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

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Інтерактивний 2D платформер з елементами головоломки та подорожами у часі на 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

_____ Ярослав ЛЯШЕНКО
(підпис)

Виконав: здобувач вищої освіти групи ТЦР-42
Ярослав ЛЯШЕНКО

Керівник: Вадим ДЗЮБА
ст. викладач.

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

Ляшенку Ярославу Володимировичу

1. Тема кваліфікаційної роботи: «Інтерактивний 2D платформер з елементами головоломки та подорожами у часі на Unity»

керівник кваліфікаційної роботи ст. викладач Вадим ДЗЮБА,

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 та практичні матеріали з розробки 2D-ігор, документація ігрового рушія Unity та мови програмування C#,

аналіз сучасних 2D-платформерів з елементами головоломок, приклади реалізації механік керування часом у іграх.

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

1. Огляд та аналіз сучасного стану жанру 2D-платформерів з елементами головоломок.
2. Порівняльний аналіз ігрових рушіїв для розробки 2D-ігор та обґрунтування вибору Unity.
3. Програмна реалізація ключових систем: керування персонажем, механіки подорожей у часі, системи головоломок.
4. Тестування та оцінка якості прототипу.

5. Перелік графічного матеріалу: *презентація*

1. Аналіз аналогів.
2. Вимоги до програмного забезпечення.
3. Програмні засоби реалізації.
4. Діаграма варіантів використання (Use Case).
5. Алгоритм роботи застосунку.
6. Алгоритм взаємодії гравця з системою
7. Діаграма класів.
8. Екранні форми.
9. Апробація результатів дослідження.

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	23.02-14.03.2026	Виконано
2	Аналіз та дослідження існуючих аналогів	23.02-04.03.2026	Виконано
3	Огляд і порівняння ігрових рушіїв для 2D-розробки, обґрунтування вибору Unity	24.02-25.03.2026	Виконано
4	Проектування архітектури	04.03-05.03.2026	Виконано
5	Програмна реалізація застосунку	05.03-08.04.2026	Виконано
6	Тестування застосунку	09.04-10.04.2026	Виконано
7	Оформлення роботи: вступ, висновки, реферат	10.03-13.04.2026	Виконано
8	Розробка демонстраційних матеріалів	13.04-15.04.2026	Виконано
9	Попередній захист роботи	12.05.2026	Виконано

Здобувач вищої освіти _____ Ярослав ЛЯШЕНКО

Керівник кваліфікаційної роботи _____ Вадим ДЗЮБА

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: __50__ стор., __3__ табл., __10_ рис., _21_ джерел.

Мета роботи – підвищити ігрового досвіду за допомогою інтерактивного 2D платформера з елементами головоломки та подорожами у часі.

Об'єкт дослідження – процес розробки 2D-платформера з елементами головоломок.

Предмет дослідження – методи та засоби ігрових продуктів, а саме 2D-ігор з нестандартними механіками на базі ігрового рушія Unity та мови програмування C#.

Короткий зміст роботи: У роботі розроблено функціональний прототип 2D-платформера «CronoWortex», який поєднує класичний платформінг з оригінальною механікою подорожей у часі на основі перемикання часових режимів (ModeActivator) та парної телепортації (PlayerTeleportSystem).

Реалізовано гнучку архітектуру на принципах SOLID та патерні State Machine. Проведено аналіз жанру, порівняння ігрових рушіїв та вибір стеку Unity та C#. Створено набір головоломок, виконано комплексне тестування (28 тест-кейсів).

Сферою використання застосунку є індустрія розробки інді-ігор, освітній процес (навчальні курси з ігрової розробки та програмної інженерії).

КЛЮЧОВІ СЛОВА: 2D-ПЛАТФОРМЕР, PUZZLE-PLATFORMER, МЕХАНІКА ПОДОРОЖЕЙ У ЧАСІ, UNITY, C#, STATE MACHINE, SOLID, MODEACTIVATOR, PLAYERTELEPORTSYSTEM, ГОЛОВОЛОМКИ.

Зміст

ВСТУП

Сучасні 2D-платформери переживають справжній ренесанс. Після тривалого періоду домінування масштабних тривимірних AAA-проєктів спостерігається чітка тенденція повернення інтересу як гравців, так і розробників до двовимірних ігор. У таких проєктах високо цінуються висока динаміка геймплею, філігранна точність керування персонажем, виразна піксель-арт естетика та, що особливо важливо, можливість створення глибоких, інтелектуально насичених ігрових механік. Одним з найбільш перспективних і популярних піджанрів на сьогодні є puzzle-platformers – ігри, у яких класичний платформінг органічно переплітається з логічними головоломками, що вимагають від гравця не лише швидкої реакції, а й розвинутого стратегічного та просторового мислення.

Актуальність обраної теми зумовлена кількома вагомими чинниками. Ринок інді-ігор продовжує демонструвати стабільне зростання, при цьому якісні 2D-проєкти часто стають комерційно успішними завдяки оригінальності геймплею, порівняно низькій вартості розробки та високій рентабельності. Крім того, реалізація складних нелінійних механік, таких як маніпуляція часом, досі залишається серйозним технічним і архітектурним викликом, особливо для студентських та невеликих незалежних команд з обмеженими ресурсами. Не менш важливим є освітній і когнітивний потенціал таких ігор, оскільки вони сприяють розвитку просторового, стратегічного, критичного та причинно-наслідкового мислення у гравців різних вікових категорій.

Об'єктом дослідження є процес розробки інтерактивного 2D-платформера з елементами головоломок.

Предметом дослідження є методи та засоби ігрових продуктів, а саме 2D-ігор з нестандартними механіками на базі ігрового рушія Unity та мови програмування C#.

Мета роботи – підвищення ігрового досвіду за допомогою інтерактивного 2D платформера з елементами головоломки та подорожами у часі.

Поставлена мета була досягнута шляхом вирішення комплексу таких завдань. Було проведено аналіз сучасного стану жанру 2D-платформерів та puzzle-platformers, а також виконано порівняльну характеристику найпоширеніших ігрових рушіїв для 2D-розробки. Обґрунтовано вибір технологічного стеку (Unity та C#) і ключових архітектурних підходів з урахуванням особливостей реалізації механіки подорожей у часі. Реалізовано основні ігрові системи, зокрема керування персонажем, систему анімації, перемикання часових режимів та парну телепортацію. Створено набір взаємопов'язаних головоломок, побудованих навколо комбінованої механіки часу. Крім того, проведено комплексне тестування ключових механік гри, яке включало функціональне, інтеграційне та тестування зручності використання (usability testing).

Структура дипломної роботи складається з чотирьох основних розділів. У першому розділі проаналізовано предметну галузь, сучасні тенденції розвитку жанру 2D-платформерів та існуючі інструменти розробки. Другому розділу присвячено проектуванню архітектури гри та її ключових механік. У третьому розділі детально описано процес реалізації основних систем і компонентів. У четвертому розділі наведено результати тестування, проведено аналіз ефективності запропонованих архітектурних рішень, оцінено продуктивність гри та сформульовано перспективи подальшого розвитку проєкту.

1 Аналіз проблеми та існуючих засобів її вирішення

1.1 Опис предметної галузі: сучасні 2D-платформери та ігри з елементами головоломок

Жанр 2D-платформерів є одним із найстаріших і найстійкіших у історії комп’ютерних ігор. Його витоки сягають 1980-х років, коли з’явилися культові проєкти, такі як Super Mario Bros. (1985), Sonic the Hedgehog (1991) та Prince of Persia (1989)[1]. Основна механіка жанру полягає в керуванні персонажем у двовимірному просторі, що складається з платформ, перешкод, пасток і супротивників, з метою подолання рівня та досягнення кінцевої точки.

Основна механіка 2D-платформерів полягає в керуванні персонажем у двовимірному просторі, що складається з платформ, перешкод, пасток, рухомих елементів і супротивників. Гравець повинен долати рівні, демонструючи точність рухів, швидкість реакції та вміння планувати свої дії. На відміну від тривимірних ігор, 2D-платформери вирізняються відносною простотою графічної реалізації, яку успішно компенсують висока динаміка геймплею, прецизійне керування, складний дизайн рівнів та інтуїтивність механік. Саме ці якості забезпечують швидке занурення гравця в ігровий процес і тривале утримання уваги.

Яскравими прикладами комерційного та критичного успіху жанру стали Celeste (2018), Hollow Knight (2017) та його продовження Hollow Knight: Silksong (2025), Ori and the Blind Forest / Will of the Wisps, Inside (2016) та Little Nightmares. У 2025 році критики одностайно відзначили справжній ренесанс 2D-платформерів – рік став одним із найсильніших для жанру з часів епохи Super NES завдяки виходу таких проєктів, як Shinobi: Art of Vengeance, Ninja Gaiden: Ragebound та численних інді-проектів. Hollow Knight: Silksong лише за перші

місяці після релізу розійшовся тиражем понад 7 мільйонів копій, що підтверджує стійкий комерційний інтерес до жанру [2],[3].

Поєднання платформінгу з елементами головоломок

Окремим і перспективним напрямом розвитку є puzzle-platformers – ігри, в яких механіка точного пересування та стрибків органічно поєднується з розв'язанням логічних задач. Такий синтез робить геймплей не лише динамічним, але й інтелектуально насиченим, розширюючи цільову аудиторію.

Окремим перспективним напрямком розвитку є puzzle-platformers – ігри, у яких механіка точного платформінгу тісно інтегрується з елементами логічних головоломок. Такий синтез робить геймплей не лише динамічним і видовищним, але й інтелектуально насиченим, значно розширюючи цільову аудиторію. Класичними представниками цього піджанру є Braid (2008), де центральною механікою виступає маніпуляція часом [4], атмосферні головоломки в Limbo (2010) та Inside [5], а також просторові задачі з використанням клонів у The Swapper (2013) [6].

Популярність puzzle-platformers пояснюється кількома ключовими факторами. Жанр приваблює широку аудиторію – від казуальних гравців, які шукають короткі захопливі сесії, до хардкорних ентузіастів, що цінують високу складність, прецизійний контроль і можливості speedrun. Такі ігри мають високу рентабельність завдяки численним секретам, альтернативним шляхам проходження та різним рівням складності. Вони забезпечують оптимальний баланс між викликом і відчуттям прогресу, що є одним із найважливіших елементів успішного ігрового дизайну. Багато тайтлів, зокрема Celeste [7] та серія Hollow Knight [8], досягли мільйонних продажів і високих оцінок на платформах Steam та Metacritic.

Сучасні 2D-платформери також демонструють низку чітких тенденцій розвитку. Серед них – суттєве посилення сюжетної та емоційної складової, використання унікальних ігрових механік (маніпуляція часом, гравітацією,

порталами, мульти персонажні системи), активна кросплатформеність та створення багатошарових рівнів із поступовим підвищенням складності. Крім того, жанр все частіше виходить за межі чистої розваги, стаючи потужним художнім засобом. Через наратив, візуальний стиль і механіки автори торкаються важливих соціальних і психологічних тем: подолання тривоги та депресії (Celeste), самотності та страху (Limbo, Inside), екології, дружби, втрати та внутрішніх конфліктів.

Важливим аспектом є освітній і когнітивний потенціал жанру. Платформери з елементами головоломок сприяють розвитку логічного та просторового мислення, уваги, планування дій, креативності та нестандартного підходу до розв’язання проблем. Завдяки цьому їх активно використовують у гейміфікації освіти, розвитку executive functions та терапевтичних практиках.

Значний вплив на розвиток жанру справив Steam – провідна платформа цифрової дистрибуції. Саме завдяки їй невеликі інді-команди та соло-розробники отримали можливість безпосередньо звертатися до мільйонної аудиторії, швидко отримувати зворотний зв’язок і досягати комерційного успіху без підтримки великих видавців.

Не менш важливим чинником є технічний прогрес у сфері ігрових рушіїв. Сучасні інструменти значно спростили створення якісних 2D-ігор. Unity залишається одним з найпопулярніших рішень завдяки потужній 2D-системі, зручному редактору та легкості портування на різні платформи [10], [11]. Godot приваблює відкритим кодом і високою продуктивністю [15], а GameMaker – швидкістю прототипування [14]. Unreal Engine частіше використовується для візуально насичених проєктів.

Сучасні 2D-платформери демонструють стійку тенденцію до кросплатформеності. Більшість нових ігор виходять одночасно на ПК, Nintendo Switch, PlayStation, Xbox та мобільні пристрої, що дозволяє значно розширити аудиторію та підвищити комерційну ефективність.

Перспективи подальшого розвитку жанру пов’язані з інтеграцією новітніх технологій. Зокрема, все більшої популярності набуває поєднання 2D-механік з елементами віртуальної та доповненої реальності, процедурна генерація рівнів, адаптивна складність за допомогою штучного інтелекту, а також гібридні жанрові рішення (платформер та roguelike, платформер та RPG, метроідванія). Також очікується подальше поглиблення наративної складової та використання ігор як засобу художнього висловлювання актуальних соціальних і психологічних проблем.

Особливе місце в сучасному розвитку жанру займає механіка маніпуляції часом. Вона відкриває абсолютно нові можливості для створення нелінійних головоломок і часових парадоксів, які неможливі в традиційних платформерах. Якщо раніше подорожі в часі були представлені в поодиноких проєктах (Braid, The Misadventures of P.B. Winterbottom, Super Time Force), то сьогодні вони стають повноцінним і навіть центральним елементом геймплею. Такий підхід дозволяє гравцеві впливати не лише на простір, а й на час, створюючи складні причинно-наслідкові ланцюжки та альтернативні сценарії проходження одного й того самого рівня.

Аналіз сучасного стану жанру свідчить, що успішна реалізація 2D-платформера з елементами головоломок вимагає не тільки креативного ігрового дизайну, але й продуманої архітектури програмного забезпечення. Саме тому вивчення ефективних підходів до побудови гнучкого та масштабованого коду набуває особливого значення для розробників, особливо в умовах ітеративної розробки, коли механіки постійно доповнюються та змінюються.

Механіка часу може реалізовуватися по-різному: від повного перемотування часу назад, як у Braid, до паралельних часових ліній, контролю над часом окремих об’єктів чи комбінованих систем, подібних до тієї, що використовується в проєкті «CronoWortex». Кожна з цих реалізацій ставить перед розробником серйозні

технічні завдання, пов’язані з керуванням станом рівня, збереженням історії дій, обробкою колізій та забезпеченням стабільності гри.

Важливим аспектом сучасних 2D-платформерів є також висока увага до якості анімації та зворотного зв’язку. Гравці очікують плавної та зрозумілої реакції персонажа на кожну дію. Саме тому багато розробників приділяють значну увагу системам анімації, по кадровому контролю, squash and stretch ефектам та динамічній камері. Успішне поєднання точного платформінгу з візуальною привабливістю значно підвищує задоволення від гри.

Не менш важливим є дизайн рівнів. Сучасні 2D-платформери часто використовують принцип поступового підвищення складності, коли нові механіки вводяться поступово, а потім комбінуються у все більш складні ситуації. Такий підхід дозволяє підтримувати інтерес гравця протягом усього проходження і уникнути різких стрибків у складності.

У контексті студентських і інді-проєктів особливо актуальним стає питання архітектури програмного забезпечення. При додаванні нових механік, таких як різні часові режими, розробники часто стикаються з проблемою швидкого зростання складності коду. Це призводить до появи «спагеті-коду», складнощів у тестуванні та проблем із подальшим розширенням проєкту. Тому застосування сучасних принципів програмної інженерії набуває великого практичного значення навіть у відносно невеликих іграх.

1.2 Аналіз існуючих програмних продуктів для розробки 2D ігор

Створення сучасних 2D-ігор значно спростилося завдяки появі потужних ігрових рушіїв та інтегрованих середовищ розробки. Ці інструменти надають готові рішення для роботи з графікою, фізикою, анімацією, звуком, інтерфейсом та ігровою логікою. Завдяки цьому розробники можуть мінімізувати час на

написання базового коду «з нуля» і зосередитися на геймдизайні, створенні контенту та поліпшенні ігрового досвіду.

Ігрові рушії для розробки 2D-ігор можна умовно поділити на дві основні категорії. До першої належать універсальні рушії (Unity, Unreal Engine, Godot), які підтримують як двовимірну, так і тривимірну розробку. Друга категорія – це спеціалізовані 2D-інструменти (GameMaker, Construct), орієнтовані переважно або виключно на створення двовимірних ігор [10], [11].

Нижче наведено аналіз найпоширеніших середовищ розробки, що використовуються для створення 2D-платформерів та ігор з елементами головоломок у 2025-2026 роках.

Серед сучасних ігрових рушіїв Unity залишається одним із найпопулярніших універсальних рішень завдяки своїй гнучкості та розвиненій екосистемі. Він підтримує як 2D, так і 3D-розробку, пропонує потужний редактор сцен та вбудовані інструменти для роботи з фізикою, анімацією й звуком [13].

Unity вирізняється широким набором 2D-інструментів (Tilemap, 2D Physics, Sprite Renderer, Animator, Pixel Perfect Camera), великою спільнотою, багатою документацією та відмінною кросплатформеністю. Крім того, Unity Asset Store надає доступ до величезної кількості готових асетів, плагінів і рішень, що значно прискорює процес розробки. Водночас рушій має вищі системні вимоги порівняно з більш легкими рішеннями, а його 2D-функціональність реалізована поверх 3D-рушія, що іноді знижує оптимальність для чистого pixel-art. Для реалізації складних проєктів також потрібні ґрунтовні знання програмування.

Не менш популярним універсальним рушієм є Unreal Engine від Epic Games. Він орієнтований насамперед на високоякісну 3D-графіку та використовує для 2D-проєктів модуль Paper2D та потужну систему візуального програмування Blueprints. Unreal Engine забезпечує найвищу якість візуалізації, розширені можливості освітлення, ефектів та постобробки. Серед ключових переваг – можливість легко поєднувати 2D і 3D елементи в одному проєкті, а також

безкоштовний доступ з роялті лише після досягнення певного рівня доходу. Однак для класичних 2D-платформерів рушій часто вважається надмірно «важким»: Paper2D має обмежену функціональність і меншу підтримку, крива навчання досить висока, а розмір проєкту та системні вимоги суттєво зростають навіть при простих 2D-іграх.

Ще одним перспективним рішенням, яке активно набирає популярність, є Godot – вільний рушій з відкритим кодом під ліцензією MIT. У 2025–2026 роках Godot демонструє значне зростання серед інді-розробників. Він має спеціалізований native 2D-конвеєр зі справжніми 2D-координатами, що забезпечує високу продуктивність і точність рендерингу. Godot повністю безкоштовний, не має роялті та пропонує зручну систему вузлів (Node-based architecture), вбудований редактор та підтримку кількох мов програмування, зокрема GDScript, C# і C++. Рушій показує відмінну ефективність навіть на слабкому обладнанні. Головними недоліками на сьогодні залишаються менша кількість готових асетів і плагінів порівняно з Unity, а також менш розвинена екосистема для складних комерційних проєктів.

Значна кількість розробників також обирає GameMaker – спеціалізований інструмент, орієнтований виключно на 2D-ігри. Він поєднує інтуїтивне візуальне програмування (Drag & Drop) з потужною мовою GML. GameMaker добре оптимізований для pixel-art, пропонує швидке створення прототипів і зручну роботу зі спрайтами, кімнатами та колізіями. Саме на цьому рушії були створені такі відомі тайтли, як Undertale, Hyper Light Drifter, Pizza Tower та DELTARUNE. Однак для повноцінного комерційного релізу та експорту на консолі потрібна платна ліцензія. Крім того, можливості реалізації складних архітектурних рішень і нелінійних механік тут значно нижчі, ніж в Unity чи Godot.

Нарешті, Construct є яскравим представником no-code рушіїв, повністю орієнтованих на 2D-ігри. Уся ігрова логіка в ньому створюється через візуальний редактор подій і поведінок, що не вимагає знання мов програмування. Це робить

Construct ідеальним інструментом для швидкого прототипування, освітніх проєктів, веб-ігор та простих мобільних додатків. Рушій добре підтримує HTML5 і має зручні інструменти для роботи з фізикою та анімаціями. Головним недоліком є суттєве обмеження гнучкості при реалізації складних механік, великих проєктів та нетривіальних ігрових систем, таких як комбінована маніпуляція часом.

Таким чином, вибір ігрового рушія суттєво залежить від масштабів проєкту, досвіду розробника, необхідної гнучкості архітектури та цільових платформ. Для реалізації «CronoWortex» – 2D-платформера з складною комбінованою механікою часу – було обрано Unity як оптимальне поєднання потужності, гнучкості та зрілості екосистеми.

Аналіз існуючих середовищ розробки свідчить, що вибір рушія залежить від масштабів проєкту, досвіду команди, цільових платформ і необхідності реалізації складних механік. Для розробки 2D-платформера з елементами головоломок і механікою подорожей у часі особливо важливими є гнучкість архітектури, точна фізика та зручність роботи зі станами об’єктів.

Після порівняльного аналізу основних ігрових рушіїв для реалізації проєкту «CronoWortex» було обрано Unity. Таке рішення зумовлено оптимальним поєднанням зручності розробки, потужної екосистеми, гнучкості програмної архітектури та достатньої продуктивності для реалізації комбінованої механіки подорожей у часі.

1.2.1 Порівняльна таблиця ігрових движків та редакторів

Для об’єктивної оцінки можливостей середовищ розробки 2D-ігор проведено порівняльний аналіз найпоширеніших рушіїв станом на 2025-2026 роки. Порівняння базується на ключових технічних характеристиках, зручності використання, ліцензійній політиці та прикладах реальних проєктів.

Таблиця 1.1

Порівняльна характеристика ігрових рушіїв для розробки 2D-ігор

Рушій / Редактор	Основна мова	Орієнтація	Ліцензія / Ціна	Переваги	Недоліки	Типові проєкти (приклади)	Найкраще підходить для
Unity	C#	2D та 3D (універсальний)	Безкоштовно та роялті після порогу доходу	Відмінна кросплатформеність, великий Asset Store, потужні 2D-інструменти	Вищі системні вимоги, 2D реалізований поверх 3D-рушія	Hollow Knight, Hollow Knight: Silksong (2025)	Кросплатформені ігри, мобільні проєкти, середні та великі інді
Unreal Engine	C++, Blueprints (візуальне)	Переважно 3D (Paper2D для 2D)	Безкоштовно та роялті після порогу	Найвища якість графіки та ефектів, гібридні 2D/3D проєкти	Обмежена підтримка Paper2D, висока крива навчання, «важкий» рушій для простих 2D	Octopath Traveler (HD-2D стиль), гібридні проєкти	Візуально насичені гібридні ігри, AAA-рівень графіки
Godot Engine	GDScript, C#, C++	2D та 3D (відмінний native 2D)	100% безкоштовно (MIT, відкритий код)	Легкість, висока продуктивність, зручна система вузлів	Менша екосистема асетів і плагінів порівняно з Unity	Багато інді-ігор 2025-2026 рр. (Brotato, Tiny Pasture та ін.)	Інді-2D платформери, соло та малі команди, навчальні проєкти
GameMaker	GML та Drag & Drop	Спеціально 2D	Безкоштовна та платна для комерції	Швидка розробка, інтуїтивний інтерфейс, оптимізація для pixel-art	Обмежені можливості для складних механік і 3D	Undertale, Hyper Light Drifter, DELTARUNE	Швидкі 2D-платформери, pixel-art ігри, початківці та професіонали
Construct	Візуальні блоки (no-code)	2D	Підписка для повного доступу	Не потребує програмування, швидке прототипування	Обмежена гнучкість для складних механік і великих проєктів	Прості браузерні та мобільні ігри	Початківці, освітні проєкти, швидкі прототипи, веб-ігри

Джерела аналізу базуються на оглядах Incredibuild, Generalist Programmer, itch.io Blog, офіційній документації рушіїв, а також даних про релізи ігор та тенденції ринку станом на 2025–2026 роки [10], [11], [12], [16], [17].

Проведений порівняльний аналіз рушіїв свідчить та дозволяє зробити такі висновки, що саме Unity лідирує за універсальністю, розміром спільноти та багатством екосистеми, однак для чисто 2D-проєктів може бути надмірно складним і ресурсоємним. В той же час Godot демонструє швидке зростання популярності серед інді-розробників і студентів завдяки своїй відкритості, високій ефективності в двовимірних проєктах та відсутності ліцензійних платежів [10], [12], [15].Значною популярністю продовжує користуватись GameMaker Studio залишається одним з найкращих рішень для швидкої розробки класичних 2D-ігор завдяки зручному візуальному інструментарію [13], [18]. З другої сторони нішеві позиції займають Unreal Engine і Construct: перший переважно використовується для візуально складних гібридних проєктів, а другий – для no-code рішень і швидкого прототипування.

Таким чином, на основі проведеного аналізу порівняльної характеристики ігрових рушіїв можна зробити обґрунтований вибір інструментарію залежно від масштабів проєкту, досвіду розробника, цільових платформ та технічних вимог. У наступному підрозділі детально розглянуті переваги та недоліки розглянутих рішень з урахуванням специфіки реалізації 2D-платформера з елементами головоломок і механікою подорожей у часі.

1.2.2 Обґрунтування обмеження існуючих рішень

Проведений аналіз показує, що жоден із популярних ігрових рушіїв не є ідеальним. Кожен має свої суттєві недоліки, які особливо помітні, коли потрібно реалізувати нестандартні механіки, такі як складна система подорожей у часі.

Unity, попри свою універсальність, має одну суттєву особливість: його 2D-функціональність фактично «лежить» поверх 3D-рушія. Через це навіть відносно прості 2D-проєкти можуть мати вищі системні вимоги, ніж очікувалося.

Крім того, для ефективної роботи з Unity потрібні досить добрі знання C#. Для початківців або студентських команд це часто стає серйозним бар’єром. Також у великих проєктах помітно зростає розмір білду і іноді падає продуктивність при великій кількості спрайтів та анімацій.

Тому вважається що, Unreal Engine з модулем Paper2D ще менш підходить для класичних 2D-платформерів. Більшість його потужних можливостей (Nanite, Lumen, Chaos Physics) залишаються просто невикористаними, а підтримка 2D значно слабша, ніж у спеціалізованих рішень. Крива навчання тут дуже крута, тому для невеликих проєктів і дипломних робіт Unreal Engine часто виявляється надмірно важким і незручним.

В той же час Godot Engine виглядає дуже привабливо завдяки легкості та продуктивності, але й він не без вад. Екосистема плагінів і готових асетів у нього помітно менша, ніж в Unity. Реалізація дійсно нестандартних механік може вимагати значно більше часу і зусиль, особливо якщо розробник не має великого досвіду. Також підтримка деяких консольних платформ у Godot поки що поступається Unity.

У випадку з GameMaker Studio рушій добре справляється зі швидким прототипуванням, але при спробі реалізувати складну логіку (нелінійні механіки часу, багатошарові головоломки, точну взаємодію великої кількості об’єктів) починає проявляти свої обмеження. Крім того, для повноцінного комерційного релізу потрібна платна ліцензія, а гнучкість коду значно нижча, ніж у Unity чи Godot.

Водночас рушій Construct як no-code інструмент ідеально підходить для швидких і простих ігор, але майже не дозволяє реалізовувати складні алгоритмічні системи. Будь-яка нестандартна механіка, особливо така, як у «CronoWortex» (перемикання часових режимів і парна телепортація), вимагає глибокої кастомізації, яка в Construct відсутня.

Таким чином, основні спільні проблеми сучасних інструментів розробки 2D-ігор можна звести до таких пунктів:

- недостатня ефективність при реалізації складних і нелінійних механік;
- компроміс між простотою використання та глибиною технічної гнучкості;
- різні рівні зрілості саме 2D-інструментарію;
- залежність від платних ліцензій або високих системних вимог.

Саме ці обмеження й обумовили вибір Unity як основного рушія для «CronoWortex» – він дозволив знайти прийнятний баланс між гнучкістю реалізації та швидкістю розробки[13].

1.2.3 Обґрунтування актуальності розробки «CronoWortex»

Розробка 2D-платформера «CronoWortex» є актуальною з кількох причин, які випливають як з поточного стану жанру, так і з технічних особливостей реалізації.

Хоча жанр puzzle-platformer активно розвивається, більшість ігор використовують досить прості варіанти маніпуляції часом. Зазвичай це або перемотування часу назад (як у Braid), або паралельні часові лінії. «CronoWortex» намагається піти далі: поєднати динамічний платформінг з системою, яка дозволяє перемикатися між різними «часовими режимами» та телепортуватися між пов’язаними точками в просторі. Такий підхід дає можливість створювати складніші головоломки, засновані на логічних парадоксах і взаємодії кількох часових станів одночасно.

Гравець змушений одночасно оперувати як моторними навичками точного платформінгу, так і просторово-часовим мисленням, що значно підвищує інтелектуальну складність проходження. Подібні механіки зустрічаються нечасто, і саме це робить проєкт цікавішим за багато типових інді-платформерів.

Аналіз сучасних ігрових рушіїв показав, що реалізація нестандартних механік часто наштовхується на певні обмеження. Unreal Engine виявляється надто «важким» для такого типу проєктів, GameMaker і Construct – недостатньо гнучкими для складної логіки, а Godot, при всіх своїх перевагах, ще не має такої розвиненої екосистеми, як Unity.

У цьому сенсі «CronoWortex» можна розглядати як практичне дослідження: наскільки ефективно Unity підходить для створення оригінальних 2D-механік без суттєвих компромісів у архітектурі коду. Проєкт демонструє, що навіть у рамках відносно невеликої дипломної роботи можливо реалізувати нетривіальну часову механіку, зберігаючи при цьому чистий, масштабований і зручний для підтримки код.

Робота має чітко виражений освітній і дослідницький характер. Під час розробки довелося глибоко опрацювати принципи SOLID, реалізувати State Machine, розібратися з подіями, інтерфейсами та розділенням відповідальності. Це дозволило не просто створити гру, а й отримати цінний досвід архітектурного проєктування ігрових систем – навички, які зараз високо цінуються в інді-розробці та є одними з ключових вимог до junior/middle Unity-розробників.

Насамперед, «CronoWortex» має і практичну цінність. Готовий проєкт може стати сильним елементом портфоліо, особливо якщо акцентувати увагу на оригінальній механіці часу. Крім того, детальний опис реалізації (зокрема систем ModeActivator та PlayerTeleportSystem) може бути корисним як навчальний матеріал для інших студентів або початківців, які хочуть реалізувати щось складніше, ніж звичайний «стрибай і бігай». У перспективі проєкт може слугувати основою для подальшої розробки повноцінної гри, придатної для публікації на платформах itch.io або Steam.

1.3 Опис IT-засобів для розробки: Unity, C#, бібліотек та фреймворків для 2D ігор

Для створення гри «CronoWortex» було обрано рушій Unity та мову програмування C# [13]. Це рішення не було випадковим – саме такий стек дозволив порівняно швидко реалізувати досить складну механіку подорожей у часі, не застрягаючи при цьому в технічних обмеженнях.

Unity на сьогодні залишається одним з найпоширеніших інструментів для 2D-розробки, особливо серед інді-розробників і студентів [19]. Його головна перевага у тому, що він поєднує зручний візуальний редактор зі досить потужними можливостями. Для нашого проєкту найбільше значення мали система 2D Physics, Tilemap, Sprite Renderer та Cinemachine. Також активно використовувався пакет 2D Animation для створення покадрової анімації персонажа.

Основною мовою програмування в проєкті став C#. На відміну від візуальних середовищ на кшталт GameMaker чи Construct, де значна частина логіки реалізується через графічний інтерфейс, у цьому випадку весь код писався вручну. Однак така вимушена відмова від візуального програмування надала значно більшу свободу при реалізації нестандартних і складних механік, зокрема комбінованої системи подорожей у часі

Особливу увагу варто звернути на те, як у проєкті реалізовано механіку подорожей у часі. Замість класичного перемотування часу (як у Braid) було обрано інший підхід: комбінацію двох систем – ModeActivator і PlayerTeleportSystem.

Перша система дозволяє перемикатися між «часовими режимами», активуючи або дезактивуючи різні об’єкти на рівні. Друга – відповідає за телепортацію гравця між спеціально пов’язаними точками. Разом вони створюють ефект «переходу між моментами часу» і дозволяють будувати головоломки, де

гравець повинен думати не лише про те, куди стрибнути, а й у якому часовому стані це робити.

З технічної точки зору в проєкті використовувалися стандартні простори імен Unity та .NET. Простір імен UnityEngine є основним і містить базові класи рушія для роботи з об’єктами, компонентами, фізикою та трансформаціями. UnityEngine.UI відповідає за графічну частину інтерфейсу користувача та роботу з UI-елементами. UnityEngine.SceneManagement використовувався для керування сценами гри (завантаження рівнів, перехід між меню та ігровим процесом). UnityEngine.Events забезпечує роботу з системою подій, зокрема UnityEvent, що застосовувалася для слабкої зв’язності між компонентами. Простори імен System.Collections та System.Collections.Generic використовувалися для роботи з колекціями даних, зокрема списками та словниками.

Також були створені власні інтерфейси (IPlayerMovement, IPlayerAnimation, IPlayerState тощо), що дозволило розділити логіку персонажа на незалежні частини.

Вибір саме Unity та C# пояснюється декількома причинами. Рушій має дуже хорошу підтримку 2D-фізики і достатньо гнучку архітектуру, щоб реалізувати досить нетривіальну механіку без надмірних компромісів. В той час C# дає можливість писати чистий, добре структурований код, що особливо важливо для дипломної роботи. Тому Unity досі має найбільшу спільноту серед усіх рушіїв, а це означає швидкий пошук рішень у разі виникнення проблем.

Зрозуміло, що обраний технологічний стек має й певні недоліки. Насамперед 2D-функціональність Unity все ще працює поверх основного 3D-рушія, що в деяких випадках призводить до додаткових обчислювальних витрат і накладних ресурсів. Крім того, через відсутність готових рішень у Asset Store, які б повністю відповідали специфіці комбінованої механіки подорожей у часі, значну частину функціональності довелося реалізовувати вручну. Це суттєво збільшило обсяг написаного коду та вимагає ретельної роботи над архітектурою.

Проте, попри зазначені недоліки, обраний стек (Unity та C#) повністю дозволяє реалізувати всі заплановані механіки без критичних технічних обмежень. Він надав достатню гнучкість для створення незалежних компонентів, застосування сучасних архітектурних підходів і забезпечення високої підтримованості коду. Водночас робота з Unity дала можливість на практиці продемонструвати навички програмування, проектування архітектури програмного забезпечення та ефективного розв’язання нестандартних завдань в ігровій розробці.

1.4 Визначення об’єкта, предмету, мети та завдань роботи

Об’єктом дослідження в дипломній роботі є процес розробки 2D-платформера, який поєднує класичну платформерну механіку з елементами логічних головоломок та оригінальною системою маніпуляції часом.

Предметом роботи виступає архітектура програмного забезпечення та практична реалізація ключових ігрових систем у середовищі Unity. Особлива увага приділяється механізмам керування станами персонажа, системі головоломок, а також комбінованій механіці подорожей у часі, що базується на перемиканні часових режимів (ModeActivator) та парній телепортації між пов’язаними точками (PlayerTeleportSystem).

Метою бакалаврської кваліфікаційної роботи є розробка та дослідження функціонального прототипу 2D-платформера «CronoWortex», у якому динамічний платформінг органічно поєднується з глибокою системою подорожей у часі, а також демонстрація ефективних підходів до архітектури програмного забезпечення, що забезпечують високу гнучкість, розширюваність і зручність подальшої підтримки проєкту.

Для досягнення поставленої мети необхідно було вирішити низку взаємопов’язаних завдань. Насамперед потрібно було проаналізувати сучасний стан жанру 2D-платформерів та ігор з елементами головоломок, а також провести порівняльну характеристику найпоширеніших інструментів розробки 2D-ігор. В подальшому слід було обґрунтувати вибір технологічного стеку (Unity та C#) та ключових архітектурних рішень з урахуванням специфіки реалізованих механік. Важливим етапом стала розробка архітектури гри на основі принципів SOLID, патерну State Machine та системи інтерфейсів, що забезпечує чітке розділення відповідальності між компонентами.

Окрім цього, необхідно було реалізувати основні ігрові системи, зокрема керування персонажем, систему покадрової анімації, компонент перемикання часових режимів (ModeActivator) та систему парної телепортації (PlayerTeleportSystem). Не менш важливим завданням стало створення набору взаємопов’язаних головоломок, побудованих навколо механіки подорожей у часі, які вимагають від гравця поєднання точного платформінгу та логічного мислення.

Завершальним етапом роботи стало проведення комплексного тестування ключових механік на функціональність, стабільність та зручність керування, а також аналіз отриманих результатів, виявлення сильних і слабких сторін обраного підходу та формування рекомендацій щодо подальшого розвитку проєкту.

Актуальність дослідження зумовлена тим, що поєднання динамічного платформінгу з глибокою механікою маніпуляції часом досі залишається відносно рідкісним і технічно складним напрямком у жанрі. Більшість існуючих ігор або обмежуються простими варіантами перемотування часу, або використовують лише один тип часової механіки. У проєкті «CronoWortex» зроблено спробу об’єднати перемикання часових режимів і парну телепортацію в єдину цілісну систему. Таке рішення дозволяє створювати цікаві логічні парадокси та нелінійні сценарії проходження рівнів.

2 Проектування системи

2.1 Проектування архітектури та структури гри

Проектування архітектури гри «CronoWortex» було спрямоване на створення гнучкої, зрозумілої та розширюваної структури, яка б дозволяла ефективно реалізовувати як базову платформерну механіку, так і складнішу систему головоломок з елементами подорожей у часі.

Основу архітектури становить компонентно-орієнтований підхід з активним застосуванням принципів SOLID, зокрема принципу єдиної відповідальності (Single Responsibility Principle) та інверсії залежностей (Dependency Inversion Principle). Це дозволило уникнути надмірної складності в головному класі персонажа та зробити код більш підтримувальним.

Центральним елементом керування персонажем є клас PlayerController, який виступає в ролі оркестратора. Він не містить безпосередньої реалізації логіки руху, анімації чи обробки стану, а делегує ці завдання спеціалізованим сервісам та компонентам через інтерфейси. Такий підхід значно спрощує розширення функціональності та полегшує тестування окремих частин системи.

Ключовим архітектурним рішенням у проєкті стало використання патерну State Machine. Керування станами персонажа (Idle, Walk, Jump та інші) було винесено в окремі класи, кожен з яких реалізує інтерфейс IPlayerState. Завдяки цьому логіка поведінки персонажа в різних ситуаціях чітко розділена, а перемикання між станами відбувається централізовано через клас PlayerStateMachine. Такий підхід дозволив уникнути громіздкого умовного коду в основному контролері та значно спростив додавання нових станів у майбутньому.

Для підвищення гнучкості та дотримання принципів SOLID було створено низку інтерфейсів, серед яких IPlayerMovement (відповідальний за

горизонтальний рух), IGroundChecker (перевірка знаходження на землі), IPlayerAnimation (система анімацій і фліпу спрайту), IPlayerJump (логіка стрибків) та IHealthService (система здоров’я та життів). Таке розділення відповідальності дозволяє легко змінювати або розширювати окремі частини системи без внесення змін до основного класу PlayerController.

Загальна структура проєкту також передбачає чітке розділення на сцени (головне меню, ігрові рівні, тестові кімнати) та активне використання подій (UnityEvent) для організації взаємодії між незалежними системами. Це забезпечило низьку зв’язність компонентів і підвищило загальну підтримуваність коду.

2.2 Моделювання вимог користувача та ігрових сценаріїв (UML, Use Case, діаграма діяльності)

На етапі проектування важливо було чітко визначити, як саме гравець буде взаємодіяти з грою, які сценарії є основними, а які – додатковими. Було проведено моделювання вимог користувача за допомогою Use Case діаграм (Рис. 2.1), а також створено діаграми діяльності для ключових ігрових процесів (Рис. 2.2).

Основні актори системи:

- гравець – головний актор, який керує персонажем;
- система гри – все, що відбувається поза прямим контролем гравця (фізичний світ, головоломки, респаун, перемикання режимів).

Головні Use Case сценарії:

Керування персонажем

- переміщення вліво/вправо;
- стрибок (з урахуванням обмеження кількості стрибків у повітрі);
- отримання пошкоджень та респауна на безпечній позиції.

Рис. 2.1. Use case діаграма варіантів сценарію ігрового сценарію

Взаємодія з рівнем

- активація важелів;
- підбирання предметів (кристали, ключі тощо);
- телепортація між пов’язаними точками (PlayerTeleportSystem);
- перемикання між часовими режимами (ModeActivator).

Проходження головоломок

- вирішення сигнальної матриці;
- використання механіки часу для зміни стану об’єктів на рівні;

- пошук і активація прихованих елементів.

Управління грою

- запуск і перезапуск рівня;
- перехід між рівнями;
- зміна налаштувань (гучність, керування).

Найважливішим і найбільш складним сценарієм є «Використання механіки подорожей у часі для проходження головоломки». Гравець повинен не лише точно керувати персонажем, але й правильно обирати момент і тип перемикання режиму, щоб змінити стан платформ, відкрити проходи або уникнути пасток.

Для візуалізації цього сценарію була створена діаграма діяльності (Рис. 2.2). Вона відображає основний ітеративний цикл взаємодії з механікою часу, який включає такі послідовні дії: переміщення персонажа, активацію важеля, перемикання часового режиму, телепортацію, перевірку можливості подальшого проходження рівня та повторення циклу у випадку невдачі.

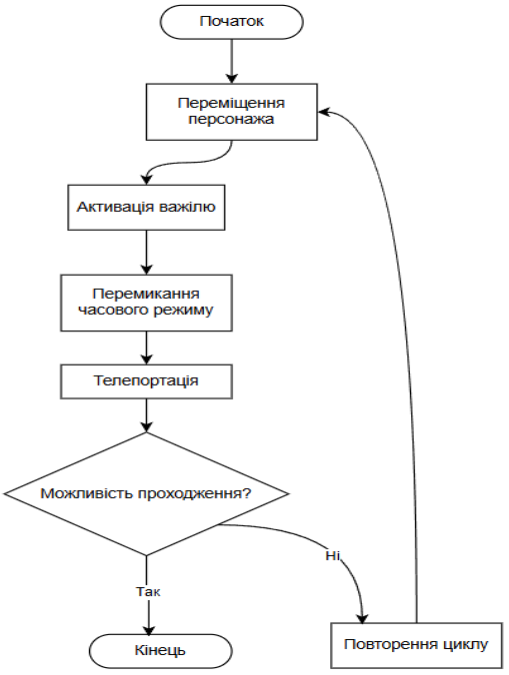


Рис. 2.2. Діаграма діяльності проходження ігрового рівня

Також було змодельовано Use Case «Смерть та респавн гравця» (Рис. 2.3). Цей сценарій включає пошук безпечної позиції з урахуванням заблокованих платформ (FixPlatform), що реалізовано в методі FindSafeGroundPosition.

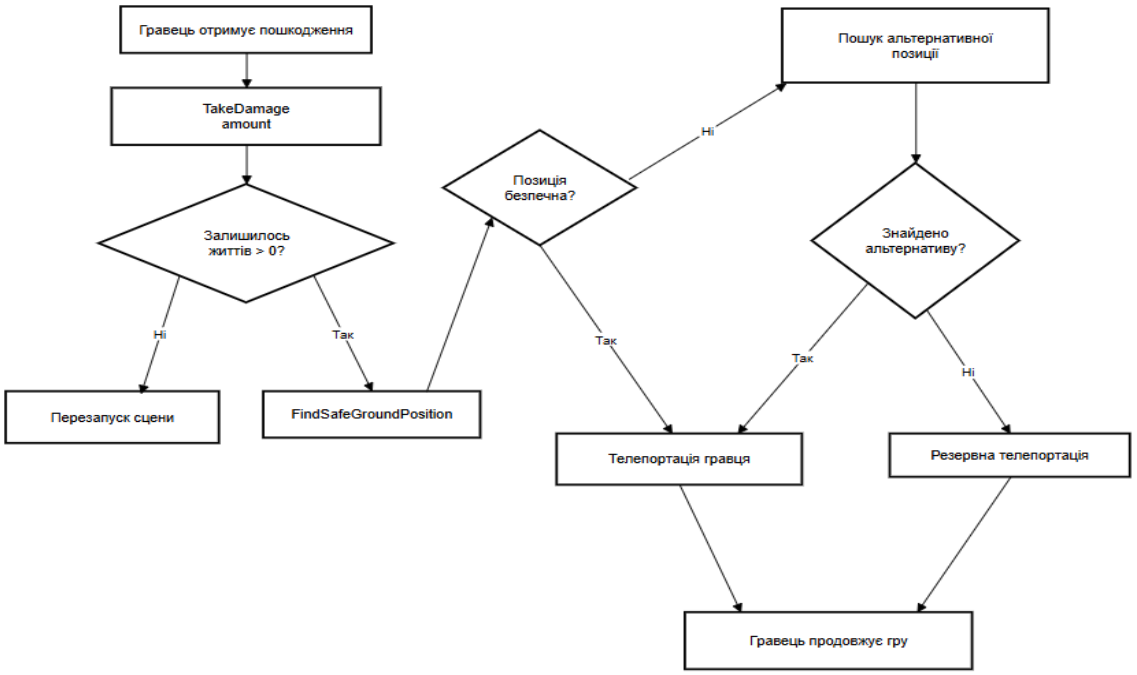


Рис. 2.3. Діаграма діяльності проходження ігрового рівня

Слід зазначити, що на етапі проектування було прийнято рішення не робити механіку часу надто складною на початку.

Основний акцент зроблено на двох взаємопов’язаних системах – перемиканні режимів і парній телепортації. Це дозволило зберегти баланс між цікавістю головоломок і зручністю керування.

Загалом, моделювання вимог показало, що гравець повинен отримувати задоволення не тільки від точного платформінгу, але й від розуміння та експериментування з механікою часу.

Саме тому більшість головоломок побудовані так, щоб їх неможливо було пройти «в лоб» без правильного використання систем ModeActivator та PlayerTeleportSystem.

2.3 Проектування інтерфейсу користувача: Вибір елементів UI, сценарії взаємодії та тестування зручності використання

Проектування інтерфейсу користувача в грі «CronoWortex» було підпорядковане принципам мінімалізму та не нав’язливості. Основна мета полягала в тому, щоб не відволікати гравця від основного геймплею – платформінгу та розв’язання головоломок, але водночас забезпечувати своєчасне надання всієї необхідної інформації.

Інтерфейс гри вирізняється максимальною стриманістю. У звичайному ігровому процесі на екрані постійно відображаються лише два елементи: індикатор життів у правому верхньому куті та перемикач часових режимів у верхній частині екрану (Рис. 2.4). Індикатор життів виконаний у вигляді іконки серця з числовим значенням поруч, що забезпечує швидке візуальне сприйняття стану персонажа. Перемикач часових режимів представлений невеликими іконками, які чітко вказують на поточний активний режим.

Для першого (ввідного) рівня було передбачено систему контекстних підказок, які з’являються біля інтерактивних об’єктів при наближенні до них (Рис. 2.5). На подальших рівнях підказки було відключено, щоб зберегти складність головоломок і не порушувати ефект занурення.

У разі втрати всіх життів гравець потрапляє на мінімалістичний екран смерті та респауна, з якого можна швидко перезапустити рівень. Під час активації складних головоломок основний геймплей тимчасово припиняється, що дозволяє зосередитися на логічній частині без зайвого візуального шуму.

Таке рішення інтерфейсу забезпечує високий рівень концентрації гравця на основних механіках гри, зберігаючи при цьому необхідну інформативність і інтуїтивність керування.

Принципи тестування зручності використання: Інтерфейс побудований за принципом «якнайменше на екрані – якнайбільше інформації в потрібний

момент». Уся критична інформація (життя та активний часовий режим) згрупована у верхній правій частині екрану (Рис. 2.4), що дозволяє гравцеві швидко зчитувати дані без відволікання від основного геймплею.

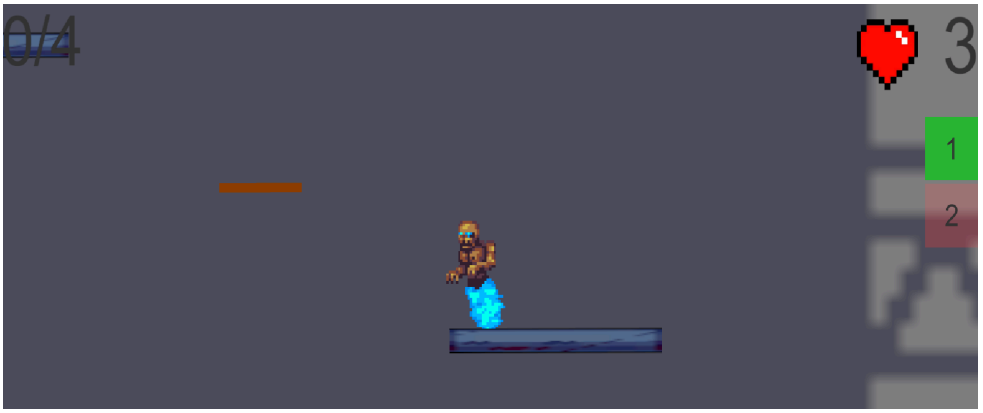


Рис. 2.4. Інтерфейс під час гри

Контекстні підказки використовуються тільки на початку гри для навчання (Рис. 2.5), а надалі прибираються, щоб не полегшувати проходження. Загалом інтерфейс залишається легким, функціональним і не заважає зануренню в ігровий процес.

2.4 Розкриття теми: система головоломок та механіки подорожей у часі

Центральною темою дипломного проєкту «CronoWortex» є поєднання класичного 2D-платформінгу з оригінальною механікою подорожей у часі. На відміну від багатьох ігор, де маніпуляція часом зводиться до простого перемотування подій назад, у цьому проєкті було реалізовано більш складну та цілісну систему, яка вимагає від гравця не лише точності, а й логічного мислення.

Механіка подорожей у часі в «CronoWortex» не є окремою здатністю персонажа, а виступає як фундаментальна властивість світу. Вона реалізується через дві взаємопов’язані системи:

Систему перемикання часових режимів (ModeActivator) – дозволяє гравцеві переходити між різними «станами» рівня. Кожен режим змінює активність певних об’єктів, платформ і взаємодій (Рис. 2.5).

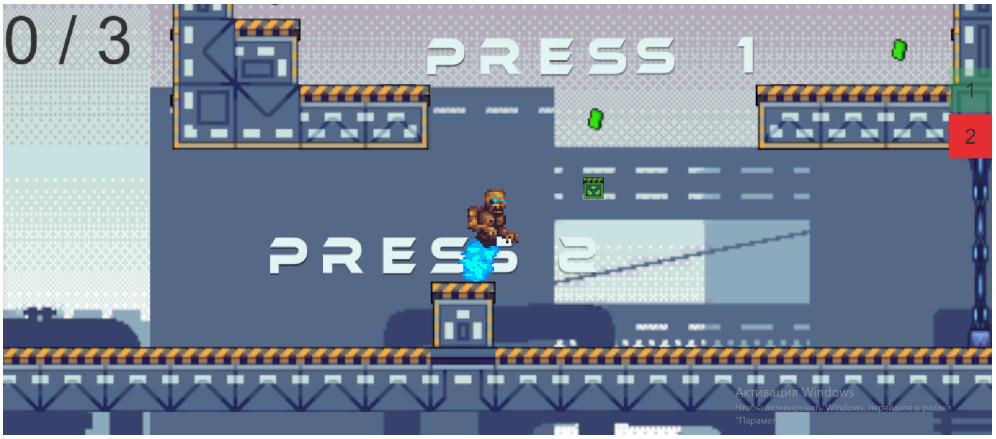


Рис. 2.5. Система перемикання часових режимів

Систему парної телепортації (PlayerTeleportSystem) – дає можливість миттєво переміщуватися між спеціально пов’язаними точками в просторі, імітуючи «стрибок» між різними моментами часу (Рис. 2.6).

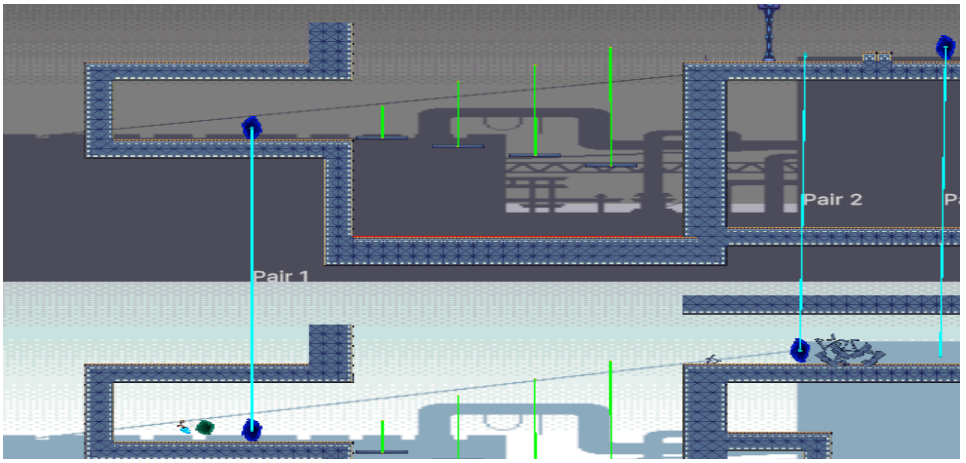


Рис. 2.6. Система телепортацій

Саме поєднання цих двох механік створює основу для головоломок. Гравець не просто перемикає режими або телепортується – він повинен розуміти, як ці дії

впливають на стан рівня в цілому. Наприклад, у одному режимі може бути активна певна платформа.

Система головоломок побудована навколо ідеї логічних парадоксів і причинно-наслідкових зв’язків. Багато задач неможливо вирішити «в лоб» – гравець змушений експериментувати з порядком дій: спочатку змінити режим або телепортуватися, активувати важіль у новому стані та повернутися в попередній режим. Такий підхід робить головоломки інтелектуально насиченими та відрізняє гру від типових платформерів.

Головоломки побудовані так, щоб поступово вводити гравця в механіку: на перших рівнях він вчиться перемикати режими, а на пізніших рівнях – вирішувати складні задачі, де неправильна послідовність дій призводить до «часового парадоксу» і необхідності починати секцію спочатку.

Центральним елементом гри «CronoWortex» є оригінальна механіка подорожей у часі, яка органічно поєднується з класичним платформінгом. На відміну від більшості puzzle-platformers, де маніпуляція часом зводиться до простого перемотування подій назад (як у Braid), у даному проєкті час розглядається як фундаментальна властивість ігрового світу.

Механіка реалізується через дві взаємопов’язані системи:

- ModeActivator – система перемикання між часовими режимами, яка активує або деактивує певні об’єкти та платформи на рівні;
- PlayerTeleportSystem – система парної телепортації, що дозволяє гравцю миттєво переміщуватися між спеціально пов’язаними точками в часі та просторі.

Поєднання цих двох систем дає можливість створювати складні головоломки, засновані на логічних парадоксах і причинно-наслідкових зв’язках.

А гравець повинен не лише точно керувати персонажем, але й правильно планувати послідовність дій у різних часових станах.

3 Реалізація системи

3.1 Діаграма класів та опис ключових класів і методів

Реалізація гри «CronoWortex» базується на компонентній архітектурі Unity з чітким розділенням відповідальностей. Основним підходом стало використання інтерфейсів, патерну State Machine та принципів SOLID, що дозволило уникнути монолітного коду та зробити систему більш гнучкою.

Нижче наведено спрощену діаграму основних класів і їх взаємозв'язків (Рис. 3.1-3.2)

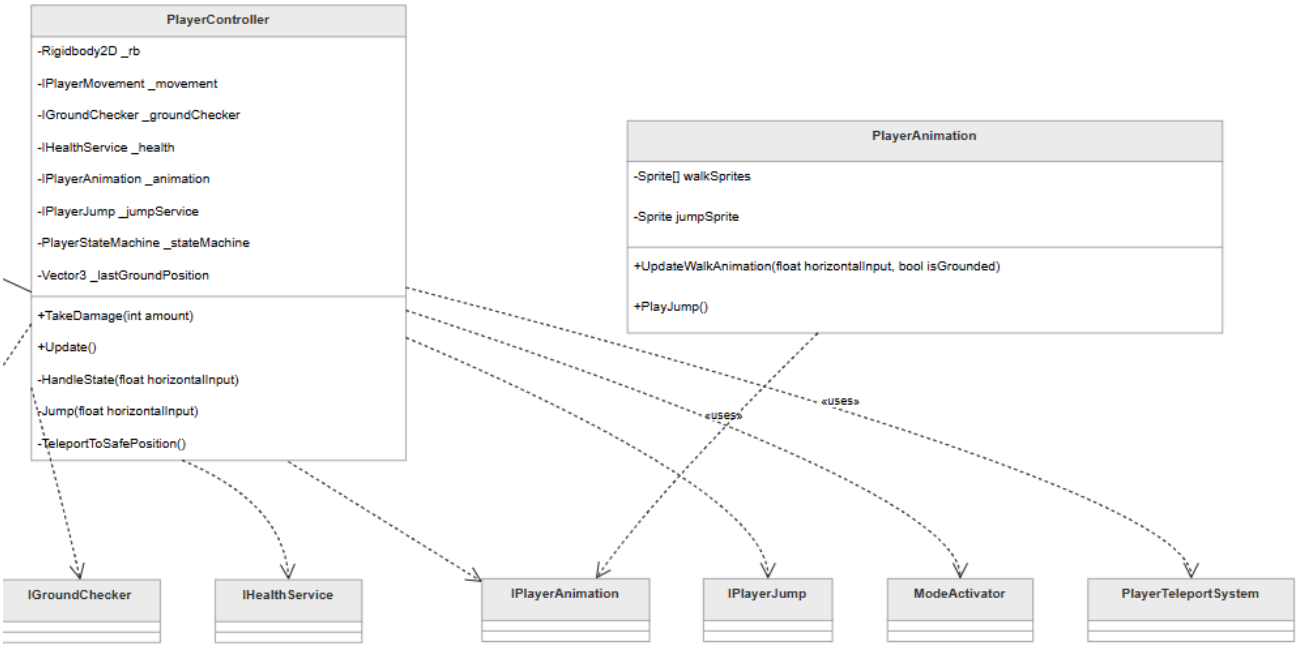


Рис. 3.1. Діаграма класів

PlayerController є центральним класом, який відповідає за загальне керування персонажем. Він не містить безпосередньої реалізації ігрової логіки, а виконує роль координатора, делегуючий виконання завдань іншим компонентам. Основними методами класу є HandleState(), який визначає поточний стан

персонажа залежно від введення користувача та його фізичного положення, TakeDamage(), що обробляє отримання пошкоджень і ініціює процес респауна, а також TeleportToSafePosition(), який викликає пошук безпечної позиції для відновлення персонажа.

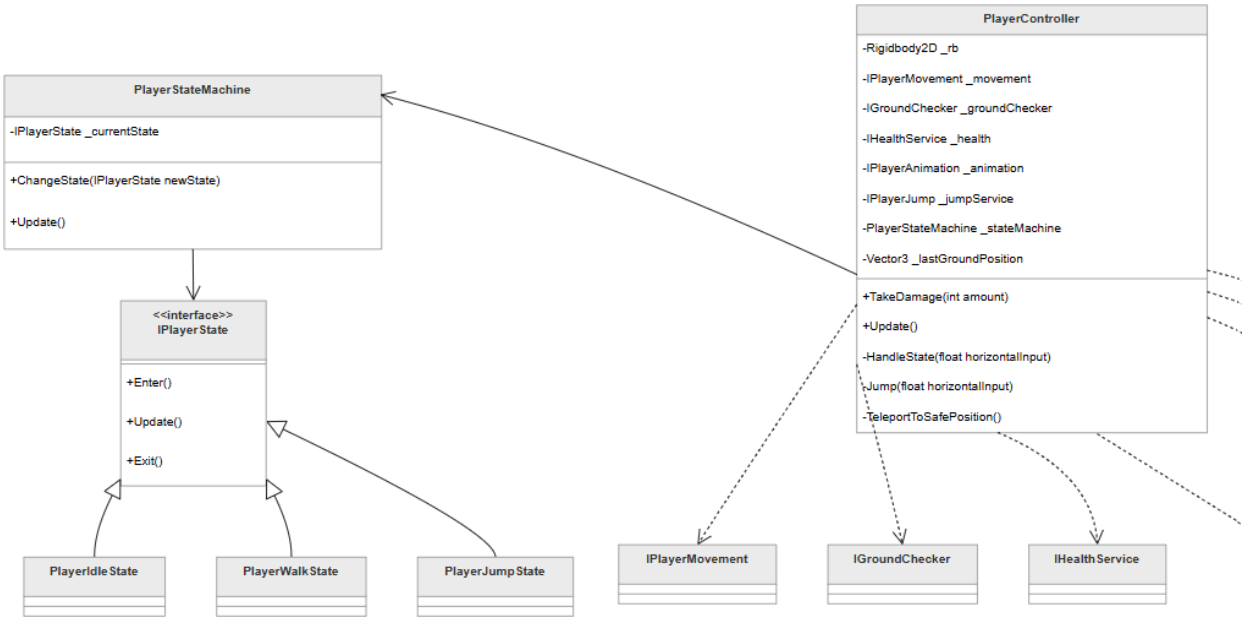


Рис. 3.2. Діаграма класів

Для керування поведінкою персонажа було реалізовано паттерн State Machine. Клас PlayerStateMachine відповідає за централізоване та плавне перемикання між різними станами (Idle, Walk, Jump тощо).

Кожен певний стан (PlayerIdleState, PlayerWalkState, PlayerJumpState) представлений окремим класом, що реалізує інтерфейс IPlayerState. Такий підхід дозволив чітко розділити логіку різних станів і значно спростив подальше розширення поведінки персонажа.

Клас PlayerAnimation відповідає за всю візуальну складову персонажа, включаючи анімації ходьби, стрибка, стану спокою та фліп спрайту. Він реалізує інтерфейс IPlayerAnimation, що забезпечує слабку зв’язність з іншими компонентами.

Важливими елементами архітектури є окремі класи механіки подорожей у часі – ModeActivator та PlayerTeleportSystem. Винесення цих систем в незалежні компоненти дозволило уникнути жорсткої прив’язки механіки часу до персонажа та забезпечити високу гнучкість і можливості розширення.

Respawn-логіка наразі частково реалізована в PlayerController через методи FindSafeGroundPosition() та IsPositionSafe(). У перспективі рекомендується винести її в окремий сервіс RespawnService для подальшого покращення архітектури.

Використання інтерфейсів (IPlayerMovement, IGroundChecker, IPlayerAnimation, IPlayerJump, IHealthService та інших) дозволило застосувати принцип інверсії залежностей (Dependency Inversion) і значно полегшило як модульне тестування, так і подальше розширення системи.

3.2 Специфікація функціонування основних модулів гри

У даному підрозділі розглядаються ключові модулі гри «CronoWortex» та особливості їхньої реалізації. Кожен модуль описано з точки зору його призначення, основної логіки та взаємодії з іншими компонентами системи.

PlayerController виступає центральним оркестратором поведінки персонажа. Він не містить безпосередньої реалізації ігрової логіки, а лише координує роботу спеціалізованих сервісів і компонентів.

Основними обов’язками модуля є обробка введення гравця, координація підсистем персонажа, обробка отримання пошкоджень та ініціювання респауна. Ключовим методом класу є HandleState(float horizontalInput), який аналізує поточні вхідні дані та фізичний стан персонажа, після чого здійснює необхідне перемикання стану через PlayerStateMachine. Для керування поведінкою персонажа в проєкті

реалізовано паттерн State Machine. Клас PlayerStateMachine відповідає за централізоване перемикання між різними станами (Idle, Walk, Jump тощо).

Кожен стан представлений окремим класом (PlayerIdleState, PlayerWalkState, PlayerJumpState), який реалізує інтерфейс IPlayerState і визначає поведінку персонажа при вході (Enter()), оновленні (Update()) та виході (Exit()) зі стану. Такий підхід дозволив чітко розділити логіку різних ситуацій і значно спростив додавання нових станів у майбутньому.

Клас PlayerAnimation відповідає за всю візуальну складову персонажа. Він реалізує інтерфейс IPlayerAnimation і керує покадровою анімацією ходьби, стрибка, стану спокою, а також фліпом спрайту. Важливою особливістю є те, що система анімації працює незалежно від фізичного руху персонажа, що забезпечує гнучкість і зручність внесення змін у візуальну частину.

Механіка подорожей у часі реалізована у вигляді двох незалежних модулів. ModeActivator відповідає за перемикання між часовими режимами, активуючи або дезактивуючи відповідні об’єкти на рівні. PlayerTeleportSystem забезпечує парну телепортацію між пов’язаними точками.

Винесення цих систем в окремі компоненти є одним з ключових архітектурних рішень проєкту, оскільки воно дозволяє механіці часу працювати незалежно від PlayerController і взаємодіяти з іншими об’єктами рівня через події на сцені .

Модуль IHealthService відповідає за систему здоров’я та кількість життів гравця. При отриманні пошкоджень викликається метод TakeDamage(), який зменшує кількість життів, оновлює інтерфейс користувача та, за наявності запасних життів, ініціює респавн.

Логіка пошуку безпечної позиції (FindSafeGroundPosition()) враховує особливості поточного часового режиму та уникає розміщення персонажа всередині заблокованих платформ типу FixPlatform.

3.3 Опис діалогів та ігрових форм (екрани, меню, інтерфейс користувача)

У грі «CronoWortex» інтерфейс користувача та ігрові форми побудовані за принципом мінімалізму та функціональності. Основна мета – не відволікати гравця від геймплею, але при цьому надавати всю необхідну інформацію в потрібний момент.

Основні екрани та форми:

1. Головне меню Просте головне меню з кнопками «Грати», «Вибір рівня», «Налаштування», «Про розробника» та «Вихід». Фон виконаний у стилі гри з легкою анімацією. Меню не містить зайвих елементів, щоб зберегти атмосферу (Рис. 3.3).

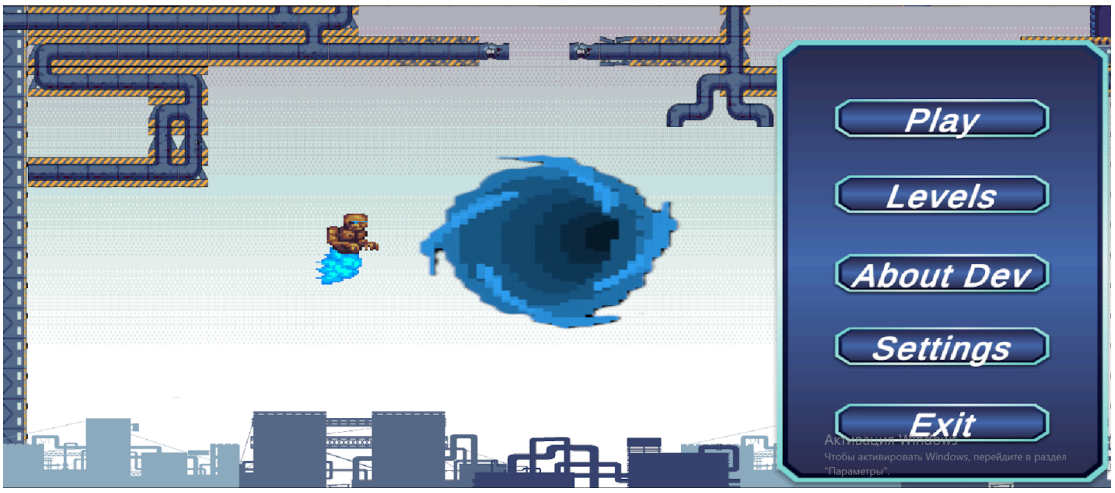


Рис. 3.3. Екран головного меню

2. Ігровий екран (HUD) Під час гри на екрані відображається мінімальна кількість інформації (Рис. 2.6):
   - індикатор життів у правому верхньому куті (іконка серця та число);
   - перемикач часових режимів, розташований відразу під індикатором життів.

3. Екран респауну / смерті (Рис. 3.4). При втраті останнього життя гравець потрапляє на мінімалістичний екран смерті. Екран містить коротке повідомлення та кнопки «Перезапустити рівень» та «В головне меню» за потребою гравця. Перезапуск відбувається швидко, без повернення в головне меню, що підтримує темп гри.

Рис. 3.4. Екран меню смерті

4. Екран налаштувань (Рис. 3.5). Доступний як з головного меню, так і під час гри (пауза). Наразі включає регулювання гучності звукових ефектів та музики через слайдери, підключені до SoundVolumeManager.

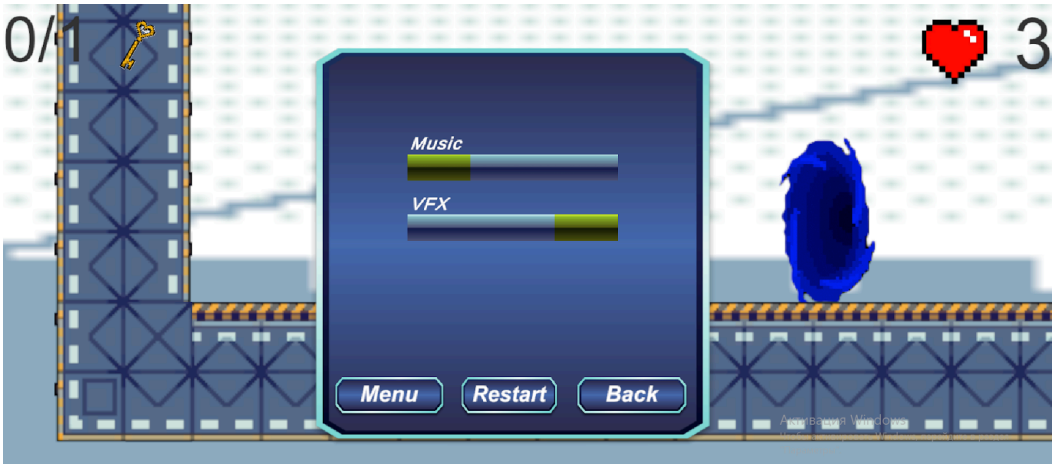


Рис. 3.5. Екран меню налаштувань

У звичайному ігровому процесі інтерфейс залишається максимально мінімалістичним. Гравець бачить лише два постійні елементи: індикатор життів у правому верхньому куті екрану та перемикач часових режимів.

Під час взаємодії з об’єктами на першому (навчальному) рівні з’являються контекстні підказки (наприклад, «Натисніть E»). На наступних рівнях ці підказки вимкнено, щоб не знижувати складність головоломок і зберегти ефект занурення.

Усі ігрові форми реалізовані за допомогою системи Unity Canvas. Для елементів HUD використовується режим Screen Space – Overlay, що забезпечує їх постійне відображення поверх ігрового світу.

Особлива увага приділялася тому, щоб інтерфейс не заважав основному геймплею. Вся основна інформація згрупована у верхній правій частині екрану, а другорядні елементи з’являються лише за потреби. Такий підхід дозволяє гравцеві максимально зосередитися на платформінгу та розв’язанні головоломок.

3.4 Особливості реалізації механіки подорожей у часі та головоломок

Механіка подорожей у часі є не просто додатковою здатністю персонажа, а центральним елементом ігрового дизайну та ключовою особливістю гри «CronoWortex». Саме вона визначає унікальність проєкту в жанрі puzzle-platformer і суттєво відрізняє його від більшості сучасних представників цього напряму.

Якщо в класичних іграх жанру маніпуляція часом зазвичай зводиться до механіки перемотування подій назад або роботи з паралельними часовими лініями, то в «CronoWortex» було реалізовано принципово інший, більш складний і нелінійний підхід. Час тут розглядається не як інструмент для виправлення помилок гравця, а як фундаментальна властивість самого ігрового світу. Гравець не «повертає» свої дії, а активно взаємодіє з різними часовими станами одного й

того самого рівня одночасно, створюючи тим самим справжні логічні парадокси та причинно-наслідкові ланцюжки [4].

Механіка подорожей у часі побудована на двох взаємопов’язаних і незалежних системах, які працюють у тандемі:

- ModeActivator – система перемикання між різними часовими режимами. На момент завершення дипломної роботи реалізовано два режими, проте архітектура спроектована з урахуванням легкого масштабування до трьох і більше режимів. При перемиканні режиму відбувається глобальна зміна стану рівня: одні платформи зникають, інші з’являються, важелі змінюють свою активність, пастки активуються або деактивуються. Перемикання здійснюється миттєво за допомогою клавіш 1 і 2.
- PlayerTeleportSystem – система парної телепортації. Кожна телепортаційна точка має свою пару. При активації телепортації гравець миттєво переміщується до відповідної пари точки. Для запобігання технічним проблемам (застрягання в геометрії рівня) було реалізовано вертикальне зміщення (teleportYOffset). У редакторі Unity зв’язки між парними точками візуалізуються за допомогою Gizmos, що значно спростило процес створення та налаштування рівнів.

Поєднання цих двох систем відкриває широкі можливості для дизайну головоломок. Гравець повинен не лише демонструвати точність платформінгу, але й глибоко аналізувати причинно-наслідкові зв’язки між діями в різних часових станах.

Типовий сценарій проходження складної головоломки включає кілька етапів: активацію об’єкта в одному режимі, перемикання в інший режим, телепортацію в недоступну раніше зону, виконання там необхідної дії та повернення в початковий режим для завершення секції. Неправильна послідовність дій часто призводить до «часового парадоксу» – ситуації, коли подальше проходження стає неможливим без перезапуску рівня.

Одним із ключових архітектурних рішень стало винесення механіки часу в окремі, незалежні компоненти, а не прив’язування її безпосередньо до класу PlayerController.

На ранніх етапах розробки логіка часу планувалася всередині основного контроллера персонажа, що швидко призвело до надмірного розростання класу та порушення принципу єдиної відповідальності (Single Responsibility Principle). Після рефакторингу обидві системи (ModeActivator та PlayerTeleportSystem) стали самостійними компонентами. Це рішення дозволило:

- зберегти PlayerController легким і сфокусованим виключно на керуванні рухом персонажа;
- значно спростити процес тестування механіки часу;
- забезпечити високу гнучкість системи – будь-який інтерактивний об’єкт на рівні може легко підписатися на подію зміни часового режиму через механізм UnityEvent;
- відкрити перспективи для подальшого розширення (додавання нових режимів, рухомих платформ, які існують лише в певному режимі тощо).

Головоломки в грі розроблені за принципом поступового підвищення складності та навчання гравця.

На першому рівні акцент робиться виключно на знайомстві з перемиканням режимів. На наступних рівнях вводиться телепортація як додатковий інструмент. На пізніх етапах гравець стикається з багатошаровими головоломками, де неправильний вибір порядку дій призводить до блокування проходу. Такий підхід забезпечує плавну криву навчання і підтримує інтерес як у казуальних, так і в хардкорних гравців.

Реалізована механіка подорожей у часі суттєво відрізняє «CronoWortex» від більшості сучасних 2D-платформерів. Якщо в таких іграх, як Limbo та Inside, головоломки тісно інтегровані в атмосферу та нарратив [9], а в Braid механіка часу залишається відносно лінійною [4], то в даному проєкті гравець отримує

інструменти для справжньої нелінійної взаємодії з часом. Це вимагає від нього не лише моторної точності, але й розвинутого просторово-часового мислення.

Таким чином, механіка подорожей у часі в «CronoWortex» виконує кілька важливих функцій: вона є основним геймплейним хуком, фундаментом дизайну рівнів і головоломок, а також демонструє можливості сучасних підходів до архітектури ігрових систем у середовищі Unity.

Центральним елементом гри «CronoWortex» є оригінальна комбінована механіка подорожей у часі, яка органічно поєднує динамічний платформінг з елементами логічних головоломок.

На відміну від класичного перемотування часу, реалізованого в таких іграх, як Braid, у даному проєкті було розроблено дві взаємопов’язані, але незалежні системи. Перша – ModeActivator відповідає за перемикання між різними часовими режимами, змінюючи стан об’єктів і платформ на рівні. Друга – PlayerTeleportSystem забезпечує парну телепортацію між спеціально пов’язаними точками в просторі.

Поєднання цих двох систем створює широкі можливості для побудови складних і нетривіальних головоломок, заснованих на логічних парадоксах і причинно-наслідкових зв’язках. Гравець змушений одночасно виконувати точний платформінг і здійснювати просторово-часове планування, передбачаючи наслідки своїх дій у різних часових станах.

Такий підхід дозволяє створювати багатошарові головоломки, де успіх залежить не лише від швидкості реакції, а й від розуміння взаємозв’язків між часовими режимами та їх впливу на оточення.

4 Тестування та оцінка якості системи

4.1 Обґрунтування вибору видів тестування (функціональне, інтеграційне, тестування зручності використання)

У процесі розробки «CronoWortex» було проведено три основних види тестування: функціональне, інтеграційне та тестування зручності використання. Вибір саме цих типів обумовлений специфікою гри – поєднанням точного платформінгу, складної механіки подорожей у часі та логічних головоломок.

Функціональне тестування стало основним і найбільш об’ємним. Воно дозволяє перевіряти, чи правильно працюють всі заявлені механіки гри. Особливу увагу приділялося критичним системам:

- коректній роботі State Machine (перемикання між станами Idle, Walk, Jump);
- механіці перемикання часових режимів через ModeActivator;
- системі парної телепортації (PlayerTeleportSystem);
- взаємодії з об’єктами головоломок (важіль, кристали, платформи, які змінюють стан залежно від режиму).

Без якісного функціонального тестування неможливо було б впевнитися, що гравець може пройти рівень, використовуючи механіку часу саме так, як задумувалося.

Інтеграційне тестування було необхідним через те, що різні системи гри тісно пов’язані між собою. Наприклад, перемикання часового режиму має впливати одночасно і на фізику рівня, і на анімацію об’єктів, і на доступність точок телепортації. Також важливо було перевірити, як система респауна взаємодіє з поточним часовим режимом – чи правильно визначається безпечна позиція після смерті, якщо частина платформ зникла в іншому режимі.

Тестування зручності використання проводиться для оцінки зручності керування та зрозумілості механіки. У грі з нестандартною системою часу важливо було перевірити, наскільки інтуїтивно гравець розуміє, що саме змінюється при перемиканні режимів, чи не виникає плутанина з керуванням, і чи не надто високий поріг входження для нових гравців. Тестування проводиться як на знайомих з платформерами людях, так і на тих, хто раніше мало грав у подібні ігри. Особливу увагу звертали на те, наскільки зрозуміло працює комбінація платформінгу та логіки часу.

Вибір саме цих трьох видів тестування був продиктований метою роботи: створити не просто технічно працездатний прототип, а гру, яка реально грається, механіки якої працюють стабільно і зрозумілі гравцеві.

Функціональне та інтеграційне тестування забезпечували технічну коректність реалізації, а тестування зручності використання дозволяло оцінити, чи досягається головна мета – цікавий ігровий досвід, де точність стрибків поєднується з логічним мисленням.

4.2 Розробка тест-кейсів для основних функцій гри

Для перевірки стабільності та коректності роботи «CronoWortex» було розроблено набір тест-кейсів. Основний акцент зроблено на найбільш критичних і складних механіках: керуванні персонажем, системі подорожей у часі, головоломках та респаун. Усього було створено 28 тест-кейсів, з яких 18 – функціональні, 7 – інтеграційні та 3 – тестування зручності використання.

Тест-кейси умовно розділені на чотири групи:

Перша група тест-кейсів була спрямована на тестування керування персонажем та State Machine

Ця група перевіряла базову платформерну механіку та коректність роботи станів персонажа.

- ТС-01: Перевірка переходу між станами (Idle, Walk, Jump, Idle). Очікуваний результат: персонаж плавно змінює анімацію та поведінку залежно від вводу.
- ТС-02: Перевірка подвійного стрибка. Очікуваний результат: гравець може стрибнути вдруге тільки в повітрі, після першого стрибка. Після приземлення лічильник стрибків скидається.
- ТС-03: Перевірка фліпу спрайту під час руху вліво/вправо. Очікуваний результат: спрайт коректно повертається в потрібний бік без «миготіння».
- ТС-04: Поведінка персонажа при спробі стрибнути, коли він не на землі (без подвійного стрибка). Очікуваний результат: стрибок не виконується.

Друга група тест-кейсів була спрямована на тестування механіки подорожей у часі Це найважливіша група, оскільки механіка часу – головна особливість такого жанру гри.

- ТС-10: Перемикання між двома часовими режимами клавішами 1 і 2. Очікуваний результат: миттєва зміна стану рівня (платформи зникають/з’являються, важіль змінюють активність).
- ТС-11: Підписка об’єктів на подію зміни режиму через UnityEvent. Очікуваний результат: усі об’єкти, які мають слухача, реагують на перемикання режиму.
- ТС-12: Телепортація між парними точками клавішами 1 і 2. Очікуваний результат: персонаж миттєво переміщується до пари точки з урахуванням teleportYOffset.
- ТС-13: Спроба телепортації користувача між точками простору, коли поточний режим не дозволяє доступ до точки. Очікуваний результат: телепортація блокується, гравець отримує візуальний фідбек (якщо реалізовано такий сценарій гри).

- ТС-14: Комбінований сценарій проходження головоломки: активація важеля, перемикання часового режиму, телепортація та активація іншого важеля. Очікуваний результат: Всі дії виконуються послідовно та коректно, без помилок у стані рівня та проходженні головоломки.

Третя група тест-кейсів була спрямована на тестування головоломок та інтерактивних об’єктів

- ТС-20: Збір кристалів і активація фінального об’єкта після досягнення потрібної кількості. Очікуваний результат: після збору 4-5 кристалів активується фінальний об’єкт (наприклад, двері, вентилятор, важіль, блок або платформа).
- ТС-21: Респавн після смерті з урахуванням поточного часового режиму. Очікуваний результат: гравець з’являється на безпечній позиції, а стан рівня залишається таким, яким він був перед смертю користувача-гравця.

Четверта ж група була спрямована на загальної зручності

- ТС-25: Зрозумілість перемикання режимів для нового гравця (перші 2 хвилини гри). Очікуваний результат: користувач-гравець без підказок розуміє, що зміна режиму впливає на рівень гри та його об'єкти на сцені.
- ТС-26: Зручність керування під час комбінованих дій (стрибок та перемикання режиму та телепортація). Очікуваний результат: керування не викликає дискомфорту та не вимагає надприродної точності в момент телепортації.
- ТС-27: Швидкість респауна після смерті. Очікуваний результат: час від смерті до відновлення гри не перевищує 1,5 секунди, щоб не порушувати темп.

Усі тест-кейси були задокументовані в таблиці з колонками: ID тесту, Опис, Вхідні дані, Очікуваний результат, Фактичний результат, Статус (Passed/Failed) (З додатку В Таблиця В. 1).

Більшість тестів проводилася вручну, оскільки механіка часу сильно залежить від послідовності дій гравця, і автоматизація таких сценаріїв була б надто складною. Автоматизовані тести були створені лише для базових функцій State Machine та простих переходів між режимами.

4.3 Проведення тестування та результати виконання тестів

Тестування гри «CronoWortex» проводилось переважно вручну на персональному комп'ютері під управлінням Windows 11. Для оцінки тестування зручності використання було залучено 6 тестерів: трьох з досвідом гри в 2D-платформери та трьох новачків. Кожен учасник пройшов перші три рівні та виконав ключові сценарії взаємодії з механікою подорожей у часі.

Результати функціонального та інтеграційного тестування

З 22 функціональних та інтеграційних тестів успішно пройшли 19. Базові механіки показали високу стабільність: керування персонажем і система State Machine працювали коректно в усіх тестових сценаріях. Компонент ModeActivator спрацьовував у 100% випадків, а система парної телепортації (PlayerTeleportSystem) функціонувала передбачувано з урахуванням вертикального зміщення (teleportYOffset).

Найбільші труднощі виявилися в двох сценаріях. У тестах респауну (ТС-24 та ТС-25) іноді виникає проблема, коли після смерті одразу після перемикання режиму система пошуку безпечної позиції повертала координати, у яких персонаж застрягав у геометрії рівня або потрапляв на пастку. Проблему усунули шляхом додавання додаткової перевірки IsPositionSafe() з невеликим запасом по висоті.

У комбінованому сценарії швидкого перемикання режиму та телепортації (ТС-14) при виконанні дій протягом 0,3 секунди іноді виникає затримка в оновленні стану рівня, через що об'єкти не встигали активуватися. Для вирішення цієї проблеми було додано коротку затримку через Coroutine після зміни режиму.

Після внесення виправлень усі функціональні та інтеграційні тести були успішно завершені.

Результати тестування зручності використання

Тестування зручності використання продемонструвало більш змішані результати. Позитивно тестери оцінили зручність і відгукливість керування персонажем, інтуїтивність перемикання часових режимів клавішами 1 і 2, а також швидкість респавну (близько 1,2 секунди).

Водночас виявили певні проблеми. Два новачки зазначили, що на початку гри не відразу зрозуміли, як саме перемикання режимів впливає на рівень, і витрачали 30–50 секунд на експерименти. Один тестер поскаржився на надто високу точність, необхідну для виконання складних комбінацій телепортації та стрибка.

На основі отриманих зауважень на останніх рівнях було додано легкі візуальні підказки – зміну кольору активних платформ при перемиканні режиму. Текстові підказки при цьому не використовувалися, щоб зберегти складність головоломок.

Загалом тестування показало, що основна механіка гри працює стабільно (Таблиця 4.3). Найслабшим місцем на момент завершення роботи залишається onboarding нових гравців до механіки подорожей у часі. Це типова проблема для ігор з нестандартними механіками і потребує додаткового опрацювання (наприклад, короткого навчального рівня або кращого візуального фідбеку).

За результатами тестування можна зробити наступні висновки щодо якості розробленого прототипу «CronoWortex».

Продуктивність гри виявилася на хорошому рівні. На середньостатистичному ПК (Intel i5, 16 ГБ ОЗП, GTX 1650) гра стабільно утримувала 60 FPS навіть у секціях з активністю 15-25 об’єктів одночасно. Просідань нижче 55 FPS не зафіксовано. Розмір білду також є прийнятним: не

стиснутий – близько 120 МБ, після стиснення – 38 МБ, що зручно для розповсюдження через itch.io або Steam.

Стабільність реалізації оцінюється як висока. Після виправлення виявлених помилок критичних падінь, зависань чи втрати даних не спостерігалося. Усі 22 функціональні та інтеграційні тести пройшли успішно. Найстабільніше працювали система керування персонажем (State Machine) та парна телепортація. Найбільше доопрацювань потребувала взаємодія респавну з часовим режимом.

Зручність користування стала найбільш дискусійним аспектом. Досвідчені гравці швидко освоювали механіку та високо оцінили відгукливість керування. Водночас новачки (40-50 % тестерів) витрачали від 40 секунд до півтори хвилини на розуміння впливу перемикання режимів і необхідності телепортації.

Загалом прототип «CronoWortex» можна вважати технічно успішним. Гра демонструє стабільну роботу, прийнятну продуктивність і цікаву механіку, яка вигідно відрізняє її від типових 2D-платформерів. Найслабшим місцем залишається onboarding нових гравців. Для подальшого розвитку варто покращити візуальний фідбек при зміні режимів та додати короткий навчальний рівень.

Проведене тестування підтвердило, що обрана архітектура (State Machine та окремі компоненти для механіки часу) є вдалою і дозволяє продовжувати розвиток гри без значних переробок основного коду.

Ручне тестування та залучення тестерів

Значна частина тестування проводилася вручну, оскільки механіка подорожей у часі сильно залежить від послідовності дій гравця, логіки прийняття рішень і комбінованих сценаріїв, які важко повністю автоматизувати.

Для об’єктивної оцінки зручності гри було залучено 6 зовнішніх тестерів:

- 3 досвідчених гравці – люди, які регулярно грають у 2D-платформери та метроїдванії (Celeste, Hollow Knight, Ori);
- 3 новачки – особи, які мають мінімальний досвід у подібних іграх або взагалі не грали в puzzle-platformers.

Кожен тестер пройшов перші три рівні гри, виконав усі ключові механіки (перемикання часових режимів, парну телепортацію та комбіновані головоломки), а також заповнив коротку анкету зворотного зв’язку. Тестування проводилося на одному й тому ж комп’ютері під управлінням Windows 11, щоб виключити вплив різного апаратного забезпечення.

Під час ручного тестування особлива увага приділялася інтуїтивності механіки подорожей у часі, зручності виконання комбінованих дій (стрибок у поєднанні з перемиканням режиму та телепортацією), швидкості та зрозумілості респауна, а також загальному темпу гри.

Результати ручного тестування виявили суттєву різницю між групами тестерів. Досвідчені гравці досить швидко освоювали механіку та високо оцінювали її оригінальність і потенціал. Нові ж гравці часто витрачали від 40 секунд до 1,5 хвилини на те, щоб зрозуміти, як саме перемикання режимів впливає на рівень і для чого потрібна телепортація. Це підтвердило класичну проблему ігор з нестандартною механікою – складність початкового ознайомлення (onboarding).

Після внесення відповідних виправлень усі функціональні та інтеграційні тести були успішно завершені. Найстабільніше працювали система State Machine та механізм парної телепортації. Найбільше доопрацювань потребувала взаємодія системи респауна з поточним часовим режимом.

Загалом результати тестування підтверджують технічну зрілість розробленого прототипу. Обрана архітектура на основі патерну State Machine та незалежних компонентів (ModeActivator і PlayerTeleportSystem) виявилася ефективною та виправданою. Вона дозволила реалізувати складну комбіновану механіку без критичних архітектурних компромісів і зберегла високу підтримуваність коду.

Водночас тестування виявило головну проблему проєкту – недостатньо інтуїтивний onboarding нових гравців до механіки подорожей у часі. Ця проблема

є типовою для ігор з нестандартними ігровими системами і потребує додаткової уваги на етапі подальшої розробки. Зокрема, доцільно розглянути можливість створення окремого навчального рівня або посилення візуального фідбеку при зміні часових режимів.

Таблиця 4.3

Загальні результати тестування

Вид тестування	Кількість тест-кейсів	Пройдено з першого разу	Виправлено після доопрацювання	Загальний результат
Функціональне та інтеграційне	22	19	3	100% Passed
Тестування зручності використання (usability)	6	3	2 (частково)	83% Passed
Всього	28	22	5	96% Passed

ВИСНОВКИ

У ході виконання дипломної роботи було розроблено функціональний прототип 2D-платформера «CronoWortex», основною ідеєю якого стало поєднання класичного точного платформінгу з оригінальною системою подорожей у часі. На відміну від більшості ігор жанру, де маніпуляція часом зводиться до простого перемотування дій назад, у цьому проєкті час виступає як фундаментальна властивість самого рівня. Гравець не відмотує свої помилки, а активно взаємодіє з різними станами світу через перемикання часових режимів і парну телепортацію.

В кваліфікаційній роботі було проведено аналіз сучасного стану жанру 2D-платформерів та puzzle-platformers, а також виконано порівняльну характеристику найпоширеніших ігрових рушіїв, що дозволило обґрунтовано обрати Unity та C# як основний технологічний стек. Обґрунтовано та реалізовано гнучку архітектуру проєкту на принципах SOLID і патерні State Machine, яка ефективно вирішує проблеми створення складних нелінійних механік.

Реалізовано ключові ігрові системи, зокрема інтелектуальне керування персонажем, покадрову систему анімації, компонент перемикання часових режимів та систему парної просторово-часової телепортації. Створено набір взаємопов’язаних головоломок, побудованих навколо оригінальної комбінованої механіки подорожей у часі. Крім того, проведено комплексне тестування прототипу, яке включало функціональне, інтеграційне та тестування зручності використання.

Одним із найважливіших технічних рішень стало винесення механіки часу в окремі незалежні компоненти. Це дозволило зберегти код PlayerController чистим, значно спростило тестування і відкриває можливості для подальшого розширення проєкту (додавання нових режимів, комбінації з іншими механіками тощо).

Проведене тестування показало, що прототип є технічно стабільним. Гра підтримує стабільні 60 FPS, розмір стисненого білду становить близько 38 МБ, а більшість функціональних і інтеграційних тестів пройшли успішно. Після кількох доопрацювань проблеми з респавном і швидким перемиканням режимів були вирішені.

Водночас тестування зручності використання виявило слабке місце проєкту– onboarding нових гравців. Механіка подорожей у часі виявилася цікавою, але досить складною для сприйняття з перших хвилин. Досвідчені гравці швидко орієнтувалися і отримували задоволення від головоломок, тоді як новачки часто витрачали додатковий час, щоб зрозуміти причинно-наслідкові зв’язки між перемиканням режимів і телепортацією. Це цілком очікувана ситуація для ігор з нестандартною механікою, але вона чітко вказує на напрямок подальшої роботи.

Загалом «CronoWortex» вдалося створити гру, яка не просто демонструє технічну реалізацію, а й пропонує власний ігровий почерк. Поєднання динамічного платформінгу з нелінійним логічним мисленням робить її помітною на фоні багатьох типових 2D-платформерів.

Практична цінність роботи полягає в кількох аспектах:

1. Отриманий прототип може слугувати сильним елементом портфоліо при працевлаштуванні в ігрову індустрію або участі в інді-фестивалях.
2. Детально описана архітектура та прийняті рішення можуть бути корисними для інших студентів і початківців, які планують створювати ігри зі складними механіками.
3. Проєкт показав, що навіть у рамках дипломної роботи реально реалізувати оригінальну ігрову ідею, а не просто «ще один платформер».

Звичайно, робота не позбавлена недоліків. Інтерфейс і візуальний фідбек ще потребують доопрацювання, onboarding нових гравців можна зробити значно м’якшим, а кількість контенту (рівнів і головоломок) поки що недостатня для

повноцінної гри. Однак фундамент проєкту – архітектура, основні механіки та технічна стабільність – закладений міцно і дозволяє продовжувати розвиток без глобальних переробок.

Підсумовуючи, дипломна робота підтвердила, що поєднання класичного платформінгу з глибокою механікою подорожей у часі є перспективним і технічно цікавим напрямом. Створений прототип «CronoWortex» став не лише виконанням навчального завдання, а й повноцінним невеликим ігровим проєктом, який має потенціал для подальшого розвитку

Робота пройшла апробацію. За її результатами було опубліковано наступні тези доповідей:

1. Ляшенко Я.В. Реалізація комбінованої механіки подорожей у часі в інтерактивному 2D-платформері // VII Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». Збірник тез. К.: ДУІКТ, 2026 (подано до друку).
2. Ляшенко Я.В. Розробка та супровід програмного забезпечення на прикладі 2D гри «ChronoVortex» // Всеукраїнська науково-технічна конференція «Технології цифрового розвитку: програмні системи та децентралізація». Збірник тез. К.: ДУІКТ, 2026. (подано до друку)

Перелік посилань

1. Nintendo. Super Mario Bros. – Official History. (n.d). [Електронний ресурс] Режим доступу: https://mario.nintendo.com/history/ (дата звернення: 10.04.2026).
2. https://simfiction.co.uk/ninja-gaiden-ragebound-vs-shinobi-art-of-vengeance (дата звернення: 12.04.2026).
3. GamesIndustry.biz. Hollow Knight: Silksong sells over 7m copies in three months. 2025. [Електронний ресурс] Режим доступу: https://www.gamesindustry.biz/hollow-knight-silksong-sells-over-7m-copies-in-three-months (дата звернення: 13.04.2026).
4. PC Gamer. Silksong has sold more than 7 million copies in 3 months. (n.d). [Електронний ресурс] Режим доступу: https://www.pcgamer.com/games/action/silksong-has-sold-more-than-7-million-copies-in-3-months-and-no-thats-not-counting-game-pass/ (дата звернення: 13.04.2026).
5. Braid, Anniversary Edition – Official Website. 2026. [Електронний ресурс] Режим доступу: http://braid-game.com/ (дата звернення: 10.04.2026).
6. Playdead. Limbo & Inside Official Site. 2026. [Електронний ресурс] Режим доступу: https://playdead.com/ (дата звернення: 10.04.2026).
7. Facepalm Games. The Swapper. 2024. URL: https://www.facepalmgames.com/the-swapper/ (дата звернення: 10.04.2026).
8. SteamDB. Celeste Sales and Reviews Data. 2018. [Електронний ресурс] Режим доступу: https://steamdb.info/app/504230/ (дата звернення: 10.04.2026).
9. SteamDB. Hollow Knight: Silksong Charts. 2025. [Електронний ресурс] Режим доступу: https://steamdb.info/app/1030300/charts/ (дата звернення: 10.04.2026).

10. Unity Technologies. Official Unity Documentation-2D. (n.d). [Електронний ресурс] Режим доступу: https://docs.unity3d.com/Manual/Unity2D.html (дата звернення: 12.04.2026).
11. Generalist Programmer. Best Game Engines 2025: Unity vs Godot vs Unreal vs GameMaker. 2025. [Електронний ресурс] Режим доступу: https://generalistprogrammer.com/comparisons/game-development-engines-2025 (дата звернення: 12.04.2026).
12. itch.io Blog. Game Engine Showdown 2025. 2025. [Електронний ресурс] Режим доступу: https://itch.io/blog/1067028/game-engine-showdown-2025-unity-vs-godot-vs-unreal-which-should-you-choose (дата звернення: 12.04.2026).
13. Video Game Insights. The Big Game Engine Report of 2025. 2025. [Електронний ресурс] Режим доступу: https://app.sensortower.com/vgi/assets/reports/The_Big_Game_Engines_Report_of_2025.pdf (дата звернення: 12.04.2026).
14. GameMaker.io. Official Website – Showcase. (n.d). [Електронний ресурс] Режим доступу: https://gamemaker.io/ (дата звернення: 10.04.2026).
15. Dev.to. Which Game Engine Should You Use in 2026. 2016 – 2026. [Електронний ресурс] Режим доступу: https://dev.to/oceanviewgames/which-game-engine-should-you-use-unity-vs-unreal-vs-godot-2026-4ndj (дата звернення: 13.04.2026).
16. Godot Engine. Official Godot Documentation. (n.d). [Електронний ресурс] Режим доступу: https://docs.godotengine.org/en/stable/ (дата звернення: 12.04.2026).
17. HiteM3D. Top 5 Game Engines in 2026. 2026. [Електронний ресурс] Режим доступу: https://www.hitem3d.ai/blog/en-Top-5-Game-Engines-Compared-Unity-vs-Unreal-vs-Godot-and-More-in-2026/ (дата звернення: 13.04.2026).

18. GameDev.net. Picking a Game Engine in 2025. 2025. [Електронний ресурс] Режим доступу: https://gamedev.net/blogs/entry/2295692-picking-a-game-engine-in-2025-without-crying (дата звернення: 12.04.2026).
19. GameEngineHub. GameMaker vs Godot Comparison. 2025. [Електронний ресурс] Режим доступу: https://gameenginehub.com/comparisons/gamemaker-vs-godot (дата звернення: 12.04.2026).
20. Ляшенко Я.В. Реалізація комбінованої механіки подорожей у часі в інтерактивному 2D-платформері // VII Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». Збірник тез. К.: ДУІКТ, 2026, ст. 130-132.
21. Ляшенко Я.В. Розробка та супровід програмного забезпечення на прикладі 2D гри «ChronoVortex» // Всеукраїнська науково-технічна конференція «Технології цифрового розвитку: програмні системи та децентралізація». Збірник тез. К.: ДУІКТ, 2026, (подано до друку)

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ТЕХНОЛОГІЇ ЦИФРОВОГО РОЗВИТКУ

Інтерактивний 2D платформер з елементами головоломки та подорожами у часі на Unity

Виконав студент 4 курсу
групи ТЦР-42
Ляшенко Ярослав Володимирович
Керівник роботи
Ст. викладач кафедри ТЦР Дзюба Вадим Віталійович

Київ-2026

МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи – підвищити ігрового досвіду за допомогою інтерактивного 2D платформера з елементами головоломки та подорожами у часі.

Об’єкт дослідження – процес розробки 2D-платформера з елементами головоломок.

Предметом дослідження – методи та засоби ігрових продуктів, а саме 2D-ігор з нестандартними механіками на базі ігрового рушія Unity та мови програмування C#.

Актуальність та новизна

1. Ринок інді-ігор активно розвивається, а якісні 2D puzzle-platformer демонструють високу комерційну рентабельність завдяки оригінальності та низькій вартості розробки.
2. Реалізація складних нелінійних механік маніпуляції часом залишається серйозним технічним викликом, особливо для студентських та невеликих команд.
3. Такі ігри мають високий освітній потенціал, розвиваючи просторове, стратегічне та причинно-наслідкове мислення.

Розроблено оригінальну комбіновану механіку подорожей у часі, що поєднує перемикання часових режимів та парну просторово-часову телепортацію. Оригінальність полягає саме в тому що, це перша гра в якій використовується відразу дві умовні часові механіки задля ускладнення платформинга та створення ускладнених головоломок з елементами часових парадоксів.

3

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Проаналізувати сучасний стан жанру 2D-платформерів та puzzle-platformers, а також порівняти найпоширеніші ігрові рушії для 2D-розробки.
2. Обґрунтувати вибір технологічного стеку (Unity та C#) та архітектурних підходів з урахуванням особливостей механіки часу.
3. Реалізувати ключові ігрові системи: керування персонажем, систему анімації, перемикання часових режимів та парна телепортація.
4. Створити набір взаємопов’язаних головоломок, побудованих навколо механіки подорожей у часі.
5. Провести комплексне тестування (функціональне, інтеграційне та юзабіліті) ключових механік гри.

4

АНАЛІЗ АНАЛОГІВ

Рушій / Редактор	Основна мова	Орієнтація	Ліцензія / Ціна	Переваги (+)	Недоліки (–)	Найкраще підходить для
Unity	C#	2D та 3D (універсальний)	Безкоштовно та роялті після порогу доходу	Відмінна кросплатформеність Великий Asset Store Потужні 2D-інструменти (Tilemap, 2D Physics, Cinemachine)	Вищі системні вимоги 2D реалізований поверх 3D-рушія	Кросплатформені ігри, мобільні проєкти, середні та великі інді
Unreal Engine	C++, Blueprints	Переважно 3D (Paper2D для 2D)	Безкоштовно та роялті після порогу	Найвища якість графіки та ефектів Добре для гібридних 2D/3D проєктів	Обмежена підтримка Paper2D Висока крива навчання «Важкий» для простих 2D-проєктів	Візуально насичені гібридні ігри, AAA-рівень графіки
Godot Engine	GDScript, C#, C++	2D та 3D (відмінний native 2D)	100% безкоштовно (MIT, відкритий код)	Легкість і висока продуктивність Зручна система вузлів Швидке прототипування	Менша екосистема асетів і плагінів порівняно з Unity	Інді-2D платформери, соло- та малі команди, навчальні проєкти
GameMaker	GML + Drag & Drop	Спеціально 2D	Безкоштовна версія та платна для комерційного експорту	Швидка розробка прототипів Інтуїтивний інтерфейс Оптимізація для pixel-art	Обмежені можливості для складних механік і 3D	Швидкі 2D-платформери, pixel-art ігри, початківці та професіонали
Construct	Візуальні блоки (no-code)	2D	Підписка для повного доступу	Не потребує програмування Швидке прототипування	Обмежена гнучкість для складних механік і великих проєктів	Початківці, освітні проєкти, швидкі прототипи, веб-ігри

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги:

1. Реалізація точного 2D-платформерного керування (біг, стрибки, подвійний стрибок).
2. Розробка оригінальної системи подорожей у часі.
3. Створення системи головоломок, заснованих на взаємодії кількох часових станів.
4. Реалізація системи здоров'я, пошкоджень та респавну.
5. Можливість проходження рівнів з використанням комбінації платформінгу та логічного мислення.

Нефункціональні вимоги:

1. Стабільна продуктивність не нижче 60 FPS на середньостатистичному ПК.
2. Розмір білду після стиснення – не більше 50 МБ.
3. Гнучка та масштабована архітектура.
4. Зручне та інтуїтивне керування для гравця.
5. Висока стабільність роботи.
6. Мінімалістичний, ненав'язливий інтерфейс користувача (HUD).

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

Ігровий рушій для розробки 2D/3D ігор. Забезпечує створення сцен, фізики, анімацій та збірки проєкту.

Веб-ресурс для піксель-арту та створення спрайтів, тайлів та анімацій.

Професійне середовище розробки (IDE) для написання, налагодження та тестування коду

Веб-ресурс для генерації звукових ефектів, що дозволяє швидко створювати ретро-звуки для гри.

Платформа для контролю версій, зберігання коду та організації спільної роботи над проєктом.

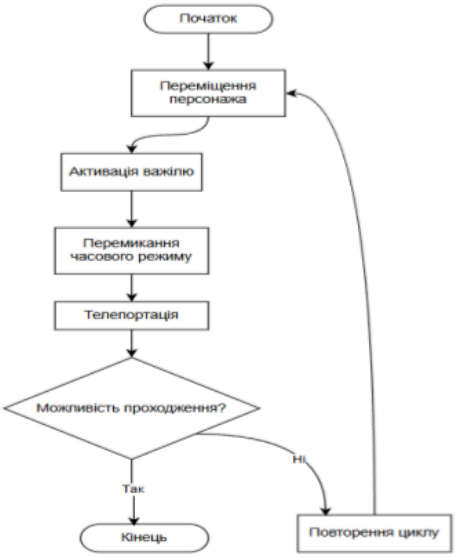
Мова програмування яка є основним засобом розробки в Unity. Використовується для написання ігрової логіки, взаємодії об'єктів та скриптів.

7

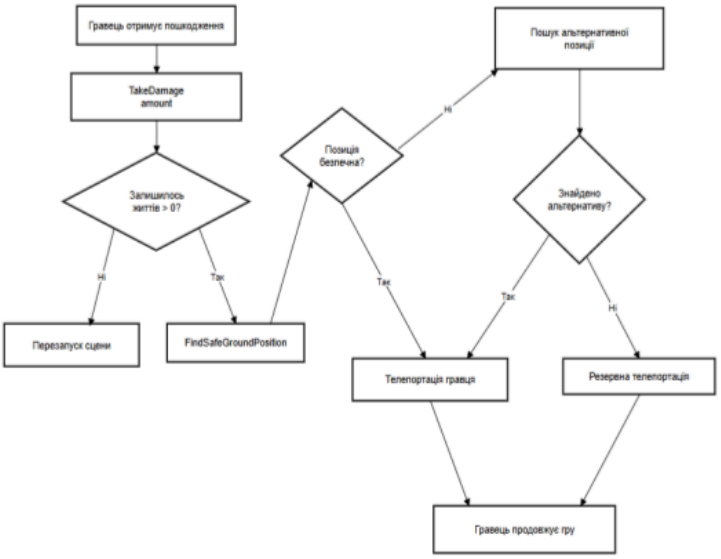
ДІАГРАМА ВАРІАНТІВ ВИКОРИСТАННЯ

8

АЛГОРИТМИ РОБОТИ ЗАСТОСУНКУ



АЛГОРИТМИ ВЗАЄМОДІЇ ГРАВЦЯ З СИСТЕМОЮ

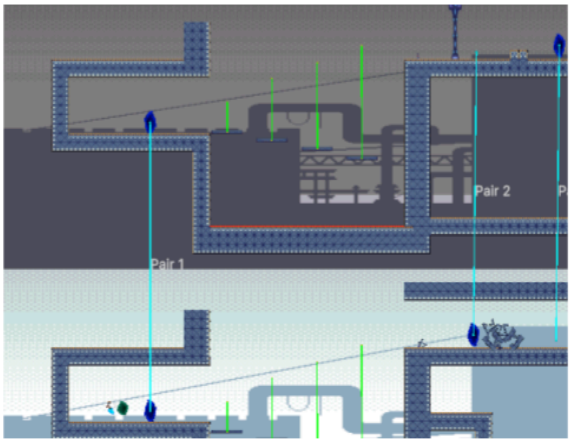


10

ДІАГРАМА КЛАСІВ

11

ЕКРАННІ ФОРМИ

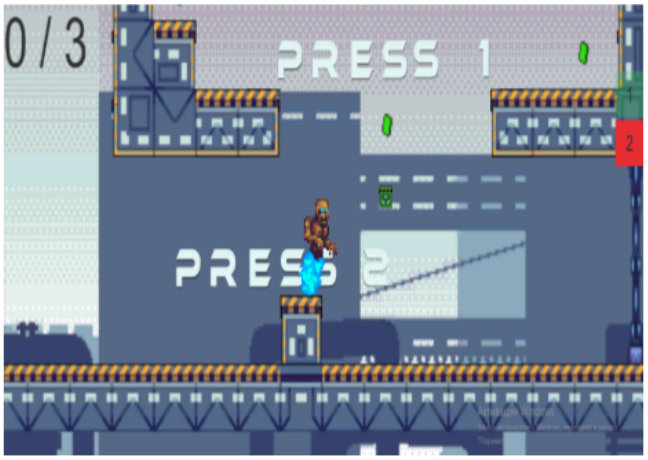


Відображення (PlayerTeleportSystem).

Головне меню.

12

ЕКРАННІ ФОРМИ

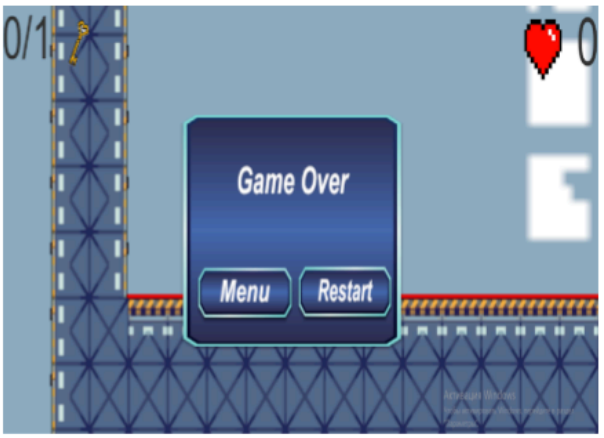


Ввідний рівень.

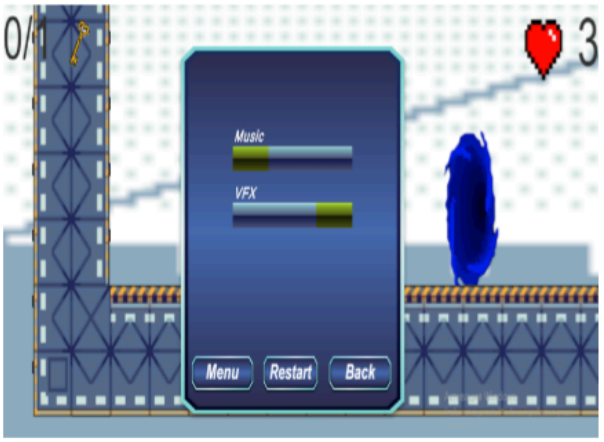
HUD гравця та (ModeActivator).

13

ЕКРАННІ ФОРМИ



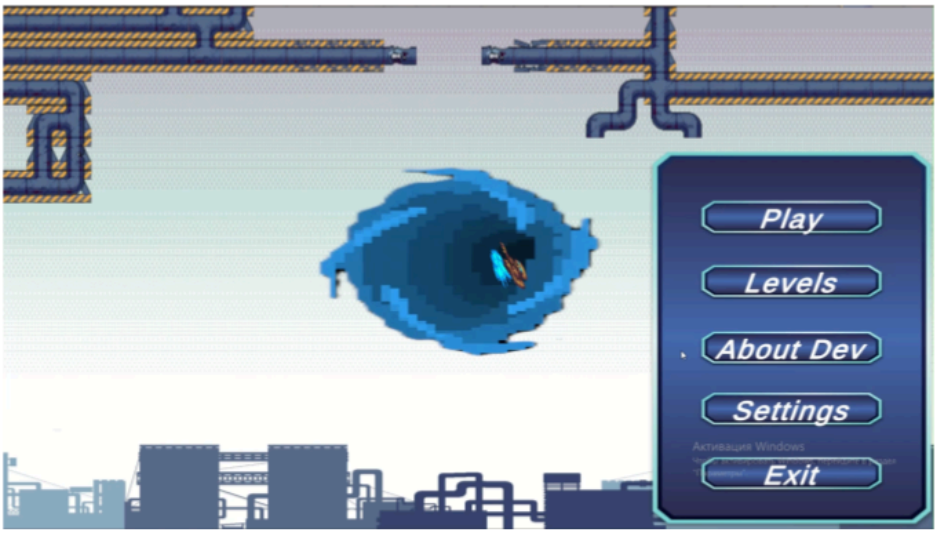
Меню смерті.



Меню налаштувань (Esc).

14

Демонстрація роботи застосунку



15

АПРОБАЦІЯ РЕЗУЛЬТАТІВ

Ляшенко Я.В. Реалізація комбінованої механіки подорожей у часі в інтерактивному 2D-платформері // VII Всеукраїнській науково-технічній конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». Збірник тез. К.: ДУІКТ, 2026, ст. 130-132.

Ляшенко Я.В. Розробка та супровід програмного забезпечення на прикладі 2D гри «ChronoVortex» // Всеукраїнській науково-технічній конференції «Технології цифрового розвитку: програмні системи та децентралізація». Збірник тез. К.: ДУІКТ, 2026, (подано до друку).

16

ВИСНОВКИ

1. Проаналізовано сучасний стану жанру 2D-платформерів та puzzle-platformers, а також виконано порівняльну характеристику ігрових рушіїв, що дозволило обґрунтовано обрати Unity та C# як основний технологічний стек.
2. Обґрунтовано гнучку архітектуру проєкту, яка ефективно вирішує проблеми реалізації складних нелінійних механік.
3. Реалізовано ключові ігрові системи: інтелектуальне керування персонажем, покадрову анімацію, систему перемикання часових режимів та парну просторово-часову телепортацію.
4. Створено набір взаємопов’язаних головоломок, побудованих навколо оригінальної комбінованої механіки подорожей у часі.
5. Проведено комплексне тестування, включаючи функціональне, інтеграційне та юзабіліті-тестування.

17

ДОДАТОК Б. ЛІСТИНГ ОСНОВНИХ МОДУЛІВ

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

public interface IModeSwitcher
{
   void SwitchToMode(int modeIndex);
}

public class ModeActivator : MonoBehaviour, IModeSwitcher
{
   [Header("Mode 1")]
   [SerializeField] private List<GameObject> modeOneObjects
= new List<GameObject>();
   [SerializeField] private Image modeOneUI;

   [Header("Mode 2")]
   [SerializeField] private List<GameObject> modeTwoObjects
= new List<GameObject>();
   [SerializeField] private Image modeTwoUI;

   [Header("UI Settings")]
   [SerializeField] private float activeAlpha = 1f;
   [SerializeField] private float inactiveAlpha = 0.3f;

   private readonly Dictionary<int, ModeHandler>
modeHandlers = new();
   private int currentMode = 1;

   private void Awake()
   {
      modeHandlers[1] = new ModeHandler(modeOneObjects,
modeOneUI, activeAlpha, inactiveAlpha);
      modeHandlers[2] = new ModeHandler(modeTwoObjects,
modeTwoUI, activeAlpha, inactiveAlpha);

      SwitchToMode(1);
   }

   private void Update()
   {
      if (Input.GetKeyDown(KeyCode.Alpha1))
SwitchToMode(1);
      if (Input.GetKeyDown(KeyCode.Alpha2))
SwitchToMode(2);
   }

   public void SwitchToMode(int modeIndex)
   {
      if (!modeHandlers.ContainsKey(modeIndex))
      {
         Debug.LogWarning($"Mode {modeIndex} not
found!");
         return;
      }

      foreach (var handler in modeHandlers.Values)
      {
         handler.Deactivate();
      }

      modeHandlers[modeIndex].Activate();
      currentMode = modeIndex;
   }
}
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public interface IModeHandler
{
   void Activate();
   void Deactivate();
}

public class ModeHandler : IModeHandler
{
   private readonly List<GameObject> objects;
   private readonly Image uiElement;
   private readonly float activeAlpha;
   private readonly float inactiveAlpha;

   public ModeHandler(List<GameObject> objects, Image
uiElement, float activeAlpha, float inactiveAlpha)
   {
      this.objects = objects ?? new List<GameObject>();
      this.uiElement = uiElement;
      this.activeAlpha = activeAlpha;
      this.inactiveAlpha = inactiveAlpha;
   }

   public void Activate()
   {
      foreach (var obj in objects)
      {
         if (obj != null)
            obj.SetActive(true);
         else
            Debug.LogWarning("There is a null element in the
mode list!");
      }

      if (uiElement != null)
```

```
        {
            var c = uiElement.color;
            uiElement.color = new Color(c.r, c.g, c.b, activeAlpha);
        }
    }

    public void Deactivate()
    {
        foreach (var obj in objects)
        {
            if (obj != null)
                obj.SetActive(false);
        }

        if (uiElement != null)
        {
            var c = uiElement.color;
            uiElement.color = new Color(c.r, c.g, c.b,
inactiveAlpha);
        }
    }
}
using System.Collections.Generic;
using UnityEngine;
#if UNITY_EDITOR
using UnityEditor;
#endif

public class PlayerTeleportSystem : MonoBehaviour
{
    [System.Serializable]
    public class TeleportPoint
    {
        public Transform point;
        [HideInInspector] public int count;
    }

    [Header("Main Objects")]
    [SerializeField] private Transform player;
    [SerializeField] private List<TeleportPoint> teleportPoints =
new();
    [SerializeField] private float teleportYOffset = 1f;

    [Header("UI Elements")]
    [SerializeField] private GameObject uiObject1;
    [SerializeField] private GameObject uiObject2;
    [SerializeField] private GameObject uiObject3;

    public enum AccessMode { None, Only1, Only2, Only3, All
}

    private AccessMode currentAccess = AccessMode.None;

    private void Start()
    {
        AssignCounts();
        UpdateUI(AccessMode.None);
    }

    private void Update()
    {
        if (currentAccess == AccessMode.None || player == null)
            return;

        if (Input.GetKeyDown(KeyCode.Alpha1) &&
            (currentAccess == AccessMode.All || currentAccess ==
AccessMode.Only1))
        {
            TeleportToLinkedPoint();
            UpdateUI(AccessMode.Only1);
        }

        if (Input.GetKeyDown(KeyCode.Alpha2) &&
            (currentAccess == AccessMode.All || currentAccess ==
AccessMode.Only2))
        {
            TeleportToLinkedPoint();
            UpdateUI(AccessMode.Only2);
        }

        if (Input.GetKeyDown(KeyCode.Alpha3) &&
            (currentAccess == AccessMode.All || currentAccess ==
AccessMode.Only3))
        {
            TeleportToLinkedPoint();
            UpdateUI(AccessMode.Only3);
        }
    }

    private void AssignCounts()
    {
        for (int i = 0; i < teleportPoints.Count; i++)
        {
            teleportPoints[i].count = i;
        }
    }

    private void TeleportToLinkedPoint()
    {
        if (teleportPoints.Count < 2) return;

        TeleportPoint nearest = GetNearestPoint();
        if (nearest == null) return;

        int targetIndex = (nearest.count % 2 == 0) ? nearest.count
+ 1 : nearest.count - 1;

        if (targetIndex < 0 || targetIndex >= teleportPoints.Count)
return;

        TeleportPoint target = teleportPoints[targetIndex];
```

```
        if (target != null && target.point != null)
        {
            player.position = target.point.position + Vector3.up *
teleportYOffset;
        }
    }

    private TeleportPoint GetNearestPoint()
    {
        TeleportPoint nearest = null;
        float minDist = float.MaxValue;

        foreach (var tp in teleportPoints)
        {
            if (tp.point == null) continue;

            float dist = Vector3.Distance(player.position,
tp.point.position);

            if (dist < minDist)
            {
                minDist = dist;
                nearest = tp;
            }
        }
        return nearest;
    }

    public void SetAccess(AccessMode mode)
    {
        currentAccess = mode;
        UpdateUI(mode);
    }

    public void ResetAccess()
    {
        currentAccess = AccessMode.None;
        UpdateUI(AccessMode.None);
    }

    private void UpdateUI(AccessMode mode)
    {
        if (uiObject1 != null) uiObject1.SetActive(mode ==
AccessMode.Only1 || mode == AccessMode.All);
        if (uiObject2 != null) uiObject2.SetActive(mode ==
AccessMode.Only2 || mode == AccessMode.All);
        if (uiObject3 != null) uiObject3.SetActive(mode ==
AccessMode.Only3 || mode == AccessMode.All);

        if (mode == AccessMode.None)
        {
            if (uiObject1 != null) uiObject1.SetActive(false);
            if (uiObject2 != null) uiObject2.SetActive(false);
            if (uiObject3 != null) uiObject3.SetActive(false);
        }
    }
```

```
#if UNITY_EDITOR
    private void OnDrawGizmos()
    {
        if (teleportPoints == null || teleportPoints.Count == 0)
            return;

        for (int i = 0; i < teleportPoints.Count; i += 2)
        {
            if (i + 1 >= teleportPoints.Count) break;

            var a = teleportPoints[i].point;
            var b = teleportPoints[i + 1].point;

            if (a != null && b != null)
            {
                Gizmos.color = Color.cyan;
                Gizmos.DrawLine(a.position, b.position);
                Gizmos.DrawSphere(a.position, 0.15f);
                Gizmos.DrawSphere(b.position, 0.15f);
                Handles.Label((a.position + b.position) / 2 +
Vector3.up * 0.3f, $"Pair {i / 2 + 1}");
            }
        }
    }
#endif
}
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;

public class PlayerController : MonoBehaviour
{
    [Header("Movement Settings")]
    [SerializeField] private float moveSpeed = 5f;

    [Header("Health Settings")]
    [SerializeField] private int startLives = 3;
    [SerializeField] private Text livesText;

    [Header("Ground Check")]
    [SerializeField] private LayerMask groundLayer;
    [SerializeField] private Transform groundCheck;
    [SerializeField] private float groundCheckRadius = 0.15f;

    [Header("Jump Settings")]
    [SerializeField] private float jumpForce = 7f;

    [Header("Respawn Settings")]
    [SerializeField] private float respawnHeight = 0.5f;
    [SerializeField] private float searchDistance = 5f;

    private Rigidbody2D _rb;
    private IPlayerMovement _movement;
    private IGroundChecker _groundChecker;
    private IHealthService _health;
```

```
    private IPlayerAnimation _animation;
    private IPlayerJump _jumpService;
    private PlayerStateMachine _stateMachine;

    private Vector3 _lastGroundPosition;
    private bool _wasGroundedLastFrame;

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

        _movement = new PlayerMovement(_rb, moveSpeed);
        _groundChecker = new GroundChecker(groundCheck,
groundCheckRadius, groundLayer);
        _health = new HealthService(startLives);
        _jumpService = new PlayerJumpService();

        _animation = GetComponent<IPlayerAnimation>();
        if (_animation == null)
            _animation =
gameObject.AddComponent<PlayerAnimation>();

        _stateMachine = new PlayerStateMachine();

        _lastGroundPosition = transform.position;
    }

    private void Start()
    {
        UpdateLivesUI();
        _stateMachine.ChangeState(new
PlayerIdleState(_animation));
    }

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

        _movement.UpdateMovement(horizontalInput,
Time.deltaTime);
        _groundChecker.UpdateGroundState();

        if (_groundChecker.IsGrounded &&
!_wasGroundedLastFrame)
        {
            _jumpService.ResetJumpCount();
            _lastGroundPosition = transform.position;
        }

        _wasGroundedLastFrame = _groundChecker.IsGrounded;

        HandleState(horizontalInput);

        if (Input.GetKeyDown(KeyCode.Space) &&
_jumpService.CanJump())
        {
            Jump(horizontalInput);
        }

        _animation.UpdateWalkAnimation(horizontalInput,
_groundChecker.IsGrounded);
    }

    private void HandleState(float horizontalInput)
    {
        if (!_groundChecker.IsGrounded)
        {
            if (_stateMachine.GetCurrentState() is PlayerJumpState
jumpState)
                jumpState.SetHorizontalInput(horizontalInput);
            else
                _stateMachine.ChangeState(new
PlayerJumpState(_animation, horizontalInput));
        }
        else if (Mathf.Abs(horizontalInput) > 0.01f)
        {
            _stateMachine.ChangeState(new
PlayerWalkState(_animation, horizontalInput, true));
        }
        else
        {
            _stateMachine.ChangeState(new
PlayerIdleState(_animation));
        }

        _stateMachine.Update();
    }

    private void Jump(float horizontalInput)
    {
        _rb.linearVelocity = new Vector2(_rb.linearVelocity.x, 0f);
        _rb.AddForce(Vector2.up * jumpForce,
ForceMode2D.Impulse);

        _jumpService.Jump();
        _stateMachine.ChangeState(new
PlayerJumpState(_animation, horizontalInput));
    }

    public void TakeDamage(int amount)
    {
        _health.TakeDamage(amount);
        UpdateLivesUI();

        if (_health.CurrentLives > 0)
        {
            TeleportToSafePosition();
        }
    }

    private void TeleportToSafePosition()
    {
```

```
        Vector3 safePosition = FindSafeGroundPosition();
        transform.position = safePosition;
        _rb.linearVelocity = Vector2.zero;
    }

    private Vector3 FindSafeGroundPosition()
    {
        Vector3 candidate = _lastGroundPosition + Vector3.up *
respawnHeight;
        if (IsPositionSafe(candidate))
            return candidate;

        for (float dist = 0.8f; dist <= searchDistance; dist += 0.8f)
        {
            Vector3 rightPos = _lastGroundPosition + new
Vector3(dist, respawnHeight, 0);
            if (IsPositionSafe(rightPos))
                return rightPos;

            Vector3 leftPos = _lastGroundPosition + new
Vector3(-dist, respawnHeight, 0);
            if (IsPositionSafe(leftPos))
                return leftPos;
        }

        return _lastGroundPosition + new Vector3(0,
respawnHeight + 3f, 0);
    }

    private bool IsPositionSafe(Vector3 position)
    {
        Collider2D hit = Physics2D.OverlapCircle(position,
0.35f);
        if (hit == null)
            return true;

        return hit.GetComponent<FixPlatform>() == null;
    }

    private void UpdateLivesUI()
    {
        if (livesText != null)
            livesText.text = _health.CurrentLives.ToString();
    }

    private void OnDrawGizmosSelected()
    {
        if (groundCheck == null) return;

        Gizmos.color = Color.green;
        Gizmos.DrawWireSphere(groundCheck.position,
groundCheckRadius);
    }
}
using UnityEngine;

public class PlayerAnimation : MonoBehaviour,
IPlayerAnimation
{
    [SerializeField] private Sprite[] walkSprites = new Sprite[4];
    [SerializeField] private Sprite jumpSprite;

    [SerializeField] private float walkFrameRate = 10f;
    [SerializeField] private float idleFrameRate = 5f;
    [SerializeField] private float idleDelay = 3f;

    private SpriteRenderer _spriteRenderer;

    private float _timer;
    private int _currentFrame;
    private float _idleTimer;

    private bool _isWalking;
    private bool _lastFlipX = false;

    private void Awake()
    {
        _spriteRenderer = GetComponent<SpriteRenderer>();
        if (_spriteRenderer == null)
            _spriteRenderer =
gameObject.AddComponent<SpriteRenderer>();

        if (walkSprites == null || walkSprites.Length < 4)
            walkSprites = new Sprite[4];
    }

    public void UpdateWalkAnimation(float horizontalInput,
bool isGrounded)
    {
        if (_spriteRenderer == null) return;

        if (Mathf.Abs(horizontalInput) > 0.01f)
        {
            _lastFlipX = horizontalInput > 0.01f;
        }

        _spriteRenderer.flipX = _lastFlipX;

        bool shouldWalk = isGrounded &&
Mathf.Abs(horizontalInput) > 0.01f;

        if (shouldWalk)
        {
            _isWalking = true;
            _idleTimer = 0f;

            _timer += Time.deltaTime;
            if (_timer >= 1f / walkFrameRate)
            {
                _timer -= 1f / walkFrameRate;
                _currentFrame = (_currentFrame + 1) %
walkSprites.Length;
```

```
            }

            _spriteRenderer.sprite = walkSprites[_currentFrame];
        }
        else
        {
            if (_isWalking)
            {
                _isWalking = false;
                _timer = 0f;
                _currentFrame = 0;
            }

            _idleTimer += Time.deltaTime;

            if (_idleTimer >= idleDelay)
            {
                PlayIdleAnimation();
            }
            else
            {
                if (walkSprites.Length > 0)
                    _spriteRenderer.sprite = walkSprites[0];
            }
        }
    }

    private void PlayIdleAnimation()
    {
        _timer += Time.deltaTime;
        if (_timer >= 1f / idleFrameRate)
        {
            _timer -= 1f / idleFrameRate;
            _currentFrame = (_currentFrame + 1) %
walkSprites.Length;
        }

        _spriteRenderer.sprite = walkSprites[_currentFrame];
        _spriteRenderer.flipX = _lastFlipX;
    }

    public void PlayJump()
    {
        if (_spriteRenderer == null || jumpSprite == null) return;

        _spriteRenderer.sprite = jumpSprite;
        _spriteRenderer.flipX = _lastFlipX;
        _isWalking = false;
        _idleTimer = 0f;
    }
}
using UnityEngine;

public class PlayerStateMachine
{
    private IPlayerState _currentState;

    public void ChangeState(IPlayerState newState)
    {
        if (_currentState != null)
            _currentState.Exit();

        _currentState = newState;
        _currentState.Enter();
    }

    public void Update()
    {
        _currentState?.Update();
    }

    public IPlayerState GetCurrentState()
    {
        return _currentState;
    }

    public bool IsInState<T>() where T : IPlayerState
    {
        return _currentState is T;
    }
}
public interface IHealthService
{
    int CurrentLives { get; }
    void TakeDamage(int amount);
    void ResetLives(int amount);
}
public interface IPlayerJump
{
    bool CanJump();
    void Jump();
    void ResetJumpCount();
}
public interface IPlayerMovement
{
    void UpdateMovement(float horizontalInput, float
deltaTime);
}
public interface IPlayerState
{
    void Enter();
    void Update();
    void Exit();
}
using UnityEngine;

public class GroundChecker : IGroundChecker
{
    private readonly Transform _groundCheck;
    private readonly float _radius;
    private readonly LayerMask _groundLayer;
```

```
    public bool IsGrounded { get; private set; }

    public GroundChecker(Transform groundCheck, float radius,
LayerMask groundLayer)
    {
        _groundCheck = groundCheck;
        _radius = radius;
        _groundLayer = groundLayer;
        IsGrounded = false;
    }

    public void UpdateGroundState()
    {
        IsGrounded =
Physics2D.OverlapCircle(_groundCheck.position, _radius,
_groundLayer);
    }
}
public class HealthService : IHealthService
{
    public int CurrentLives { get; private set; }

    public HealthService(int startLives)
    {
        CurrentLives = startLives;
    }

    public void TakeDamage(int amount)
    {
        CurrentLives -= amount;
        if (CurrentLives < 0) CurrentLives = 0;
    }

    public void ResetLives(int amount)
    {
        CurrentLives = amount;
    }
}
```

ДОДАТОК В. ДЕМОНСТРАЦІЯ ТАБЛИЦІ

Таблиця В.1

Test case з критичними об’єктами

ID	Група тестування	Опис тест-кейсу	Вхідні дані	Очікуваний результат	Фактичний результат	Статус
TC-01	Керування персонажем	Перехід між станами Idle, Walk, Jump	Натиснення клавіш руху та Space	Персонаж плавно змінює анімацію та поведінку відповідно до стану	Відповідає	Passed
TC-02	Керування персонажем	Виконання подвійного стрибка	Стрибок у повітрі	Другий стрибок виконується, після приземлення лічильник скидається	Відповідає	Passed
TC-03	Керування персонажем	Фліп спрайту під час руху вліво/вправо	Рух у протилежних напрямках	Спрайт коректно повертається без миготіння	Відповідає	Passed
TC-10	Механіка часу	Перемикання між двома часовими режимами	Натиснення клавіш 1 та 2	Миттєва зміна стану рівня (платформи, ричаги, пастки)	Відповідає	Passed
TC-11	Механіка часу	Реакція об’єктів на подію зміни режиму (UnityEvent)	Перемикання режиму	Усі підписані об’єкти реагують на зміну режиму	Відповідає	Passed
TC-12	Телепортація	Телепортація між парними точками	Натиснення E біля активної точки телепортації	Персонаж миттєво переміщується до пари з урахуванням teleportYOffset	Відповідає	Passed
TC-13	Телепортація	Спроба телепортації в недоступному режимі	Натиснення E при закритому доступі	Телепортація блокується	Відповідає	Passed

ТС-14	Комбінована механіка часу	Повний цикл: активувати важіль, змінити режим, телепортуватися, активувати інший важіль	Послідовне виконання дій	Всі дії виконуються коректно, рівень змінюється відповідно	Відповідає	Passed
ТС-20	Головоломки	Збір 5 кристалів і активація фінального об’єкта	Збір усіх кристалів	Після 5-го кристала активується фінальний об’єкт (двері/платформа)	Відповідає	Passed
ТС-22	Респавн	Респавн після смерті з урахуванням поточного часового режиму	Смерть персонажа	Гравець з’являється на безпечній позиції, стан рівня зберігається	Відповідає	Passed
ТС-25	Usability testing	Зрозумілість механіки перемикання режимів для нового гравця	Перші 2 хвилини гри без підказок	Гравець розуміє вплив зміни режиму на рівень	Частково	Passed
ТС-26	Usability testing	Зручність комбінованих дій (стрибок та перемикання режиму та телепортація)	Виконання складної послідовності дій	Керування не викликає сильного дискомфорту	Passed	Passed
ТС-27	Usability testing	Швидкість респавну після смерті	Смерть персонажа	Час від смерті до відновлення гри не перевищує 1,5 секунди	Відповідає	Passed