller.VelocityY = _jumpForce * 0.5f;
    }
}

bool JumpInputBuffer()
{
    if (_jumpInput)
    {
        if (_jumpBuffer < _maxJumpBuffer)
        {
            _jumpBuffer += deltaTime;
            return true;
```

```
        }
    }
    else
    {
        _jumpBuffer = 0;
    }
    return false;
}

bool CoyoteTime()
{
    if (controller.IsGrounded)
    {
        _coyoteTime = 0;
        return true;
    }
    else if (_coyoteTime < _maxCoyoteTime)
    {
        _coyoteTime += deltaTime;
        return true;
    }
    return false;
}

void HandleFallingAndLanding()
{
    if (controller.VelocityY < -5f)
    {
        _isFalling = true;
    }
    else if (controller.IsGrounded && _isFalling)
    {
        _isFalling = false;
        SoundManager.instance.SoundEffectPlay(_stepSound);
        ObjectPoolManager.instance.GetPoolObject("JumpDust", actorTransform.position);
        GamepadVibrationManager.instance.GamepadRumbleStart(0.25f, 0.05f);
    }
}

void HandleWallSlide()
{
    if (!PlayerLearnedSkills.hasLearnedClimbingWall || !_canWallSliding || _isAttacking) return;

    if (!_isWallSliding)
    {
        if (controller.IsWalled && !controller.IsGrounded && controller.VelocityY < 0)
        {
            int wallDirection = controller.IsLeftWalled ? -1 : 1;
            if (_xAxis == wallDirection)
            {
                StartWallSliding();
            }
        }
    }
    else
    {
        ContinueWallSliding();
    }
}

void StartWallSliding()
{
    _isWallSliding = true;
```

```
    _isDodging = _canMove = false;
    controller.SlideCancle();
}

void ContinueWallSliding()
{
    int wallDirection = controller.IsLeftWalled ? -1 : 1;
    float dustEffectXPos = actorTransform.position.x + (wallDirection == 1 ? 0.625f : -0.625f);
    Vector2 dustEffectPos = new Vector2(dustEffectXPos, actorTransform.position.y + 1.25f);

    controller.VelocityY = Mathf.Clamp(controller.VelocityY, -_wallSlidingSpeed, float.MaxValue);

    bool backInput = (wallDirection == 1 && _leftMoveInput) ||
                (wallDirection == -1 && _rightMoveInput);

    if (backInput)
    {
        _timeHeldBackInput += deltaTime;
        if (_timeHeldBackInput >= 0.2f)
        {
            WallSlidingCancle();
        }
    }
    else
    {
        _timeHeldBackInput = 0;
    }

    if (_jumpDownInput)
    {
        _isJumped = true;
        _coyoteTime = _maxCoyoteTime;
        _maxJumpHeight = actorTransform.position.y + _wallJumpHeight;

        Flip();
        controller.SlideMove(_wallJumpXForce, -wallDirection, 30f);
        _moveX = 1.0f;

        ObjectPoolManager.instance.GetPoolObject("WallJumpDust", dustEffectPos, -actorTransform.localScale.x);
        GamepadVibrationManager.instance.GamepadRumbleStart(0.5f, 0.05f);
        SoundManager.instance.SoundEffectPlay(_stepSound);

        WallSlidingCancle();
    }

    if (!controller.IsWalled || controller.IsGrounded || isBeingKnockedBack)
    {
        WallSlidingCancle();
    }

    _nextWallSlideDustEffectTime -= deltaTime;
    if (_nextWallSlideDustEffectTime <= 0)
    {
        _nextWallSlideDustEffectTime = _wallSlideDustEffectDelay;
        ObjectPoolManager.instance.GetPoolObject("WallSlideDust", dustEffectPos, actorTransform.localScale.x);
        GamepadVibrationManager.instance.GamepadRumbleStart(0.1f, 0.033f);
        SoundManager.instance.SoundEffectPlay(_swingSound);
    }
}

void WallSlidingCancle()
{
    _isWallSliding = false;
```

```
    _canMove = true;
    _timeHeldBackInput = 0;
    _nextWallSlideDustEffectTime = _wallSlideDustEffectDelay;
}

void HandleAttack()
{
    if (_isAttacking || _isWallSliding || _isDodging || isBeingKnockedBack) return;

    if (_attackInput)
    {
        StartCoroutine(BasicAttack());
    }
    else if (_specialInput)
    {
        if (_drivingForce.TryConsumeDrivingForce(1))
        {
            StartCoroutine(FireBall());
        }
    }
}

IEnumerator BasicAttack()
{
    _isAttacking = true;

    bool isHit = false;
    bool isNextAttacked = false;

    animator.SetTrigger(GetAnimationHash("BasicAttack"));
    animator.ResetTrigger(GetAnimationHash("AnimationEnd"));

    _basicAttackKnockBack.direction = actorTransform.localScale.x;

    if (controller.IsGrounded)
    {
        controller.SlideMove(_moveSpeed, actorTransform.localScale.x, 65f);
    }

    yield return null;

    SoundManager.instance.SoundEffectPlay(_swingSound);

    while (!IsAnimationEnded())
    {
        if (isBeingKnockedBack) break;

        if (!isHit)
        {
            if (_attack.IsAttacking(_power, _basicAttackKnockBack))
            {
                for (int i = 0; i < _attack.HitCount; i++)
                {
                    _drivingForce.IncreaseDrivingForce();
                }

                SoundManager.instance.SoundEffectPlay(_attackHitSound);
                GamepadVibrationManager.instance.GamepadRumbleStart(0.5f, 0.05f);

                controller.SlideMove(11.5f, -actorTransform.localScale.x);

                if (!controller.IsGrounded)
                {
```

```
                controller.VelocityY = 15;
            }

            isHit = true;
        }
    }

    if (!Physics2D.Raycast(_backCliffChecked.position, Vector2.down, 1.0f, LayerMask.GetMask("Ground")))
    {
        controller.SlideCancle();
    }

    _canMove = !controller.IsGrounded;

    if (IsAnimatorNormalizedTimeInBetween(0.4f, 0.87f))
    {
        if (_attackInput)
        {
            isNextAttacked = true;
        }
    }
    else if (IsAnimatorNormalizedTimeInBetween(0.87f, 1.0f))
    {
        if (isNextAttacked)
        {
            if (_xAxis == actorTransform.localScale.x)
            {
                controller.SlideMove(_moveSpeed, actorTransform.localScale.x, 65f);
            }

            animator.SetTrigger(GetAnimationHash("NextAttack"));
            SoundManager.instance.SoundEffectPlay(_swingSound);

            isHit = isNextAttacked = false;
        }
    }

    yield return null;
}

AttackEnd();
}

IEnumerator FireBall()
{
    _isAttacking = true;

    bool hasSpawnedFireBall = false;

    animator.SetTrigger(GetAnimationHash("FireBall"));
    animator.ResetTrigger(GetAnimationHash("AnimationEnd"));
    _fireBallKnockBack.direction = actorTransform.localScale.x;

    yield return null;

    SoundManager.instance.SoundEffectPlay(_fireBallSound);

    while (!IsAnimationEnded())
    {
        if (isBeingKnockedBack) break;

        if (controller.IsGrounded)
        {
```

```
            _moveX = 0;
            _canMove = false;
        }

        if (IsAnimatorNormalizedTimeInBetween(0.28f, 0.4f))
        {
            if (!hasSpawnedFireBall)
            {
                float scaleX = actorTransform.localScale.x;
                float addPosX = actorTransform.position.x + (0.66f * scaleX);
                float addPosY = actorTransform.position.y + 1.12f;
                Vector2 addPos = new Vector2(addPosX, addPosY);
                float angle = scaleX == 1 ? 180 : 0;

                ObjectPoolManager.instance.GetPoolObject("FireBall", addPos, scaleX, angle);
                hasSpawnedFireBall = true;
            }
        }

        yield return null;
    }

    AttackEnd();
}

void AttackEnd()
{
    animator.SetTrigger(GetAnimationHash("AnimationEnd"));
    _isAttacking = false;
    _canMove = true;
}

void HandleDodge()
{
    if (_isAttacking || _isWallSliding || _isDodging || isBeingKnockedBack) return;

    if (_dodgeInput && _canDodge && controller.IsGrounded)
    {
        StartCoroutine(Dodging());
    }
}

IEnumerator Dodging()
{
    float nextDodgeDustEffectTime = 0;
    float dodgeInvinsibleTime = _dodgeInvinsibleTime;
    float dodgeDuration = _dodgeDuration;

    _damage.IsDodged = _isDodging = true;
    _canDodge = _canMove = false;

    animator.SetTrigger(GetAnimationHash("Sliding"));
    animator.ResetTrigger(GetAnimationHash("AnimationEnd"));

    yield return null;

    controller.SlideMove(_slidingForce, actorTransform.localScale.x);

    while (dodgeDuration > 0)
    {
        if (isBeingKnockedBack) break;

        if (nextDodgeDustEffectTime <= 0)
```

```
        {
            ObjectPoolManager.instance.GetPoolObject("RunDust", actorTransform.position, actorTransform.localScale.x);
            nextDodgeDustEffectTime = _dodgeDustEffectDelay;
            SoundManager.instance.SoundEffectPlay(_jumpSound);
        }
        else
        {
            nextDodgeDustEffectTime -= deltaTime;
        }

        if (dodgeInvinsibleTime <= 0)
        {
            _damage.IsDodged = false;
        }
        else
        {
            dodgeInvinsibleTime -= deltaTime;
        }

        dodgeDuration -= deltaTime;
        yield return null;
    }

    DodgeEnd();
}

void DodgeEnd()
{
    controller.SlideCancle();
    animator.SetTrigger(GetAnimationHash("AnimationEnd"));

    _isDodging = false;
    _canMove = true;

    StartCoroutine(DodgeCooldown());
}

IEnumerator DodgeCooldown()
{
    yield return YieldInstructionCache.WaitForSeconds(_dodgeCooldown);
    _canDodge = true;
}

public void RestAtCheckPoint(Vector2 checkPointPos)
{
    if (!controller.IsGrounded || _isResting) return;

    GameManager.instance.playerResurrectionPos = checkPointPos;
    GameManager.instance.resurrectionScene = SceneManager.GetActiveScene().name;

    DeadEnemyManager.ClearDeadEnemies();
    GameManager.instance.GameSave();

    StartCoroutine("Resting");
}

IEnumerator Resting()
{
    float red = _healingEffect.color.r;
    float green = _healingEffect.color.g;
    float blue = _healingEffect.color.b;
    float alpha = 0.5f;
```

```
    _isResting = true;
    animator.ResetTrigger(GetAnimationHash("AnimationEnd"));
    animator.SetTrigger(GetAnimationHash("Rest"));
    yield return YieldInstructionCache.WaitForSecondsRealtime(0.2f);

    _healingEffect.enabled = AccessibilitySettingsManager.screenFlashes;
    _healingEffect.color = new Color(red, green, blue, alpha);
    _damage.HealthRecovery(_damage.maxHealth);
    yield return YieldInstructionCache.WaitForSecondsRealtime(0.1f);

    while (alpha > 0f)
    {
        alpha -= 0.025f;
        _healingEffect.color = new Color(red, green, blue, alpha);
        yield return YieldInstructionCache.WaitForSecondsRealtime(0.01f);
    }

    yield return YieldInstructionCache.WaitForSecondsRealtime(0.1f);

    animator.SetTrigger(GetAnimationHash("AnimationEnd"));
    _isResting = false;
}

void OnKnockedBack(KnockBack knockBack)
{
    controller.UseGravity = true;
    actorTransform.Translate(new Vector2(0, 0.1f));

    controller.SlideMove(knockBack.force, knockBack.direction);
    _canMove = false;

    StopCoroutine("Resting");
    _isResting = false;

    _isAttacking = _isDodging = _isWallSliding = _isJumped = false;
    isBeingKnockedBack = true;
    GamepadVibrationManager.instance.GamepadRumbleStart(0.8f, 0.068f);

    if (!isDead)
    {
        controller.VelocityY = 24f;
        if (_knockedBackCoroutine != null)
        {
            StopCoroutine(_knockedBackCoroutine);
            _knockedBackCoroutine = null;
        }
        _knockedBackCoroutine = StartCoroutine(KnockedBackCoroutine());
    }
}

IEnumerator KnockedBackCoroutine()
{
    animator.SetTrigger(GetAnimationHash("KnockBack"));

    while (controller.IsSliding)
    {
        yield return null;
    }

    _canMove = true;
    isBeingKnockedBack = false;

    animator.SetTrigger(GetAnimationHash("KnockBackEnd"));
```

```
        _knockedBackCoroutine = null;
    }

    IEnumerator OnDied()
    {
        isDead = true;
        GameManager.instance.HandlePlayerDeath();
        animator.SetTrigger(GetAnimationHash("Die"));
        ScreenEffect.instance.BulletTimeStart(0.3f, 1.0f);
        yield return YieldInstructionCache.WaitForSecondsRealtime(2.5f);
        GameManager.instance.playerCurrentHealth = _damage.maxHealth;
        GameManager.instance.playerCurrentDrivingForce = 0;
        SceneTransition.instance.LoadScene(GameManager.instance.resurrectionScene);
    }

    void AnimationUpdate()
    {
        animator.SetFloat(GetAnimationHash("MoveX"), _moveX);
        animator.SetFloat(GetAnimationHash("FallSpeed"), controller.VelocityY);
        animator.SetBool(GetAnimationHash("IsGrounded"), controller.IsGrounded);
        animator.SetBool(GetAnimationHash("IsWallSliding"), _isWallSliding);
    }
}
```