ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ
ТЕХНОЛОГІЙ НАВЧАЛЬНО-НАУКОВИЙ
ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ТЕХНОЛОГІЙ ЦИФРОВОГО РОЗВИТКУ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Метод використання компонентного підходу при розробці 2D платформера в Unity»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Технології цифрового розвитку»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Гліб СИДОРОВ
(підпис)

Виконав: здобувач(ка) вищої освіти групи ТЦР-42

Гліб СИДОРОВ

Керівник: Іван ОЛЕЙНІКОВ

Рецензент: _____

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Технологій цифрового розвитку

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Технології цифрового розвитку»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Технологій цифрового розвитку
_____Вікторія ЖЕБКА
« ____» _____2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Сидорову Глібу Сергійовичу

1. Тема кваліфікаційної роботи: «2D платформер на базі рушія Unity»
керівник кваліфікаційної роботи старший викладач кафедри Іван ОЛЕЙНІКОВ,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 року № 51.

2. Строк подання кваліфікаційної роботи «18» травня 2026 року.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про розробку 2D-ігор, принципи побудови платформерів та ігрових механік, особливості використання рушія Unity для створення 2D-додатків, методи реалізації фізики, анімації та взаємодії об’єктів у грі, опис підходів до проєктування рівнів, технічна документація з використання бібліотек і вбудованих інструментів Unity, а також матеріали щодо оптимізації продуктивності та тестування ігрових додатків.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Огляд та аналіз існуючих методів та технологій розробки 2D-ігор та платформерів.
Розгляд сучасних підходів до створення 2D-ігор, аналіз популярних

ігрових рушіїв, принципи реалізації ігрових механік у жанрі платформер, особливості фізики, анімації та управління персонажем.
2. Проєктування 2D платформера на базі рушія Unity.
Розробка архітектури застосунку, визначення основних модулів гри, проєктування ігрових рівнів (level design), механік персонажа, ворогів та взаємодії об'єктів.
3. Програмна реалізація та опис функціонування 2D платформера.
Опис реалізації гри з використанням Unity, створення скриптів на C#, інтеграція фізики, анімацій, системи керування, UI та логіки гри.
Також у розділі висвітлюється процес створення ігрових рівнів, налаштування камери, реалізація логіки проходження та взаємодії персонажа з елементами середовища. Окремо розглядається створення користувацького інтерфейсу, включаючи головне меню, кнопки керування, екрани паузи та завершення гри. Додатково описується використання компонентного підходу Unity та принципи організації скриптів для забезпечення гнучкості й можливості подальшого розширення функціоналу гри.
4. Тестування та налагодження гри.
Проведення тестування працездатності гри, перевірка ігрових механік, виявлення та усунення помилок, оптимізація продуктивності. Особлива увага приділяється процесу виявлення помилок у програмному коді, фізичній моделі та анімаційній системі. Розглядаються способи усунення логічних і технічних помилок, оптимізація структури скриптів та покращення продуктивності гри. Також висвітлюється використання інструментів Unity для аналізу навантаження на систему, перевірки стабільності роботи застосунку та оптимізації частоти кадрів. У результаті проведеного тестування забезпечується стабільна робота гри та підвищується якість кінцевого програмного продукту.

5. Перелік графічного матеріалу: *презентація*
1. Мета, об'єкт та предмет дослідження.
2. Задачі кваліфікаційної роботи.
3. Вимоги до програмного забезпечення.
4. Аналіз існуючих аналогів.
5. Програмні засоби реалізації.
6. Діаграма класів.
7. Екранні форми гри.
8. Апробація результатів.

6. Дата видачі завдання «20» лютого 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	20.02.2026	виконано
2	Аналіз та дослідження існуючих аналогів	02.03.2026	виконано
3	Огляд сучасних методів та технологій розробки 2D ігор	12.03.2026	виконано
4	Проєктування архітектури 2D платформера та ігрових механік	22.03.2026	виконано
5	Програмна реалізація 2D платформера засобами Unity	15.04.2026	виконано
6	Реалізація системи анімацій, фізики та колізій	22.04.2026	виконано
7	Оформлення роботи: вступ, висновки, реферат	01.05.2026	виконано
8	Розробка демонстраційних матеріалів	08.05.2026	виконано
9	Попередній захист роботи	11.05-18.05.2026	виконано

Здобувач(ка) вищої освіти _____ Гліб СИДОРОВ
(підпис)

Керівник
кваліфікаційної роботи _____ Іван ОЛЕЙНІКОВ
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 71 стор., 2 табл., 26 рис., 8 джерел.

Мета роботи - підвищення ефективності процесу розробки 2D ігор жанру платформер за допомогою створення програмного застосунку на базі ігрового рушія Unity із реалізацією основних ігрових механік та сучасних підходів до проєктування ігрових систем.

Об'єкт дослідження - процес розробки 2D ігрових застосунків.

Предмет дослідження - методи та засоби створення 2D платформера з використанням рушія Unity.

Короткий зміст роботи: У роботі проаналізовано сучасні підходи до розробки 2D ігор та особливості жанру платформер. Досліджено існуючі аналоги платформерів, принципи реалізації фізики, системи колізій, анімацій та взаємодії персонажа з ігровим середовищем. Виконано проєктування архітектури програмного застосунку та розроблено основні ігрові механіки, зокрема систему руху персонажа, механіку стрибків, взаємодію з платформами, перешкодами та елементами рівня. Програмну реалізацію виконано із використанням рушія Unity та мови програмування C#. У процесі розробки використано компоненти Physics2D, Animator, Tilemap та систему обробки введення користувача. Проведено функціональне тестування та оптимізацію програмного продукту для забезпечення стабільної роботи та плавності ігрового процесу.

Сферою використання розробленого застосунку є навчальні та демонстраційні проєкти у сфері розробки комп'ютерних ігор, а також практичне вивчення технологій створення 2D платформерів.

КЛЮЧОВІ СЛОВА: 2D ПЛАТФОРМЕР, UNITY, C#, GAME DEVELOPMENT, PHYSICS2D, TILEMAP, ANIMATOR, ІГРОВІ МЕХАНІКИ, КОЛІЗІЇ, АНІМАЦІЯ.

ЗМІСТ

ВСТУП **11**
РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ РІШЕНЬ ... **14**
1.1 Загальна характеристика 2D ігор ***14***
1.2 Особливості жанру платформер ***15***
1.3 Аналіз існуючих аналогів ***16***
1.3.1 Класичні 2D платформери ***17***
1.3.2 Сучасні інді-платформери ***20***
1.3.3 AAA-платформери ***23***
1.3.4 Мобільні платформери ***26***
1.3.5 Роль рушіїв у реалізації платформерів ***29***
1.3.6 Узагальнення аналізу ***33***
РОЗДІЛ 2 ПРОЄКТУВАННЯ 2D ПЛАТФОРМЕРА **35**
2.1Постановка задачі ***35***
2.2 Визначення функціональних вимог ***36***
2.3 Визначення нефункціональних вимог ***37***
2.4 Архітектура застосунку ***38***
2.5 Проєктування ігрових механік ***40***
2.5.1 Рух персонажа ***40***
2.5.2 Стрибки ***41***
2.5.3 Система прогресії ***43***
2.5.4 Взаємодія з платформами ***44***
2.5.5 Уникнення або подолання перешкод ***46***
2.5.6 Пастки ***47***
2.5.7 Унікальні навички персонажа ***48***
2.6 Проєктування рівнів ***50***
РОЗДІЛ 3 РОЗРОБКА ТА РЕАЛІЗАЦІЯ ЗАСТОСУНКУ **52**
3.1 Середовище розробки та інструменти ***52***
3.1.1 Основне середовище розробки ***53***
3.1.2 Інструменти програмування та допоміжні засоби... *Ошибка! Закладка не определена.*

3.2 Реалізація керування персонажем ... 58
3.2.1 Загальна структура системи керування ... 59
3.2.2 Горизонтальний рух персонажа ... 61
3.2.3 Вертикальний рух та перевірка поверхні ... 62
3.2.4 Механіка стрибка ... 63
3.2.5 Присідання та зміна колайдерів ... 65
3.2.6 Анімації та візуальний зворотний зв'язок ... 66
3.3 Реалізація фізики та колізій ... 67
3.4 Реалізація анімацій ... 69
РОЗДІЛ 4 ТЕСТУВАННЯ ТА ОПТИМІЗАЦІЯ ... 71
4.1 Методи тестування ... 71
4.2 Функціональне тестування ... 72
4.3 Виявлення та усунення помилок ... 74
4.3.1 Помилки ігрової логіки та системи керування ... 75
4.3.2 Фізичні помилки та колізії ... 76
4.3.3 Анімації, сцени та продуктивність ... 78
ВИСНОВОК ... 80
ПЕРЕЛІК ПОСИЛАНЬ ... 82
ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ ... 83
ДОДАТОК Б. ЛІСТИНГИ ОСНОВНИХ МОДУЛІВ ... 93

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

Unity - ігровий рушій для розробки 2D та 3D застосунків

C# - об’єктно-орієнтована мова програмування, що використовується в Unity

2D - двовимірна графіка

UI - користувацький інтерфейс (User Interface)

GameObject - базовий об’єкт у сцені

Unity Sprite - двовимірне зображення ігрового

б’єкта

Scene - ігрова сцена (рівень) у Unity

Prefab - заготовка (шаблон) ігрового об’єкта

Physics2D - система фізики для 2D об’єктів у Unity

Collider - компонент для обробки зіткнень об’єктів

Rigidbody2D - компонент фізичного тіла у 2D фізиці

Animator - система керування анімаціями

Script - програмний скрипт, що реалізує логіку гри

FPS - кадри за секунду (Frames Per Second)

ВСТУП

У нинішній добі інформаційних технологій індустрія комп’ютерних ігор виступає як одна з найбільш динамічних та фінансово успішних сфер. Її стрімкий розвиток зумовлений безперервним прогресом у сфері апаратної

складової, виникненням новітніх програмних засобів та зростаючим попитом аудиторії на інтерактивні розваги. Особливе місце тут займають двовимірні (2D) ігри, які, попри розквіт тривимірної графіки, зберігають свою привабливість завдяки простоті, досяжності та високому темпу ігрового процесу

Жанр платформерів є одним із найвідоміших і найпоширеніших у сфері 2D-ігор. Його основою є взаємодія персонажа з ігровим середовищем, подолання перешкод, дослідження рівнів та виконання різноманітних завдань. Такі ігри вимагають ретельного проєктування механік руху, фізики, системи колізій, а також продуманого дизайну рівнів. Завдяки цьому платформери є зручним об’єктом для дослідження основ розробки ігрових застосунків.

Значний внесок у спрощення процесу гейм деву роблять актуальні ігрові рушії. Серед них надзвичайно популярним є Unity, що пропонує широкий

арсенал інструментарію для розробки як двовимірних, так і тривимірних проєктів. Застосування Unity дає змогу розробникам сфокусуватися на втіленні ігрової логіки, економлячи суттєві ресурси, які інакше пішли б на створення базових модулів, таких як механізми рендерингу, фізичні симуляції чи обробка вхідних даних. Більше того, рушій підтримує мову програмування C#, що

забезпечує високу гнучкість та можливість розширення програмного коду.

Актуальність теми обумовлена необхідністю вивчення та практичного застосування сучасних інструментів розробки ігор, а також потребою у

створенні навчальних проєктів, які дозволяють ефективно засвоювати принципи побудови програмного забезпечення. Розробка 2D платформера є доцільною, оскільки поєднує в собі як теоретичні аспекти (алгоритми, архітектура, моделювання), так і практичні навички (програмування, тестування, оптимізація).

На додаток, розробка ігрових додатків вимагає подолання низки

нетривіальних проблем. Сюди входять такі елементи, як налагодження ігрового циклу, впровадження механізмів керування протагоністом, опрацювання

зіткнень, забезпечення злагодженості анімаційних послідовностей, підвищення швидкодії та формування інтуїтивно зрозумілого інтерфейсу. Усі ці складові

роблять процес створення платформера багатоаспектним і складним, що суттєво збільшує як наукову, так і практичну значущість нашого дослідження.

Мета роботи - розробка 2D платформера на базі рушія Unity з

реалізацією основних ігрових механік та дослідженням ефективних підходів до створення ігрових застосунків.

Для успішного досягнення окресленої мети доцільно виконати такі кроки:

— здійснити огляд актуальних технологічних рішень для створення 2D ігор;

— детально вивчити характерні риси жанру платформер;

— дослідити функціонал, який надає ігровий рушій Unity;

— розробити архітектурну схему кінцевого програмного продукту;

— втілити базові ігрові функції(переміщення героя, взаємодія з оточенням, виявлення колізій, анімаційні ефекти);

— створити ігрові локації та елементи користувацького середовища;

— провести випробування проєкту та його подальше вдосконалення. Об’єкт дослідження - процес розробки 2D ігрових застосунків.

Предмет дослідження – методи та засоби створення 2D платформера з використанням рушія Unity.

Методологія дослідження. Залучено підходи системного аналізу, принципи об'єктно-орієнтованого підходу, техніки моделювання ігрових механік, а також різноманітні методи тестування програмних продуктів.

Новизна наукових здобутків полягає у вибудові архітектури двовимірного платформера, де компоненти взаємодіють найбільш доцільним чином, що гарантує

якісне керування елементами гри та покращення швидкодії програми.

Практична цінність досягнутих висновків полягає у формуванні завершеного ігрового продукту, який здатний слугувати зразком для опанування методологій створення ігор, а також може бути використаний як фундамент для майбутнього удосконалення та розширення можливостей.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ РІШЕНЬ

1.1 Загальна характеристика 2D ігор

2D-ігри є одним із базових та найпоширеніших напрямів у сфері розробки комп'ютерних ігор. Вони характеризуються використанням двовимірної графіки, де всі об'єкти розташовуються у площині, що визначається двома координатами — по горизонталі та вертикалі. Незважаючи на активний розвиток тривимірних технологій, 2D-ігри залишаються актуальними завдяки своїй простоті, доступності та ефективності реалізації.

Основною особливістю 2D-ігор є спосіб представлення ігрового світу.

Графічні елементи, такі як персонажі, фони, об'єкти та ефекти, зазвичай реалізуються у вигляді спрайтів або тайлів. Це дозволяє значно спростити процес розробки та оптимізувати використання обчислювальних ресурсів. Крім того, 2D-графіка часто має стилізований вигляд, що надає розробникам широкі можливості для творчої реалізації.

Важливим аспектом 2D-ігор є організація ігрового процесу. Більшість таких ігор базується на простих та зрозумілих механіках, які легко сприймаються користувачем. До них належать переміщення персонажа, взаємодія з об'єктами, подолання перешкод та досягнення певних цілей. Завдяки цьому 2D-ігри часто мають інтуїтивно зрозуміле керування та не потребують тривалого навчання.

Ще однією характерною рисою є використання ігрового циклу, який включає обробку введення користувача, оновлення стану гри та відображення результату на екрані. Цей підхід забезпечує безперервність ігрового процесу та дозволяє реалізувати динамічну взаємодію між гравцем і системою. У 2D-іграх особлива увага приділяється швидкодії та плавності анімацій, оскільки це безпосередньо впливає на якість сприйняття гри.

Суттєве значення має також фізична модель гри. Хоча у 2D-іграх вона є спрощеною порівняно з тривимірними системами, вона все ж забезпечує

реалістичність руху об’єктів, обробку зіткнень та взаємодію між елементами ігрового середовища. Використання фізичних рушіїв дозволяє автоматизувати ці процеси та підвищити точність розрахунків.

З точки зору розробки, 2D-ігри мають ряд переваг. Вони потребують менше ресурсів як на етапі створення, так і під час виконання, що робить їх доступними для широкого кола розробників. Крім того, такі ігри можуть ефективно працювати на різних платформах, включаючи персональні комп’ютери, мобільні пристрої та веб-середовище.

1.2 Особливості жанру платформер

Платформери є одним із найпоширеніших жанрів 2D-ігор, основною характеристикою яких є пересування ігрового персонажа по рівнях, що складаються з платформ, перешкод та інтерактивних об’єктів. Геймплей у таких іграх базується на поєднанні точного керування рухом, стрибків та взаємодії з оточенням, що формує основну ігрову механіку.

Ключовою особливістю жанру є необхідність подолання вертикальних та горизонтальних перешкод за допомогою стрибків і маневрування між платформами. Це вимагає від гравця точності у керуванні персонажем та швидкої реакції на зміни ігрової ситуації. У зв’язку з цим важливу роль відіграє коректна реалізація фізики руху, включаючи гравітацію, прискорення та зіткнення з об’єктами.

Ігрові рівні у платформерах зазвичай мають лінійну або частково нелінійну структуру, де гравець поступово просувається до кінцевої мети. При цьому рівні можуть містити різні типи перешкод, таких як рухомі платформи, пастки, вороги або інші елементи, що ускладнюють проходження. Така структура сприяє поступовому підвищенню складності гри.

Важливим елементом жанру є система взаємодії з ігровими об’єктами, персонаж може збирати предмети, активувати механізми або вступати у взаємодію з ворогами. Це дозволяє урізноманітнити ігровий процес та підвищити його

динамічність. Крім того, часто використовуються системи очок, здоров’я або прогресу, що додає елемент змагання або розвитку.

Окрему увагу у платформерах приділяють дизайну рівнів. Від правильного розташування платформ, ворогів і об’єктів залежить баланс складності та зручність проходження. Добре спроєктований рівень забезпечує поступове навчання гравця та підтримує інтерес до гри протягом всього ігрового процесу.

1.3 Аналіз існуючих аналогів

Аналіз існуючих аналогів у жанрі 2D платформерів є важливим етапом передпроєктного дослідження, оскільки дозволяє визначити актуальні підходи до реалізації ігрових механік, архітектури програмних рішень, організації ігрових рівнів та взаємодії користувача з системою. Результати такого аналізу використовуються для формування вимог до розроблюваного застосунку, вибору технологічного стеку та визначення рівня складності реалізації основних функцій.

Сучасні 2D платформери є одним із найстабільніших та найпоширеніших жанрів ігрової індустрії. Вони охоплюють широкий спектр реалізацій — від простих аркадних проєктів до складних багаторівневих ігор із розвиненою фізикою, сюжетом та художнім оформленням. Незважаючи на відмінності в реалізації, всі вони базуються на спільному принципі двовимірного ігрового простору, де основна взаємодія відбувається через переміщення персонажа по платформах та взаємодію з об’єктами середовища.

Важливою особливістю жанру є еволюція ігрових механік. Якщо ранні представники жанру мали обмежений набір дій та просту фізичну модель, то сучасні аналоги використовують складні системи керування рухом, інерцію, прискорення, гравітацію та точну обробку колізій. Це дозволяє створювати більш реалістичний та динамічний ігровий процес, який підвищує залученість гравця.

Окрему увагу в сучасних платформерах приділяють дизайну рівнів. Рівні стають багатошаровими, містять приховані області, альтернативні шляхи проходження та інтерактивні елементи. Це забезпечує нелінійність проходження та

підвищує реіграбельність гри. Крім того, активно використовуються системи прогресії, які можуть включати покращення персонажа, відкриття нових здібностей або механік.

Ще одним важливим аспектом є розвиток художнього стилю та аудіовізуального оформлення. Сучасні 2D платформери часто використовують стилізовану графіку, яка може варіюватися від мінімалістичної до детально промальованої. Візуальні ефекти, анімації та звуковий супровід відіграють важливу роль у формуванні атмосфери гри та емоційного сприйняття користувачем.

Також слід зазначити значний вплив ігрових рушіїв на розвиток жанру. Використання сучасних технологій, таких як Unity або подібні платформи, дозволяє значно спростити процес розробки, забезпечити кросплатформеність та підвищити продуктивність реалізації. Це сприяє появі як інді-проєктів, так і комерційно успішних ігор.

1.3.1 Класичні 2D платформери

Класичні 2D платформери є фундаментом усього жанру та визначили

основні принципи його подальшого розвитку. Вони з’явилися на ранніх етапах розвитку ігрової індустрії та стали одними з перших ігрових форм, що широко використовували двовимірну графіку та просту, але чітко структуровану ігрову логіку.

Основною особливістю класичних платформерів є їх орієнтація на простий та інтуїтивно зрозумілий ігровий процес. Базові дії гравця зазвичай обмежуються переміщенням персонажа вліво/вправо, стрибками та взаємодією з об’єктами (збір предметів, уникнення ворогів, активація простих механізмів). Такий набір дій дозволяє сфокусувати увагу на точності керування та реакції гравця.

Рівні у класичних платформерах, як правило, мають лінійну структуру з поступовим ускладненням. На початкових етапах гравця знайомлять із базовими механіками, після чого складність підвищується за рахунок більш складного розташування платформ, появи рухомих об’єктів та збільшення швидкості ігрового

процесу. Такий підхід формує логічну криву навчання (learning curve), яка поступово адаптує гравця до гри. Важливою характеристикою є спрощена фізична модель. У класичних іграх гравітація, зіткнення та рух персонажа реалізовані таким чином, щоб забезпечити передбачуваність і стабільність. Це дозволяє уникнути складних обчислень та зосередитися на геймплейних механіках, а не на фізичній реалістичності.

З точки зору графіки, класичні платформери використовують піксельну або спрайтову візуалізацію. Це було зумовлено технічними обмеженнями апаратного забезпечення того часу, однак згодом стало впізнаваним стилем жанру. Незважаючи на простоту, така графіка забезпечує високу читабельність ігрових елементів.

Одним із найвідоміших представників класичних 2D платформерів є гра *Super Mario Bros*, яка заклала основи сучасного розуміння жанру.

Геймплей гри базується на переміщенні персонажа Mio (Mario) по рівнях, що складаються з платформ, блоків та ворогів. Основною метою є проходження рівня до кінця, долаючи перешкоди та збираючи монети. Гравець повинен точно розраховувати стрибки, уникати ворогів та використовувати особливості рівня для просування вперед.

Особливістю гри є поступове введення нових елементів геймплею. Спочатку гравець навчається базовому руху та стрибкам, після чого з’являються вороги, рухомі платформи та більш складні комбінації перешкод. Це створює природний процес навчання без необхідності додаткових інструкцій.

Рис. 1.1 Загальний вигляд рівня Super Mario Bros

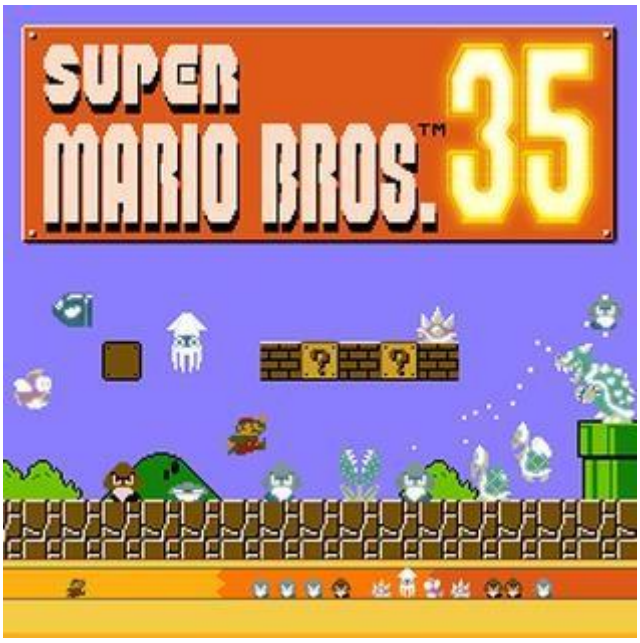


Рис. 1.2 Момент стрибка через перешкоду

Також важливим елементом є система винагороди, яка реалізована через збір монет, бонусів та прихованих об’єктів. Це стимулює дослідження рівнів та повторне проходження.

Фізика гри є простою та передбачуваною: персонаж має фіксовану гравітацію, обмежену інерцію та чітку реакцію на зіткнення з об’єктами. Це забезпечує високу точність керування, що є критично важливим для платформерів.

Рис. 1.3 Взаємодія з ворогами (Goomba/Koopa)

Рис. 1.4 Збір монет та бонусів

1.3.2 Сучасні інді-платформери

Сучасні інді-платформери є важливим етапом розвитку жанру, який характеризується поєднанням класичних ігрових механік із новими підходами до дизайну, фізики та подачі ігрового матеріалу. На відміну від класичних ігор, інді-проєкти створюються невеликими командами або окремими

розробниками, що дозволяє їм експериментувати з ігровим процесом, художнім стилем та структурою рівнів.

Однією з ключових особливостей інді-платформерів є їх орієнтація на унікальний ігровий досвід. Це досягається шляхом впровадження

нестандартних механік, таких як подвійні стрибки, взаємодія з середовищем, зміна станів персонажа або нові способи переміщення. Такі механіки значно розширюють класичну модель платформера.

Інді-ігри часто мають підвищений рівень складності, що вимагає від гравця високої точності керування та швидкої реакції. При цьому складність реалізується поступово, що дозволяє гравцеві адаптуватися до нових умов.

Важливу роль відіграє дизайн рівнів. У сучасних інді-платформерах рівні часто є нелінійними, містять альтернативні маршрути, приховані елементи та інтерактивні об’єкти. Це підвищує інтерес до гри та стимулює дослідження.

Також слід відзначити значення художнього стилю. Інді-платформери часто мають унікальну візуальну ідентичність, яка відрізняє їх від інших ігор. Використання кольору, освітлення та анімації дозволяє створити атмосферу, що є важливою частиною ігрового процесу.

Гра *Hollow Knight* є одним із найуспішніших інді-платформерів, яка поєднує класичні механіки платформера з елементами дослідження та бойової системи.

Основною особливістю гри є велика відкрита карта, яка складається з взаємопов’язаних локацій. Гравець поступово відкриває нові області, отримуючи доступ до них через здобуття нових здібностей персонажа. Це створює ефект нелінійного проходження.

Рис. 1.5 Загальний вигляд ігрового світу Hollow Knight

Рис. 1.6 Переміщення персонажа (стрибки, ривки)

Механіка руху включає не лише стандартні стрибки, але й додаткові можливості, такі як ривок, стрибок від стін та інші здібності, які відкриваються в процесі гри. Це значно розширює можливості переміщення та взаємодії з рівнями.

Окрему роль відіграє бойова система, яка додає глибині геймплею. Гравець взаємодіє з ворогами, використовуючи атаки та ухилення, що вимагає точного контролю та планування дій.

Рис. 1.7 Взаємодія з ворогами

Рис. 1.8 Карта світу

1.3.3 AAA-платформери

AAA-платформери є найбільш технологічно складними представниками жанру, які розробляються великими студіями з використанням значних фінансових і людських ресурсів. Такі ігри характеризуються високим рівнем графічної деталізації, складною анімацією, продуманою фізичною моделлю та широким набором ігрових механік.

Однією з основних особливостей ААА-платформерів є орієнтація на якість візуального та аудіального оформлення. Графіка в таких іграх часто включає деталізовані текстури, динамічне освітлення, ефекти частинок та плавні анімації персонажів. Це дозволяє створити глибоку ігрову атмосферу та підвищити рівень занурення гравця.

Ігровий процес у ААА-платформерах є значно складнішим у порівнянні з класичними та інді-іграми. Окрім базових механік (рух, стрибки), такі проєкти часто включають бойову систему, взаємодію з оточенням, головоломки, кооперативні елементи та сюжетні вставки. Це робить ігровий процес більш різноманітним і багатогранним.

Важливим аспектом є дизайн рівнів. У ААА-платформерах рівні мають складну структуру, можуть бути як лінійними, так і напіввідкритими, містити альтернативні шляхи проходження, секрети та інтерактивні елементи. Значна увага приділяється балансу складності та логіці розташування об’єктів.

З технічної точки зору, такі ігри використовують складні системи керування станами персонажа (state machines), фізичні рушії, системи колізій та оптимізації продуктивності. Важливу роль відіграє також кросплатформеність, оскільки ААА-проєкти зазвичай випускаються для різних платформ.

Гра *Rayman Legends* є одним із найяскравіших представників ААА 2D платформерів, яка поєднує високоякісну графіку, динамічний геймплей та продуманий дизайн рівнів.

Основною особливістю гри є плавна анімація персонажів та високий рівень деталізації графіки. Усі елементи гри — від персонажів до фону — виконані у художньому стилі з використанням сучасних технологій рендерингу.

Геймплей базується на класичних механіках платформера, але доповнений додатковими елементами, такими як взаємодія з об’єктами, кооперативний режим та ритмічні рівні. Останні синхронізовані з музикою, що додає унікальності ігровому процесу.

Рівні у *Rayman Legends* відзначаються великою різноманітністю. Вони містять рухомі платформи, пастки, ворогів та інтерактивні елементи, що змінюють

поведінку гравця. Дизайн рівнів побудований таким чином, щоб постійно змінювати умови гри та підтримувати інтерес користувача.

Рис. 1.9 Загальний вигляд рівня Rayman Legends

Особливістю гри є також кооперативний режим, який дозволяє декільком гравцям взаємодіяти в межах одного рівня. Це додає новий рівень складності та варіативності проходження.

З технічної точки зору, гра демонструє ефективну оптимізацію, що дозволяє зберігати високу продуктивність навіть при великій кількості об’єктів та ефектів на екрані.

Рис. 1.10 Анімація персонажів

Рис. 1.11 Ритмічний рівень

Рис. 1.12 Кооперативний режим

1.3.4 Мобільні платформери

Мобільні платформери є окремим напрямом розвитку жанру, який сформувався з появою смартфонів та планшетів. Основною особливістю таких ігор є адаптація ігрового процесу до сенсорного керування, обмежених обчислювальних ресурсів мобільних пристроїв та коротких ігрових сесій.

На відміну від класичних ігор, мобільні платформери орієнтовані на максимальну доступність для широкого кола користувачів. Це досягається шляхом спрощення механік керування, мінімізації кількості кнопок та використання інтуїтивних дій, таких як натискання або свайпи. У багатьох випадках керування

зводиться до однієї дії, наприклад, стрибка або зміни напрямку руху.

Однією з характерних рис мобільних платформерів є швидкий темп гри та короткі рівні або ігрові сесії. Це пов’язано з особливостями використання мобільних пристроїв, коли користувач взаємодіє з грою протягом обмеженого часу. У зв’язку з цим активно застосовуються механіки нескінченного бігу (endless runner) або рівнів із високою щільністю подій.

Важливим аспектом є оптимізація продуктивності. Мобільні пристрої мають обмежені ресурси, тому графіка в таких іграх зазвичай є спрощеною, але стилізованою. Часто використовуються мінімалістичні або геометричні форми, які дозволяють зменшити навантаження на систему.

Також мобільні платформери відрізняються підходами до монетизації.

Більшість таких ігор використовують модель free-to-play, що передбачає безкоштовний доступ із можливістю здійснення внутрішніх покупок або перегляду реклами. Це впливає на дизайн гри, зокрема на систему прогресії, винагород та повторюваності ігрового процесу.

З точки зору технічної реалізації, мобільні платформери також використовують сучасні ігрові рушії, зокрема Unity, що дозволяє забезпечити кросплатформеність та ефективну оптимізацію під різні типи пристроїв.

Гра *Geometry Dash* є популярним представником мобільних платформерів, що поєднує просту механіку керування з високою складністю проходження.

Основною особливістю гри є автоматичний рух персонажа вперед. Гравець не керує напрямком руху, а лише виконує стрибки у потрібний момент. Це значно спрощує керування, але водночас підвищує вимоги до точності реакції.

Геймплей побудований на синхронізації дій гравця з музичним супроводом. Рівні створені таким чином, щоб перешкоди відповідали ритму музики, що додає динамічності та унікальності ігровому процесу.

Дизайн рівнів у *Geometry Dash* характеризується високою щільністю перешкод та швидким темпом. Гравець повинен швидко реагувати на зміну ситуації, що робить гру складною навіть для досвідчених користувачів.

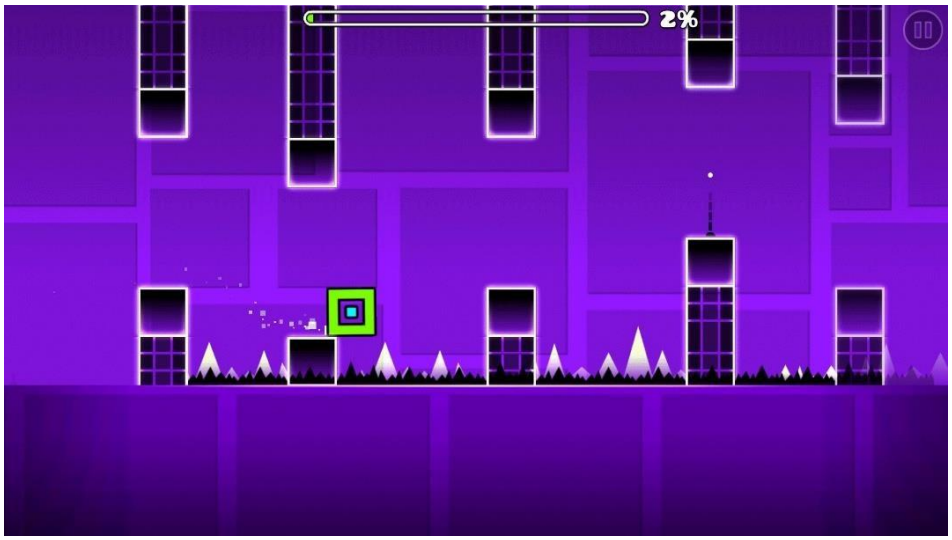


Рис. 1.13 Загальний вигляд рівня Geometry Dash

Рис. 1.14 Геймплей

Графічний стиль гри є мінімалістичним і базується на геометричних формах та яскравих кольорах. Це дозволяє зменшити навантаження на пристрій та забезпечити стабільну продуктивність.

Особливістю гри є також наявність редактора рівнів, який дозволяє користувачам створювати власний контент. Це значно розширює можливості гри та підвищує її популярність.

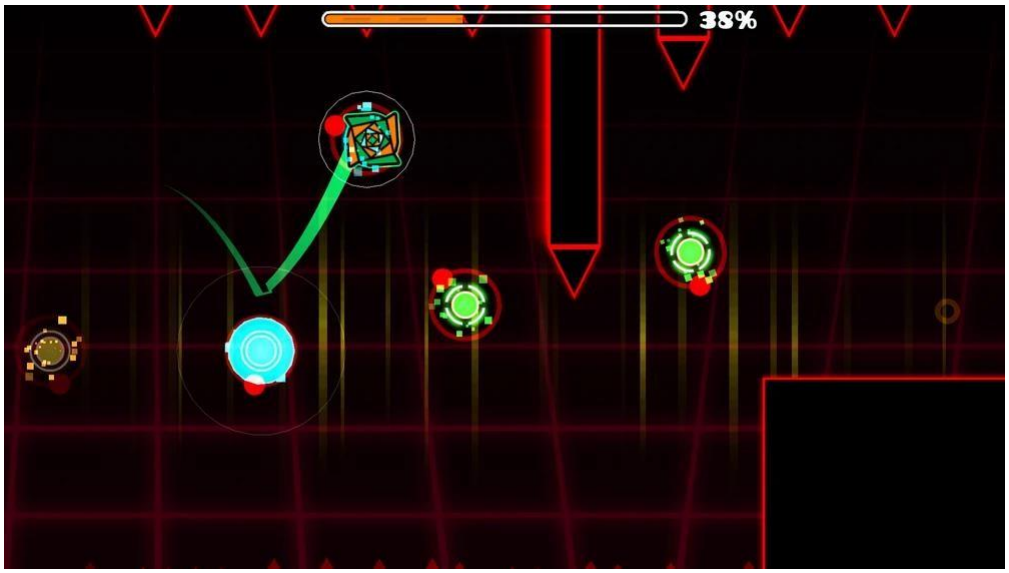


Рис. 1.15 Користувацький рівень

Рис. 1.16 Редактор рівнів

1.3.5 Роль рушіїв у реалізації платформерів

Сучасна розробка 2D платформерів практично неможлива без використання ігрових рушіїв, які забезпечують базову інфраструктуру для створення, тестування та запуску ігрових застосунків. Ігровий рушій являє собою програмне середовище, що містить набір інструментів і бібліотек для реалізації ключових компонентів гри, таких як графіка, фізика, обробка введення, анімація та управління ресурсами.

Однією з основних функцій рушія є забезпечення рендерингу графіки. У

випадку 2D платформерів це включає відображення спрайтів, тайлів, анімацій та візуальних ефектів. Завдяки цьому розробник може зосередитися на логіці гри, не витрачаючи час на реалізацію низькорівневих графічних алгоритмів.

Важливим компонентом є система фізики, яка відповідає за моделювання руху об’єктів, гравітацію, інерцію та обробку зіткнень. У платформерах це критично важливо, оскільки від коректності фізичної моделі залежить точність керування персонажем і якість ігрового процесу. Рушії надають готові фізичні модулі (наприклад, Physics2D), що значно спрощує реалізацію цих функцій.

Також рушії забезпечують систему компонентів та об’єктно-орієнтовану структуру, яка дозволяє організувати код гри у вигляді взаємодіючих модулів. Це спрощує розробку, тестування та подальше розширення функціоналу. Наприклад, персонаж може складатися з окремих компонентів: руху, анімації, фізики та обробки введення.

Ще одним важливим аспектом є система анімації. Ігрові рушії надають інструменти для створення та керування анімаційними станами персонажів, що дозволяє реалізувати плавні переходи між діями, такими як біг, стрибок або атака.

Крім того, рушії забезпечують обробку введення користувача, що є критично важливим для платформерів, де точність керування має вирішальне значення. Це включає підтримку різних пристроїв введення — клавіатури, геймпадів або сенсорних екранів.

Важливою перевагою сучасних рушіїв є кросплатформеність. Вони дозволяють створювати ігри, які можуть запускатися на різних платформах (ПК, мобільні пристрої, консолі) без суттєвих змін у коді. Це значно розширює можливості використання розробленого продукту.

Серед найбільш популярних рушіїв для створення 2D платформерів можна виділити Unity, Godot та Unreal Engine (з підтримкою 2D). Unity є одним із найпоширеніших рішень завдяки своїй гнучкості, наявності великої кількості готових компонентів та активній спільноті розробників.

Рушій Unity широко використовується для розробки 2D платформерів завдяки своїм можливостям та зручності використання. Він надає повний набір

інструментів для створення ігрового застосунку — від редактора сцен до системи фізики та анімації.

Основною структурною одиницею в Unity є об’єкт (GameObject), до якого додаються компоненти. Для створення платформера зазвичай використовуються такі компоненти, як Transform (позиція об’єкта), Sprite Renderer (відображення графіки), Rigidbody2D (фізика) та Collider (обробка зіткнень).

Реалізація руху персонажа здійснюється за допомогою скриптів на мові C#, які обробляють введення користувача та змінюють параметри фізичного тіла. Це дозволяє реалізувати точне керування, що є критично важливим для платформерів.

Система анімації (Animator) використовується для створення плавних переходів між станами персонажа, такими як стояння, біг або стрибок. Це підвищує якість візуального сприйняття гри.

Також Unity надає інструменти для створення рівнів, зокрема Tilemap, що дозволяє швидко формувати ігрове середовище з використанням тайлів. Це значно спрощує процес level design.

Завдяки вбудованим інструментам тестування та профілювання, розробник може аналізувати продуктивність гри та оптимізувати її для різних платформ.

Рис. 1.17 Інтерфейс Unity

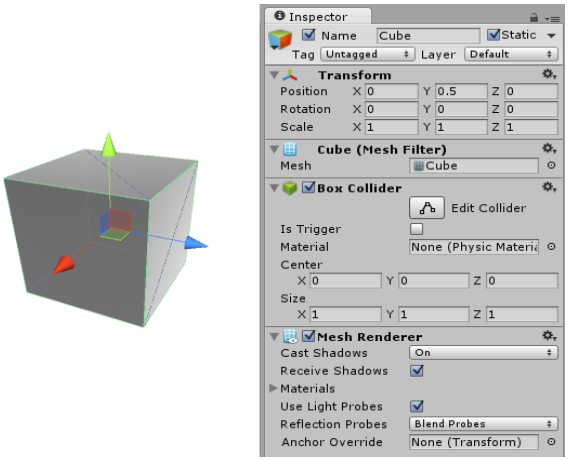


Рис. 1.18 Структура GameObject з компонентами

Рис. 1.19 Приклад Tilemap рівня

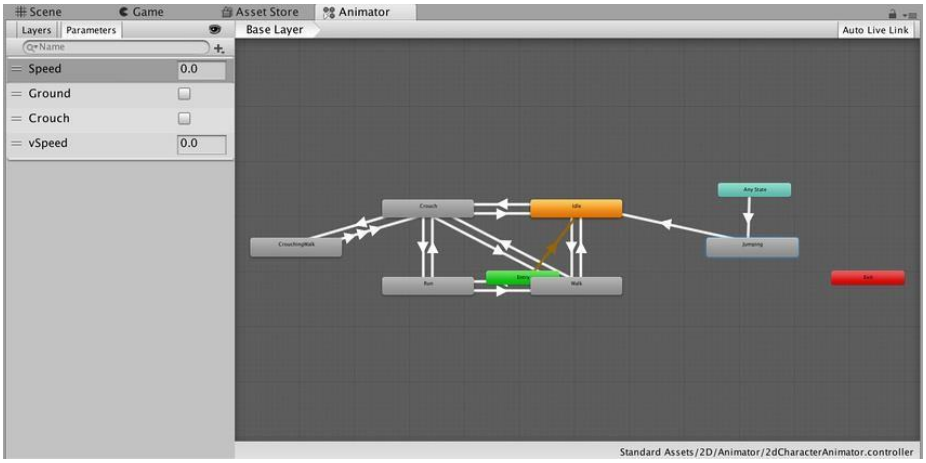


Рис. 1.20 Animator Controller

1.3.6 Узагальнення аналізу

Проведений аналіз існуючих аналогів у жанрі 2D платформерів дозволяє зробити висновок, що даний напрям є добре сформованим, але водночас продовжує активно розвиватися під впливом нових технологій та підходів до розробки ігор. Розглянуті категорії — класичні, інді, AAA та мобільні платформери — демонструють різні рівні складності, цільові аудиторії та підходи до реалізації ігрового процесу.

Класичні платформери заклали базові принципи жанру, зокрема просту та зрозумілу механіку керування, лінійну структуру рівнів і поступове ускладнення геймплею. Ці підходи залишаються актуальними і сьогодні, оскільки забезпечують зручність сприйняття гри та швидке навчання користувача.

Сучасні інді-платформери значно розширили можливості жанру, впроваджуючи нові механіки, складніший дизайн рівнів та унікальні художні рішення. Вони орієнтовані на створення індивідуального ігрового досвіду та часто експериментують із геймплеєм, що дозволяє формувати нові тенденції розвитку жанру.

AAA-платформери демонструють найвищий рівень технічної реалізації, поєднуючи складну фізику, деталізовану графіку, розвинені анімації та різноманітні ігрові механіки. Вони задають стандарти якості та впливають на розвиток індустрії в цілому.

Мобільні платформери, у свою чергу, адаптують жанр до особливостей мобільних пристроїв, спрощуючи керування та структуру гри, але водночас забезпечуючи високий рівень динаміки та доступності. Вони орієнтовані на масову аудиторію та короткі ігрові сесії.

Окрему роль у розвитку платформерів відіграють ігрові рушії, які значно спрощують процес розробки, забезпечують кросплатформеність та надають готові інструменти для реалізації фізики, графіки та логіки гри. Використання таких рушіїв, як Unity, дозволяє реалізовувати як прості, так і складні проєкти з мінімальними витратами часу та ресурсів.

Узагальнюючи результати аналізу, можна виділити основні тенденції розвитку жанру 2D платформерів:

— ускладнення ігрових механік та фізичної моделі;

— підвищення ролі дизайну рівнів;

— розвиток художнього стилю та анімації;

— використання сучасних рушіїв розробки;

— адаптація під різні платформи та пристрої.

Отримані результати аналізу є основою для формування вимог до розроблюваного програмного продукту. Зокрема, при створенні 2D платформера доцільно враховувати класичні принципи побудови геймплею, доповнюючи їх сучасними механіками та використовуючи можливості ігрового рушія Unity для реалізації ефективної та масштабованої системи.

2 ПРОЄКТУВАННЯ 2D ПЛАТФОРМЕРА

2.1 Постановка задачі

Метою даної кваліфікаційної роботи є проєктування та розробка 2D платформера на базі ігрового рушія Unity, який реалізує основні механіки

жанру, забезпечує інтерактивну взаємодію користувача з ігровим середовищем та відповідає сучасним вимогам до якості програмного забезпечення.

У рамках поставленої задачі необхідно створити програмний застосунок, що дозволяє користувачу керувати ігровим персонажем у двовимірному просторі, долати перешкоди, взаємодіяти з об’єктами та проходити рівні різної складності. Особливу увагу слід приділити точності керування, стабільності фізичної моделі та логіці взаємодії між елементами ігрового середовища.

Основна задача розробки полягає у створенні цілісної ігрової системи, яка включає:

— модуль керування персонажем із реалізацією руху та стрибків;

— систему фізики для обробки гравітації та зіткнень;

— механізми взаємодії з платформами, перешкодами та іншими об’єктами;

— систему рівнів із поступовим ускладненням;

— інтерфейс користувача для відображення ігрової інформації.

Крім того, важливим аспектом є забезпечення логічної структури гри, що передбачає чіткий поділ на сцени (меню, ігровий процес, завершення рівня), а також реалізацію механізмів переходу між ними.

У процесі виконання роботи необхідно вирішити наступні задачі:

— провести аналіз предметної області та існуючих аналогів;

— визначити функціональні та нефункціональні вимоги до застосунку;

— спроєктувати архітектуру програмного забезпечення;

— розробити ігрові механіки та систему рівнів;

— реалізувати програмний продукт із використанням Unity;

— провести тестування та оцінку працездатності застосунку.

Додатково необхідно врахувати вимоги до продуктивності та оптимізації, оскільки ігровий застосунок повинен стабільно працювати без затримок навіть при наявності великої кількості об’єктів на сцені.

2.2 Визначення функціональних вимог

Функціональні вимоги визначають перелік можливостей та дій, які повинен забезпечувати розроблюваний 2D платформер. Вони описують поведінку системи з точки зору користувача та визначають основний функціонал гри, що реалізується в процесі розробки.

У контексті даного проєкту функціональні вимоги охоплюють усі ключові аспекти ігрового процесу: керування персонажем, взаємодію з ігровим середовищем, організацію рівнів, а також відображення інформації користувачу.

Однією з основних вимог є реалізація системи керування персонажем. Користувач повинен мати можливість переміщувати персонажа у горизонтальному напрямку, виконувати стрибки, а також здійснювати додаткові дії, передбачені механікою гри. Керування має бути чутливим і забезпечувати точну реакцію на дії гравця.

Важливою функціональною складовою є система фізики, яка забезпечує реалістичну взаємодію об’єктів. Вона повинна включати моделювання гравітації, обробку зіткнень між персонажем і об’єктами, а також коректну реакцію на ці взаємодії (зупинка, відштовхування, падіння).

Система рівнів є ще одним ключовим елементом. Гра повинна містити набір рівнів із поступовим ускладненням. Користувач повинен мати можливість проходити рівні, переходити до наступних після виконання умов та повертатися до попередніх у разі необхідності.

Необхідно також реалізувати взаємодію з ігровими об’єктами. Це включає:

— взаємодію з платформами (статичними та рухомими);

— взаємодію з перешкодами (пастки, небезпечні зони);

— взаємодію з ворогами або іншими активними елементами гри;

— збір предметів або бонусів (за наявності).

Окрему роль відіграє система станів гри. Вона повинна забезпечувати можливість початку гри, паузи, завершення рівня, а також перезапуску після поразки. Це передбачає наявність відповідних сцен або режимів роботи застосунку.

Інтерфейс користувача також входить до функціональних вимог. Гра повинна відображати необхідну інформацію, зокрема:

— поточний стан гри (меню, гра, завершення);

— показники гравця (життя, очки, час);

— повідомлення або підказки.

2.3 Визначення нефункціональних вимог

Нефункціональні вимоги визначають якісні характеристики програмного продукту, які описують умови його роботи, обмеження, а також загальні

параметри продуктивності та зручності використання. Вони не описують безпосередній функціонал гри, але є критично важливими для забезпечення стабільності, ефективності та користувацького досвіду.

Для розроблюваного 2D платформера нефункціональні вимоги

спрямовані на забезпечення плавного ігрового процесу, коректної роботи фізичної моделі, зручного інтерфейсу та сумісності з різними платформами.

Однією з ключових вимог є продуктивність. Гра повинна забезпечувати стабільну частоту кадрів без помітних затримок під час ігрового процесу. Особливо це важливо при одночасній обробці великої кількості об’єктів, анімацій та фізичних взаємодій.

Наступною вимогою є оптимізація використання ресурсів. Програмний продукт повинен ефективно використовувати процесорний час та оперативну пам’ять, що особливо актуально при запуску гри на пристроях із різними

технічними характеристиками.

Важливою нефункціональною вимогою є зручність використання.

Інтерфейс користувача повинен бути інтуїтивно зрозумілим, не перевантаженим зайвими елементами та забезпечувати швидкий доступ до основних функцій гри. Керування персонажем має бути логічним і передбачуваним.

Також необхідно враховувати надійність програмного забезпечення. Гра повинна коректно обробляти помилки, уникати критичних збоїв та забезпечувати стабільну роботу протягом усього ігрового сеансу.

Окремо слід виділити кросплатформеність. Оскільки використовується рушій Unity, гра повинна мати можливість запуску на різних платформах без суттєвих змін у вихідному коді.

Важливою вимогою є масштабованість системи. Архітектура проєкту повинна дозволяти розширення функціоналу, зокрема додавання нових рівнів, механік, типів ворогів або ігрових об’єктів без необхідності суттєвого перепроєктування всієї системи.

Також враховується підтримуваність коду. Структура проєкту повинна бути модульною, з чітким розподілом відповідальності між компонентами, що спрощує подальше тестування, модифікацію та розширення функціоналу.

2.4 Архітектура застосунку

Архітектура застосунку визначає загальну структуру програмної системи, взаємозв’язок її компонентів та принципи організації коду. Для розробки 2D платформера на базі Unity доцільно використовувати компонентно-орієнтовану архітектуру, яка є базовою для даного рушія та забезпечує високу гнучкість і масштабованість.

Основою архітектури є об’єкти сцени (GameObject), які представляють усі елементи гри: персонажа, платформи, ворогів, інтерфейс та інші об’єкти. Кожен об’єкт складається з набору компонентів, що визначають його поведінку. Такий

підхід дозволяє розділити функціональність на незалежні модулі та спростити управління системою.

Архітектуру застосунку можна умовно розділити на декілька основних підсистем.

Першою є підсистема керування персонажем. Вона відповідає за обробку введення користувача, рух персонажа, виконання стрибків та взаємодію з фізичним середовищем. Ця підсистема реалізується за допомогою скриптів, що працюють із компонентами Rigidbody2D та Collider.

Другою важливою складовою є фізична підсистема. Вона забезпечує моделювання гравітації, обробку зіткнень та взаємодію між об’єктами. Фізична модель повинна бути стабільною та передбачуваною, оскільки від неї залежить точність керування.

Наступною є підсистема анімації. Вона відповідає за відображення станів персонажа та інших об’єктів гри. За допомогою Animator Controller реалізуються переходи між анімаціями (наприклад, стояння, рух, стрибок), що забезпечує плавність візуального відображення.

Підсистема управління рівнями відповідає за завантаження, збереження та зміну ігрових сцен. Вона забезпечує перехід між рівнями, обробку умов завершення рівня та перезапуск гри у разі поразки.

Інтерфейс користувача (UI) є окремою підсистемою, яка відповідає за відображення інформації користувачу. До неї належать меню, індикатори стану, повідомлення та інші елементи взаємодії.

Також важливою частиною архітектури є система обробки подій. Вона забезпечує взаємодію між різними компонентами гри, наприклад, реагування на зіткнення, завершення рівня або отримання бонусів.

З точки зору організації коду, доцільно використовувати модульний підхід із розділенням логіки на окремі класи:

— клас керування персонажем;

— клас обробки фізики;

— клас управління рівнями;

— клас інтерфейсу користувача;

— допоміжні класи для обробки подій.

Така структура дозволяє забезпечити підтримуваність коду та спрощує його подальшу модифікацію.

2.5 Проєктування ігрових механік

2.5.1 Рух персонажа

Рух персонажа є базовою та критично важливою механікою 2D

платформера, оскільки саме від його реалізації залежить загальне відчуття керування, точність дій гравця та якість ігрового процесу. У межах даного проєкту рух персонажа реалізується у горизонтальній площині з використанням фізичної моделі, що забезпечує природну та передбачувану поведінку.

Основою реалізації руху є обробка введення користувача, яка дозволяє визначати напрямок переміщення (вліво або вправо). На основі цих даних змінюється швидкість персонажа з урахуванням параметрів прискорення та гальмування. Такий підхід дозволяє уникнути різких змін швидкості та забезпечує плавність руху.

Важливим аспектом є використання фізичного компонента, який дозволяє моделювати інерцію та взаємодію з іншими об’єктами. Рух реалізується не шляхом прямої зміни позиції, а через застосування сили або зміни швидкості, що забезпечує узгодженість із фізичною системою гри.

Для забезпечення комфортного керування вводиться обмеження максимальної швидкості персонажа. Це дозволяє уникнути ситуацій, коли персонаж стає важкокерованим при високих швидкостях. Додатково реалізується механізм гальмування, який активується при відсутності введення, що дозволяє персонажу поступово зупинятися.

Особливу увагу приділено точності керування. Важливо забезпечити

мінімальну затримку між дією користувача та реакцією персонажа. Це досягається шляхом оптимізації обробки введення та правильного вибору моменту оновлення фізичних параметрів.

Також враховується орієнтація персонажа у просторі. При зміні напрямку руху виконується віддзеркалення спрайта, що дозволяє візуально відобразити напрямок руху. Це підвищує зрозумілість та зручність сприйняття гри.

Додатково може бути реалізована перевірка стану персонажа, зокрема

визначення, чи знаходиться він на землі. Це важливо для обмеження деяких дій та забезпечення коректної логіки руху.

З технічної точки зору, реалізація руху передбачає:

— обробку введення користувача;
— розрахунок швидкості руху;
— застосування фізичних параметрів;
— обмеження швидкості;
— оновлення стану персонажа;
— синхронізацію з анімацією.

Синхронізація з анімаційною системою є важливою складовою, оскільки стан руху (рухається або стоїть) впливає на відтворення відповідних анімацій. Це дозволяє забезпечити узгодженість між логікою гри та її візуальним представленням.

2.5.2 Стрибки

Механіка стрибків є однією з ключових у 2D платформері, оскільки вона визначає можливості переміщення персонажа у вертикальній площині та безпосередньо впливає на складність проходження рівнів. Якість реалізації стрибка суттєво впливає на відчуття керування, точність дій гравця та загальний баланс гри.

У межах даного проєкту реалізується базовий стрибок, який виконується при натисканні відповідної клавіші або кнопки. Стрибок ініціюється лише за умови, що

персонаж знаходиться на поверхні (на платформі), що визначається за допомогою перевірки контакту з землею (ground check). Це запобігає некоректній поведінці та забезпечує контрольованість механіки.

Реалізація стрибка базується на використанні фізичної моделі. При виконанні стрибка до персонажа застосовується імпульс сили у вертикальному напрямку, що визначає початкову швидкість підйому. Подальший рух персонажа відбувається під впливом гравітації, яка поступово зменшує швидкість підйому та переводить рух у фазу падіння.

Одним із важливих параметрів є сила стрибка, яка визначає максимальну висоту, яку може досягти персонаж. Цей параметр підбирається таким чином, щоб забезпечити баланс між складністю та зручністю проходження рівнів. Занадто високий стрибок може спростити гру, тоді як занадто низький - ускладнити її.

Особлива увага приділяється налаштуванню гравітації. Вона визначає швидкість падіння та загальну динаміку руху. У деяких випадках може використовуватися збільшена гравітація під час падіння, що дозволяє зробити керування більш чутливим і точним.

Для підвищення якості геймплею можуть застосовуватися додаткові технічні прийоми, такі як:

— невелика затримка після сходу з платформи (coyote time), що дозволяє виконати стрибок навіть після втрати контакту;

— буферизація введення (jump buffer), яка враховує натискання кнопки до моменту приземлення.

Ці підходи дозволяють зробити керування більш «поблажливим» до користувача та підвищують комфорт гри.

Також важливим є коректне визначення стану персонажа: підйом, падіння або перебування на поверхні. Ці стани використовуються для синхронізації з анімацією, що дозволяє відображати відповідні візуальні ефекти (наприклад, анімацію стрибка або приземлення).

З технічної точки зору, реалізація механіки стрибків включає:

— перевірку контакту з поверхнею;

— обробку введення користувача;
— застосування імпульсу сили;
— обробку гравітації;
— оновлення стану персонажа;
— синхронізацію з анімаційною системою.

2.5.3 Система прогресії

Система прогресії є важливою складовою 2D платформера, оскільки вона визначає логіку розвитку ігрового процесу, послідовність проходження рівнів та поступове ускладнення гри. Її основне призначення полягає у підтримці інтересу гравця, формуванні відчуття досягнення та стимулюванні до подальшого проходження.

У межах даного проєкту система прогресії реалізується через поетапне проходження рівнів, кожен з яких має власні особливості, складність та унікальні механіки. Гравець починає з простих рівнів, які знайомлять його з базовими елементами керування та взаємодії, після чого складність поступово зростає за рахунок додавання нових перешкод, пасток і умов проходження.

Одним із ключових елементів прогресії є структурована побудова рівнів. Кожен рівень виконує навчальну функцію, поступово вводячи нові елементи гри. Такий підхід дозволяє гравцеві адаптуватися до змін та розвивати свої навички без різких стрибків складності.

Особливістю системи прогресії є впровадження унікальних навичок на кожному рівні. Кожен новий рівень відкриває нову можливість, яка змінює спосіб взаємодії з ігровим середовищем. Це дозволяє не лише ускладнювати гру, але й урізноманітнювати геймплей, запобігаючи його одноманітності.

Також важливим аспектом є контроль складності. Система прогресії повинна забезпечувати баланс між викликом і доступністю. Для цього

складність зростає поступово, а нові механіки вводяться в контрольованих умовах, що дозволяє гравцеві їх освоїти.

У межах реалізації передбачається можливість переходу між рівнями після виконання певних умов, наприклад:

— досягнення кінцевої точки рівня;

— подолання всіх основних перешкод;

— використання отриманих навичок.

У разі невдачі гравець має можливість повторного проходження рівня, що дозволяє вдосконалити навички та знайти оптимальний спосіб проходження.

Додатково система прогресії може включати облік результатів, таких як час проходження або кількість спроб. Це створює додаткову мотивацію для покращення власних результатів.

З технічної точки зору, система прогресії включає:

— управління переходами між рівнями;

— збереження стану проходження;

— контроль умов завершення рівня;

— інтеграцію з іншими механіками гри.

2.5.4 Взаємодія з платформами

Взаємодія з платформами є фундаментальною складовою ігрового процесу в 2D платформері, оскільки саме платформи формують основну структуру рівнів та визначають можливості переміщення персонажа. Якість реалізації цієї механіки безпосередньо впливає на точність керування, передбачуваність поведінки персонажа та загальний комфорт гри.

У межах даного проєкту передбачено використання двох основних типів платформ: статичних та тимчасових (зникаючих). Статичні платформи є базовими елементами рівня, які забезпечують стабільну поверхню для пересування персонажа. Тимчасові платформи, у свою чергу, зникають через певний час після контакту з персонажем, що створює додаткову складність та вимагає швидкого прийняття рішень.

Реалізація взаємодії з платформами базується на використанні системи колізій. Кожна платформа має колайдер, який визначає її фізичні межі, а персонаж — відповідний колайдер і фізичний компонент. При контакті цих об’єктів відбувається обробка зіткнення, що дозволяє персонажу стояти на платформі або взаємодіяти з нею іншим чином.

Одним із ключових аспектів є коректне визначення стану «на землі» (grounded). Це необхідно для забезпечення правильної роботи інших механік, зокрема стрибків. Перевірка може здійснюватися за допомогою додаткових сенсорів або перевірки контактів із поверхнею.

Важливим елементом є обробка країв платформ. Персонаж повинен коректно реагувати на ситуації, коли він знаходиться на межі платформи, не провалюючись і не зависаючи у повітрі. Це досягається завдяки точному налаштуванню колайдерів та фізичних параметрів.

Для тимчасових платформ реалізується механізм активації, який запускається при контакті з персонажем. Після активації платформа може:

— зникати через певний проміжок часу;

— втрачати колізію;

— змінювати свій візуальний стан.

Це створює додатковий динамізм у грі та змушує гравця швидко реагувати на зміну умов.

Також важливо враховувати стабільність взаємодії при русі персонажа. Переміщення по платформі повинно бути плавним, без ривків або несподіваних змін положення. Це забезпечується коректною обробкою фізичних параметрів та синхронізацією руху.

З технічної точки зору, механіка взаємодії з платформами включає:

— використання колайдерів для визначення меж об’єктів;

— обробку зіткнень між персонажем і платформами;

— визначення стану контакту з поверхнею;

— реалізацію логіки поведінки тимчасових платформ;

— синхронізацію з іншими механіками гри (рух, стрибки).

2.5.5 Уникнення або подолання перешкод

Механіка уникнення або подолання перешкод є однією з ключових складових геймплею 2D платформера, оскільки вона формує основний рівень складності та визначає вимоги до реакції, точності та стратегічного мислення гравця. Перешкоди виступають як обмежувальні або небезпечні елементи ігрового середовища, що змушують користувача адаптуватися до умов рівня.

У розроблюваній грі перешкоди реалізуються як статичні або умовно-активні об’єкти, розміщені в ігровому просторі таким чином, щоб ускладнювати проходження рівня. Основна мета цієї механіки полягає не лише у створенні складності, але й у формуванні логіки проходження рівнів через поступове навчання гравця розпізнаванню небезпечних ситуацій. Перешкоди можуть мати різні форми та властивості, зокрема:

— фізичні бар’єри, що блокують шлях персонажа;

— небезпечні зони, які призводять до втрати прогресу або перезапуску рівня;

— обмежені простори, що вимагають точного керування рухом;

— елементи середовища, які змінюють темп або ритм проходження.

Основним способом подолання перешкод є використання базових механік персонажа, зокрема руху та стрибків. Гравець повинен оцінювати відстані, таймінг та розташування об’єктів, щоб правильно виконати необхідні дії. Це сприяє розвитку координації та навичок точного керування.

Важливим аспектом є дизайн розташування перешкод. Вони не повинні бути випадковими, а мають формувати логічні послідовності, які поступово ускладнюються. Таким чином забезпечується природне навчання гравця без різких стрибків складності.

Окрему роль відіграє інтеграція перешкод із системою прогресії. На ранніх етапах вони виконують навчальну функцію, тоді як на пізніших рівнях комбінуються між собою, створюючи складні сценарії, які вимагають швидкого

прийняття рішень.

З технічної точки зору реалізація перешкод включає:

— використання колайдерів для визначення небезпечних зон;

— обробку подій зіткнення з персонажем;

— реалізацію логіки втрати життя або перезапуску рівня;

— налаштування візуального індикатора небезпеки;

— синхронізацію з системою рівнів та прогресії.

2.5.6 Пастки

Пастки є важливим елементом геймплейного дизайну 2D платформера, який використовується для підвищення складності рівнів та створення додаткових викликів для гравця. Вони формують ризиковані ділянки ігрового простору, де неправильний рух або невдалий таймінг призводить до негативних наслідків, таких як перезапуск рівня або повернення до контрольної точки.

У структурі гри пастки виступають як інтерактивні або статичні об’єкти, що активують певну логіку при взаємодії з персонажем. Вони можуть бути розміщені як на основному шляху проходження, так і в прихованих або альтернативних маршрутах, що змушує гравця уважно досліджувати рівень.

До основних типів пасток належать:

— шипи, що завдають миттєвої поразки при контакті;

— падаючі об’єкти, які активуються при наближенні персонажа або через таймер;

— небезпечні зони, які автоматично спричиняють втрату прогресу або перезапуск рівня.

Кожен тип пастки має власну логіку роботи та спосіб активації. Деякі пастки є постійно активними, тоді як інші спрацьовують лише за певних умов, що дозволяє урізноманітнити ігровий процес.

Особлива увага приділяється візуальному оформленню пасток. Вони повинні

бути достатньо помітними для гравця, щоб забезпечити можливість реагування, але водночас не втрачати елементу складності. Баланс між помітністю та небезпекою є важливим фактором геймдизайну.

Розташування пасток у рівнях є структурованим і логічно обґрунтованим.

Вони використовуються для:

— перевірки навичок гравця після засвоєння нових механік;

— ускладнення проходження у ключових точках рівня;

— створення комбінацій із платформами та іншими елементами середовища.

З технічної точки зору реалізація пасток включає:

— використання колайдерів для визначення контакту з персонажем;

— обробку подій зіткнення;

— запуск сценаріїв шкоди або перезапуску рівня;

— можливість активації через тригери або таймери;

— зміну візуального стану при активації (за потреби).

2.5.7 Унікальні навички персонажа

Унікальні навички персонажа є розширеною ігровою механікою, яка суттєво впливає на різноманітність геймплею та поступове ускладнення проходження рівнів. Вони вводяться як елемент прогресії та змінюють спосіб взаємодії гравця з ігровим середовищем, додаючи нові можливості для дослідження рівнів, подолання перешкод та вирішення ігрових задач.

У межах ігрової структури кожна навичка активується або відкривається на певному рівні, що дозволяє поступово знайомити гравця з новими механіками. Такий підхід забезпечує логічне навчання та зменшує різкий стрибок складності, оскільки кожна нова здатність впроваджується у контрольованих умовах.

До основних унікальних навичок належать:

— Ввімкнення освітлення — механіка, яка дозволяє активувати джерела світла або тимчасово покращувати видимість у темних ділянках рівня. Це може

використовуватися для виявлення прихованих платформ, пасток або альтернативних шляхів проходження. Дана навичка додає елемент дослідження та орієнтації в просторі.

— Стан духа — спеціальний режим персонажа, у якому він може проходити крізь певні об’єкти або перешкоди, недоступні у звичайному стані. Ця механіка використовується для вирішення просторових задач та подолання складних ділянок рівня, які неможливо пройти стандартними методами.

— Телепортація між спеціальними алтарями — механіка швидкого переміщення, що дозволяє переносити персонажа між заздалегідь визначеними точками рівня. Алтарі виступають як вузлові елементи маршруту, які відкривають нові шляхи або дозволяють обходити небезпечні зони.

Інтеграція унікальних навичок у геймплей вимагає ретельного балансу, оскільки вони не повинні повністю спрощувати проходження рівнів. Кожна навичка вводиться разом із відповідними обмеженнями або умовами використання, що зберігає ігровий виклик.

З точки зору ігрового дизайну, унікальні навички виконують декілька функцій:

— урізноманітнюють геймплей;

— розширюють можливості проходження рівнів;

— формують нові типи ігрових задач;

— підсилюють відчуття прогресії та розвитку персонажа.

З технічної реалізації система унікальних навичок включає:

— механізм активації здібностей;

— контроль умов використання (тригери, рівні, об’єкти);

— взаємодію з ігровими об’єктами та середовищем;

— зміну стану персонажа;

— інтеграцію з системою UI для відображення активної навички.

2.6 Проєктування рівнів

Проєктування рівнів є одним із ключових етапів розробки 2D платформера, оскільки саме рівні формують основний ігровий досвід, визначають темп проходження та задають логіку використання ігрових механік. Якість Level Design безпосередньо впливає на баланс складності, зрозумілість гри та загальне сприйняття гравцем ігрового процесу.

Рівні проєктуються як послідовність взаємопов’язаних сцен, які поступово ускладнюються та вводять нові елементи геймплею. Кожен рівень має власну структуру, мету проходження та набір механік, які використовуються для її досягнення. Такий підхід дозволяє забезпечити логічну прогресію та поступове навчання гравця.

При проєктуванні рівнів важливо дотримуватися принципу поступового ускладнення. На початкових етапах гравець знайомиться з базовими елементами керування та простими платформами, тоді як на пізніших рівнях вводяться комбіновані механіки, пастки та складні маршрути проходження. Це дозволяє уникнути різких стрибків складності та забезпечує природну адаптацію до ігрового процесу.

Важливу роль відіграє структура рівня. Зазвичай рівень умовно поділяється на кілька логічних зон: стартову, основну, ускладнену та фінальну. Стартова зона використовується для орієнтації гравця та введення в умови рівня. Основна зона містить базові елементи геймплею. У зоні ускладнення комбінуються різні механіки та перешкоди, а фінальна зона завершує рівень і підводить підсумок проходження.

Окрему увагу приділено розміщенню ігрових об’єктів. Платформи, пастки та перешкоди повинні формувати логічні маршрути руху гравця. Вони не розміщуються випадково, а створюють послідовності дій, які змушують гравця використовувати всі доступні механіки. Також важливо враховувати ритм рівня, чергуючи складні та прості ділянки для підтримки динаміки гри.

Інтеграція ігрових механік у рівні є важливою складовою дизайну. Нові

здібності персонажа вводяться через структуру рівня та обов’язкові ігрові ситуації. Наприклад, рівні можуть бути побудовані так, що використання освітлення необхідне для проходження темних зон, стан духа - для проходження крізь перешкоди, а телепортація - для переміщення між віддаленими частинами рівня.

Баланс складності забезпечується через поступове збільшення кількості та складності перешкод. На кожному етапі рівень має залишатися прохідним, але вимагати більшої точності, реакції та розуміння механік. Це досягається шляхом поєднання різних типів ігрових елементів та ретельного тестування проходження.

Важливим аспектом є візуальна навігація. Рівні повинні містити елементи, які допомагають гравцю інтуїтивно визначати напрямок руху. Це можуть бути світлові акценти, контрастні платформи або композиційне виділення ключових об’єктів. Такий підхід зменшує потребу у додаткових поясненнях і покращує сприйняття ігрового простору.

3 РОЗРОБКА ТА РЕАЛІЗАЦІЯ ЗАСТОСУНКУ

3.1 Середовище розробки та інструменти

Середовище розробки та набір інструментів визначають технологічну основу створення 2D платформера, впливають на продуктивність розробки, якість програмного коду та можливості подальшого розширення функціоналу. Обрані інструменти повинні забезпечувати повний цикл розробки: від проєктування та реалізації до тестування і супроводу програмного продукту.

Для створення гри використовується сучасний підхід, який поєднує ігровий рушій, мову програмування, редактор коду та систему контролю версій. Така комбінація дозволяє ефективно організувати процес розробки, розділити відповідальність між компонентами системи та забезпечити гнучкість у внесенні змін.

Важливою вимогою до середовища розробки є підтримка 2D-графіки, фізичного моделювання, анімації та обробки подій. Крім того, інструменти повинні забезпечувати можливість швидкого тестування, що дозволяє оперативно перевіряти реалізовані механіки та виявляти помилки на ранніх етапах.

Суттєвим аспектом є інтеграція всіх інструментів між собою. Рушій повинен коректно взаємодіяти з редактором коду, система контролю версій відслідковувати зміни у проєкті, а засоби налагодження — забезпечувати аналіз роботи програми. Така інтеграція дозволяє створити єдине середовище розробки, яке підвищує ефективність роботи.

Також враховується зручність використання інструментів. Інтерфейс середовища повинен бути інтуїтивно зрозумілим, а функціональні можливості доступними без значних витрат часу на навчання. Це особливо важливо при розробці ігор, де значну роль відіграє швидкість ітерацій - процесу створення, тестування та вдосконалення ігрових елементів. Не менш важливою є підтримка спільної роботи та керування версіями проєкту. Це дозволяє уникнути втрати

даних, контролювати зміни та забезпечити стабільність розробки навіть при внесенні значної кількості модифікацій.

3.1.1 Основне середовище розробки

Основним інструментом розробки 2D платформера є Unity - багатофункціональний ігровий рушій, який забезпечує повний цикл створення інтерактивних застосунків. Його використання дозволяє об’єднати процеси програмування, візуального проєктування, налаштування фізики та тестування в межах одного середовища, що значно спрощує розробку.

Unity підтримує роботу з 2D-графікою на високому рівні, зокрема через використання спрайтів, тайлсетів (Tilemap) та шарів сортування (Sorting Layers). Це дозволяє ефективно організовувати ігровий простір, керувати відображенням об’єктів та створювати складні сцени з великою кількістю елементів. Однією з ключових особливостей середовища є редактор сцен, який забезпечує візуальне проєктування рівнів. Розробник може у реальному часі розміщувати об’єкти, змінювати їх параметри та одразу бачити результат. Це значно пришвидшує процес створення рівнів і дозволяє швидко вносити зміни.

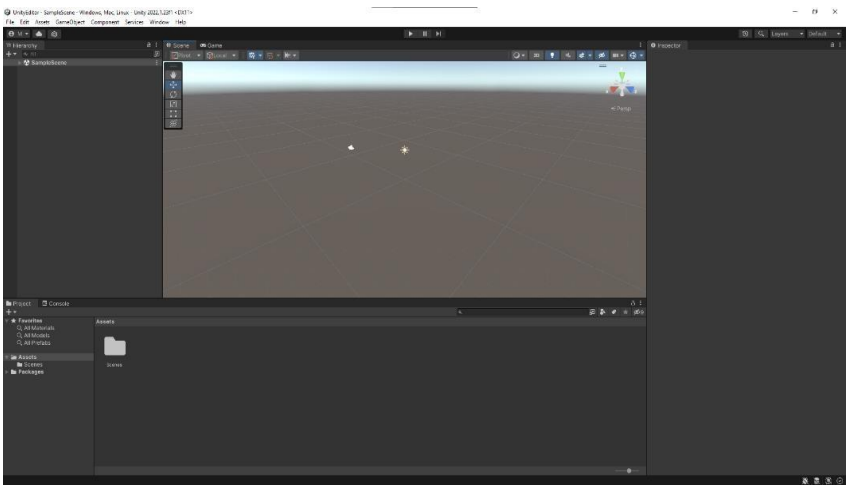


Рис. 3.1 Редактор сцени Unity

Важливим компонентом Unity є система об’єктів GameObject та компонентів (Component). Кожен об’єкт у грі складається з набору компонентів, які визначають його властивості та поведінку. Наприклад, для реалізації фізики додаються

компоненти Rigidbody2D та Collider, для відображення - Sprite Renderer, а для логіки - скрипти на мові C#. Такий підхід забезпечує модульність і гнучкість архітектури.

Інспектор (Inspector) дозволяє налаштовувати всі параметри об’єктів безпосередньо в редакторі, що зменшує необхідність зміни коду для дрібних коригувань. Це особливо важливо при налаштуванні фізичних параметрів, анімацій та взаємодій між об’єктами.

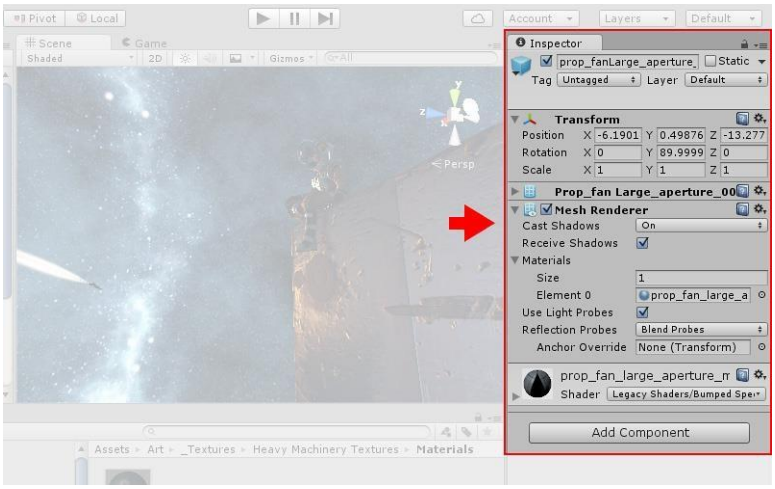


Рис. 3.2 Вікно Inspector

Unity має вбудовану фізичну систему для 2D, яка базується на обчисленні сил, зіткнень та гравітації. Це дозволяє реалізувати реалістичну поведінку персонажа та інших об’єктів без необхідності створення складних алгоритмів з нуля.

Окрему увагу слід приділити системі анімації. Компонент Animator дозволяє створювати анімаційні стани та переходи між ними, що використовується для відображення руху персонажа, стрибків та інших дій.

Анімації можуть змінюватися залежно від стану гри, що забезпечує динамічність візуального представлення.

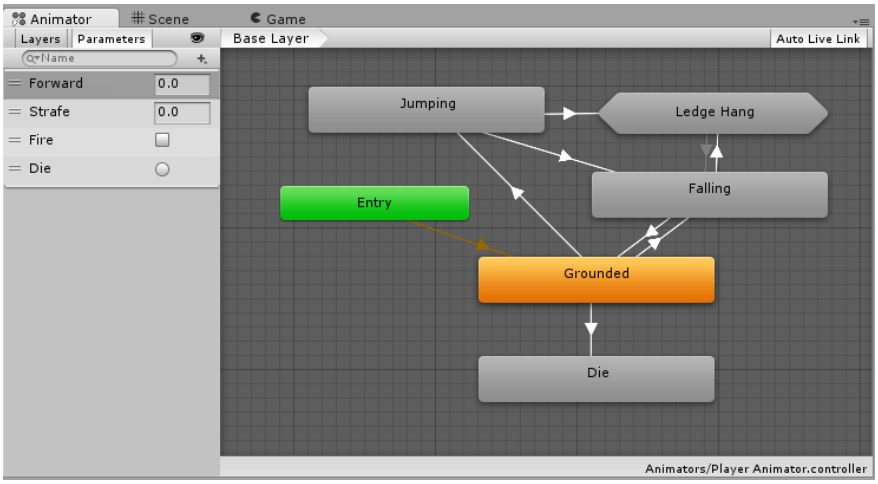


Рис. 3.3 Вікно Animator

Ще одним важливим елементом є система префабів (Prefab). Вона дозволяє створювати шаблони об’єктів, які можуть багаторазово використовуватися у різних сценах. Це значно спрощує розробку та підтримку проєкту, оскільки зміни в одному префабі автоматично застосовуються до всіх його екземплярів.

Unity також підтримує систему сцен (Scenes), яка використовується для організації рівнів гри. Кожен рівень може бути окремою сценою, що дозволяє керувати завантаженням ресурсів, переходами між рівнями та структурою гри.

Рис. 3.4 Список сцен у проєкті

Для тестування використовується режим Play Mode, який дозволяє

запускати гру безпосередньо в редакторі. Це дає змогу миттєво перевіряти зміни, аналізувати поведінку об’єктів та знаходити помилки. Такий підхід значно скорочує час розробки та підвищує якість продукту. Крім того, Unity має розвинену систему логування (Console), яка відображає помилки, попередження та службову інформацію. Це є важливим інструментом для налагодження та оптимізації гри.

3.1.2. Інструменти програмування та допоміжні засоби

Реалізація логіки 2D платформера потребує використання ефективних інструментів програмування та допоміжних засобів, які забезпечують написання, налагодження, тестування та підтримку коду. Від їх вибору залежить якість програмної реалізації, зручність розробки та можливість подальшого розширення функціоналу.

Основною мовою програмування є C#, яка використовується у середовищі Unity для створення скриптів. Вона підтримує об’єктно- орієнтований підхід, що дозволяє реалізовувати складні системи через класи, об’єкти, наслідування та інкапсуляцію. Це особливо важливо при розробці ігрових механік, де необхідно чітко структурувати логіку та забезпечити взаємодію між різними компонентами.

C# у контексті Unity використовується для:

— обробки введення користувача;

— керування рухом персонажа;

— реалізації фізики та взаємодії об’єктів;

— управління станами гри;

— створення систем прогресії та UI.

Скрипти підключаються до об’єктів як компоненти, що дозволяє безпосередньо інтегрувати програмну логіку у сцену та забезпечити гнучкість архітектури.

Як редактор коду використовується Visual Studio Code. Даний інструмент забезпечує комфортне середовище для розробки та підтримує широкий набір функцій, що значно підвищують продуктивність.

Основні можливості Visual Studio Code:

— підсвічування синтаксису та виявлення помилок у реальному часі;

— інтелектуальне автодоповнення коду (IntelliSense);

— інтеграція з C# через відповідні розширення;

— підтримка налагодження (debugging);

— швидка навігація по проєкту;

— можливість роботи з великими кодовими базами.

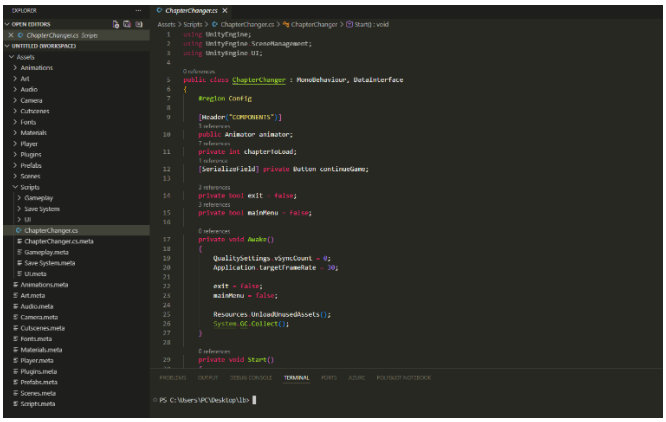


Рис. 3.5 Інтерфейс Visual Studio Code

Одним із ключових етапів розробки є налагодження програмного коду. Використання точок зупину дозволяє поетапно виконувати програму, аналізувати значення змінних та виявляти логічні помилки.

```
using System;

1 reference
public class BossHealth
{
    2 references
    public event EventHandler OnDamaged;
    2 references
    public event EventHandler OnDefeated;

    7 references
    private int currentHealth;
    2 references
    private int maxHealth;

    0 references
    public BossHealth(int maxHealth)
    {
        this.maxHealth = maxHealth;
        currentHealth = maxHealth;
    }

    0 references
    public int GetHealth()
    {
        return currentHealth;
    }

    0 references
    public void ResetHealth()
    {
        currentHealth = maxHealth;
    }

    0 references
    public void Damage(int damageAmount)
    {
        currentHealth -= damageAmount;
        if (currentHealth < 0)
            currentHealth = 0;

        if (OnDamaged != null) OnDamaged(this, EventArgs.Empty);

        if (IsDead())
            if (OnDefeated != null) OnDefeated(this, EventArgs.Empty);
    }
```

Рис. 3.6 Приклад коду

Для аналізу роботи програми використовується вбудована консоль Unity (Console), яка відображає повідомлення про помилки, попередження та службову інформацію. Це дозволяє оперативно реагувати на проблеми та контролювати виконання коду.Окрім цього, застосовуються вбудовані інструменти Unity для

тестування та оптимізації. Зокрема, режим Play Mode дозволяє запускати гру безпосередньо в редакторі, що значно пришвидшує перевірку реалізованих механік. Також використовується Unity Profiler - інструмент для аналізу продуктивності, який дозволяє оцінювати навантаження на процесор, пам’ять та інші ресурси системи. Це особливо важливо для оптимізації гри та забезпечення її стабільної роботи. Додатково важливу роль відіграє правильна організація структури коду. Використання окремих класів для різних механік (рух, стрибки, взаємодія) дозволяє спростити підтримку проєкту та забезпечити можливість подальшого розширення функціоналу.

3.2 Реалізація керування персонажем

Реалізація керування персонажем є центральною частиною ігрової логіки 2D платформера, оскільки саме вона визначає взаємодію гравця з ігровим світом. Від якості цієї системи залежить відчуття керованості, точність виконання дій та загальна динаміка гри. У даному проєкті керування реалізовано у вигляді єдиного скрипта PlayerController, який об’єднує фізику руху, обробку введення, перевірку станів, анімаційні переходи та візуальні ефекти.

Система побудована на базі рушія Unity, який забезпечує інтеграцію фізичного моделювання, компонентної архітектури та системи подій. Основою руху персонажа є компонент Rigidbody2D, що дозволяє забезпечити інерційність, гравітацію та коректну взаємодію з колайдерами сцени. Загальна логіка керування організована як безперервний цикл, де:

— у кожному кадрі обробляється введення користувача;

— у фізичному оновленні застосовуються розраховані швидкості;

— паралельно оновлюються стани анімації та ефекти;

— виконується перевірка станів персонажа (земля/повітря/присідання).

Такий підхід забезпечує стабільність роботи системи та синхронізацію між фізикою і візуальною частиною.

Важливою особливістю реалізації є те, що керування не є прямим

переміщенням об’єкта. Замість цього використовується модель керування через фізичні сили та швидкості, що дозволяє отримати більш природну поведінку персонажа. Це особливо важливо для платформера, де точність стрибків і рухів є критичною.

Окрему роль відіграє система адаптації керування до станів персонажа. Наприклад, при зміні положення (присідання або перебування в повітрі)змінюються параметри руху, швидкість та доступні дії. Це дозволяє створити різні стилі поведінки персонажа в залежності від ситуації.

Система також інтегрована з іншими підсистемами гри:

— анімаційною системою (для відображення станів);

— звуковою системою (для відтворення подій);

— системою частинок (для візуальних ефектів);

— системою здібностей персонажа (тимчасові модифікації параметрів).

Додатково реалізовано механізми згладжування руху, які запобігають різким змінам швидкості та забезпечують більш плавне управління. Це підвищує комфорт гравця та робить керування більш передбачуваним.

Особливу увагу приділено синхронізації логіки керування з фізичним рушієм. Оновлення фізичних параметрів відбувається у відповідний момент фізичного циклу, що виключає нестабільність та “ривки” у русі персонажа.

3.2.1 Загальна структура системи керування

Система керування персонажем у 2D платформері є одним із ключових

елементів ігрової логіки, оскільки саме вона визначає спосіб взаємодії гравця з ігровим середовищем та формує базове відчуття “керованості” персонажа. У даному проєкті реалізовано централізований підхід, де вся логіка керування зосереджена у скрипті PlayerController, який координує рух, фізику, стани, анімації та візуальні ефекти.

Реалізація базується на можливостях рушія Unity, який надає компонентну

архітектуру та дозволяє розділяти функціональність на незалежні модулі.

Завдяки цьому кожен аспект поведінки персонажа (рух, стрибки, присідання, взаємодія з оточенням) реалізується як окремий логічний блок, але при цьому всі вони працюють узгоджено в межах одного контролера.

Загальна структура системи керування побудована таким чином, щоб забезпечити баланс між простотою керування та складністю ігрових механік. З одного боку, гравець взаємодіє лише через прості елементи вводу (джойстик), з іншого - всередині системи виконується значна кількість розрахунків, пов’язаних із фізикою, інерцією, станами та анімаціями.

Уся система умовно поділяється на кілька взаємопов’язаних рівнів. Перший рівень відповідає за зчитування введення користувача. Другий - за перетворення цього введення у фізичні параметри руху. Третій рівень забезпечує перевірку стану персонажа (на землі чи в повітрі). Четвертий рівень відповідає за візуальне відображення станів через анімації та ефекти. Така багаторівнева структура дозволяє ізолювати логіку та уникнути перевантаження одного модуля.

Важливим аспектом є те, що система керування тісно пов’язана з фізичною моделлю персонажа. Рух не реалізується через пряме змінення позиції, а відбувається через керування швидкістю фізичного тіла (Rigidbody2D). Це забезпечує більш природну поведінку персонажа, врахування інерції та коректну взаємодію з платформами, перешкодами та іншими об’єктами рівня.

Також система враховує поточний стан персонажа, який безпосередньо впливає на доступні дії. Наприклад, під час перебування в повітрі змінюються правила виконання стрибка, а під час присідання обмежується швидкість руху та змінюється фізичний розмір колайдера. Це дозволяє створювати більш складну та реалістичну поведінку без ускладнення керування для гравця. Окрему роль у структурі відіграє взаємодія з іншими системами гри. Керування персонажем не є ізольованим модулем - воно постійно передає дані до анімаційної системи, звукового менеджера та системи ефектів. Завдяки цьому забезпечується повна синхронізація між тим, що відбувається фізично, і тим, що бачить гравець.

3.2.2 Горизонтальний рух персонажа

Горизонтальний рух персонажа є базовою механікою переміщення у 2D платформері та визначає основну динаміку взаємодії гравця з рівнем. У даному проєкті реалізація руху побудована на фізичному підході, де зміна положення персонажа відбувається через керування швидкістю фізичного об’єкта (Rigidbody2D), а не через пряме встановлення координат. Це забезпечує більш реалістичну поведінку персонажа, врахування інерції та коректну взаємодію з платформами.

Горизонтальний рух тісно пов’язаний із введенням користувача через джойстик. Значення осі вводу безперервно зчитується та перетворюється у зміну швидкості персонажа. При цьому система не використовує миттєве прискорення, а застосовує поступове нарощування або зменшення швидкості, що дозволяє уникнути різких ривків та забезпечує плавність керування.

Особливістю реалізації є використання згладжування руху. Залежно від ситуації застосовуються різні коефіцієнти уповільнення: при відсутності вводу персонаж поступово зупиняється, при зміні напрямку — гаситься попередня швидкість, а при стабільному русі підтримується рівномірне переміщення. Такий підхід дозволяє створити більш «вагому» та контрольовану фізику руху.

Додатково система враховує стан персонажа. Наприклад, під час присідання швидкість руху зменшується, що логічно обмежує мобільність у цьому стані.

Також передбачені тимчасові модифікатори швидкості, які можуть змінювати поведінку персонажа залежно від ігрових подій або активних здібностей.

Важливим елементом є механізм зміни напрямку руху. Персонаж автоматично розвертається відповідно до напрямку руху, що реалізується через зміну масштабу по горизонталі. Це дозволяє коректно відображати анімації та забезпечує природну поведінку спрайта.

Основні особливості реалізації:

— рух базується на зміні фізичної швидкості, а не координат; застосовується згладжування прискорення та зупинки;

— враховується зміна напрямку руху з гасінням інерції;

— швидкість залежить від стану персонажа (звичайний стан, присідання, здібності).

Результат реалізації:

— забезпечено плавне та інтуїтивне керування рухом;

— усунено різкі зміни швидкості та “ривки”;

— досягнуто природної інерційної поведінки персонажа;

— реалізовано адаптацію руху до ігрових станів.

3.2.3 Вертикальний рух та перевірка поверхні

Вертикальний рух персонажа є критично важливою частиною ігрової механіки 2D платформера, оскільки він безпосередньо визначає стрибки, падіння та взаємодію з платформами. У даному проєкті вертикальна складова руху реалізована на основі фізичної моделі з використанням Rigidbody2D, що дозволяє коректно враховувати дію гравітації та забезпечує природну траєкторію руху в повітрі.

Основою роботи вертикального руху є визначення стану персонажа відносно поверхні. Для цього використовується система перевірки контакту із землею через фізичну область під персонажем. Такий підхід дозволяє визначати, чи знаходиться персонаж на платформі, чи перебуває у стані падіння або стрибка. Коректне визначення цього стану є обов’язковим для запуску стрибків та обмеження певних дій у повітрі.

Важливою частиною реалізації є механізм «запам’ятовування землі» (coyote time). Він дозволяє гравцю виконати стрибок навіть через короткий проміжок часу після втрати контакту з поверхнею. Це значно покращує ігровий досвід, оскільки робить керування більш прощаючим до неточних дій користувача та підвищує комфорт гри.

Також у системі реалізовано відстеження часу перебування персонажа в

повітрі. Цей параметр використовується не лише для логіки стрибків, але й для анімаційної системи та візуальних ефектів, що дозволяє більш точно відображати стан персонажа.

Гравітація відіграє важливу роль у вертикальному русі. У проєкті застосовується модифікована модель падіння, яка підсилює прискорення під час руху вниз. Це дозволяє уникнути ефекту “плавання” персонажа в повітрі та робить падіння більш швидким і контрольованим, що особливо важливо для жанру платформера.

Окрему увагу приділено моменту приземлення. Система визначає перехід із повітряного стану в контакт із поверхнею та активує відповідні реакції: запуск частинок пилу, відтворення звукового ефекту та оновлення анімаційного стану. Це створює більш виразний візуальний зворотний зв’язок для гравця.

Основні особливості реалізації:

— визначення контакту із землею через фізичну перевірку області під персонажем;

— реалізація «coyote time» для покращення чутливості стрибків;

— відстеження часу перебування в повітрі для синхронізації логіки та анімацій;

— використання модифікованої гравітації для динамічного падіння. Результат реалізації:

— забезпечено стабільне визначення стану «земля / повітря»;

— покращено відчуття керування під час стрибків;

— підвищено точність та комфорт вертикального руху;

— реалізовано плавний перехід між станами руху.

3.2.4 Механіка стрибка

Механіка стрибка є однією з ключових ігрових систем у 2D платформері, оскільки саме вона формує вертикальну динаміку проходження рівнів та визначає

складність подолання перешкод. У даному проєкті реалізовано розширену систему стрибків, яка поєднує базовий стрибок, додаткові стрибки в повітрі, «грейс-період» після відриву від землі та модифіковану фізику падіння.

Реалізація побудована на фізичному компоненті Rigidbody2D, що дозволяє керувати вертикальною швидкістю персонажа шляхом зміни її значення, а не через пряме переміщення. Це забезпечує природну траєкторію руху, врахування гравітації та коректну взаємодію з ігровими об’єктами.

Особливістю системи є використання декількох механізмів, які підвищують точність і комфорт керування. Зокрема, реалізовано систему додаткових стрибків, яка дозволяє виконувати обмежену кількість стрибків у повітрі. Це додає гнучкості геймплею та дозволяє гравцю коригувати траєкторію руху після початкового стрибка.

Також використовується механізм «coyote time» - короткий проміжок часу після втрати контакту з поверхнею, протягом якого стрибок ще залишається доступним. Це значно покращує ігровий досвід, оскільки компенсує неточності у таймінгу натискання кнопки стрибка.

Для підвищення динаміки руху застосовується модифікована гравітація. Вона збільшує швидкість падіння персонажа при русі вниз, що дозволяє уникнути ефекту «плавання» у повітрі та робить керування більш чітким і відчутним.

Таблиця 3.1

Основні параметри механіки стрибка

Параметр	Опис	Призначення
Jump Force	Сила початкового стрибка	Визначає висоту стрибка
Extra Jumps	Кількість додаткових стрибків	Дозволяє стрибати в повітрі
Coyote Time	Час “прощення” після падіння	Покращує точність стрибка
Fall Multiplier	Множник прискорення падіння	Робить падіння швидшим і контрольованішим

Логіка роботи стрибка

— перевірка стану персонажа (на землі або в повітрі);

— перевірка доступності стрибка (кількість додаткових стрибків);
— аналіз короткого часу після втрати контакту з поверхнею;
— застосування вертикальної швидкості для виконання стрибка;
— оновлення станів анімації та звукових ефектів;
— активація частинок при приземленні. Результат реалізації
— забезпечено точне та передбачуване виконання стрибків;
— підвищено комфорт керування завдяки додатковим механікам;
— реалізовано гнучку систему вертикального руху;
— досягнуто балансу між складністю та зручністю геймплею.

3.2.5 Присідання та зміна колайдерів

Механіка присідання у 2D платформері є важливим елементом керування персонажем, оскільки вона впливає як на фізичні характеристики об’єкта, так і на його взаємодію з ігровим середовищем. У даному проєкті присідання реалізовано як окремий стан персонажа, який активується на основі введення користувача через вертикальну вісь джойстика.

Основною особливістю реалізації є динамічна зміна фізичної форми персонажа через перемикання колайдерів. У звичайному стані використовується один набір колайдерів, який відповідає повній висоті персонажа. При переході в стан присідання активується альтернативний колайдер з меншою висотою, а основний колайдер деактивується. Це дозволяє коректно змінювати фізичні розміри персонажа без порушення взаємодії з фізичною системою рушія.

Зміна колайдерів безпосередньо впливає на поведінку персонажа в ігровому світі. Завдяки цьому персонаж може проходити через вузькі простори, уникати певних перешкод або змінювати тактичну позицію під час проходження рівня. При цьому зберігається коректна взаємодія з платформами та іншими об’єктами сцени.

Окрему роль відіграє перевірка простору над персонажем. Перед переходом зі стану присідання в звичайний стан виконується перевірка на наявність перешкод

зверху. Це необхідно для запобігання ситуаціям, коли персонаж може “вставати” всередині об’єктів, що призвело б до фізичних помилок або некоректної поведінки колізій.

Присідання також впливає на швидкість горизонтального руху. У цьому стані швидкість пересування зменшується, що логічно відображає обмежену мобільність персонажа. Це створює додатковий тактичний елемент у геймплеї, коли гравець повинен обирати між швидкістю пересування та можливістю уникати перешкод.

З візуальної точки зору стан присідання супроводжується зміною анімації персонажа. Анімаційна система синхронізується зі зміною фізичного стану, що дозволяє забезпечити узгодженість між поведінкою об’єкта та його зовнішнім виглядом. Це є важливим для формування цілісного сприйняття персонажа гравцем.

3.2.6 Анімації та візуальний зворотний зв’язок

Анімаційна система та візуальний зворотний зв’язок є важливою складовою системи керування персонажем, оскільки вони забезпечують відповідність між фізичною поведінкою об’єкта та його зовнішнім відображенням. У 2D платформері саме візуальна частина формує у гравця відчуття “живості” персонажа, а також дозволяє швидко зчитувати його стан без необхідності аналізу ігрової логіки.

У даному проєкті анімації реалізовані за допомогою системи Animator рушія Unity, яка дозволяє керувати станами персонажа через параметри та переходи між анімаційними кліпами. Кожна анімація відповідає конкретному стану персонажа, зокрема руху, стрибку, перебуванню в повітрі та присіданню.

Основний принцип роботи полягає у тому, що фізична логіка персонажа не напряму керує анімаціями, а лише передає набір параметрів, які описують поточний стан. На основі цих параметрів Animator автоматично визначає, яку анімацію необхідно відтворювати. Такий підхід забезпечує розділення логіки та візуалізації, що є важливим для масштабованих ігрових проєктів.

Особливу роль відіграє параметр швидкості руху, який використовується для

плавного переходу між станами “стоячий” та “біг”. Це дозволяє уникнути різких змін анімацій і створює більш природну поведінку персонажа. Аналогічно, вертикальна швидкість використовується для визначення стану стрибка або падіння, що дозволяє анімаційній системі реагувати на фізику в реальному часі.Окремо реалізовано логіку відображення стану перебування на землі. Вона використовується для визначення моменту переходу між повітряними та наземними анімаціями. Це забезпечує коректну синхронізацію між фізичним приземленням та візуальним ефектом контакту із поверхнею.

Візуальний зворотний зв’язок також включає систему частинок і звукових ефектів. Наприклад, при приземленні активується ефект пилу, який підсилює відчуття удару об поверхню. При виконанні стрибка або приземленні відтворюються відповідні звуки, що доповнюють візуальне сприйняття та підвищують занурення у гру.

Додатково важливим аспектом є плавність переходів між анімаціями. У системі використовуються параметризовані переходи, які дозволяють уникати різких змін кадрів і забезпечують природний вигляд руху персонажа. Це особливо важливо для платформера, де персонаж постійно змінює стан у залежності від взаємодії з рівнем.

3.3 Реалізація фізики та колізій

Реалізація фізики та системи колізій у 2D платформері є однією з ключових складових ігрової логіки, оскільки саме вона забезпечує коректну взаємодію персонажа з ігровим середовищем. Від якості фізичної моделі залежить точність руху, стабільність стрибків, поведінка при зіткненнях з платформами та загальна передбачуваність ігрового процесу.

У даному проєкті фізика реалізована на базі вбудованої фізичної системи рушія Unity, з використанням компонента Rigidbody2D для персонажа та Collider2D для всіх об’єктів рівня. Такий підхід дозволяє використовувати готову фізичну симуляцію, включаючи гравітацію, зіткнення та обмеження руху.

Основою фізичної моделі є взаємодія між колайдерами різних типів. Персонаж має власний набір колайдерів, які змінюються залежно від його стану (наприклад, звичайний стан або присідання). Оточення рівня складається з платформ, стін та перешкод, які представлені статичними колайдерами. Це забезпечує стабільну фізичну взаємодію без необхідності складних розрахунків вручну.

Окрему увагу приділено системі визначення контакту з поверхнею. Вона реалізована через перевірку області під персонажем, що дозволяє визначати, чи знаходиться він на землі. Цей механізм є критично важливим для роботи стрибків, анімацій та переходів між станами. Завдяки цьому система точно визначає момент приземлення та відриву від поверхні.

Фізика руху персонажа не базується на прямому переміщенні координат. Замість цього використовується зміна швидкості через Rigidbody2D, що дозволяє враховувати інерцію, прискорення та гальмування. Такий підхід забезпечує більш природну поведінку персонажа та покращує відчуття керування.Також у системі реалізовано модифіковану гравітацію, яка впливає на швидкість падіння персонажа. Це дозволяє уникнути надто повільного спуску та робить вертикальний рух більш динамічним. Гравітація підсилюється під час падіння, що покращує контроль гравця над персонажем у повітрі.

Колізії між об’єктами обробляються фізичною системою рушія, однак додатково реалізовано логіку реагування на зіткнення. Наприклад, при приземленні активуються звукові та візуальні ефекти, а також оновлюється стан персонажа. Це дозволяє поєднати фізичну модель з ігровою логікою.

Окремо слід відзначити обробку вузьких просторів і зміни розмірів персонажа. При переході у стан присідання змінюється активний колайдер, що дозволяє персонажу проходити через низькі ділянки рівня. Це вимагає точного узгодження фізичної форми персонажа з його візуальним представленням.

3.4 Реалізація анімацій

Реалізація анімацій у 2D платформері є важливим елементом, який забезпечує візуальне відображення станів персонажа та створює відчуття його “живості” в ігровому світі. Анімаційна система безпосередньо не впливає на фізичну поведінку об’єкта, однак є критично важливою для формування зворотного зв’язку між діями гравця та реакцією персонажа.

У проєкті анімації реалізовані за допомогою системи Animator у рушії Unity.

Вся логіка керування анімаціями побудована на основі параметрів, які передаються зі скрипта PlayerController. Це дозволяє відокремити ігрову логіку від візуального представлення та забезпечує гнучке керування станами персонажа.

Основний принцип роботи полягає в тому, що Animator не приймає прямих команд на зміну анімації, а реагує на зміну параметрів. Залежно від значень цих параметрів відбувається автоматичний перехід між анімаційними станами.

Такий підхід дозволяє створити плавну та логічно структуровану систему анімацій. Важливу роль відіграє синхронізація анімацій із фізичними параметрами персонажа. Наприклад, горизонтальна швидкість використовується для визначення переходу між станами стояння та руху, а вертикальна швидкість - для визначення станів стрибка та падіння. Це забезпечує відповідність між фізикою та візуальним відображенням.

Окремо реалізовано анімацію присідання, яка активується при зміні фізичного стану персонажа. Також передбачено анімацію перебування в повітрі, яка дозволяє візуально відрізняти фазу стрибка від фази падіння.

Окрім основних станів, система анімацій підтримує плавні переходи між ними. Це досягається за рахунок налаштування переходів у Animator Controller, що дозволяє уникати різких змін кадрів і робить рух персонажа більш природним.

Таблиця 3.2

Основні анімаційні стани персонажа

Стан	Умова активації	Опис поведінки
Idle	Відсутність руху	Персонаж перебуває у спокої
Run	Горизонтальний рух	Активується при переміщенні
Jump	Початок вертикального руху вгору	Перехід у повітряний стан
Fall	Від’ємна вертикальна швидкість	Фаза падіння після стрибка
Crouch	Вертикальний ввід вниз	Зменшення висоти персонажа

Також анімаційна система тісно пов’язана з іншими підсистемами гри. Вона реагує на зміни фізичного стану, отримує дані про швидкість руху та синхронізується з системою ефектів. Наприклад, при приземленні може активуватися не лише відповідна анімація, але й звуковий ефект та частинки.

4 ТЕСТУВАННЯ ТА ОПТИМІЗАЦІЯ

4.1 Методи тестування

Під час розробки 2D платформера значна увага приділялася перевірці працездатності окремих елементів гри та стабільності всього застосунку загалом. Тестування проводилося протягом усього процесу створення проєкту, оскільки це дозволяло своєчасно знаходити помилки та уникати проблем у подальшій роботі програми.

Насамперед перевірялися основні ігрові механіки. Було протестовано рух персонажа, систему стрибків, взаємодію з платформами та реакцію гри на дії користувача. Особлива увага приділялася коректності обробки зіткнень, адже помилки у фізичній взаємодії об’єктів могли негативно впливати на ігровий процес. Наприклад, на початкових етапах розробки виникали ситуації, коли персонаж міг частково проходити крізь платформу або некоректно реагував на зіткнення з перешкодами. Для усунення цих проблем змінювалися параметри колайдерів та перевірялася логіка роботи фізики.

Окремо тестувалася система анімацій. Перевірялося, чи правильно змінюються анімаційні стани під час руху, стрибків або бездіяльності персонажа. У деяких випадках анімації запускалися із затримкою або відображалися некоректно, тому довелося змінювати налаштування Animator та коригувати переходи між станами.

Також проводилася перевірка взаємодії між різними компонентами гри. Після додавання нових елементів тестувалося, чи не виникають конфлікти між системою керування, фізикою та анімаціями. Це було необхідно для забезпечення стабільної роботи всіх механік одночасно.

Під час створення рівнів додатково перевірялися розташування платформ, перешкод та інших об’єктів. Тестування допомогло виявити місця, де персонаж міг застрягати або виходити за межі ігрової сцени. Після цього структура рівнів

частково змінювалася для покращення ігрового процесу.

Крім функціональності, перевірялася і продуктивність гри. Аналізувалася плавність роботи застосунку, стабільність частоти кадрів та навантаження на систему. Найбільше уваги приділялося сценам із великою кількістю об’єктів, оскільки саме в таких ситуаціях могли виникати просідання FPS. Для оптимізації було зменшено кількість зайвих обчислень у скриптах та спрощено окремі елементи сцени.

У процесі тестування використовувалися стандартні засоби Unity, зокрема консоль для пошуку помилок і профайлер для аналізу продуктивності. Це дозволило швидше знаходити проблемні ділянки коду та контролювати використання ресурсів.

Окрім технічної перевірки, проводилося і звичайне ручне тестування. Гра неодноразово проходилася повністю для оцінки зручності керування, плавності руху персонажа та загального сприйняття ігрового процесу. Саме під час такоготестування вдалося помітити частину недоліків, які не проявлялися під час окремої перевірки компонентів.

Після внесення будь-яких змін тестування виконувалося повторно. Це було необхідно для того, щоб переконатися у відсутності нових помилок після виправлення попередніх. Завдяки такому підходу вдалося покращити стабільність роботи гри та забезпечити коректну роботу основних механік платформера.

4.2 Функціональне тестування

Функціональне тестування проводилося з метою перевірки коректності роботи всіх основних можливостей гри та відповідності реалізованого функціоналу поставленим вимогам. Основна увага приділялася стабільності роботи ігрових механік, правильності взаємодії між компонентами та реакції гри на дії користувача. Перевірка виконувалася на різних етапах розробки, починаючи від створення базових механік і завершуючи фінальною оптимізацією проєкту.

Одним із головних елементів тестування була перевірка системи керування

персонажем. Аналізувалася реакція героя на натискання клавіш, плавність руху та швидкість виконання дій. Тестувалися переміщення вліво та вправо, стрибки, приземлення, а також зміна станів під час руху. Окремо перевірялося, чи не виникають затримки між натисканням клавіш і реакцією персонажа, оскільки навіть незначна затримка могла негативно впливати на комфорт гри.

Значна увага приділялася тестуванню фізики та системи колізій. Перевірялося, чи правильно персонаж взаємодіє з платформами, стінами, перешкодами та іншими об’єктами сцени. Під час ранніх етапів тестування були виявлені випадки некоректної обробки зіткнень: герой міг частково проходити крізь платформу або застрягати між об’єктами. Для усунення цих проблем змінювалися параметри фізичних компонентів та вдосконалювалася логіка перевірки колізій.

Окремо перевірялася система стрибків. Аналізувалася висота стрибка, коректність приземлення та поведінка персонажа під час контакту з різними поверхнями. Також тестувалася можливість повторного стрибка та обмеження, які не дозволяли виконувати некоректні дії у повітрі. Це було важливо для забезпечення збалансованого та передбачуваного ігрового процесу.

Під час функціонального тестування значна увага приділялася роботі анімацій. Перевірялося, чи відповідають анімаційні стани поточним діям персонажа. Аналізувалися переходи між анімаціями руху, стрибка, падіння та бездіяльності. У деяких випадках анімації могли запускатися повторно або не змінювалися вчасно, тому виконувалося додаткове налаштування параметрів Animator. Також тестувалася система взаємодії персонажа з ігровим середовищем. Перевірялося підбирання предметів, реакція на зіткнення з небезпечними об’єктами, активація певних тригерів та робота контрольних точок. Особлива увага приділялася правильності виконання подій після взаємодії з різними елементами рівня.

Для перевірки стабільності роботи гри тестувалися переходи між сценами та рівнями. Аналізувалося завантаження нових сцен, коректність збереження стану гри та відсутність помилок під час зміни рівнів. Також перевірялося, чи не виникає

повторного завантаження об’єктів або втрати даних після переходу між сценами.

Окрему частину функціонального тестування займала перевірка користувацького інтерфейсу. Аналізувалася робота кнопок меню, коректність відображення тексту, реакція елементів інтерфейсу на дії користувача та правильність роботи ігрових меню. Перевірялося відкриття паузи, перезапуск рівня та повернення до головного меню. Також тестувалася адаптація інтерфейсу до різних роздільних здатностей екрана. Під час тестування неодноразово перевірялася продуктивність гри у різних умовах. Аналізувалася плавність роботи під час великої кількості об’єктів на сцені, стабільність частоти кадрів та використання ресурсів системи. Для цього використовувалися стандартні інструменти Unity, які дозволяли контролювати навантаження на процесор та оперативну пам’ять.

Функціональне тестування проводилося багаторазово після внесення змін до коду або структури рівнів. Це дозволяло переконатися, що нові елементи не спричиняють появу додаткових помилок та не впливають негативно на вже реалізовані механіки. Повторна перевірка після кожного етапу розробки допомогла поступово підвищити стабільність і якість роботи програмного продукту.

4.3 Виявлення та усунення помилок

У процесі розробки 2D платформера етап виявлення та усунення помилок мав критичне значення, оскільки дозволяв забезпечити стабільність роботи всіх ігрових механік та уникнути критичних збоїв у фінальній версії продукту. Помилки виникали на різних рівнях: у логіці програмного коду, фізичній взаємодії об’єктів, роботі анімацій, а також під час інтеграції окремих систем у єдину архітектуру гри. Саме тому процес тестування та виправлення помилок проводився безперервно протягом усього циклу розробки.

Для виявлення помилок активно використовувалися інструменти Unity, зокрема консоль для відстеження помилок, система Debug.Log для аналізу виконання коду та Unity Profiler для оцінки продуктивності. Ці засоби дозволяли

визначати проблемні ділянки коду, контролювати навантаження на систему та аналізувати поведінку об’єктів у реальному часі. Окрім цього, значна частина помилок виявлялася шляхом ручного тестування, що включало багаторазове проходження рівнів та перевірку всіх можливих ігрових сценаріїв.

У ході роботи було встановлено, що помилки умовно можна поділити на три основні групи: логічні помилки ігрового процесу, фізичні помилки взаємодії об’єктів та технічні помилки реалізації сцен і продуктивності. Такий поділ дозволив більш системно підходити до їх аналізу та виправлення, а також пришвидшив процес оптимізації гри.

4.3.1 Помилки ігрової логіки та системи керування

Під час розробки 2D платформера значна частина виявлених проблем була пов’язана з ігровою логікою та системою керування персонажем. Саме ці елементи є основою ігрового процесу, тому навіть незначні неточності у їх реалізації суттєво впливали на зручність гри та відчуття керування.

На початкових етапах тестування спостерігалися затримки у реакції персонажа на натискання клавіш. Це проявлялося у тому, що рух або стрибок виконувалися із невеликою затримкою, особливо при швидкому змінюванні команд. Також іноді персонаж продовжував рухатися після відпускання

клавіші, що створювало відчуття «інерційного» керування, яке не відповідало задуму гри. Подібні проблеми виникали через надмірну кількість перевірок у методі оновлення кадру та недостатньо чітке розмежування логіки вводу і фізичної поведінки об’єкта. Ще однією проблемою було некоректне поєднання різних дій персонажа. Наприклад, у деяких випадках можна було одночасно ініціювати рух і стрибок у момент, коли персонаж уже перебував у повітрі. Це призводило до нелогічної поведінки, коли фізика гри не відповідала очікуваному результату. Причиною таких ситуацій була відсутність централізованої системи контролю станів персонажа, через що різні дії виконувалися незалежно одна від одної.

Для усунення цих проблем було переглянуто архітектуру системи керування.

Було впроваджено чітку модель станів персонажа, яка обмежує доступні дії залежно від поточного стану (наприклад, рух, стрибок, падіння або стан спокою). Це дозволило уникнути конфліктів між діями та зробити поведінку персонажа більш передбачуваною. Також було оптимізовано обробку вводу користувача: тепер усі команди зчитуються централізовано та передаються до системи логіки без дублювання перевірок у різних частинах коду.

Окремі помилки виникали через неправильне використання змінних стану. У деяких випадках значення не оновлювалися своєчасно, що призводило до «зависання» персонажа в певному режимі (наприклад, стан стрибка не переходив у стан падіння або стояння). Для вирішення було введено додаткові перевірки умов переходів між станами та уточнено логіку їх зміни в залежності від фізичних параметрів об’єкта.

Також виявлялися проблеми з чутливістю керування. На різних етапах тестування відчувалася або надмірна різкість руху, або недостатня швидкість реакції персонажа. Це було пов’язано з невдало підібраними коефіцієнтами швидкості та прискорення, а також із відсутністю плавної інтерполяції руху. Після коригування цих параметрів керування стало більш збалансованим і комфортним для користувача.

Додатково проводився аналіз сценаріїв, у яких гравець виконує швидку зміну команд. Саме в таких ситуаціях найчастіше проявлялися приховані помилки логіки. Було виявлено, що без додаткового контролю станів система могла «переписувати» попередні дії новими командами, що призводило до нестабільної поведінки персонажа. Виправлення цієї проблеми дозволило значно підвищити точність керування.

4.3.2 Фізичні помилки та колізії

Під час розробки 2D платформера значна частина технічних проблем була пов’язана з фізичною моделлю гри та системою колізій. Саме ці компоненти відповідають за взаємодію персонажа з ігровим середовищем, тому будь-які

неточності в їх роботі безпосередньо впливали на якість геймплею. Помилки проявлялися як у некоректній поведінці персонажа, так і в нестабільній роботі об’єктів сцени.

Однією з найпоширеніших проблем було часткове проходження персонажа крізь платформи або їх краї. У деяких випадках герой міг «провалюватися» вниз під час приземлення або застрягати на стиках об’єктів. Також спостерігалися ситуації, коли персонаж не розпізнавав поверхню як тверду, через що стрибок або рух по платформі працював некоректно. Подібні помилки виникали через неточне налаштування Collider2D та невідповідність фізичних і візуальних розмірів об’єктів.

Ще однією проблемою була нестабільна поведінка персонажа при зіткненні зі стінами або кутами платформ. У таких ситуаціях герой міг різко зупинятися, «прилипати» до поверхні або змінювати напрямок руху без чіткої логіки. Причиною цього було неправильне налаштування параметрів тертя та маси об’єктів, а також відсутність додаткових перевірок контакту з поверхнею. Після аналізу було скориговано фізичні матеріали та оптимізовано взаємодію колайдерів.

Окремі помилки виникали при роботі з гравітацією та стрибками. Іноді персонаж зависав у повітрі або падав із некоректною швидкістю. Це було пов’язано з нестабільними значеннями сили стрибка та гравітаційного впливу, а також із конфліктами між фізичними розрахунками та логікою керування. Для виправлення було виконано точне налаштування параметрів Rigidbody2D, що дозволило стабілізувати рух персонажа у повітрі.

Також виявлялися проблеми, пов’язані з рухомими платформами та динамічними об’єктами. У деяких випадках персонаж не «приєднувався» до рухомої платформи і залишався на місці або, навпаки, зміщувався неприродно під час її руху. Для вирішення цієї проблеми було вдосконалено логіку взаємодії з рухомими об’єктами та додано перевірку батьківських зв’язків під час контакту.

Додатково під час тестування було виявлено проблеми з точністю спрацювання колізій. Іноді зіткнення реєструвалися із затримкою або не спрацьовували при дуже швидкому русі персонажа. Це було пов’язано з

недостатньою частотою фізичних оновлень та високою швидкістю переміщення об’єкта. Для усунення цієї проблеми було оптимізовано параметри фізичного рушія та скориговано швидкість руху персонажа.

4.3.3 Анімації, сцени та продуктивність

Під час розробки 2D платформера окремий клас проблем був пов’язаний із роботою анімаційної системи, переходами між сценами та загальною продуктивністю гри. Ці аспекти безпосередньо впливають як на візуальне сприйняття гри, так і на її стабільність, тому їх тестування та оптимізація виконувалися особливо ретельно.

У процесі перевірки анімацій було виявлено низку неточностей у роботі Animator Controller. Зокрема, анімація руху персонажа іноді продовжувала відтворюватися після зупинки, а перехід у стан «стрибок» або «падіння» міг відбуватися із затримкою. Також спостерігалися ситуації, коли одночасно активувалися кілька анімаційних станів, що призводило до візуальних артефактів та некоректного відображення персонажа. Причиною таких проблем було неправильне налаштування умов переходів між станами та недостатня синхронізація анімацій із логікою керування персонажем.

Додаткові труднощі виникали при роботі зі швидкою зміною станів, коли гравець різко змінював напрямок руху або часто чергував стрибки та рух. У таких випадках анімаційна система не завжди встигала коректно оновити поточний стан, що призводило до «залипання» анімацій або їх некоректного завершення. Для усунення цієї проблеми було оптимізовано параметри переходів та введено чіткі умови активації кожної анімації, що забезпечило більш стабільну роботу системи.

Певні проблеми виникали і під час переходів між сценами. У деяких випадках після перезавантаження рівня або переходу на іншу сцену частина об’єктів дублювалася або не видалялася коректно. Це призводило до накопичення зайвих елементів у сцені та поступового зниження продуктивності. Причиною таких помилок було неправильне управління життєвим циклом об’єктів, а також

недостатній контроль процесу ініціалізації сцен. Після аналізу було оптимізовано порядок завантаження сцен та перевірено коректність роботи методів ініціалізації об’єктів.

Окрему увагу було приділено продуктивності гри. Під час тестування виявлялося, що при збільшенні кількості об’єктів на сцені або активному використанні фізики відбувалося зниження частоти кадрів. Це було пов’язано з надмірною кількістю обчислень у скриптах, частими викликами методів оновлення кожного кадру та недостатньою оптимізацією фізичних взаємодій. Для вирішення проблеми було зменшено кількість операцій у методі Update(), оптимізовано фізичні розрахунки та спрощено деякі елементи сцени без втрати функціональності.

Також проводився аналіз продуктивності за допомогою вбудованого профайлера Unity. Це дозволило визначити найбільш ресурсомісткі процеси, зокрема роботу анімацій, фізики та системи колізій. Після цього було виконано додаткову оптимізацію коду, що зменшило навантаження на процесор і підвищило стабільність роботи гри.

ВИСНОВОК

У результаті виконання кваліфікаційної роботи було проведено дослідження особливостей розробки 2D платформерів та реалізовано власний ігровий застосунок на базі рушія Unity. У процесі роботи проаналізовано сучасні підходи до створення двовимірних ігор, розглянуто основні характеристики жанру платформер, а також досліджено можливості сучасних ігрових рушіїв для реалізації ігрових механік, фізики та анімації.

У першому розділі було виконано аналіз предметної області та існуючих аналогів. Розглянуто класичні, інді, AAA та мобільні платформери, визначено їхні ключові особливості, підходи до організації геймплею та дизайну рівнів. Аналіз показав, що сучасні платформери характеризуються розвитком фізичних механік, складнішою взаємодією з ігровим середовищем, використанням анімацій та високою роллю візуального оформлення. Також встановлено, що використання рушія Unity є доцільним завдяки його кросплатформеності, наявності вбудованої фізики, системи анімації та широкого набору інструментів для створення 2D-ігор.

У другому розділі було сформовано функціональні та нефункціональні вимоги до програмного продукту, спроєктовано архітектуру застосунку та визначено основні ігрові механіки. Було розроблено структуру системи керування персонажем, механіку руху і стрибків, взаємодію з платформами, пастками та іншими об’єктами. Також було визначено підходи до організації рівнів та структури сцен гри.

У третьому розділі виконано програмну реалізацію 2D платформера. Для розробки використано мову програмування C# та інструменти Unity.

Реалізовано систему керування персонажем, фізику руху, механіку стрибків, систему колізій та анімацій. Також створено базовий користувацький інтерфейс та реалізовано взаємодію між основними компонентами гри. У процесі розробки застосовано компонентно-орієнтований підхід, що забезпечив модульність та зручність подальшого розширення функціоналу.

У четвертому розділі проведено тестування та налагодження застосунку.

Було перевірено коректність роботи ігрових механік, стабільність фізичної системи, правильність обробки зіткнень та плавність анімацій. Проведена оптимізація дозволила забезпечити стабільну продуктивність гри та коректну роботу застосунку без критичних помилок.

Отримані результати підтверджують досягнення поставленої мети роботи — розробки 2D платформера з реалізацією основних механік жанру та дослідженням ефективних підходів до створення ігрових застосунків на базі Unity. Розроблений програмний продукт може бути використаний як навчальний приклад для вивчення технологій створення 2D-ігор, а також як основа для подальшого розвитку проєкту шляхом додавання нових рівнів, ворогів, механік та елементів взаємодії.

Сидоров Г.С. Компонентний підхід до архітектури 2D платформера в Unity як основа сучасної розробки ігор // VI Всеукраїнській науково-практичній конференції «Сучасні інтелектуальні інформаційні технології в науці та освіті». Збірник тез К.: ДУІКТ, 2026. (подано до друку)

Сидоров Г.С. Метод компонентного підходу в розробці 2D платформера на базі Unity // Всеукраїнськf науково-технічнf конференціz «Технології цифрового розвитку: програмні системи та децентралізація» Збірник тез К.: ДУІКТ, 2026. (подано до друку)

ПЕРЕЛІК ПОСИЛАНЬ

1. Москаленко І. А. Розробка 2D гри на Unity — Київський національний університет технологій та дизайну, 2023. У роботі досліджено процес створення 2D платформера, використання Pixel Art та рушія Unity. URL: *університетський архів* (дата звернення: 15.05.2026).

2. Герасименюк І. В. Програмне забезпечення для 2D гри на Unity — КПІ ім. Ігоря Сікорського, 2022. Робота присвячена аналізу рушіїв та створенню 2D платформера на Unity. URL: архів КПІ (дата звернення: 15.05.2026).

3. Чернов І. І. Розробка 2D гри на платформі Unity — Чорноморський національний університет ім. Петра Могили, 2022. Реалізація платформера з алгоритмами пошуку шляху та аналізом AI у відеоіграх. URL: архів ЧНУ (дата звернення: 15.05.2026).

4. Khanh Hong. Developing a 2D Platform Endless Runner Game with Unity Engine — Vaasan University of Applied Sciences, 2023. Бакалаврська робота англійською мовою з прикладом гри «Kaizo». URL: https://www.theseus.fi/bitstream/handle/10024/798989/KhanhHong_thesis.pdf?sequence=2 (дата звернення: 15.05.2026).

5. Robert Nystrom. Game Programming Patterns — Genever Benning, 2014. Книга про архітектурні патерни у геймдеві. URL: https://gameprogrammingpatterns.com (дата звернення: 15.05.2026).

6. Joseph Hocking. Unity in Action: Multiplatform Game Development in C# — Manning Publications, 2018. Практичний посібник із розробки на Unity. URL: https://www.manning.com/books/unity-in-action-second-edition (manning.com in Bing) (дата звернення: 15.05.2026).

7. Unity Technologies. Unity Manual & Scripting API. URL: https://docs.unity3d.com (дата звернення: 15.05.2026).

8. Unity Learn. Official Tutorials and Courses. URL: https://learn.unity.com (дата звернення: 15.05.2026).

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Магістерська робота

«Метод симуляції екосистеми з використанням нейронних мереж»

Виконав: студент групи ТЦР-42 Гліб СИДОРОВ

Керівник: старший викладач кафедри ТЦР Іван ОЛЕЙНІКОВ

Київ - 2026

Мета, об’єкт та предмет дослідження

- **Мета роботи** – підвищення ефективності процесу розробки 2D ігор жанру платформер за допомогою створення програмного застосунку на базі рушія Unity із реалізацією основних ігрових механік та сучасних підходів до проєктування ігрових систем.
- **Об’єкт дослідження** – процес розробки 2D ігрових застосунків.
- **Предмет дослідження** – методи та засоби створення 2D платформера з використанням рушія Unity.

АПРОБАЦІЯ РЕЗУЛЬТАТІВ

Сидоров Г.С. Компонентний підхід до архітектури 2D платформера в Unity як основа сучасної розробки ігор // VI Всеукраїнській науково-практичній конференції «Сучасні інтелектуальні інформаційні технології в науці та освіті». Збірник тез К.: ДУІКТ, 2026(подано до друку)

Сидоров Г.С. Метод компонентного підходу в розробці 2D платформера на базі Unity // Всеукраїнська науково-технічну конференція «Технології цифрового розвитку: програмні системи та децентралізація». Збірник тез К.: ДУІКТ, 2026(подано до друку)

Актуальність роботи

Метод / Підхід	Принцип роботи	Недоліки	Переваги
Класичні 2D рушії без фізики	Ручна реалізація руху та колізій	- Складність масштабування- Велика кількість ручного коду- Обмежені можливості анімації	- Простота базової реалізації- Низькі системні вимоги
Використання готових шаблонів платформерів	Адаптація вже готових рішень	- Обмежена гнучкість- Складність внесення змін- Повторюваність механік	- Швидкий старт розробки- Економія часу
2D розробка на універсальних рушіях	Створення гри через компонентну архітектуру	- Необхідність вивчення рушія- Потреба в оптимізації продуктивності	- Велика кількість інструментів- Підтримка фізики та анімацій- Кросплатформеність
Розробка 2D платформера в Unity	Використання Physics2D, Tilemap, Animator та компонентного підходу	- Високі вимоги до структури проєкту- Необхідність оптимізації ресурсів	- Швидка розробка прототипів- Зручна система анімацій- Вбудована фізика та колізії- Можливість масштабування проєкту- Велика спільнота та документація

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Провести аналіз сучасних 2D платформерів та особливостей жанру.
2. Дослідити сучасні засоби та рушії для розробки 2D ігор.
3. Виконати порівняльний аналіз ігрових рушіїв та обґрунтувати вибір Unity для реалізації проєкту.
4. Сформувати функціональні та нефункціональні вимоги до програмного застосунку.
5. Спроєктувати архітектуру 2D платформера та структуру взаємодії його компонентів.
6. Реалізувати систему керування персонажем та основні ігрові механіки.
7. Реалізувати фізику руху, систему колізій та взаємодію об’єктів у середовищі гри.
8. Розробити систему анімацій персонажа та елементів ігрового середовища.
9. Створити ігрові рівні із використанням Tilemap та інших інструментів Unity.
10. Реалізувати базовий користувацький інтерфейс гри.
11. Провести функціональне тестування та оптимізацію програмного продукту.
12. Оцінити результати розробки та визначити можливості подальшого розвитку проєкту.

АНАЛІЗ АНАЛОГІВ

Назва гри	Жанр / Тип	Основні особливості	Переваги	Недоліки
Super Mario Bros.	Класичний 2D платформер	Лінійні рівні, проста фізика, динамічний геймплей	- Просте керування- Висока динаміка гри- Зрозумілий level design	- Обмежені механіки- Проста система взаємодії
Celeste	Indie 2D платформер	Складні механіки руху, швидкий геймплей, сюжетна складова	- Точне керування- Плавні анімації- Висока складність рівнів	- Високий поріг складності- Потребує швидкої реакції
Hollow Knight	Metroidvania платформер	Відкритий світ, бойова система, дослідження локацій	- Атмосферність- Великий світ- Якісна анімація	- Складна навігація- Високі вимоги до дизайну рівнів
Ori and the Blind Forest	Пригодницький платформер	Кінематографічна анімація, фізика руху, сюжет	- Висока якість графіки- Плавність анімацій- Емоційна атмосфера	- Високі системні вимоги- Складність реалізації подібної графіки
Geometry Dash	Аркадний платформер	Автоматичний рух, ритмічний геймплей	- Простота механік- Динамічність- Популярність серед мобільних платформ	- Одноманітність геймплею- Обмежена свобода дій
Розроблений програмний продукт	2D платформер	Реалізація базових механік жанру, фізики, анімацій та системи колізій	- Модульна архітектура- Простота розширення- Оптимізована 2D фізика- Використання можливостей Unity	- Обмежений функціонал у порівнянні з ААА-проєктами- Відсутність складної бойової системи

ВИМОГИ ДО 2D ГРИ

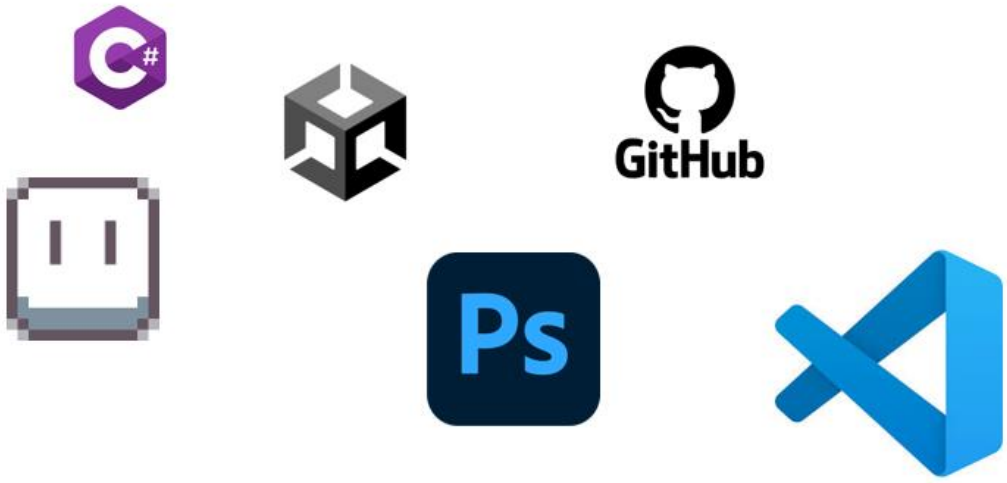
Функціональні вимоги

- Гра повинна забезпечувати керування персонажем та його взаємодію з ігровим середовищем.
- Необхідно реалізувати систему руху персонажа: переміщення, стрибки та присідання.
- Повинна бути реалізована система колізій між персонажем, платформами та іншими об’єктами.
- Гра має містити ігрові рівні з можливістю їх проходження.
- Необхідно реалізувати систему анімацій для основних станів персонажа.
- Повинен бути створений базовий користувацький інтерфейс: меню, кнопки та екран завершення гри.
- Має підтримуватись взаємодія з клавіатурою для керування персонажем.
- Повинна бути реалізована система переходу між сценами гри.
- Гра повинна підтримувати обробку зіткнень із перешкодами та платформами.
- Необхідно реалізувати систему завершення рівня або гри.

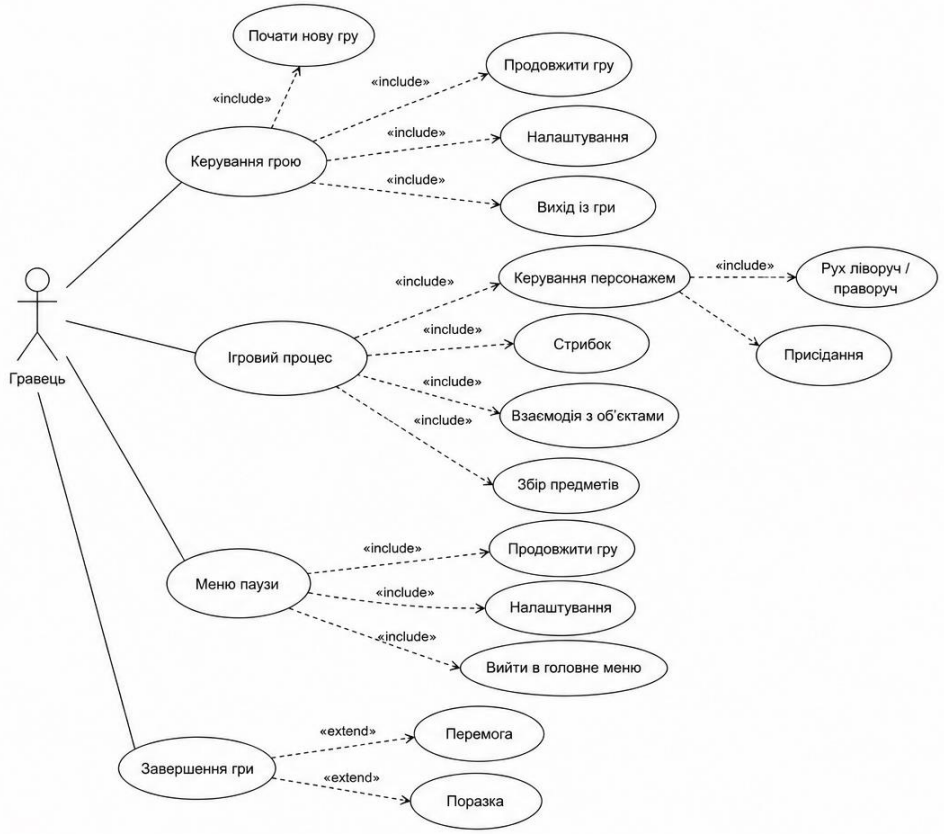
Нефункціональні вимоги

- Гра повинна працювати стабільно без критичних помилок та аварійного завершення роботи.
- Необхідно забезпечити плавний ігровий процес та стабільну частоту кадрів.
- Архітектура застосунку повинна бути модульною для можливості подальшого розширення функціоналу.
- Повинна бути забезпечена оптимізація використання оперативної пам’яті та процесорних ресурсів.
- Час завантаження сцен та рівнів повинен бути мінімальним.

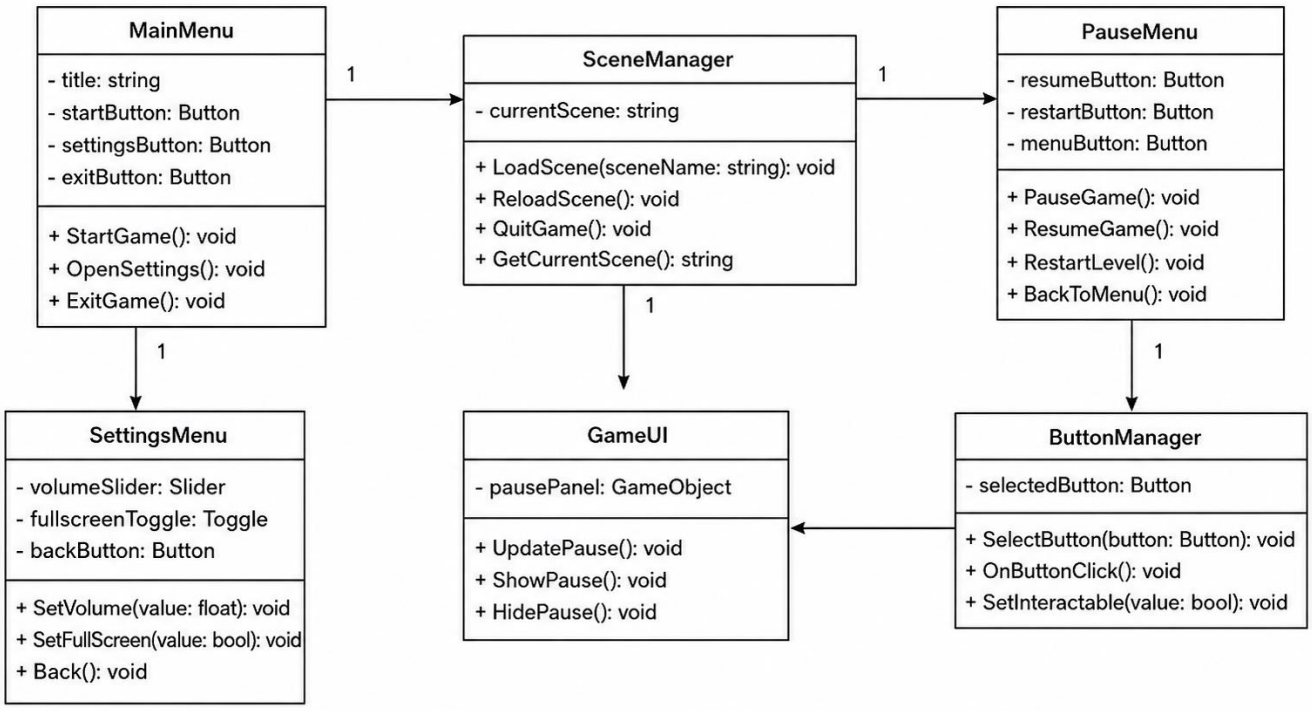
Програмні засоби реалізації



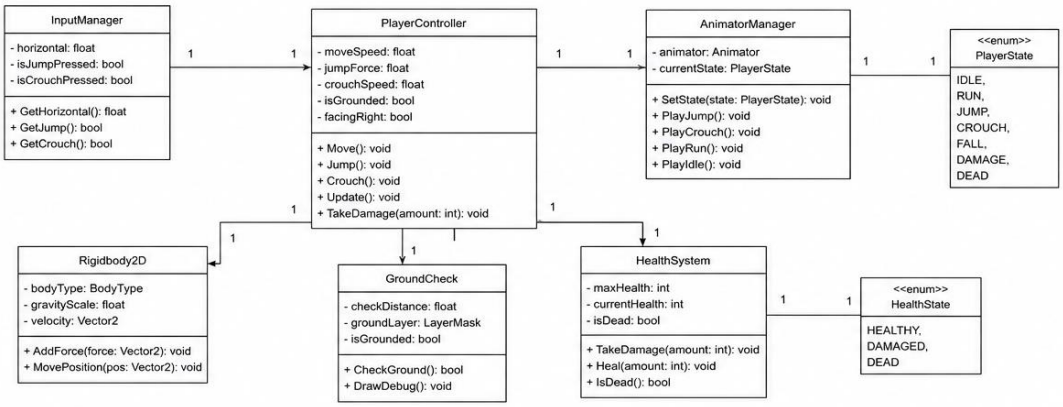
ДІАГРАМА ВАРІАНТІВ ВИКОРИСТАННЯ (ГРАВЕЦЬ)



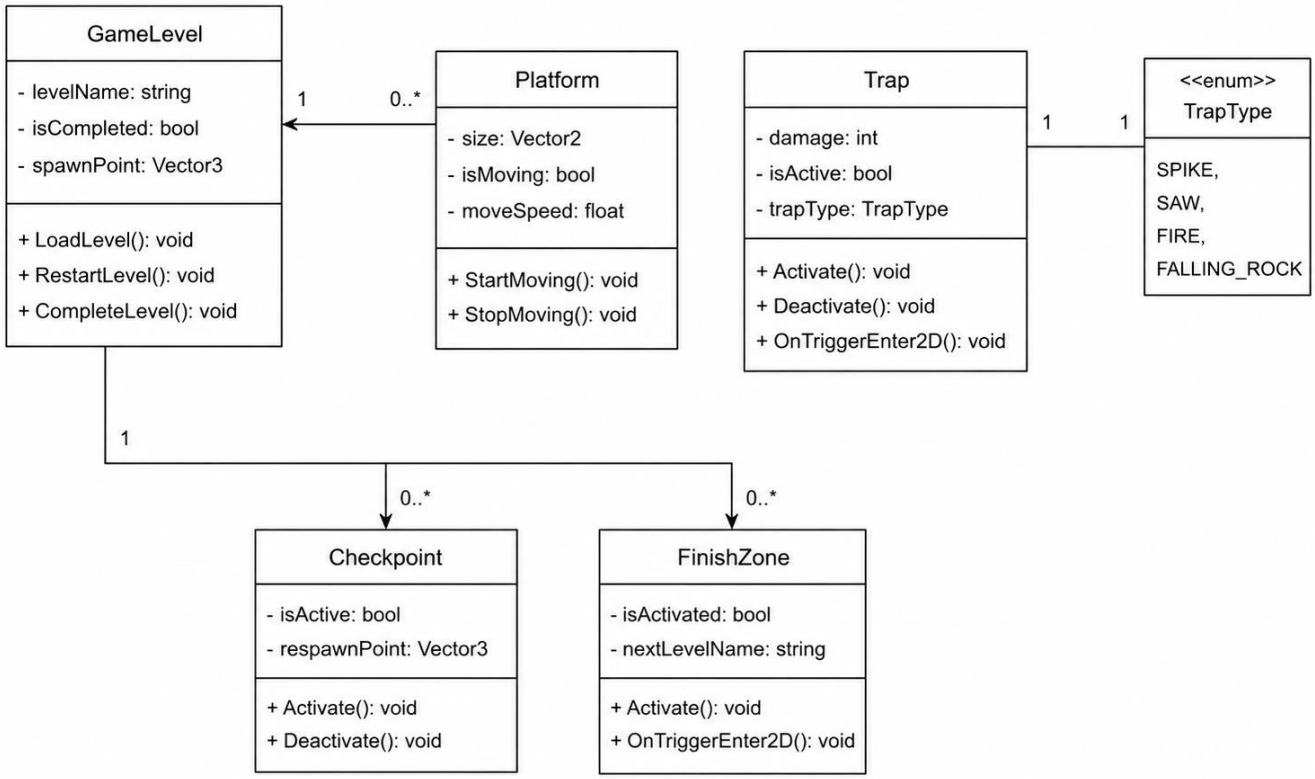
ДІАГРАМА КЛАСІВ СИСТЕМИ ІНТЕРФЕЙСУ



ДІАГРАМА КЛАСІВ СИСТЕМИ КЕРУВАННЯ ПЕРСОНАЖЕМ



ДІАГРАМА КЛАСІВ ІГРОВОГО РІВНЯ



ЕКРАННІ ФОРМИ

ЕКРАННІ ФОРМИ

ЕКРАННІ ФОРМИ

ЕКРАННІ ФОРМИ

ВИСНОВКИ

1. Проаналізовано особливості жанру 2D платформерів та сучасні підходи до їх розробки.
2. Досліджено можливості рушія Unity для створення 2D ігор.
3. Спроєктовано архітектуру програмного застосунку та структуру взаємодії компонентів.
4. Реалізовано основні ігрові механіки: рух персонажа, стрибки, систему колізій та фізику.
5. Створено систему анімацій персонажа та базовий користувацький інтерфейс.
6. Виконано тестування та оптимізацію програмного продукту.
7. Забезпечено стабільну роботу гри та можливість подальшого розширення функціоналу.
8. Усі поставлені завдання виконані, а мета кваліфікаційної роботи досягнута.

ДОДАТОК Б. ЛІСТИНГИ ОСНОВНИХ МОДУЛІВ

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

public class PlayerController : MonoBehaviour
{
    public Joystick joystick;
    public new ParticleSystem particleSystem;
    public Animator animator;
    public Rigidbody2D rb;
    private float horizontalVelocity;

    void Awake()
    {
        extraJumps = extraJumpsValue;
        rb = GetComponent<Rigidbody2D>();
    }

    void FixedUpdate()
    {
        isPlayerOnGround();
        Move();
        Jump();
        Crouch();
    }

    void Update()
    {
        animator.SetBool("Jump", !isGrounded);
        animator.SetFloat("Vertical", rb.velocity.y);

        CheckLand();
        CheckAirTime();
    }

    #region Horizontal Movement

    [Header("HORIZONTAL")]
    [Range(0, 1)] public float dampingBasic = 0.5f;
    [Range(0, 1)] public float dampingStopping = 0.5f;
    [Range(0, 1)] public float dampingTurning = 0.5f;
    [Range(1f, 1.5f)] public float movementBoost = 1.05f;
    private bool facingRight = true;

    private void Move()
    {
        horizontalVelocity = rb.velocity.x;
        horizontalVelocity += joystick.Horizontal;

        if (isCrouched && !Abilities.isCorruptionActive)
            horizontalVelocity *= crouchSpeed;

        if (Abilities.isCorruptionActive)
            horizontalVelocity *= movementBoost;

        if (Mathf.Abs(joystick.Horizontal) < 0.01f)
        {
            horizontalVelocity *= Mathf.Pow(1f - dampingStopping, Time.deltaTime * 10f);
        }
        else if (Mathf.Sign(joystick.Horizontal) != Mathf.Sign(horizontalVelocity))
        {
            horizontalVelocity *= Mathf.Pow(1f - dampingTurning, Time.deltaTime * 10f);
        }
        else
        {
            horizontalVelocity *= Mathf.Pow(1f - dampingBasic, Time.deltaTime * 10f);
        }
        rb.velocity = new Vector2(horizontalVelocity, rb.velocity.y);
```

```
    if (facingRight == false && horizontalVelocity >= 0.1f)
    {
        Flip();
    }
    else if (facingRight == true && horizontalVelocity <= -0.1f)
    {
        Flip();
    }

    animator.SetFloat("Velocity", Mathf.Abs(horizontalVelocity));
}

private void Flip()
{
    facingRight = !facingRight;
    Vector3 Scaler = transform.localScale;
    Scaler.x *= -1;
    transform.localScale = Scaler;
}
#endregion

#region Ground & Landing

[Header("GROUND")]
public float checkRadius;
private bool isGrounded;
[SerializeField] private Transform groundCheck;
public LayerMask whatIsGround;
public float groundedRemember = 0;
public float groundedRememberTime = 0.25f;
private float airTime = 0f;

private void isPlayerOnGround()
{
    isGrounded = Physics2D.OverlapCircle(groundCheck.position, checkRadius, whatIsGround);
}

private void CheckAirTime()
{
    if (isGrounded) airTime = 0f;
    else airTime += Time.deltaTime;
}

private void CheckLand()
{
    if (airTime > 0)
    {
        if (isGrounded)
        {
            CreateDust();
            FindObjectOfType<AudioManager>().Play("Landing");
        }
    }
}

private void CreateDust()
{
    particleSystem.Play();
}
#endregion

#region Jump

[Header("JUMP")]
[Range(0f, 50f)] public float jumpForce;
public float fallMultiplier = 2.5f;
private int extraJumps;
public int extraJumpsValue;

private void Jump()
{
    float vercticalMove = joystick.Vertical;
```

```
    if (crouchDisableCollider.enabled && !Abilities.isCorruptionActive)
    {
        groundedRemember -= Time.deltaTime;
        if (isGrounded)
        {
            groundedRemember = groundedRememberTime;
            extraJumps = extraJumpsValue;
        }

        if (vercticalMove >= 0.25f && extraJumps > 0)
        {
            rb.velocity = new Vector2(rb.velocity.x, jumpForce);
            extraJumps--;
            animator.SetBool("Jump", true);
            FindObjectOfType<AudioManager>().Play("Jump");
        }
        else if (vercticalMove >= 0.25f && extraJumps == 0 && isGrounded)
        {
            rb.velocity = new Vector2(rb.velocity.x, jumpForce);
            animator.SetBool("Jump", true);
            FindObjectOfType<AudioManager>().Play("Jump");
            CreateDust();
        }
        else if (vercticalMove >= 0.25f && groundedRemember > 0)
        {
            groundedRemember = 0;
            rb.velocity = new Vector2(rb.velocity.x, jumpForce);
            animator.SetBool("Jump", true);
            FindObjectOfType<AudioManager>().Play("Jump");
            CreateDust();
        }

        //fall multiplier
        if (rb.velocity.y < 0)
        {
            rb.velocity += Vector2.up * Physics2D.gravity.y * (fallMultiplier - 1) * Time.deltaTime;
        }
    }
}

#endregion

#region Crouch

[Header("CROUCH")]
[Range(0.9f, 1f)] public float crouchSpeed = 0.9f;
[SerializeField] private Transform ceilingCheck;
[SerializeField] private Collider2D crouchDisableCollider;
[SerializeField] private Collider2D crouchEnableCollider;
const float ceilingRadius = 0.2f;
private bool isCrouched = false;

private void Crouch()
{
    float vercticalMove = joystick.Vertical;

    if (vercticalMove <= -0.5f && isGrounded)
    {
        if (crouchDisableCollider != null && crouchEnableCollider != null)
        {
            crouchDisableCollider.enabled = false;
            crouchEnableCollider.enabled = true;
        }
        isCrouched = true;
        animator.SetBool("Crouch", true);
    }
    else
    {
        if (crouchDisableCollider != null && crouchEnableCollider != null)
        {
            crouchDisableCollider.enabled = true;
            crouchEnableCollider.enabled = false;
        }
        isCrouched = false;
```

```
            animator.SetBool("Crouch", false);
        }

        if (Physics2D.OverlapCircle(ceilingCheck.position, ceilingRadius, whatIsGround))
        {
            animator.SetBool("Crouch", true);
            isCrouched = true;
            if (crouchDisableCollider != null && crouchEnableCollider != null)
            {
                crouchDisableCollider.enabled = false;
                crouchEnableCollider.enabled = true;
            }
        }
    }
    #endregion

}


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

public class Abilities : MonoBehaviour, DataInterface
{
    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
        animator = GetComponent<Animator>();

        if (corruptionImage != null && corruption) corruptionImage.fillAmount = 1f;
        else corruptionImage.fillAmount = 0f;

        if (insightImage != null && insight) insightImage.fillAmount = 1f;
        else insightImage.fillAmount = 0f;
    }

    void Update()
    {
        if (corruptionCooldown)
        {
            corruptionImage.fillAmount -= 1f / darkDuration * Time.deltaTime;
            if (corruptionImage.fillAmount <= 0f) corruptionImage.fillAmount = 0f;
        }
    }

    #region Ability: Corruption

    [Header("CORRUPTION SETTINGS")]
    [Range(0f, 8f)] public float darkDuration = 5f;
    public static bool corruption = false;
    public static bool isCorruptionActive;
    private bool corruptionCooldown = false;
    public Image corruptionImage;
    private Rigidbody2D rb;
    private Animator animator;

    public void Corruption()
    {
        if (corruption && !corruptionCooldown && !DeathRespawn.isDead)
        {
            StartCoroutine(Ability3(darkDuration));
        }

        IEnumerator Ability3(float duration)
        {
            corruptionCooldown = true;
            isCorruptionActive = true;
            animator.SetBool("Corruption", true);
            rb.gravityScale = 0f;

            yield return new WaitForSeconds(duration);
            isCorruptionActive = false;
```

```
        rb.gravityScale = 3.3f;
        animator.SetBool("Corruption", false);
        Invoke("ResetCorruption", 8f);
    }
}
void ResetCorruption()
{
    corruptionCooldown = false;
    corruptionImage.fillAmount = 1f;
}

#endregion

#region Ability: Granted Eyes

[Header("EYE OF WISDOM SETTINGS")]
public GameObject flashlight;
public static bool eyeOfWisdom = false;
public bool isEyeOfWisdomActive;
[Range(0f, 60f)] public float lightDuration = 30f;

[Header("INSIGHT SETTINGS")]
public ParticleSystem vision;
public static bool insight = false;
private bool insightCooldown = false;
public Image insightImage;
[SerializeField] private BossFightManager theDarkness;

public void ElderSigns()
{
    //Eye of Wisdom
    if (eyeOfWisdom && !insight && !isEyeOfWisdomActive && !DeathRespawn.isDead)
    {
        StartCoroutine(Ability2(lightDuration));
    }

    //Insight
    if (eyeOfWisdom && insight && BossFightManager.inTheFight && !DeathRespawn.isDead)
    {
        if (!insightCooldown)
        {
            vision.Play();
            insightImage.fillAmount = 0f;
            if (flashlight != null) Destroy(flashlight);
            if (theDarkness != null) theDarkness.DamageDarkness();
            Invoke("ResetInsight", 30f);
            insightCooldown = true;
        }
    }

    IEnumerator Ability2(float duration)
    {
        isEyeOfWisdomActive = true;
        flashlight.SetActive(true);

        yield return new WaitForSeconds(duration);
        flashlight.SetActive(false);
        isEyeOfWisdomActive = false;
    }
}
void ResetInsight()
{
    insightCooldown = false;
    insightImage.fillAmount = 1f;
}

#endregion

#region Save System

public void LoadData(GameData data)
{
    foreach(KeyValuePair<string, bool> pair in data.abilityCollected)
    {
```

```
            if (pair.Key == "eye-of-wisdom")
            {
                eyeOfWisdom = pair.Value;
            }

            if (pair.Key == "corruption")
            {
                corruption = pair.Value;
            }

            if (pair.Key == "insight")
            {
                insight = pair.Value;
            }
        }
    }
    public void SaveData(GameData data)
    {
        //Nothing to save here...
    }
    #endregion
}

using System.Collections;
using System.Collections.Generic;
using UnityEngine.Events;
using UnityEngine.EventSystems;
using UnityEngine;
using System;

public class ShadowPlatform : MonoBehaviour, DataInterface
{
    #region Line Drawing Ability

    [Header("TIMERS")]
    [Range(0f, 15f)] public float cooldown;
    [Range(0f, 5f)] public float despawnTimer;

    [Header("LINE")]
    [Range(0f, 30f)] public int lengthOfLineRenderer;
    public GameObject linePrefab;
    public static bool shadowPlatform = false;

    [Header("OBJECTS")]
    public GameObject currentLine;
    public LineRenderer lineRenderer;
    public EdgeCollider2D edgeCollider;
    public List<Vector2> touchPositions;
    private bool isShadowSpawned = false;
    private float canUse = -1f;
    private bool isFingerDown = false;

    void Update()
    {
        foreach (Touch touch in Input.touches)
        {
            int id = touch.fingerId;
            if (shadowPlatform && !EventSystem.current.IsPointerOverGameObject(id) && (touch.phase == TouchPhase.Began) &&
        !PauseMenu.GameIsPaused)
            {
                if (!isShadowSpawned && Time.time > canUse && Input.touchCount == 1)
                {
                    canUse = Time.time + cooldown;

                    currentLine = Instantiate(linePrefab, Vector3.zero, Quaternion.identity);
                    lineRenderer = currentLine.GetComponent<LineRenderer>();
                    edgeCollider = currentLine.GetComponent<EdgeCollider2D>();
                    touchPositions.Clear();

                    touchPositions.Add(Camera.main.ScreenToWorldPoint(touch.position));
                    touchPositions.Add(Camera.main.ScreenToWorldPoint(touch.position));

                    lineRenderer.SetPosition(0, touchPositions[0]);
                    lineRenderer.SetPosition(1, touchPositions[1]);
```

```
                edgeCollider.points = touchPositions.ToArray();
                isShadowSpawned = true;
                isFingerDown = true;
            }
        }
        if ((touch.phase == TouchPhase.Moved) && !PauseMenu.GameIsPaused && lineRenderer != null)
        {
            Vector2 tempFingerPos = Camera.main.ScreenToWorldPoint(touch.position);
            if (Vector2.Distance(tempFingerPos, touchPositions[touchPositions.Count - 1]) > .1f)
            {
                touchPositions.Add(tempFingerPos);
                if (lineRenderer.positionCount < lengthOfLineRenderer && isFingerDown && Input.touchCount == 1)
                {
                    lineRenderer.positionCount++;
                    lineRenderer.SetPosition(lineRenderer.positionCount - 1, tempFingerPos);
                    edgeCollider.points = touchPositions.ToArray();
                }
            }
        }
        if (touch.phase == TouchPhase.Ended)
        {
            isFingerDown = false;
            DeleteShadow();
        }
    }
}

private void DeleteShadow()
{
    Destroy(currentLine, despawnTimer);
    isShadowSpawned = false;
}
#endregion

#region Save System
public void LoadData(GameData data)
{
    foreach(KeyValuePair<string, bool> pair in data.abilityCollected)
    {
        if (pair.Key == "shadow-platform")
        {
            shadowPlatform = pair.Value;
        }
    }
}

public void SaveData(GameData data)
{

}
#endregion
}
```