ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

КВАЛІФІКАЦІЙНА РОБОТА
на тему: «ПЛАТФОРМЕР З БІЧНИМ СКРОЛІНГОМ НА ОСНОВІ ІГРОВОГО РУШІЯ GODOT»

на здобуття освітнього ступеня бакалавр
зі спеціальності 126 Інформаційні системи та технології
(код, найменування спеціальності)
освітньо-професійної програми Інформаційні системи та технології
(назва)

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

_____ Євген СИТНИК
(підпис) *(ім'я, ПРІЗВИЩЕ здобувача)*

Виконав: здобувач вищої освіти гр. ІСД-42
Євген СИТНИК
(ім'я, ПРІЗВИЩЕ)

Керівник: Олег СЕНЬКОВ
к.т.н.,доцент *(ім'я, ПРІЗВИЩЕ)*

Рецензент: _____
(ім'я, ПРІЗВИЩЕ)

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інформаційні системи та технології

Ступінь вищої освіти Бакалавр

Спеціальність Інформаційні системи та технології

Освітньо-професійна програма Інформаційні системи та технології

ЗАТВЕРДЖУЮ
Завідувач кафедрою ІСТ

_____ Каміла СТОРЧАК
«_____» _____ 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Ситник Євген Олегович
(прізвище, ім'я, по батькові здобувача)

1. Тема кваліфікаційної роботи: Платформер з бічним скролінгом на основі ігрового рушія Godot

керівник кваліфікаційної роботи Олег Сеньков кандидат технічних наук,
(Ім'я, ПРІЗВИЩЕ науковий ступінь, вчене звання)
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 р. №51.

2. Строк подання кваліфікаційної роботи «1» червня 2026 р.

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література з розробки відеоігор, офіційна документація ігрового рушія Godot та мови GDScript, вимоги до проєктування двовимірних ігор жанру платформер з бічним скролінгом, вихідні графічні матеріали (спрайти персонажа, тайлсети, кадри анімації) для реалізації демонстраційної гри.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
Теоретичні основи розробки двовимірних платформерів з бічним скролінгом
Аналіз існуючих рішень та проєктування архітектури платформера

Реалізація та тестування платформера з бічним скролінгом

5. Перелік графічного матеріалу: *презентація*

1. Теоретична частина
2. Аналіз існуючих рішень та проєктування архітектури платформера
3. Реалізація та тестування платформера з бічним скролінгом

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

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз науково-технічної літератури з тематики розробки двовимірних ігор та жанру платформер	21.02-08.03.26	
2	Огляд та порівняльний аналіз сучасних ігрових рушіїв, обґрунтування вибору Godot	09.03-22.03.26	
3	Дослідження архітектури рушія Godot, системи вузлів і сцен, мови GDScript	23.03-05.04.26	
4	Проєктування загальної архітектури гри, ігрових рівнів та інтерфейсу користувача	06.04-19.04.26	
5	Реалізація системи керування персонажем, ігрової фізики та механіки бічного скролінгу камери	20.04-03.05.26	
6	Розробка ігрових рівнів засобами TileMap, реалізація анімацій, системи збирання предметів та HUD	04.05-15.05.26	
7	Тестування розробленої гри, оформлення вступу, висновків, реферату та списку джерел	16.05-21.05.26	
8	Підготовка демонстраційних матеріалів та презентації до захисту	21.05-26.05.26	

Здобувач вищої освіти _____ Євген СИТНИК
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Керівник
кваліфікаційної роботи _____ Олег СЕНЬКОВ
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 57 стор., 26 рис., 21 джерел.

Мета роботи - розробка двовимірного платформера з бічним скролінгом засобами ігрового рушія Godot з реалізацією базових механік жанру: керування персонажем з прийомами coyote time, буферизації та змінної висоти стрибка, бічного скролінгу камери, побудови демонстраційного ігрового рівня, анімацій персонажа і збираних предметів та мінімального ігрового інтерфейсу.

Об'єкт дослідження - процес розробки двовимірних ігор жанру платформер з бічним скролінгом із застосуванням сучасних ігрових рушіїв з відкритим вихідним кодом.

Предмет дослідження - методи та інструментальні засоби розробки платформера з бічним скролінгом на основі ігрового рушія Godot та мови програмування GDScript, зокрема використання вузлів CharacterBody2D, Camera2D, TileMapLayer, AnimatedSprite2D, Area2D та CanvasLayer для реалізації фізичної моделі руху персонажа, ігрової камери, рівня, анімацій та інтерфейсу користувача.

Короткий зміст роботи: Досліджено теоретичні аспекти жанру платформер та на основі порівняльного аналізу обґрунтовано вибір ігрового рушія Godot. Спроєктовано модульну архітектуру гри з використанням патерну Singleton (GlobalDataStore) та системи сигналів для обміну даними. Практично реалізовано просунуті механіки керування персонажем (coyote time, буферизація стрибка), демонстраційний ігровий рівень на базі TileMapLayer, інтерактивні об'єкти та графічний інтерфейс користувача. Проведено комплексне тестування розробленого програмного застосунку.

КЛЮЧОВІ СЛОВА: ПЛАТФОРМЕР, БІЧНИЙ СКРОЛІНГ, ІГРОВИЙ РУШІЙ, GODOT, GDSCRIPT, РОЗРОБКА ІГОР, ДВОВИМІРНА ГРАФІКА, АНІМАЦІЯ.

ЗМІСТ

ВСТУП

У сучасних умовах стрімкого розвитку індустрії відеоігор та демократизації інструментів розробки програмного забезпечення питання створення якісного ігрового контенту незалежними розробниками набуває дедалі більшої актуальності. Ігрова індустрія є одним із найдинамічніших сегментів глобальної цифрової економіки[1], а жанр платформера з бічним скролінгом, попри свою багаторічну історію, не втрачає популярності й залишається затребуваним як серед гравців, так і серед розробників-початківців. Поява відкритих та безкоштовних ігрових рушіїв суттєво знизила поріг входу у сферу геймдеву, створивши умови для розробки повноцінних ігрових проєктів без значних фінансових витрат. Саме тому дослідження методів та інструментів реалізації двовимірних платформерів із застосуванням сучасних відкритих рушіїв є актуальним завданням для фахівців у галузі інформаційних технологій.

Актуальність дослідження зумовлена низкою проблем, характерних для сучасного ринку інструментів розробки ігор. Серед них - надмірна складність і комерційна закритість провідних ігрових рушіїв, що ускладнює їх освоєння студентами та незалежними розробниками; відсутність уніфікованих методичних підходів до проєктування архітектури двовимірних платформерів; а також недостатнє висвітлення практичних аспектів використання рушія Godot для створення ігор жанру платформер із бічним скролінгом. Окремим викликом є необхідність поєднання ефективної реалізації ігрової фізики, системи анімацій та механіки скролінгу в межах єдиної, логічно структурованої архітектури проєкту.

Мета роботи - розробка двовимірного платформера з бічним скролінгом на основі ігрового рушія Godot із реалізацією повного циклу створення гри: від проєктування архітектури та ігрових рівнів до впровадження системи керування персонажем, анімацій, збирання предметів та ігрового інтерфейсу користувача.

Об'єктом дослідження є процеси розробки двовимірних ігор жанру платформер з бічним скролінгом засобами сучасних ігрових рушіїв.

Предметом дослідження є методи та програмні засоби реалізації платформера з бічним скролінгом на основі ігрового рушія Godot із застосуванням мови програмування GDScript.

Методи дослідження. Під час написання кваліфікаційної роботи були використані методи системного аналізу та порівняльного узагальнення при огляді існуючих рішень і рушіїв, методи об'єктно-орієнтованого та компонентного проєктування при побудові архітектури гри на основі системи вузлів і сцен Godot, а також методи функціонального та ігрового тестування для перевірки коректності реалізованих механік.

Наукова новизна одержаних результатів. У ході дослідження систематизовано підхід до проєктування та реалізації двовимірного платформера з бічним скролінгом засобами рушія Godot. Запропонована архітектура проєкту, що ґрунтується на ієрархії вузлів і сцен із використанням мови GDScript, забезпечує модульність, легкість розширення функціоналу та ефективну взаємодію між ігровими підсистемами - фізикою, анімацією, камерою та інтерфейсом - без надмірного ускладнення кодової бази.

Практична значущість одержаних результатів. Розроблений платформер з бічним скролінгом є повноцінним ігровим проєктом, що може слугувати методичною основою для вивчення принципів геймдеву на рушії Godot. Сформована архітектура та реалізовані ігрові підсистеми можуть бути адаптовані й розширені для створення більш масштабних ігрових продуктів, а отримані результати - використані у навчальному процесі підготовки фахівців у галузі розробки програмного забезпечення.

Апробація результатів та публікації. Основні положення і результати бакалаврської роботи доповідались на науково-практичних конференціях, що проходили на базі Державного університету інформаційно-комунікаційних технологій:

1. Ситник Є.О. - «ЦИФРОВИЙ ДВІЙНИК МІСТА: ВІД РЕАКТИВНОГО ГАСІННЯ ПОЖЕЖ ДО ПРОАКТИВНОГО МОДЕЛЮВАННЯ». Тези доповіді

на VII Міжнародній науково-технічній конференції «Сучасний стан та перспективи розвитку IoT»

2. Ситник Є.О. - «ЗАСТОСУВАННЯ ІГРОВОГО РУШІЯ GODOT ENGINE ДЛЯ РОЗРОБКИ 2D-ПЛАТФОРМЕРА З БІЧНИМ СКРОЛІНГОМ». Тези доповіді на VII Всеукраїнській науково-технічній конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях»

1. ТЕОРЕТИЧНІ ОСНОВИ РОЗРОБКИ ДВОВИМІРНИХ ПЛАТФОРМЕРІВ З БІЧНИМ СКРОЛІНГОМ

1.1. Ігри жанру платформер: поняття, класифікація та еволюція жанру

Платформер - це жанр відеоігор, у якому основою геймплею є переміщення керованого персонажа по розташованих на різній висоті поверхнях, як правило із застосуванням стрибків для подолання прірв, перешкод чи ворогів. Сама назва жанру походить від слова «платформа», адже саме платформи різного типу (нерухомі, рухомі, ті, що руйнуються, або такі, що зникають за певний час) формують основу ігрового простору.

Виникнення платформерів датується початком 1980-х років. Прийнято вважати, що першим повноцінним представником жанру стала гра Donkey Kong (Nintendo, 1981)[2], де персонаж переміщувався вгору по конструкції, перестрибуючи бочки. Згодом, з появою бічного скролінгу, жанр здійснив значний крок уперед, що знайшло відображення у грі Super Mario Bros (1985), яка фактично сформувала канон жанру і стала еталоном на десятиліття[3]. Поява таких ігор, як Sonic the Hedgehog, Castlevania та Mega Man, продемонструвала, що в межах одного жанру можуть співіснувати дуже різні підходи до темпу, складності та подачі сюжету.

Класифікувати платформери можна за кількома ознаками. За напрямком скролінгу виділяють ігри з бічним (горизонтальним) скролінгом, з вертикальним скролінгом, з вільним переміщенням камери у двох напрямках, а також одноекранні платформери, у яких рівень повністю поміщається на одному екрані. За способом подачі графіки розрізняють 2D-платформери, 2.5D (псевдотривимірні, де ігрова механіка лишається двовимірною, а зображення тривимірне) та повноцінні 3D-платформери. Окремо у багатьох класифікаціях виділяють метроідванії, тобто платформери з нелінійним великим світом і поступовим відкриванням нових здібностей персонажа, екшн-платформери з

акцентом на бойову систему, та паззл-платформери, де основою є вирішення головоломок під час руху(Рис. 1.1).

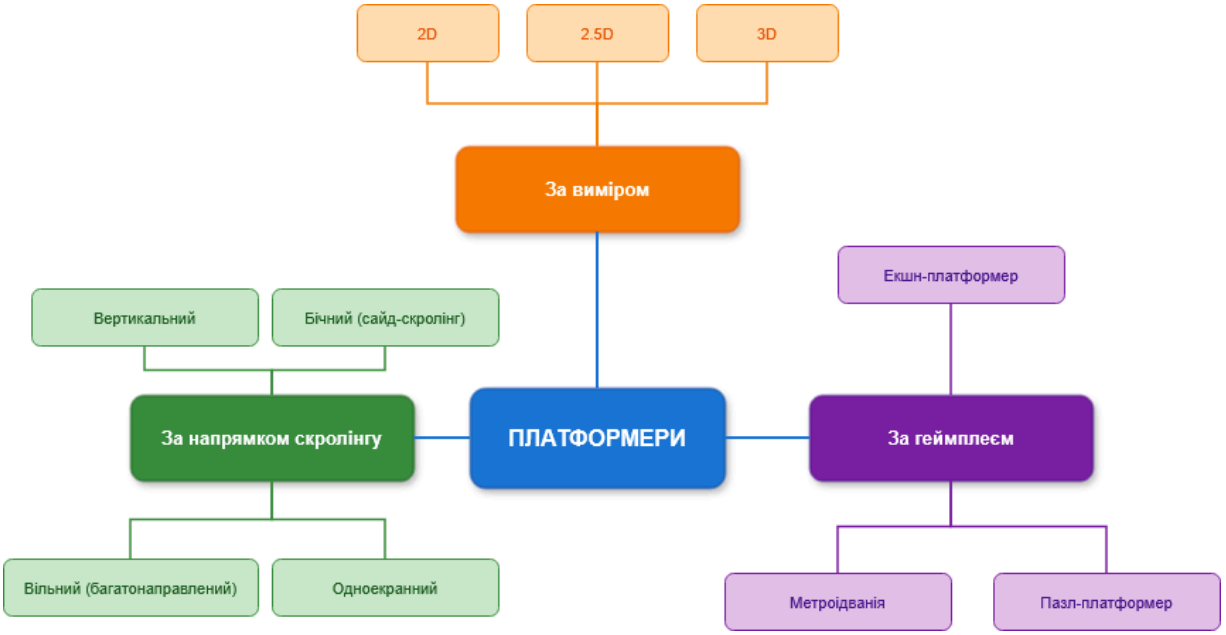

Рис. 1.1 - Класифікація платформерів за напрямком скролінгу та виміром

Еволюція жанру тісно пов'язана з технічним прогресом. У ранній період (1980-ті) платформери розроблялись з урахуванням обмежень аркадних автоматів і консолей того часу: невелика палітра кольорів, спрайтова графіка, прості рівні. У 1990-х роках, із розвитком 16-бітних консолей, з'явились складніші рівні, паралакс-фон, динамічна музика. Перехід до 3D у кінці 1990-х (Super Mario 64, Crash Bandicoot) на час відсунув класичну формулу на другий план, проте у 2010-х роках, із розквітом інді-розробки та поверненням моди на ретро-естетику, 2D-платформери з бічним скролінгом знову стали одним із найпопулярніших жанрів. Сучасні представники, такі як Celeste, Hollow Knight, Shovel Knight, Ori and the Blind Forest, демонструють, що цей жанр здатен поєднати класичну механіку з сучасним рівнем художньої та технічної реалізації[6](Рис. 1.2).

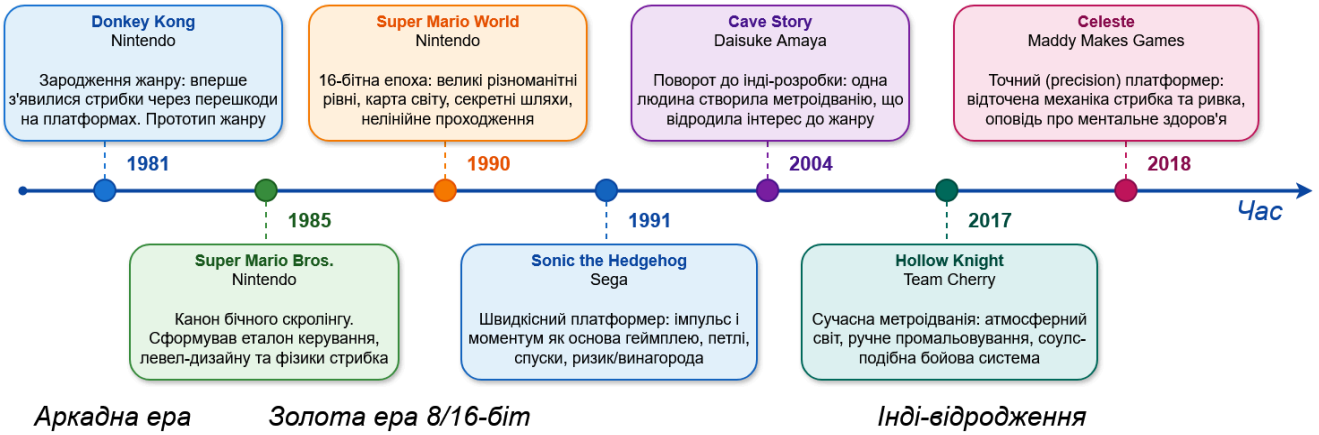

Рис. 1.2 - Часова шкала еволюції жанру платформер

Таким чином, попри понад сорокарічну історію, жанр платформера зберігає свою актуальність, постійно адаптуючись до нових технологій та смаків аудиторії.

1.2. Ключові ігрові механіки платформерів з бічним скролінгом

Хоча візуально платформери з бічним скролінгом сприймаються як прості ігри, у їх основі лежить досить великий набір тісно пов'язаних механік. Якість гри визначається не стільки кількістю цих механік, скільки тим, наскільки добре вони відчуваються при керуванні, тобто наскільки персонаж є «живим» в руках гравця.

Базовою механікою є переміщення персонажа по горизонталі. Звичайно реалізується керування за допомогою клавіш зі стрілками або WASD. Важливим параметром є не лише максимальна швидкість руху, а й те, як швидко персонаж її набирає і як швидко зупиняється. Якщо персонаж миттєво досягає максимальної швидкості, керування відчувається жорстким; якщо ж розгін занадто повільний, гра здається сонною. У більшості якісних платформерів застосовується невелика інерція, що робить рух природним.

Стрибок є, мабуть, найважливішою механікою жанру. Простий стрибок описується початковою вертикальною швидкістю, після чого на персонажа діє сила тяжіння. Однак у дійсності реалізація стрибка набагато складніша. У сучасних платформерах часто використовуються прийоми, які покращують відчуття керування[7]: змінна висота стрибка (чим довше утримується клавіша,

тим вище стрибок)(Рис. 1.3), койот-тайм[4] (можливість стрибнути ще кілька кадрів після того, як персонаж зійшов з краю платформи), буферизація вводу[5] (якщо гравець натиснув стрибок трохи раніше приземлення, стрибок виконується автоматично відразу після торкання землі). Ці прийоми компенсують неточність людської реакції та роблять гру справедливішою.

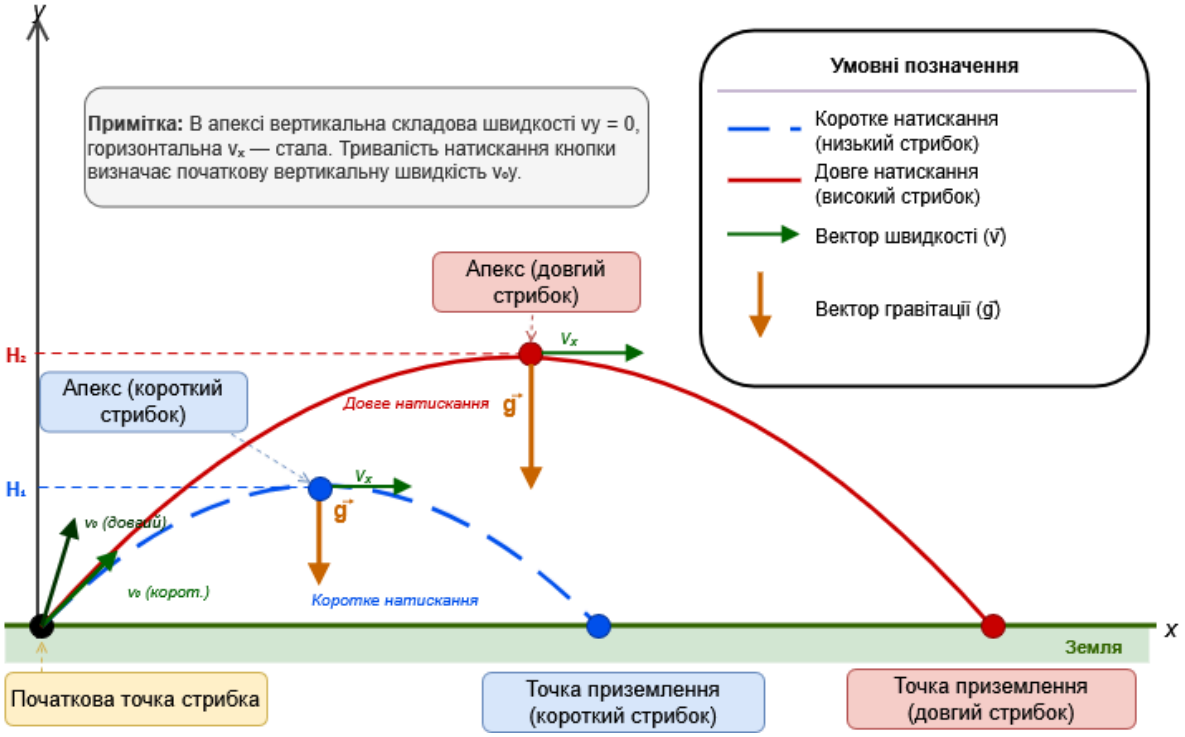

Рис. 1.3 - Діаграма траєкторії стрибка персонажа

Фізика у платформері рідко є реалістичною[6]. Вона аркадна, тобто такою, яка добре відчувається, навіть якщо порушує закони фізики. Так, сила тяжіння у Super Mario приблизно у три рази більша за земну, а максимальна швидкість падіння обмежена штучно, щоб персонаж не «провалювався» крізь тонкі платформи. Тертя та опір повітря також умовні і налаштовуються виключно під відчуття керування.

Система колізій відповідає за визначення зіткнень персонажа з оточенням. Найчастіше колізії реалізуються через прямокутні (AABB) або багатокутні форми, прив'язані до спрайтів. Окрема увага приділяється напрямку зіткнення: персонаж, який зіткнувся з блоком знизу, повинен зупинитись і впасти, а той, що зіткнувся

збоку, лише втратити горизонтальну швидкість. Помилки у логіці колізій є однією з найчастіших причин «дивних» багів у платформерах.

Камера у платформері з бічним скролінгом виконує функцію не лише слідкування за персонажем, а й створення відчуття простору. У більшості ігор камера дещо випереджає рух гравця у горизонтальному напрямку, щоб він бачив, що його чекає попереду. Часто застосовується згладжування руху камери (camera smoothing) і так звані «мертві зони» (dead zone), у межах яких камера не реагує на дрібні переміщення персонажа. Для створення глибини використовується паралакс-фон: задні шари переміщуються повільніше, ніж передні, що імітує перспективу.

Допоміжними механіками, які присутні майже в кожному платформері, є збирання предметів (монет, кристалів, бонусів), система очок та життів, чекпоїнти (точки збереження прогресу в межах рівня), а також взаємодія з ворогами. Вороги можуть діяти за простими алгоритмами: патрулювання між двома точками, переслідування персонажа в межах певної дистанції, атаки з певною періодичністю. Складніші ігри додають елементи штучного інтелекту, проте для класичних платформерів цього часто не вимагається.

Усі перелічені механіки повинні бути узгоджені між собою. Наприклад, висота стрибка має дозволяти подолати типову прірву на рівні плюс певний запас; швидкість руху повинна відповідати темпу появи перешкод; розмір персонажа має узгоджуватись із розміром тайлів. Саме ця узгодженість і робить платформер приємним у грі.

1.3. Огляд сучасних ігрових рушіїв для розробки 2D-ігор

Сучасна розробка ігор практично завжди відбувається із застосуванням ігрового рушія - спеціалізованого програмного середовища, яке бере на себе вирішення типових задач: візуалізацію графіки, обробку фізики, керування ресурсами, обробку введення з пристроїв, програвання звуку. Завдяки рушієві розробник може зосередитись на ігровому дизайні та логіці, не витрачаючи місяці

на написання низькорівневого коду. На ринку існує велика кількість рушіїв, орієнтованих як на 2D, так і на 3D, як комерційних, так і з відкритим вихідним кодом.

Unity є одним із найвідоміших ігрових рушіїв у світі. Він підтримує і 2D, і 3D, має потужну спільноту, велику бібліотеку готових ассетів та плагінів у Asset Store, документацію, велику кількість навчальних матеріалів. Основна мова програмування - C#. Серед недоліків можна назвати відносно великий розмір збірок, не зовсім прозору модель ліцензування (особливо після зміни умов у 2023 році)[8], а також надлишковість можливостей для невеликих 2D-проєктів. Крім того, Unity не є рушієм з відкритим вихідним кодом, і користувач не має повного контролю над тим, що відбувається «під капотом».

Unreal Engine - другий за популярністю рушій. Він орієнтований переважно на високобюджетні 3D-проєкти і відомий своєю якістю графіки. Для 2D-розробки Unreal Engine також придатний (існує підсистема Paper2D), однак на практиці використання цього рушія для невеликих 2D-платформерів є нераціональним: складність налаштування проєкту, великий розмір збірок і вимоги до апаратного забезпечення не відповідають масштабам задачі.

GameMaker Studio - рушій, який багато років вважався фактичним стандартом для 2D-ігор. Він має простий вбудований редактор спрайтів, рівнів, тайлів, власну скриптову мову GML, і добре підходить для початківців. Серед недоліків - закрита екосистема, обмежена гнучкість у складніших проєктах та платна ліцензія для повноцінного використання.

Construct - візуальний рушій, орієнтований на тих, хто не має навичок програмування. Замість коду використовується система подій. Це робить його зручним для прототипування, але обмежує можливості при реалізації нетипових механік.

Phaser - бібліотека для розробки браузерних ігор на JavaScript. Її основна перевага - орієнтованість на веб і можливість запуску гри без встановлення. Однак Phaser не є рушієм у класичному розумінні: у ньому немає графічного редактора, рівні та сцени створюються кодом, що збільшує час розробки.

Godot - молодий, але швидко зростаючий ігровий рушій з відкритим вихідним кодом[9], розповсюджується за ліцензією MIT[10]. Він підтримує 2D і 3D, має невеликий розмір (близько 100 МБ), власну скриптову мову GDScript, підтримку C#, GDExtension для C++. Godot активно розвивається спільнотою і є повністю безкоштовним, без роялті та обмежень. Порівняння наведено у таблиці 1.1.

Таблиця 1.1 Порівняння ігрових рушіїв за основними характеристиками

Назва рушія	Ліцензія	Мова програмування	Підтримка 2D	Розмір рушія
Unity	Proprietary / Runtime Fee	C#	Повна	~1-2 ГБ
Unreal Engine	Proprietary (Royalty 5%)	C++/Blueprints	Часткова	~10-50 ГБ
GameMaker	Proprietary	GML/GameMaker	Відмінна	~300 МБ
Construct	Proprietary	JavaScript/No-code	Відмінна	~200 МБ (веб)
Phaser	MIT	JavaScript/TypeScript	Відмінна	~1 МБ
Godot	MIT	GDScript/C#/C++	Відмінна	~65-100 МБ

З огляду на специфіку поставленої у роботі задачі (розробка невеликого 2D-платформера з бічним скролінгом), очевидно, що використання Unity або Unreal було б надлишковим. GameMaker та Construct мають обмеження ліцензування і екосистеми. Godot, навпаки, поєднує всі необхідні можливості, не накладаючи комерційних обмежень, і має нативну підтримку 2D. Це і обумовлює його вибір для подальшого дослідження.

1.4. Ігровий рушій Godot: архітектура, система вузлів і сцен

Godot - це безкоштовний кросплатформний ігровий рушій з відкритим вихідним кодом, перша версія якого з'явилася у 2014 році. Розробку розпочав аргентинський програміст Хуан Лінєцкі ще в 2001 році для внутрішніх потреб

студії, проте у 2014 році проєкт був відкритий під ліцензією MIT[11]. На момент написання роботи актуальною є гілка Godot 4.x, у якій було повністю переписано рендерер на основі Vulkan, додано підтримку сучасних графічних можливостей, а також значно оновлено інтерфейс редактора.

Архітектурно Godot побудований на двох основних концепціях: вузол (Node) і сцена (Scene). Вузол - це базовий елемент гри, що відповідає за певну функцію: відображення спрайту, програвання звуку, обробку фізики, керування камерою тощо. Кожен вузол є об'єктом певного класу, який успадковує базовий клас Node. Усі вузли проєкту організовані у дерево, у якому є один корінь і будь-яка кількість дочірніх вузлів. Така ієрархія дозволяє природно описувати складні об'єкти: наприклад, персонаж може бути представлений вузлом CharacterBody2D, до якого приєднано вузли Sprite2D (для зображення), CollisionShape2D (для форми колізії), AnimationPlayer (для анімації), Camera2D (для камери, що слідкує)(Рис. 1.4).

Рис. 1.4 - Приклад ієрархії вузлів сцени гравця у Godot

Сцена у Godot - це збережене у файл (.tscn або .scn) дерево вузлів. Будь-яка сцена може бути використана як самостійна сцена, як підсцена іншої сцени або як шаблон для створення багатьох екземплярів. Це фактично замінює концепцію префабів, поширену в інших рушіях. Наприклад, сцена ворога створюється один раз, а потім може бути розміщена на рівні у скільки завгодно екземплярах, кожен з яких є незалежним.

Передача інформації між вузлами у Godot реалізована через систему сигналів (signals). Сигнал - це повідомлення, яке вузол може випустити при настанні певної події (наприклад, зіткнення, натискання кнопки, завершення анімації). Інші вузли можуть підписатися на цей сигнал і виконати свій код у відповідь. Така архітектура зменшує жорсткі зв'язки між компонентами і робить код гнучкішим.

Окремо варто зазначити вбудовану в Godot фізичну систему. Для 2D-фізики передбачено три основні типи фізичних тіл: StaticBody2D (нерухомі тіла, наприклад, стіни), RigidBody2D (тіла, що рухаються під впливом сил), CharacterBody2D (тіла, керовані вручну, призначені для персонажів)[14]. Останній є оптимальним для платформерів, оскільки дає повний контроль над рухом, але водночас бере на себе обробку колізій з оточенням.

Редактор Godot має модульну будову. Окрім стандартного вікна сцени, у ньому є вбудовані інструменти для роботи з анімаціями (AnimationPlayer), тайловими картами (TileMap)[13], частинками, графічними шейдерами(Рис. 1.5). Скриптовий редактор підтримує підсвічування синтаксису, автодоповнення, інтерактивну документацію.

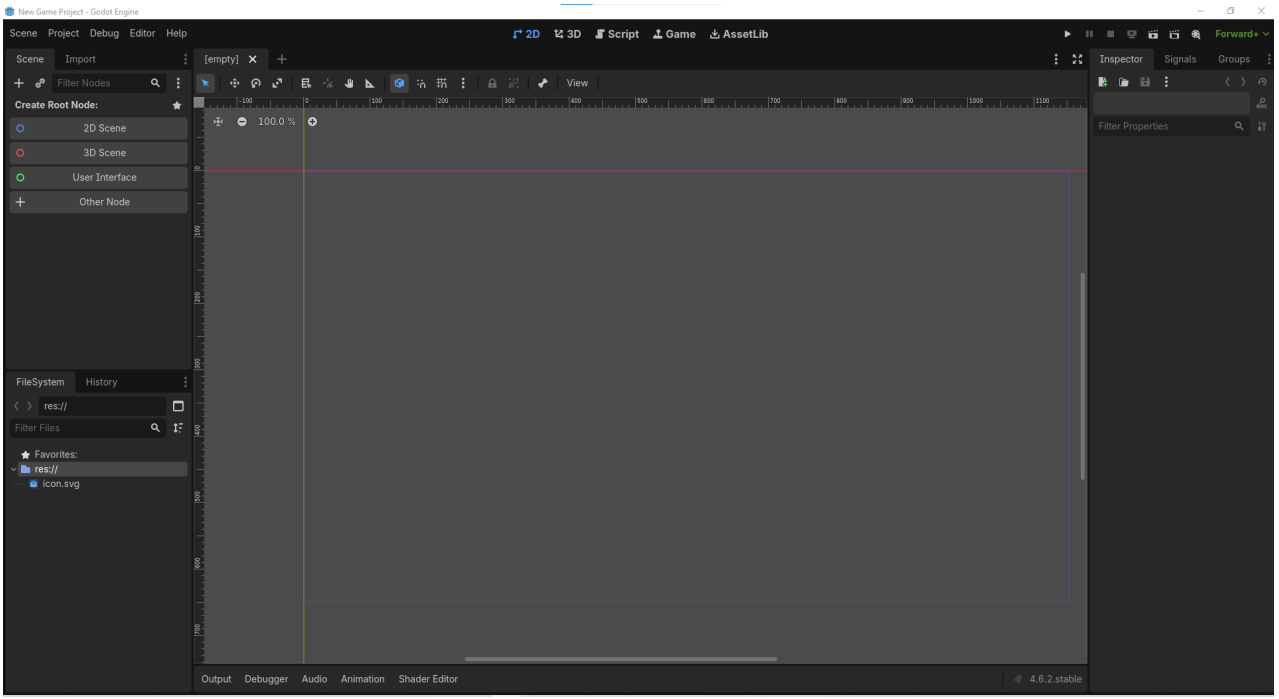

Рис. 1.5 - Інтерфейс редактора Godot 4

Godot будує гру у вигляді виконуваного файлу для цільової платформи. Підтримуються Windows, Linux, macOS, Android, iOS, веб (HTML5), а також консолі через сторонні рішення. Розмір зібраної гри загалом невеликий, що робить рушій придатним і для веб-розгортання, і для мобільних платформ.

Сукупність цих особливостей - відкритість, легкість, орієнтованість на 2D, гнучка архітектура - робить Godot привабливим вибором як для початківців, так і для досвідчених розробників невеликих та середніх проєктів.

1.5. Мова програмування GDScript як інструмент реалізації ігрової логіки

GDScript - це власна скриптова мова рушія Godot, розроблена спеціально для нього і тісно інтегрована з його архітектурою. Синтаксично GDScript нагадує Python: він використовує відступи для позначення блоків коду, динамічну типізацію (з можливістю статичної), пайтонівські ключові слова на кшталт def (хоча у GDScript це func), if, else, for, while. Така подібність робить мову інтуїтивно зрозумілою для тих, хто вже працював з Python, і простою для вивчення новачкам[12].

Основна перевага GDScript - максимально природна інтеграція з рушієм. Усі вузли, сигнали, ресурси проєкту доступні безпосередньо з коду, без необхідності використання обгорток, додаткових бібліотек або складних механізмів зв'язування. Звертання до дочірнього вузла виглядає лаконічно (наприклад, $Sprite2D або get_node("Sprite2D")), підключення до сигналу виконується одним рядком коду або через інспектор, властивості вузлів автоматично з'являються в автодоповненні.

Кожен скрипт у Godot прив'язується до конкретного вузла і розширює його поведінку. У скрипті є набір спеціальних функцій, які викликаються рушієм автоматично. Найважливіші з них: _ready() - викликається один раз після того, як вузол додано у сцену та готовий до роботи; _process(delta) - викликається щокадру, призначена для логіки, не пов'язаної з фізикою; _physics_process(delta) - викликається з фіксованою частотою (за замовчуванням 60 разів на секунду), використовується для коду, що працює з фізикою і рухом; _input(event) -

викликається при отриманні події введення. На рисунку 1.6 наведено приклад скрипту керування персонажем, реалізованого в рамках розробленої гри.

ame character_body_2d

Debug JacksonMoves.gd

```gdscript
extends CharacterBody2D
@onready var animated_sprite_2d: AnimatedSprite2D = $AnimatedSprite2D


const SPEED = 300.0
const JUMP_VELOCITY = -400.0

func _ready():
    var label: Label = $Label

    label.text = GlobalDataStore.char_name

func _physics_process(delta: float) -> void:
    if not is_on_floor():
        velocity += get_gravity() * delta

    if Input.is_action_just_pressed("ui_accept") and is_on_floor():
        velocity.y = JUMP_VELOCITY

    var direction := Input.get_axis("Walk_Left", "Walk_Right")
    if direction:
        if direction < 0:
            animated_sprite_2d.play("Walk_Left")
        elif direction > 0:
            animated_sprite_2d.play("Walk_Right")
        velocity.x = direction * SPEED
    else:
        animated_sprite_2d.play("default")
        velocity.x = move_toward(velocity.x, 0, SPEED)

    move_and_slide()
```

Рис. 1.6 - Приклад скрипта на GDScript для керування персонажем

Параметр delta у функціях _process і _physics_process - це час, який пройшов з попереднього виклику. Множення на delta при розрахунках руху забезпечує однакову швидкість гри на пристроях з різною продуктивністю. Якщо ж кадр

повторюється рідше, кожне переміщення стає більшим, і персонаж рухається з тією самою швидкістю.

GDScript підтримує об'єктно-орієнтоване програмування: можна оголошувати класи, успадковуватись від існуючих класів рушія або своїх власних, перевизначати методи, користуватись модифікаторами доступу. Хоча типи змінних можна не вказувати, у новіших версіях GDScript рекомендується використовувати статичну типізацію (var speed: float = 100.0), оскільки це покращує читаність коду, дозволяє редактору ловити помилки на етапі написання та підвищує продуктивність.

Окрім GDScript, у Godot можна писати на C#, а також підключати модулі на C++ через механізм GDExtension. Однак для невеликих та середніх 2D-проєктів GDScript є оптимальним вибором з кількох причин. По-перше, він не потребує компіляції, що прискорює цикл розробки. По-друге, він простіший для опанування. По-третє, його тісна інтеграція з рушієм означає, що багато речей реалізуються меншою кількістю коду, ніж аналогічна логіка на C#. Знизити продуктивність GDScript може у дуже навантажених проєктах із складними обчисленнями, проте у платформерах це практично не зустрічається.

Сама мова продовжує розвиватись разом із рушієм. У версії 4.x GDScript отримав підтримку лямбда-функцій, корутин на основі ключового слова await, покращену систему типів та інші можливості. Це робить GDScript повноцінним інструментом для реалізації як простої, так і складної ігрової логіки, що повністю відповідає задачам, поставленим у даній кваліфікаційній роботі.

2. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ТА ПРОЄКТУВАННЯ АРХІТЕКТУРИ ПЛАТФОРМЕРА

2.1. Порівняльний аналіз існуючих платформерів з бічним скролінгом

Перш ніж приступати до проєктування власної гри, доцільно проаналізувати кілька представників жанру, які вже довели свою якість і популярність. Такий аналіз дає змогу побачити, які механіки є базовими, а які виступають родзинкою конкретного проєкту, які підходи до дизайну рівнів та подачі складності вважаються вдалими, а які навпаки залишають у гравців негативні враження. Для аналізу обрано чотири гри різного періоду і стилю: Super Mario Bros, Celeste, Hollow Knight, Shovel Knight(Рис. 2.1).

Super Mario Bros (Nintendo, 1985) є фактично еталоном жанру. Геймплей побудовано на простих, але добре відточених механіках: переміщення вліво-вправо, стрибок змінної висоти, можливість стрибнути на ворога зверху. Рівні мають лінійну структуру, поступово вводять нові елементи (труби, прірви, рухомі платформи, ворогів різних типів), мають чітку часову мету. Графіка спрайтова, типова для NES. Особливість цієї гри полягає у бездоганному балансі складності та у тому, як дизайн рівнів навчає гравця механік без єдиного навчального екрану.

Celeste (Maddy Makes Games, 2018) - сучасний платформер з акцентом на точність керування. До базових механік додано повітряний ривок (dash), здатність хвататись за стіни і драпатися по них. Гра славиться надзвичайно точним керуванням: координати персонажа обчислюються з точністю до субпікселя, що робить можливими дуже складні маневри. Рівні короткі, поділені на окремі кімнати, при загибелі гравець миттєво відроджується на початку кімнати, що дозволяє безболісно експериментувати. Графіка піксельна, але стилізована, з виразною колірною палітрою. Окремо відзначається рівень аудіо- та сюжетного супроводу[6].

Hollow Knight (Team Cherry, 2017) є представником піджанру метроідванія. Світ гри відкритий, поділений на регіони, доступ до більшості з яких відкривається поступово - у міру отримання нових здібностей персонажа (стіну подвійного стрибка, занурення, телепортації). Бойова система розвинена: персонаж має кілька типів атаки, заклинання, систему «душ» (валюта для лікування і магії). Графіка намальована вручну, у виразному темному стилі. Hollow Knight цікавий тим, як він поєднує класичні платформерні механіки з елементами екшн-RPG.

Shovel Knight (Yacht Club Games, 2014) свідомо стилізований під ери 8-бітних консолей, навіть зберігаючи відповідні обмеження палітри. Геймплей нагадує суміш Mega Man і Castlevania: персонаж атакує лопатою, може використовувати її як трамплін, стрибаючи на ворогів та об'єктів зверху. Гра має нелінійну карту з вибором послідовності проходження рівнів, кожен рівень закінчується боєм з босом. Цікаво, що Shovel Knight показує, як ретро-естетика може органічно поєднуватись із сучасним дизайном рівнів та удосконаленнями зручності керування.

Рис. 2.1 - Скріншоти аналізованих платформерів

Порівняння аналізованих ігор за низкою критеріїв подано у таблиці 2.1.

Таблиця 2.1 Порівняння обраних платформерів

Гра	**Тип рівнів**	**Камера**	**Стиль графіки**	**Особливі механіки**	**Складність**
Super Mario Bros	Лінійні	Бічна, з випередженням	Спрайтова, 8-біт	Стрибок на ворога	Помірна
Celeste	Кімнати	Бічна, перехід між кімнатами	Піксельна, сучасна	Ривок, лазіння по стінах	Висока
Hollow Knight	Відкритий світ	Вільна	Намальована	Подвійний стрибок, бойова система	Висока
Shovel Knight	Лінійні з вибором	Бічна	Спрайтова, ретро	Стрибок-удар, бойова система	Помірна

На основі проведеного аналізу для власного проєкту обрано підхід, близький до класичного: лінійні рівні з бічним скролінгом, набір базових механік (рух, стрибок, збирання предметів, врахування зіткнень з небезпеками), спрайтова графіка. Складніших елементів на кшталт подвійного стрибка, ривка чи бойової системи у мінімальній версії гри не передбачено, проте архітектура повинна дозволяти додати їх у майбутньому без переписування коду.

2.2. Порівняльний аналіз інструментів розробки 2D-платформерів

У підрозділі 1.3 був наданий загальний огляд ігрових рушіїв. Тепер же необхідно провести детальніше порівняння саме з точки зору розробки 2D-платформера, оскільки загальна оцінка можливостей рушія і його придатність для конкретної задачі - речі різні. У межах цього порівняння враховуються такі

критерії: наявність нативних 2D-можливостей, підтримка тайлових карт, фізика для 2D, інструменти роботи з анімаціями, складність вивчення, ціна та умови ліцензування, активність спільноти і доступність документації.

Unity на момент написання роботи залишається одним із найпопулярніших рушіїв загального призначення. У ньому є підсистема 2D, яка включає спрайтовий рендерер, фізику Box2D, інструмент Tilemap для роботи з тайловими картами, систему анімацій Mecanim. Все це потужне, але реалізоване дещо обхідним шляхом: внутрішньо Unity залишається тривимірним рушієм, у якому 2D - це окремий режим роботи з ортогональною камерою і пласкими спрайтами. Це означає, що базова збірка проєкту все одно тягне за собою повний 3D-стек. Для невеликого 2D-платформера це надлишково. Ціновий аспект також є важливим: безкоштовна версія Unity Personal підходить для непрофесійного використання, але історія зі зміною умов ліцензування у 2023 році показала, що умови можуть змінюватись несподівано.

GameMaker Studio спеціально створений під 2D і має дуже якісну реалізацію основних потреб платформера: вбудований редактор спрайтів і тайлів, проста але достатня фізика, скриптова мова GML, орієнтована саме на ігрову логіку. У ньому історично розроблено величезну кількість успішних 2D-ігор, серед яких Hotline Miami, Undertale, Hyper Light Drifter. Однак екосистема GameMaker є закритою, ліцензії на повноцінне використання платні, а GML як мова менш універсальна, ніж GDScript або C#.

Construct 3 - візуальне середовище, у якому замість програмування використовується система Event Sheets. Це робить його надзвичайно швидким для прототипування простих платформерів, але обмежує гнучкість при реалізації нестандартних механік. Construct орієнтований переважно на освітні цілі та невеликі браузерні ігри. Також він є хмарним сервісом за підпискою, що може не влаштовувати частину розробників.

Phaser - JavaScript-бібліотека для веб-ігор. Її основна перевага - можливість запуску гри прямо в браузері без встановлення. Однак Phaser не має повноцінного редактора: рівні, об'єкти, поведінка - все це описується кодом. Для платформера це

означає, що або потрібно використовувати сторонні редактори (наприклад, Tiled) і завантажувати їх формати, або писати власну систему рівнів. Це збільшує обсяг роботи і ускладнює супровід проєкту.

Godot, як було сказано раніше, є відкритим рушієм. У контексті 2D-розробки він має ряд переваг: 2D-рендерер є нативним, а не побудованим поверх 3D, що дає кращу продуктивність на слабких пристроях; вбудований редактор TileMap (а у Godot 4 - TileMapLayer) дозволяє створювати рівні безпосередньо у редакторі без потреби у сторонніх інструментах; фізична система пропонує спеціальний клас CharacterBody2D з вбудованим методом move_and_slide(), оптимізованим саме для платформерів; AnimationPlayer та AnimatedSprite2D дають дві альтернативні моделі реалізації анімацій. Розмір самого редактора становить близько 100 МБ, він повністю переносний і не потребує встановлення.

Підсумовуючи, можна зазначити: для невеликого 2D-платформера, який розробляється у межах кваліфікаційної роботи, рушій Godot забезпечує оптимальне поєднання можливостей, простоти і умов ліцензування. Жоден з конкурентів не має переваги одночасно за всіма критеріями, проте Godot не поступається їм за жодним з тих, що є важливими саме для цього класу проєктів.

2.3. Обґрунтування вибору ігрового рушія Godot для реалізації проєкту

Хоча у попередньому підрозділі вже сформульовано загальний висновок про доцільність вибору Godot, у цьому підрозділі необхідно навести детальніше обґрунтування саме у контексті поставленої задачі(Рис. 2.2).

По-перше, ліцензійні умови. Godot розповсюджується за ліцензією MIT, що є однією з найвільніших ліцензій з-поміж відомих. Це означає, що рушій можна використовувати без обмежень як для навчальних, так і для комерційних цілей, без сплати роялті, без обов'язку розкривати вихідний код своєї гри. Будь-який проєкт, розроблений у Godot, повністю належить його автору. Це особливо важливо для кваліфікаційної роботи, яка може у майбутньому бути розширена та опублікована.

По-друге, нативна орієнтованість на 2D. У Godot 2D і 3D реалізовані як два паралельні рендерери з власними наборами вузлів і своєю фізикою. Це не «3D з вимкненою глибиною», а саме 2D у природному сенсі: ортогональні координати, цілочислові пікселі, оптимізована підсистема відображення. Для платформера це означає кращу продуктивність і простішу організацію коду.

По-третє, наявність всіх необхідних інструментів «з коробки». У редакторі рушія вже є: TileMap для побудови рівнів з тайлів, AnimationPlayer для анімацій, CharacterBody2D з готовою логікою руху і колізій для персонажів, Camera2D зі згладжуванням та обмеженнями, ParallaxBackground для паралакс-фону, Area2D для зон взаємодії (наприклад, монети або тригери)[13]. Реалізація аналогічної функціональності в інших рушіях часто вимагала б додаткових плагінів або написання власного коду.

По-четверте, простота вивчення мови GDScript. Як було показано у підрозділі 1.5, GDScript за синтаксисом нагадує Python, не вимагає компіляції, надає прямий доступ до всіх можливостей рушія. У межах кваліфікаційної роботи це означає, що значна частина часу буде витрачена не на боротьбу з мовою і її особливостями, а на власне ігрову логіку.

По-п'яте, легкість самого рушія. Інсталяція Godot полягає у завантаженні одного виконуваного файлу розміром приблизно 100 МБ[9] і не потребує реєстрації облікового запису, активації ліцензії чи окремого процесу встановлення. Це робить рушій придатним для роботи на маловимогливому апаратному забезпеченні, що теж є плюсом для студентського проєкту.

По-шосте, активність спільноти. Документація Godot ведеться англійською, частково перекладена іншими мовами, доступна онлайн і у вигляді локального файлу. Активна спільнота на Reddit, Discord, форумі і в YouTube робить пошук відповідей на типові запитання простим. Кількість туторіалів зросла особливо після 2023 року на тлі ліцензійних суперечок навколо Unity.

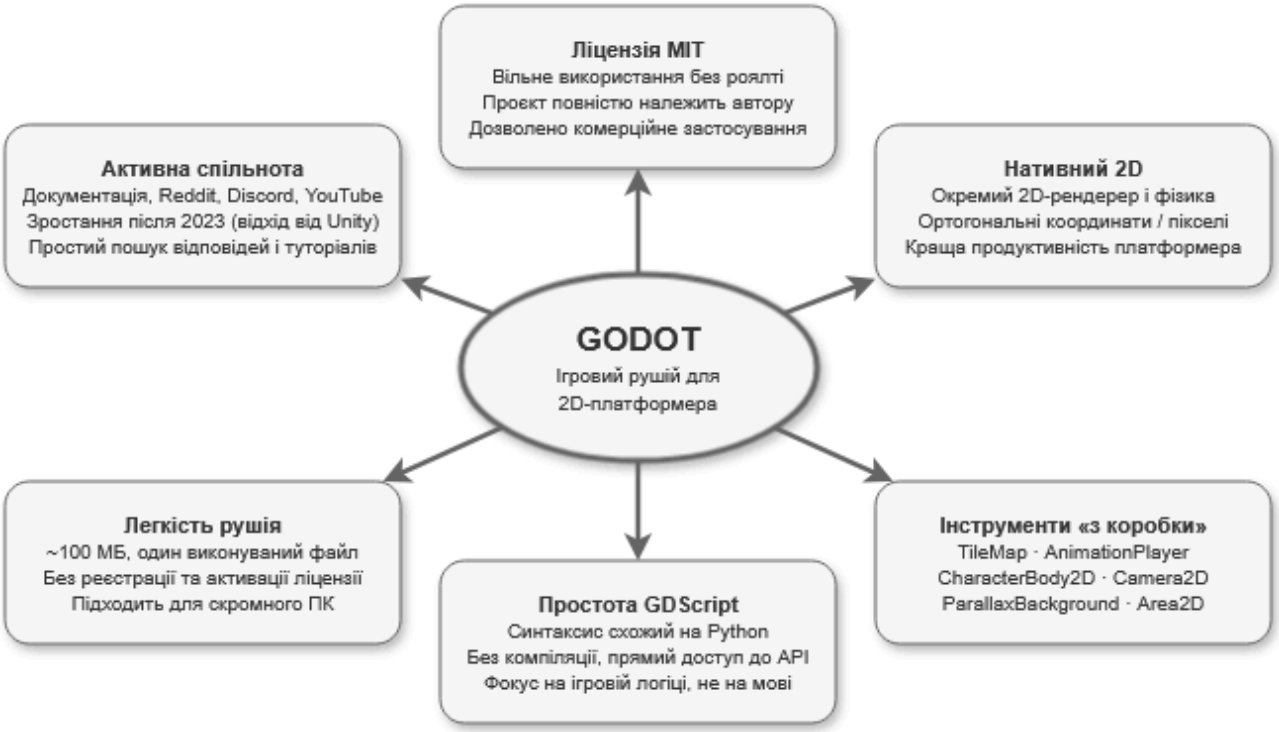

Рис. 2.2 - Зведена діаграма переваг Godot для проєкту

Сукупність наведених аргументів робить вибір Godot не просто прийнятним, а оптимальним для конкретної задачі - розробки 2D-платформера з бічним скролінгом у межах кваліфікаційної роботи бакалавра.

2.4. Проєктування загальної архітектури гри

Перш ніж переходити до реалізації, необхідно спроєктувати загальну архітектуру гри: які сцени будуть створені, як вони взаємодіють між собою, як зберігаються глобальні дані, через які канали відбувається обмін інформацією між компонентами. Добре продумана архітектура економить час на пізніших етапах розробки і значно полегшує внесення змін.

В основі архітектури лежить ієрархія сцен. Головною (стартовою) сценою проєкту є Main, яка завантажує головне меню. З головного меню гравець може перейти до сцени конкретного рівня (Level1, Level2 і так далі), переглянути таблицю результатів або вийти з гри. Кожен рівень є самостійною сценою, у якій присутні: екземпляр сцени гравця (Player), розмічена за допомогою TileMap карта

рівня, екземпляри сцен предметів (монети, серця, бонуси), екземпляри сцен ворогів та небезпек, точка фінішу рівня, а також вузол HUD - інтерфейс гравця.

Сцена гравця (Player.tscn) винесена у самостійний файл і повторно використовується на кожному рівні. Це дозволяє один раз реалізувати логіку керування персонажем, а потім просто додавати готову сцену гравця у будь-який рівень. Аналогічно сцени монети, ворога, рухомої платформи та інших об'єктів проєктуються як окремі багаторазово використовувані сцени.

Глобальні дані гри (ім'я персонажа та рахунок) зберігаються у спеціальному об'єкті GlobalDataStore, реалізованому за допомогою механізму AutoLoad[17]. AutoLoad у Godot - це режим, у якому скрипт автоматично завантажується при запуску і залишається в пам'яті протягом усього сеансу, доступний з будь-якої сцени. Фактично це реалізація патерну Singleton. У базовій версії GlobalDataStore відповідає за зберігання імені персонажа та підрахунок зібраних предметів; розширення до повноцінного менеджера зі станом життів і переходами між рівнями віднесено до перспектив розвитку.

Зв'язок між компонентами реалізовано через сигнали[18]. Наприклад, коли гравець стикається з монетою, сцена монети випускає сигнал collected, на який реагує GlobalDataStore, який збільшує рахунок і повідомляє HUD про необхідність оновлення. Такий підхід забезпечує слабкий зв'язок між компонентами: монета не знає, хто саме на неї підписаний, і не залежить від реалізації HUD або GameManager.

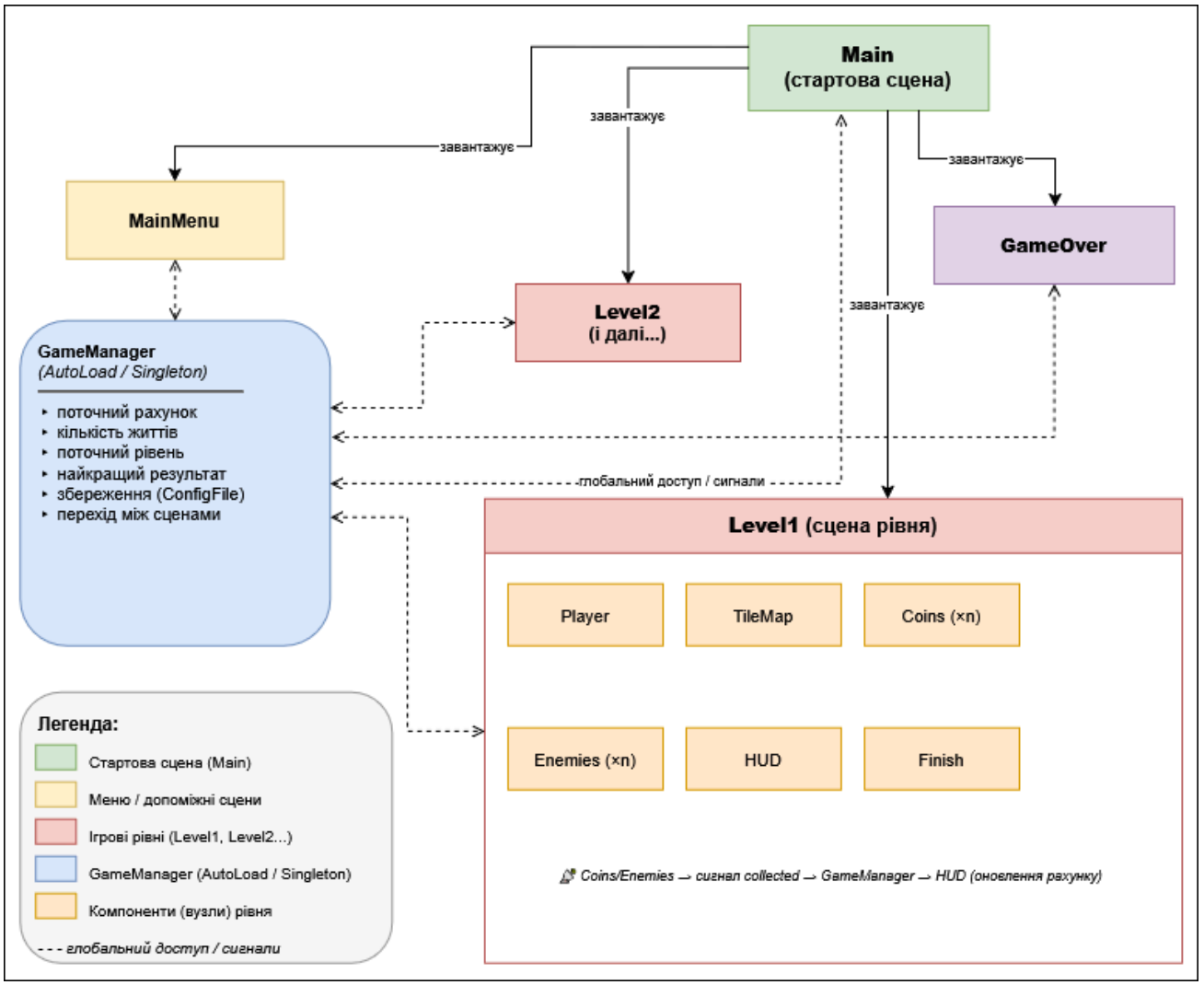

Рис. 2.3 - Загальна архітектура гри у вигляді діаграми сцен

Структура папок проєкту планується наступною: у корені знаходяться папки scenes (усі сцени .tscn), scripts (скрипти .gd), assets (всередині якої - sprites, sounds, music, fonts, tilesets). Файл project.godot з налаштуваннями і файл AutoLoad-скриптів - у корені. Такий поділ полегшує орієнтацію у проєкті та підтримує його впорядкованість при збільшенні кількості файлів.

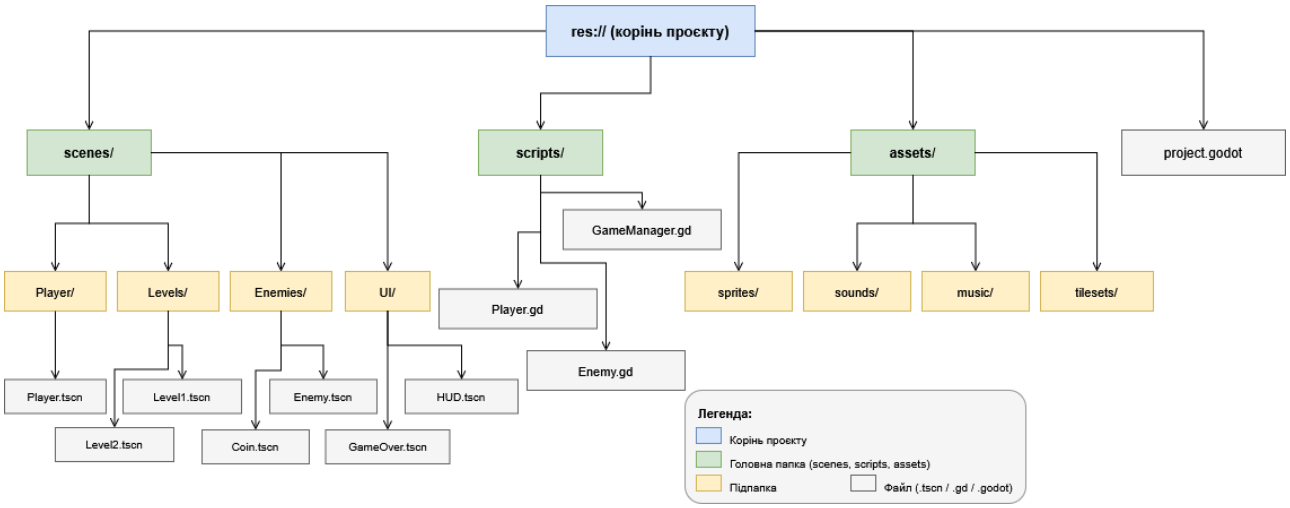

Рис. 2.4 - Структура папок проєкту

Потік виконання гри планується так: запуск рушія завантажує сцену Main, у якій відображається головне меню. При натисканні кнопки «Грати» завантажується сцена першого рівня методом get_tree().change_scene_to_file(). Під час гри стан відстежує GlobalDataStore, а у базовій версії після проходження рівня керування повертається до головного меню. Перехід між кількома рівнями та екран завершення гри передбачені як напрям подальшого розвитку.

Описана архітектура є модульною, що дозволяє у разі необхідності додавати нові рівні, нові типи ворогів, нові механіки без втручання у вже написаний код. Це відповідає принципу відкритості/закритості і робить проєкт придатним для подальшого розвитку.

2.5. Проєктування ігрових рівнів та ігрового інтерфейсу користувача

Проєктування рівнів є тим етапом, на якому теоретичні концепції перетворюються на конкретне ігрове наповнення. У цій роботі планується створити невелику серію рівнів, які поступово вводять гравця у механіки гри і збільшують її складність[20]. Орієнтовно передбачено три рівні, оскільки такої кількості достатньо для демонстрації архітектури, при цьому не виникає надмірного обсягу контенту, з яким складно впоратись у межах однієї роботи.

Перший рівень виконує роль навчального. Він короткий, складається переважно з рівних ділянок, містить кілька монет, одну або дві прості небезпеки, кілька маленьких прірв, які можна перестрибнути базовим стрибком. Завдання цього рівня - дати гравцю звикнути до керування, відчути фізику персонажа, побачити, як працює збирання предметів і відображення рахунку(Рис. 2.5).

Другий рівень додає більше різноманіття: вертикальний рух (підйоми та спуски), кілька рівнів платформ, рухомі платформи, простіші вороги, які патрулюють певну ділянку. Тут уже від гравця потрібна певна вправність, але рівень все ще лишається проходимим без занадто складних маневрів.

Третій рівень виступає як підсумковий: він поєднує всі вивчені раніше елементи, додає більше прірв з гострими краями, більше ворогів, чергує тісні ділянки з відкритими просторами. Закінчується рівень фінішною точкою, що символізує перемогу.

Принципи побудови рівнів, на яких слід наголосити:

- кожна нова механіка вводиться у безпечному середовищі, де гравець може експериментувати без покарання;
- візуальні підказки (наприклад, монети, розкладені вздовж очікуваної траєкторії стрибка) допомагають гравцю інтуїтивно розуміти, що від нього вимагається;
- невдала спроба не повинна означати втрати великого прогресу: контрольні точки розставляються так, щоб повторне проходження ділянки не виснажувало.

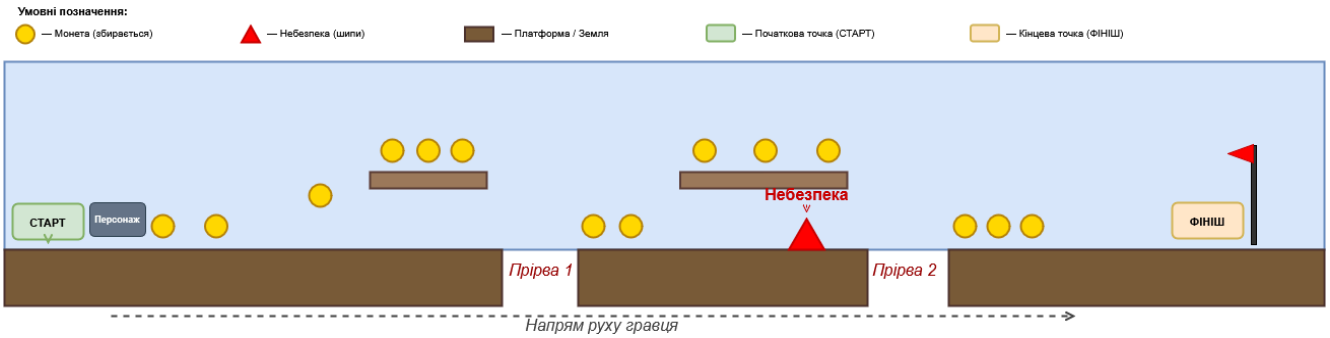

Рис. 2.5 - Концептуальний ескіз першого рівня.

Ігровий інтерфейс користувача (HUD - Heads-Up Display) проєктується мінімалістичним, щоб не відволікати від основного процесу гри. У верхньому лівому куті розміщується індикатор поточного рахунку у форматі «Очки: 0» з оновленням під час гри. У верхньому правому куті відображається кількість життів, що залишились, у вигляді ряду маленьких пікселів-сердець. У центрі верхньої частини за потреби розміщується назва рівня, яка зникає через кілька секунд після початку рівня.

Основні екрани, окрім ігрового, включають(Рис. 2.6):

- головне меню: назва гри, кнопки «Грати», «Налаштування», «Вихід»;
- екран паузи: викликається клавішею Esc, містить кнопки «Продовжити», «Перезапуск рівня», «Головне меню»;
- екран Game Over: повідомлення про втрату життів, фінальний рахунок, кнопки «Спробувати знову», «Головне меню»;
- екран перемоги: повідомлення про проходження останнього рівня, підсумковий рахунок, кнопки «Грати знову», «Головне меню».

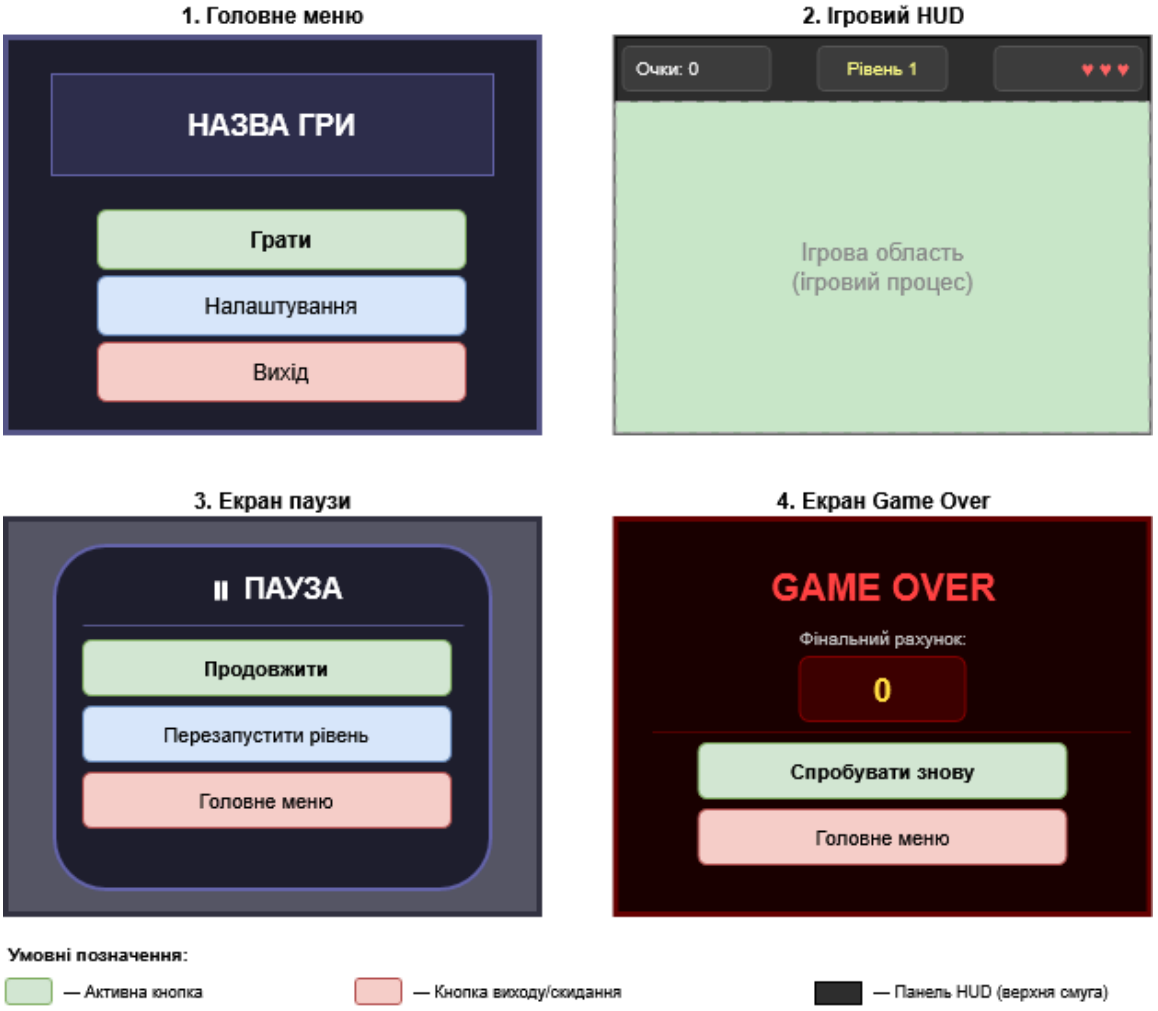

Рис. 2.6 - Wireframe-схеми основних екранів інтерфейсу.

Кольорова палітра інтерфейсу узгоджується з графічним стилем гри. Шрифт обирається моноширинний або стилізований під піксельну графіку, щоб органічно поєднуватись зі спрайтами. Розмір тексту достатній для читабельності на типовому розширенні екрану (1920x1080), при цьому HUD не повинен займати більше ніж 10 відсотків площі екрану.

Окрема увага приділяється навігації клавіатурою. Усі меню повинні підтримувати керування як мишею, так і клавіатурою (стрілки + Enter), оскільки гра в цілому орієнтована на керування з клавіатури і вимагати переходів на мишу між меню і геймплеєм було б незручно для гравця.

Підсумовуючи, спроєктована схема рівнів і інтерфейсу формує цілісне уявлення про те, як виглядатиме готова гра, які екрани побачить гравець і у якій

послідовності, як буде організовано перехід між ними. Це створює надійну основу для переходу до етапу безпосередньої реалізації, описаної у третьому розділі.

3. РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ПЛАТФОРМЕРА З БІЧНИМ СКРОЛІНГОМ

3.1. Налаштування середовища розробки та структури проєкту в Godot

Розробка гри розпочинається з підготовки робочого середовища. Godot завантажується з офіційного сайту як один виконуваний файл, який не потребує встановлення в систему. Для роботи було використано Godot 4.6 (стандартну збірку зі скриптуванням на GDScript, без додаткових залежностей C#). Виконуваний файл рушія розміщено у зручній папці, при першому запуску відкривається менеджер проєктів, у якому створюється новий проєкт.

Під час створення проєкту вказано назву (HalfRealProject), шлях до папки і обрано рендерер Forward+[21]. Для двовимірної гри з невибагливою графікою цей рендерер є дещо надлишковим, і рендерер Compatibility теж повністю задовольнив би потреби проєкту, проте Forward+ обрано як стандартний варіант за замовчуванням, що не створює жодних проблем для 2D і працює стабільно.

Після створення проєкту виконано базові налаштування у вікні Project Settings (меню Project / Project Settings). У розділі Display / Window встановлено базову роздільну здатність 1280 на 720 пікселів. Це не означає, що гра запускатиметься тільки у такому вікні, йдеться про логічну роздільну здатність, до якої прив'язуються позиції елементів. Для піксельної графіки у налаштуваннях імпорту текстур вимкнено фільтрацію (режим Nearest), оскільки інакше спрайти розмиваються при масштабуванні.

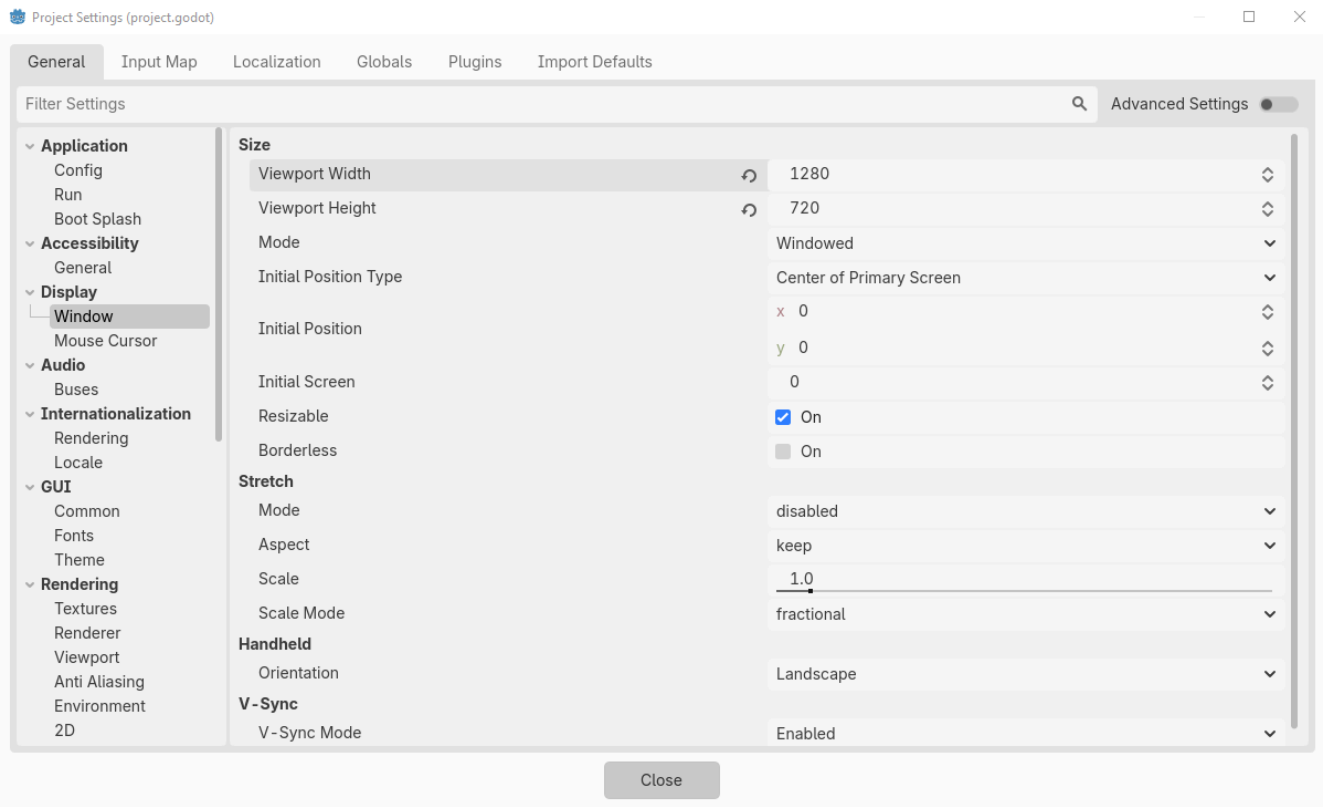

Рис. 3.1 - Вікно Project Settings з обраними параметрами

Налаштування введення виконано у Project Settings / Input Map. Створено чотири дії: move_left (прив'язана до клавіші A), move_right (клавіша D), jump (клавіша Space) та pause (клавіша Escape). Така прив'язка дозволяє у коді звертатись не до конкретних клавіш, а до назв дій, що робить керування легко змінюваним без переписування логіки. Дія pause зарезервована для майбутнього меню паузи і на поточному етапі у коді не задіяна.

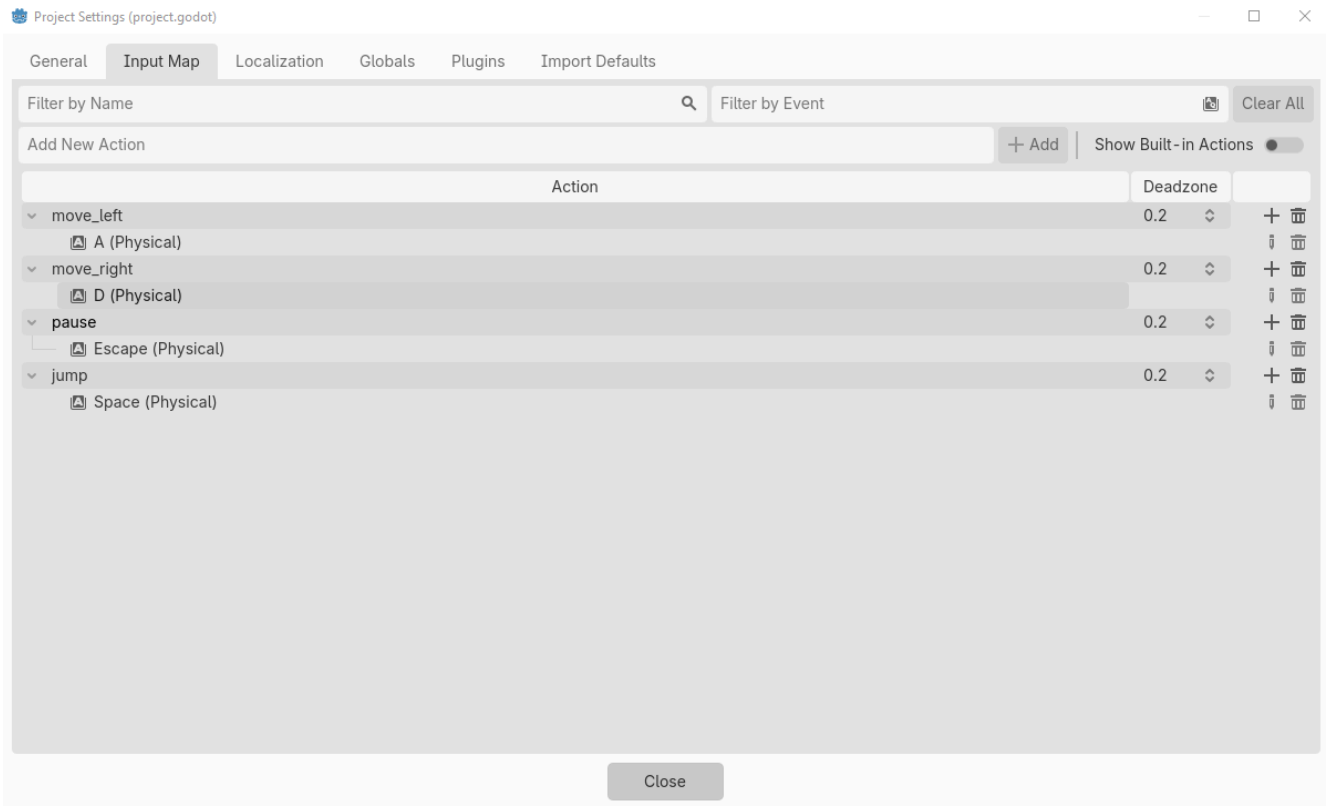

Рис. 3.2 - Вікно Input Map з налаштованими діями.

Глобальний стан гри винесено в окремий скрипт GlobalDataStore.gd, який підключено у режимі AutoLoad (Project Settings / Globals / Autoload). Завдяки цьому об'єкт існує протягом усього сеансу і доступний з будь-якої сцени за іменем. Саме він виконує роль того глобального сінглтона, що був спроєктований у підрозділі 2.4.

Ассети проєкту (спрайт персонажа fsm_ex_chara02.png, набір кадрів монети у папці CUBE, тайлові текстури wafu_A4.png та A.png) розміщено у корені проєкту. При додаванні зображень рушій автоматично створює файли імпорту .import, які зберігають налаштування фільтрації і стиснення для кожного зображення. На цьому підготовка середовища завершена і можна переходити до реалізації основних компонентів гри.

3.2. Реалізація системи керування персонажем та ігрової фізики

Персонаж гравця реалізовано як окрему сцену (character_body_2d.tscn) з кореневим вузлом типу CharacterBody2D[14]. Цей тип обрано тому, що він дає повний контроль над рухом (на відміну від RigidBody2D, який рухається сам під дією сил) і одночасно бере на себе виявлення зіткнень з оточенням, що суттєво спрощує код.

До кореневого вузла прикріплено такі дочірні вузли: AnimatedSprite2D для відображення анімованого спрайту, CollisionShape2D з капсульною формою колізії (CapsuleShape2D), а також Label, який виводить над персонажем його ім'я. У сцені присутні також вузли AnimationPlayer і AnimationTree, зарезервовані для майбутнього змішування анімацій, проте на поточному етапі робоча анімація реалізована через AnimatedSprite2D. Камера (Camera2D) приєднується до персонажа вже у головній ігровій сцені, про що йдеться у підрозділі 3.3.

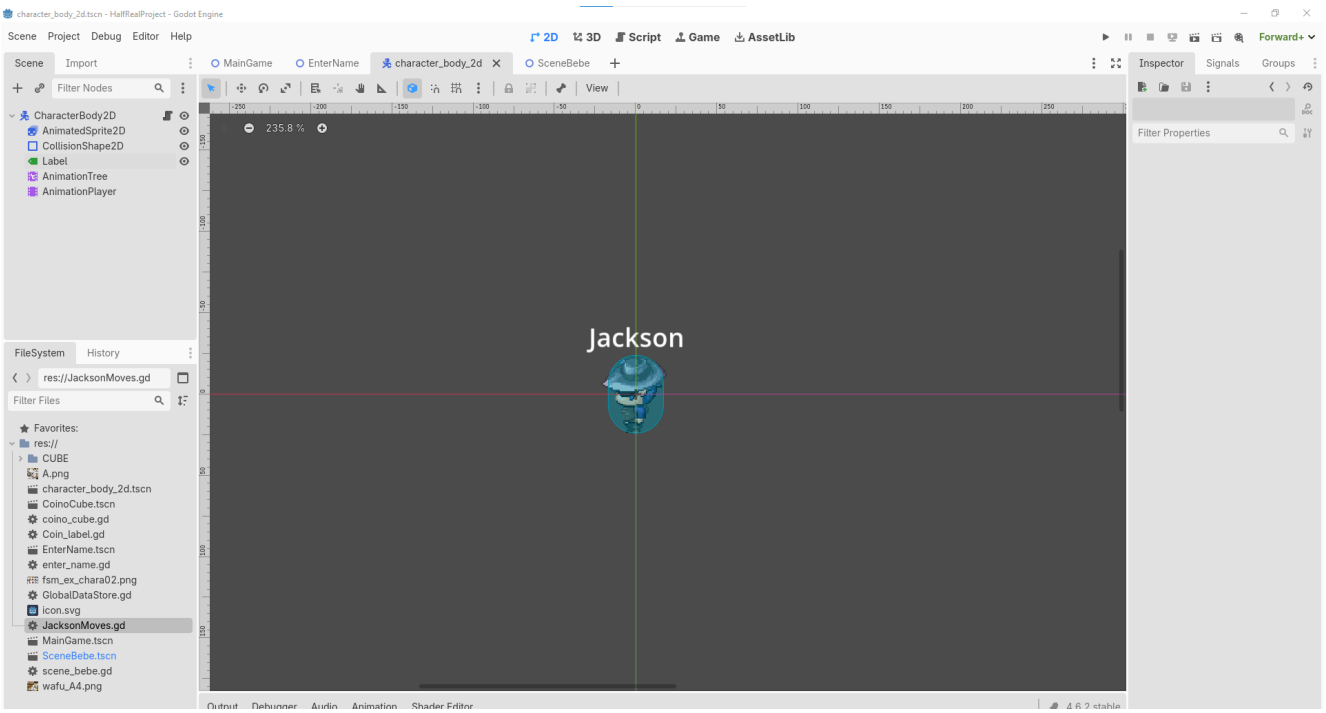

Рис. 3.3 - Структура сцени персонажа у редакторі.

На кореневий вузол прикріплено скрипт JacksonMoves.gd, який реалізує логіку руху. Окрім базового переміщення і стрибка, у нього від початку закладено три прийоми покращення відчуття керування, описані теоретично у підрозділі 1.2:

coyote time, буферизацію стрибка і змінну висоту стрибка. Повний код наведено у лістингу 3.1.

```
extends CharacterBody2D

@onready var animated_sprite_2d: AnimatedSprite2D = $AnimatedSprite2D
@onready var label: Label = $Label

const SPEED := 300.0
const JUMP_VELOCITY := -400.0
const GRAVITY := 980.0
const COYOTE_TIME := 0.1
const JUMP_BUFFER_TIME := 0.1
const JUMP_CUT := 0.5

var coyote_timer := 0.0
var jump_buffer_timer := 0.0

func _ready() -> void:
    add_to_group("player")
    label.text = GlobalDataStore.char_name

func _physics_process(delta: float) -> void:
    if not is_on_floor():
        velocity.y += GRAVITY * delta
        coyote_timer -= delta
    else:
        coyote_timer = COYOTE_TIME

    if Input.is_action_just_pressed("jump"):
        jump_buffer_timer = JUMP_BUFFER_TIME
    else:
        jump_buffer_timer -= delta

    if jump_buffer_timer > 0.0 and coyote_timer > 0.0:
        velocity.y = JUMP_VELOCITY
        coyote_timer = 0.0
        jump_buffer_timer = 0.0

    if Input.is_action_just_released("jump") and velocity.y < 0.0:
        velocity.y *= JUMP_CUT

    var direction := Input.get_axis("move_left", "move_right")
    if direction != 0.0:
        velocity.x = direction * SPEED
    else:
        velocity.x = move_toward(velocity.x, 0.0, SPEED)

    move_and_slide()
    update_animation(direction)

func update_animation(direction: float) -> void:
    if direction < 0.0:
        animated_sprite_2d.play("move_left")
    elif direction > 0.0:
        animated_sprite_2d.play("move_right")
    else:
        animated_sprite_2d.play("default")
```

Лістинг 3.1 - Скрипт керування персонажем JacksonMoves.gd

Розберемо ключові моменти. Константи SPEED, JUMP_VELOCITY та GRAVITY винесено окремо, щоб їх легко було підлаштувати під відчуття гри. JUMP_VELOCITY має від'ємне значення, оскільки у Godot вісь Y направлена вниз: щоб персонаж стрибнув угору, його швидкість по Y має зменшитись.

Функція _physics_process() викликається з фіксованою частотою (60 разів на секунду). Спочатку, якщо персонаж не на землі, до вертикальної швидкості додається складова гравітації, помножена на delta, і зменшується таймер coyote_timer. Якщо персонаж на землі, цей таймер поповнюється до повного значення. Завдяки цьому стрибок дозволений ще протягом 0.1 секунди після того, як персонаж зійшов з краю платформи, що і є ефектом coyote time.

Буферизація стрибка працює через таймер jump_buffer_timer[15]: при натисканні клавіші він встановлюється у максимум, а далі поступово зменшується. Стрибок виконується тоді, коли обидва таймери додатні. Це означає, що навіть якщо гравець натиснув стрибок на частку секунди раніше приземлення, намір не губиться, а спрацьовує одразу після контакту з землею.

Змінна висота стрибка реалізована через перевірку відпускання клавіші: якщо гравець відпустив стрибок, поки персонаж ще рухається вгору, вертикальна швидкість множиться на коефіцієнт JUMP_CUT, тобто підйом вкорочується. Коротке натискання дає низький стрибок, утримання - повний.

Горизонтальний рух отримується через Input.get_axis(), що повертає значення від -1 до 1. У разі руху швидкість встановлюється прямо, у разі відсутності вводу плавно гаситься через move_toward(), що дає невелику інерцію при зупинці[19]. Останнім викликається move_and_slide(), який переміщує тіло і обробляє зіткнення. У _ready() персонаж додається до групи «player» (це знадобиться монетам для його впізнавання) і над ним виводиться ім'я, збережене у глобальному сховищі.

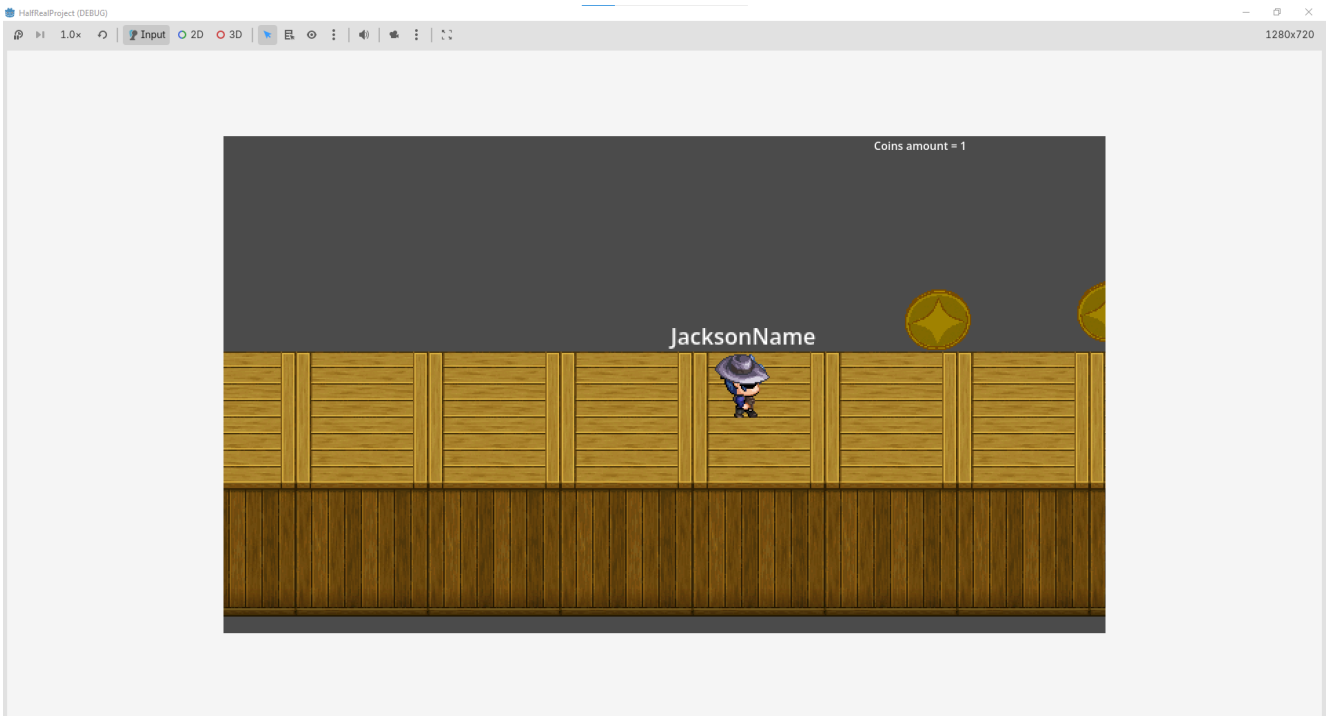

Рис. 3.4 - Скріншот гри: персонаж на рівні з підписаним над ним ім'ям

3.3. Реалізація механіки бічного скролінгу камери

Бічний скролінг камери у Godot реалізується завдяки вузлу Camera2D. У головній ігровій сцені він доданий як дочірній вузол персонажа. Оскільки координати дочірнього вузла відносні до батьківського, камера автоматично залишається на тій самій позиції відносно гравця і показує область сцени навколо нього. Саме так і працює бічний скролінг у грі: персонаж рухається вздовж рівня, а камера слідує за ним.

Просте парентування вже дає робочий скролінг, проте для якіснішого результату вузол Camera2D має кілька важливих параметрів, які налаштовуються в інспекторі. Властивість Position Smoothing вмикає згладжування: камера не «прилипає» до позиції гравця жорстко, а наздоганяє його з невеликою затримкою, що візуально приємніше при різких змінах напрямку. Параметри Limit (Left, Right, Top, Bottom) задають межі, за які камера не виходить, щоб на краях рівня не з'являлась порожня область поза картою.

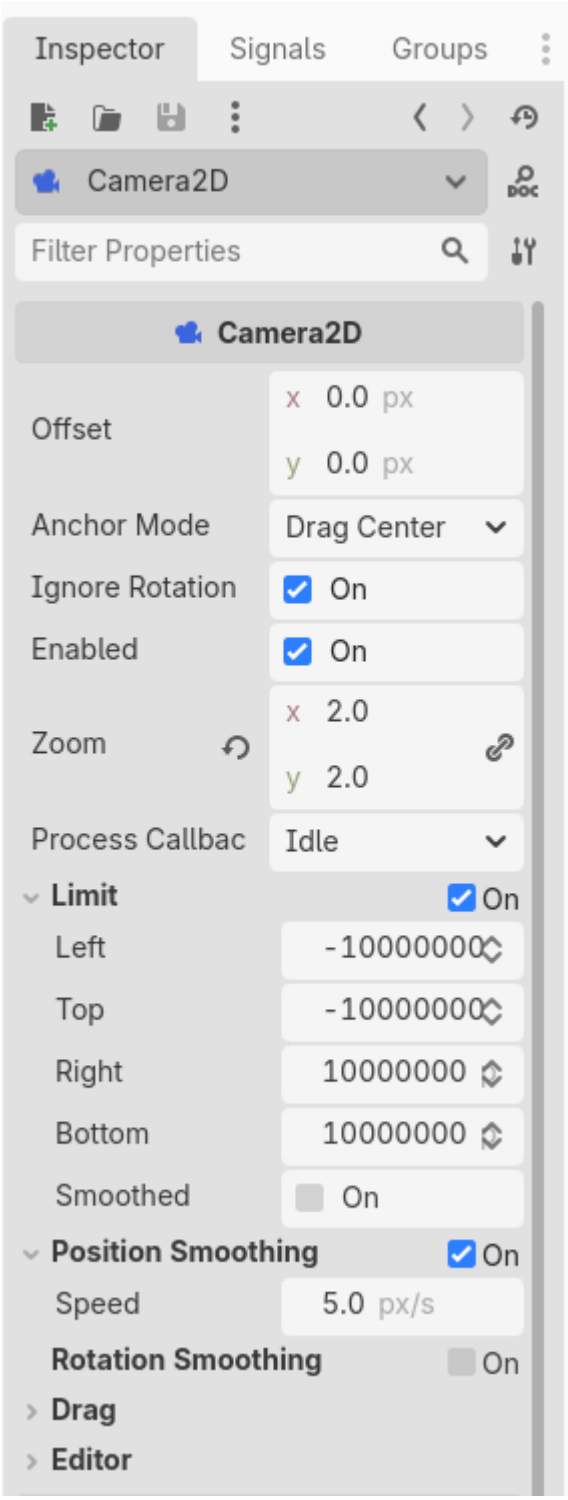

Рис. 3.5 - Налаштування вузла Camera2D в інспекторі.

Слід чесно зазначити, що паралакс-фон, описаний як бажаний елемент у підрозділі 2.4, у поточній версії гри не реалізований: фон сцени статичний. Додавання багатошарового паралаксу через вузли ParallaxBackground і ParallaxLayer є природним напрямом подальшого розвитку проєкту і не потребує

зміни вже написаної логіки руху чи камери, оскільки паралакс реагує на переміщення камери самостійно.

3.4. Розробка ігрових рівнів засобами редактора TileMap

Рівень гри розроблено з використанням вбудованого редактора тайлових карт. У Godot 4 для цього використовується вузол TileMapLayer[13], який прийшов на зміну монолітному TileMap з попередніх версій і дозволяє тримати кожен шар карти окремим вузлом.

Робота починається зі створення ресурсу TileSet, що містить набір доступних для розміщення тайлів. У проєкті тайли формуються з текстурного атласа wafu_A4.png. Редактор автоматично нарізає атлас на тайли за сіткою, після чого для тих тайлів, на яких повинні працювати фізичні зіткнення, у вкладці Physics додається колізійна форма (зазвичай прямокутник, що збігається з тайлом). Саме ці форми згодом не дають персонажу провалюватись крізь землю.

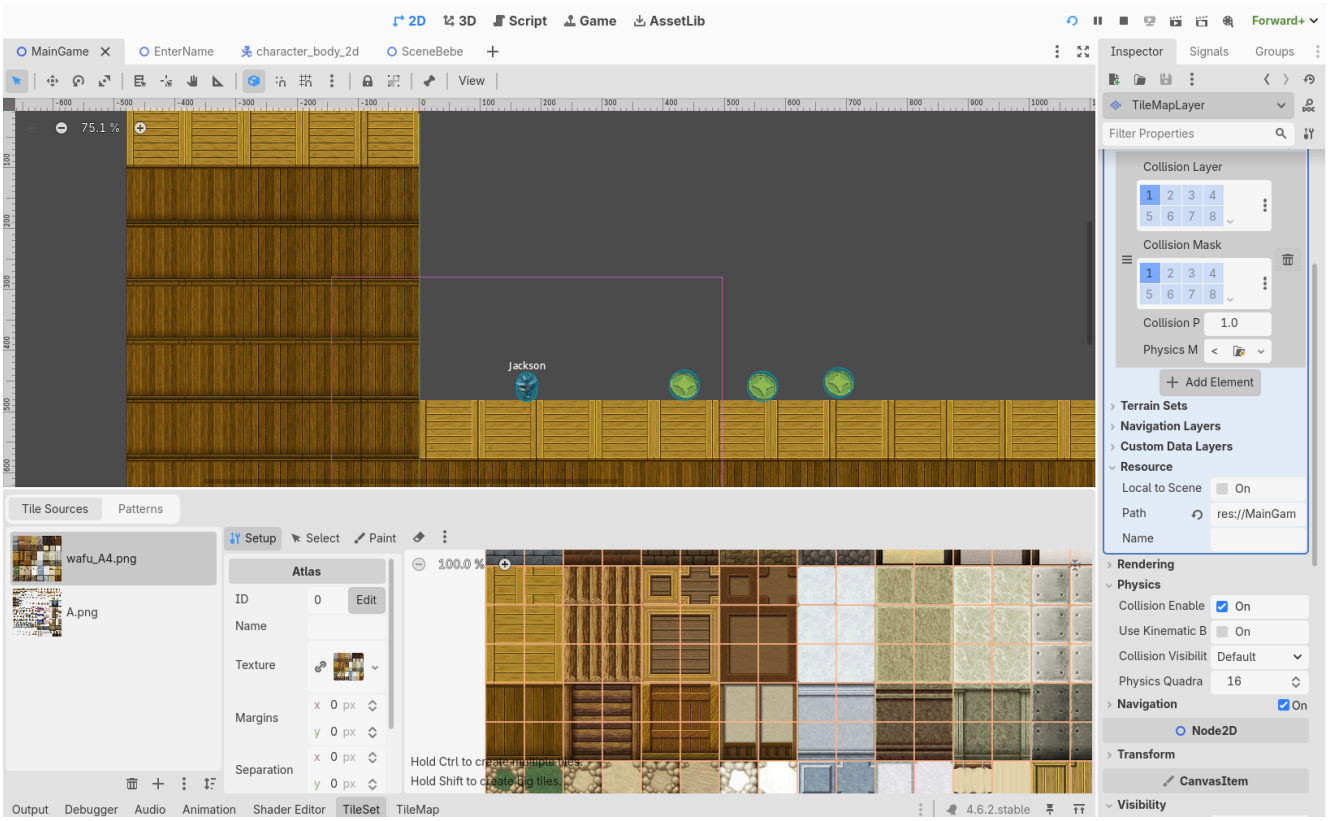

Рис. 3.6 - Вікно редактора TileSet з налаштованими тайлами.

Після підготовки TileSet карта малюється безпосередньо у вікні редактора: обирається TileMapLayer, у нижній панелі активується режим малювання, обирається потрібний тайл і ним «фарбується» область рівня[16]. Доступні режими одиничного розміщення, лінії, прямокутника і заливки. У поточній версії гри рівень побудовано на одному шарі TileMapLayer з колізіями, який формує і фон, і тверду поверхню. Поділ карти на окремі шари (декоративний фон без колізій і окремий шар твердих платформ) є можливим розширенням, передбаченим архітектурою.

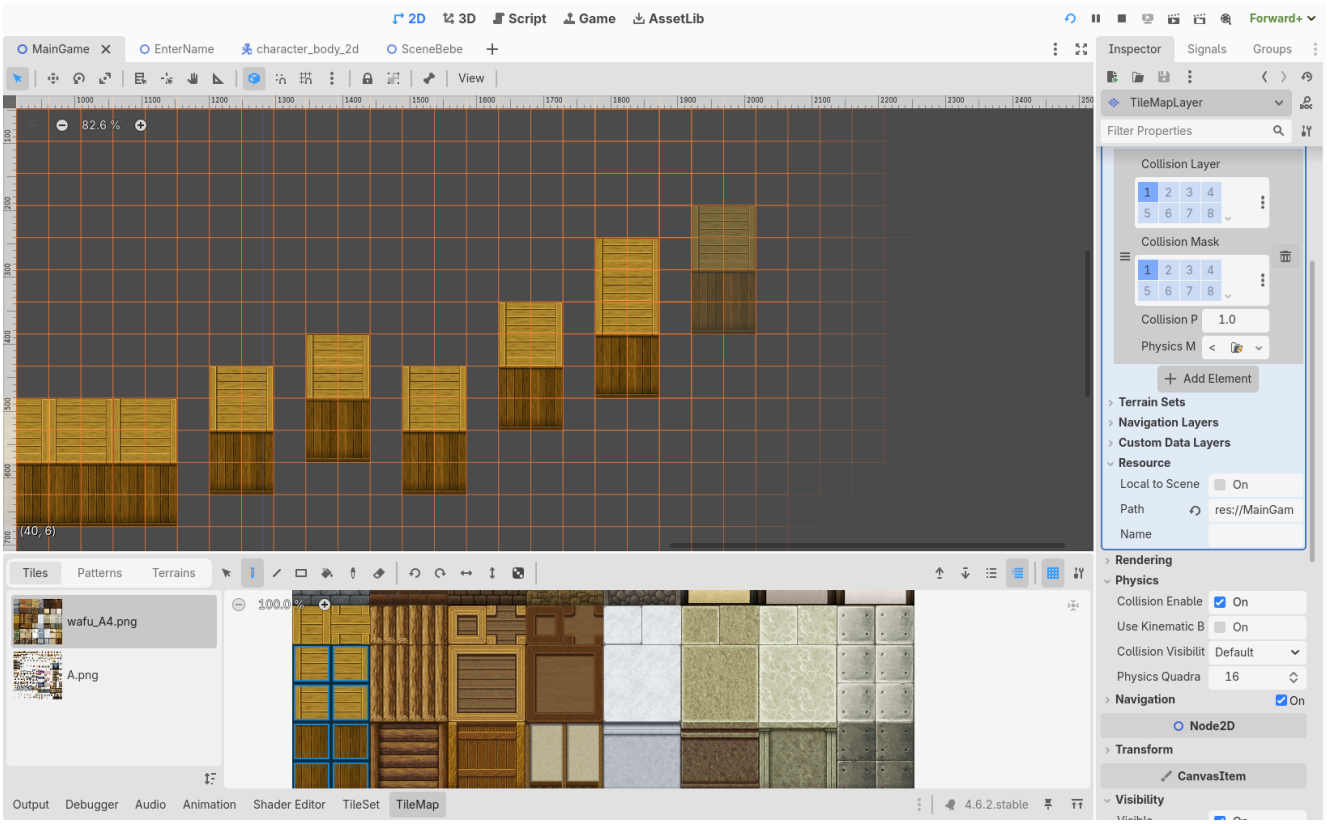

Рис. 3.7 - Скріншот процесу малювання рівня у редакторі.

Окрім тайлів, на рівні розставлено готові екземпляри сцен. Сцена монети (Coin.tscn) перетягується з панелі FileSystem на робочу область, що автоматично створює її екземпляр; у головній сцені таких екземплярів три. Персонаж також доданий як екземпляр своєї сцени, що дозволяє один раз описати його логіку і повторно використовувати без дублювання коду. Загальна структура головної сцени MainGame така: кореневий Node2D містить TileMapLayer (геометрія рівня),

екземпляр персонажа з дочірньою камерою, три екземпляри монети та шар CanvasLayer з лічильником зібраних монет.

Рис. 3.8 - Готовий рівень гри у редакторі.

3.5. Реалізація системи анімацій персонажа та ігрових об'єктів

Анімації у грі реалізовано через вузол AnimatedSprite2D[12], який добре підходить для покадрових спрайтових анімацій і є простішим у налаштуванні, ніж AnimationPlayer. Покадрові набори зберігаються у ресурсі SpriteFrames, що містить колекцію іменованих анімацій, кожна з власною швидкістю і прапорцем циклічності.

Для персонажа підготовлено три основні анімації, які реально використовуються в логіці: default (стан спокою), move_left (рух ліворуч) і move_right (рух праворуч). Кадри вирізаються з атласа fsm_ex_chara02.png. Перемикання між анімаціями виконується у функції update_animation() скрипта персонажа (див. лістинг 3.1): залежно від напрямку руху викликається відповідна анімація, а за відсутності руху програється default. Метод play() запускає анімацію лише тоді, коли вона ще не відтворюється, тому виклик щокадру не призводить до постійного перезапуску.

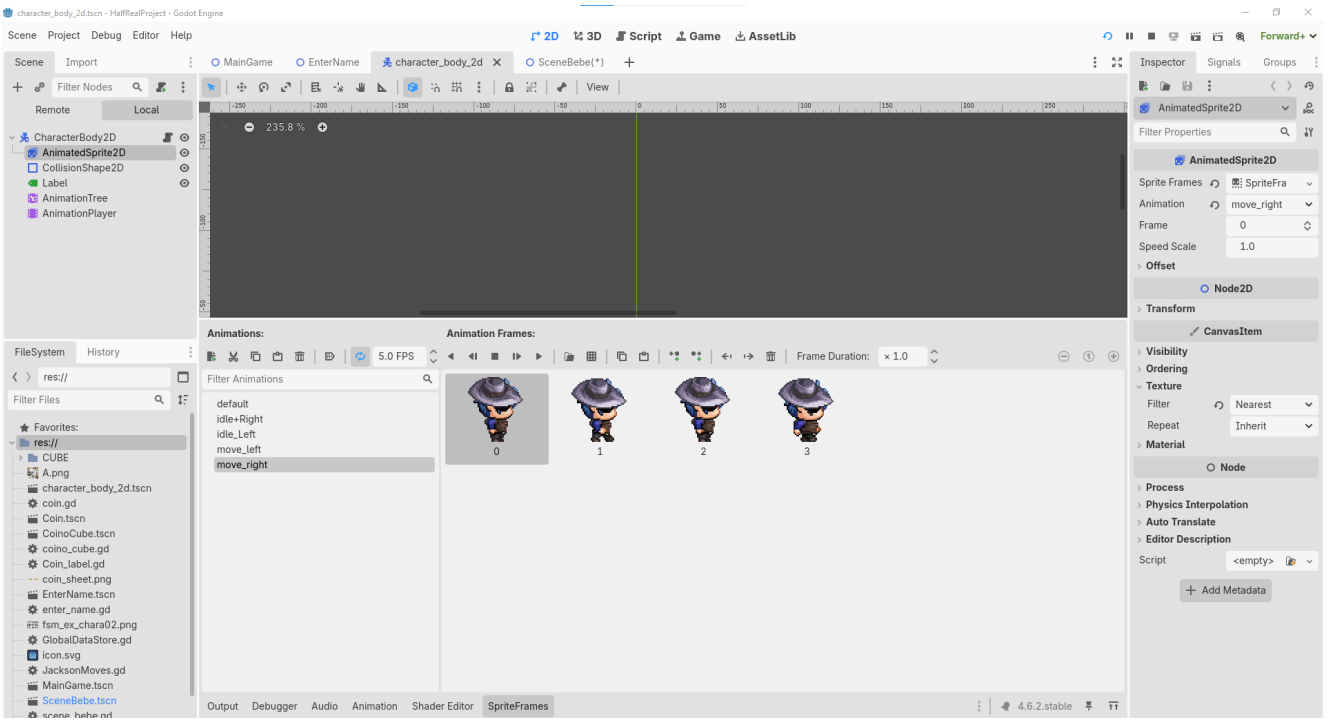

Рис. 3.9 - Вікно SpriteFrames з анімаціями персонажа.

Монета має власну анімацію обертання. Вона побудована на спрайтовому листі coin_sheet.png, який містить вісім послідовних кадрів. У редакторі SpriteFrames ці кадри вирізаються за допомогою об'єктів AtlasTexture (кожен описує прямокутну область листа розміром 48 на 48 пікселів) і об'єднуються в одну анімацію default з увімкненою циклічністю та автозапуском. Кадри відтворюються по колу: монета послідовно повертається від повної лицьової сторони до тонкого ребра і назад, що створює переконливе враження обертання у просторі. Завдяки автозапуску анімація стартує одразу при появі монети на рівні, без жодного додаткового коду.

Варто окремо згадати, що вигляд збираного предмета змінювався у процесі розробки. На ранньому етапі як предмет для збору використовувалась проста обертова модель-заглушка, яка слугувала лише для перевірки самої механіки збирання і роботи лічильника. Заглушка була зручна тим, що дозволяла швидко налагодити логіку зіткнення та нарахування очок, не відволікаючись на художню частину. Після того як механіка була відпрацьована, заглушку замінено повноцінною анімованою монетою з власним набором кадрів обертання, що

краще відповідає візуальній мові жанру і робить мету збирання інтуїтивно зрозумілою для гравця. Така послідовність (спочатку проста заглушка, потім фінальний ассет) є типовою практикою ітеративної розробки, коли спершу перевіряється працездатність механіки, а вже потім доводиться її оформлення.

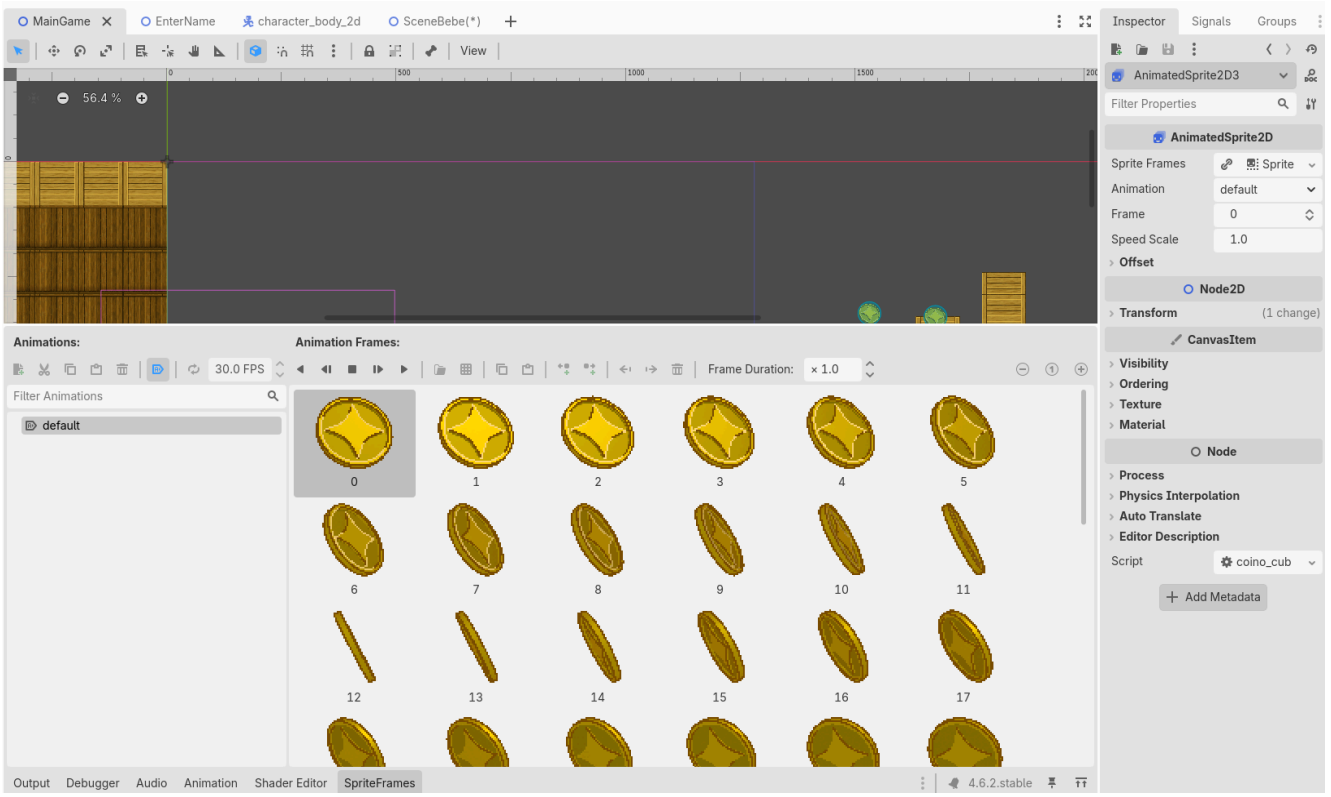

Рис. 3.10 - Анімація обертання монети.

Слід зазначити, що описане у цьому підрозділі є проєктним задумом. У базовій реалізації, наведеній у розділі 3, втілено один демонстраційний рівень і мінімальний інтерфейс з лічильником предметів, тоді як решту запланованих рівнів та екранів винесено в перспективи розвитку.

3.6. Реалізація системи збирання предметів, обліку очок та ігрового інтерфейсу

Система збирання предметів побудована за принципом «один предмет - одна сцена». Монету реалізовано сценою Coin.tscn, кореневим вузлом якої є

AnimatedSprite2D (він же відповідає за анімацію обертання). До нього прикріплено вузол Area2D[14] з дочірньою круглою формою колізії (CircleShape2D). Тип Area2D обрано тому, що він не блокує рух гравця, а лише фіксує факт перетину з ним через сигнал body_entered.

Логіку збору містить скрипт coin.gd (лістинг 3.2). При вході тіла у зону перевіряється, чи це гравець (за належністю до групи «player», з запасною перевіркою за іменем вузла). Прапорець is_collected захищає від повторного спрацювання. Якщо умова виконана, монета повідомляє глобальне сховище про збір, випускає власний сигнал collected (залишений для можливого підключення звуку чи ефекту ззовні) і видаляє себе з пам'яті.

```
extends AnimatedSprite2D

signal collected

var is_collected := false

func _on_area_2d_body_entered(body: Node2D) -> void:
    if is_collected:
        return
    if body.is_in_group("player") or body.name == "CharacterBody2D":
        is_collected = true
        GlobalDataStore.add_coin()
        collected.emit()
        queue_free()
```

Лістинг 3.2 - Скрипт збору монети coin.gd

Підрахунком зібраних монет займається сінглтон GlobalDataStore[17] (лістинг 3.3), реалізований через AutoLoad. Він зберігає ім'я персонажа (char_name) та кількість зібраних монет (coins_count), яка по суті є рахунком гравця. При зміні рахунку сінглтон не змушує інтерфейс опитувати себе щокадру, а випускає сигнал coins_changed, на який підписуються зацікавлені вузли[18]. Метод reset() обнуляє рахунок на старті нової гри.

```
extends Node

signal coins_changed(new_count)

var char_name: String = "Michael JackSon"
var coins_count: int = 0

func add_coin(amount: int = 1) -> void:
    coins_count += amount
    coins_changed.emit(coins_count)

func reset() -> void:
    coins_count = 0
    coins_changed.emit(coins_count)
```

Лістинг 3.3 - Глобальне сховище даних GlobalDataStore.gd

Така побудова реалізує спроєктований у підрозділі 2.4 слабкий зв'язок між компонентами: монета не знає про інтерфейс, інтерфейс не знає про монету, обидва спілкуються тільки через сінглтон. Завдяки цьому додавання нового типу предмета зводиться до створення нової сцени, без зміни наявних скриптів.

Ігровий інтерфейс на поточному етапі складається з лічильника зібраних монет. Це вузол Label, розміщений у шарі CanvasLayer головної сцени (CanvasLayer гарантує, що елемент не залежить від положення камери і завжди лишається на місці екрана). Скрипт лічильника (лістинг 3.4) у _ready() підписується на сигнал сховища і одразу виставляє початкове значення, після чого оновлює текст лише при фактичній зміні рахунку.

```
extends Label

func _ready() -> void:
    GlobalDataStore.coins_changed.connect(_on_coins_changed)
    _on_coins_changed(GlobalDataStore.coins_count)

func _on_coins_changed(new_count: int) -> void:
    text = "Coins: " + str(new_count)
```

Лістинг 3.4 - Скрипт лічильника монет Coin_label.gd

Рис. 3.11 - Лічильник монет у грі.

Окремою частиною інтерфейсу є екранні сцени поза рівнем. Головне меню (SceneBebe.tscn) містить кнопки запуску гри і виходу; їх обробники наведено у лістингу 3.5. Кнопка запуску веде на екран вводу імені, кнопка виходу завершує гру. У меню присутня також кнопка HardMode, зарезервована під майбутній складніший режим, який поки не реалізовано.

```
extends Node2D

func _on_start_button_pressed() -> void:
    get_tree().change_scene_to_file("res://EnterName.tscn")

func _on_quit_button_3_pressed() -> void:
    get_tree().quit()
```

Лістинг 3.5 - Скрипт головного меню scene_bebe.gd

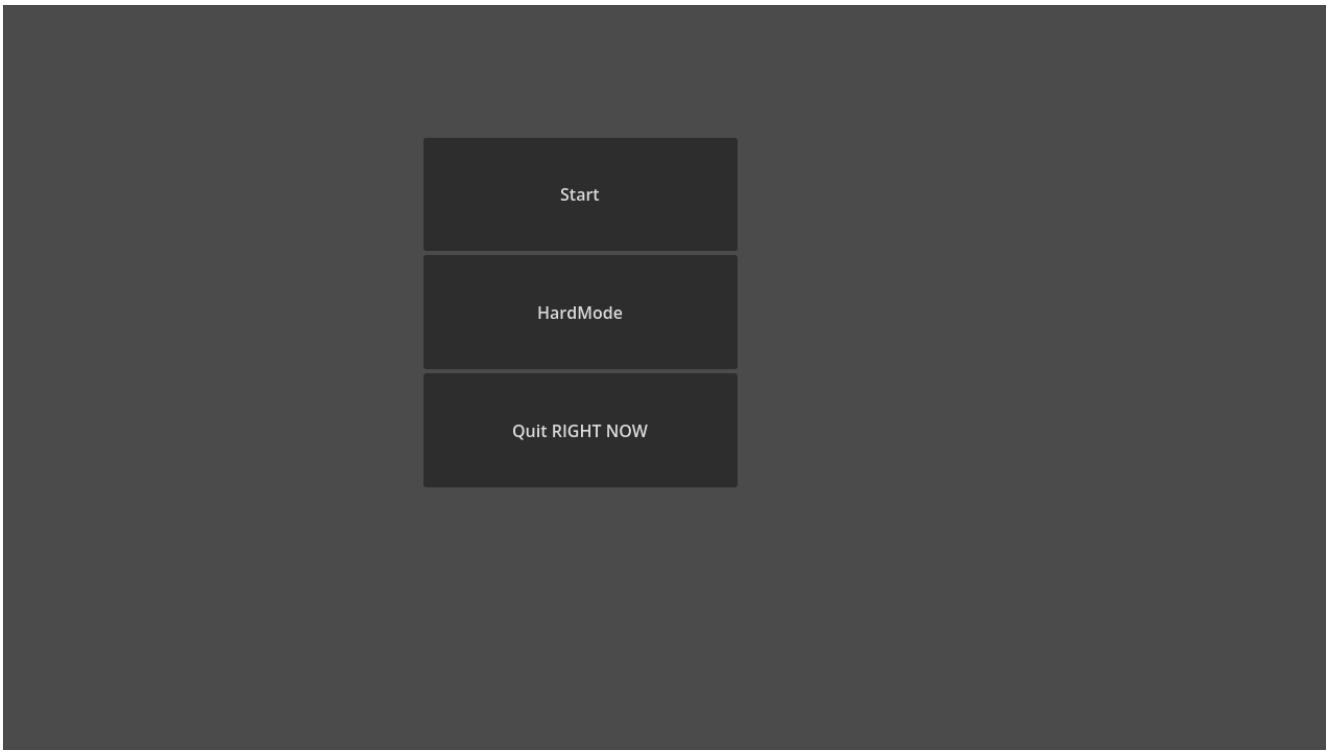

Рис. 3.12 - Головне меню гри.

Екран вводу імені (EnterName.tscn) дозволяє гравцю задати ім'я персонажа, яке згодом відображається над ним у грі. Поле вводу LineEdit і кнопка підтвердження обробляються скриптом enter_name.gd (лістинг 3.6): введене ім'я зберігається у сховище (порожнє ім'я залишає значення за замовчуванням), рахунок скидається методом reset(), після чого завантажується ігрова сцена MainGame.

```
extends Node2D

@onready var line_edit: LineEdit = $LineEdit

func _on_char_name_button_pressed() -> void:
    var entered := line_edit.text.strip_edges()
    if entered != "":
        GlobalDataStore.char_name = entered
    GlobalDataStore.reset()
    get_tree().change_scene_to_file("res://MainGame.tscn")
```

Лістинг 3.6 - Скрипт екрана вводу імені enter_name.gd

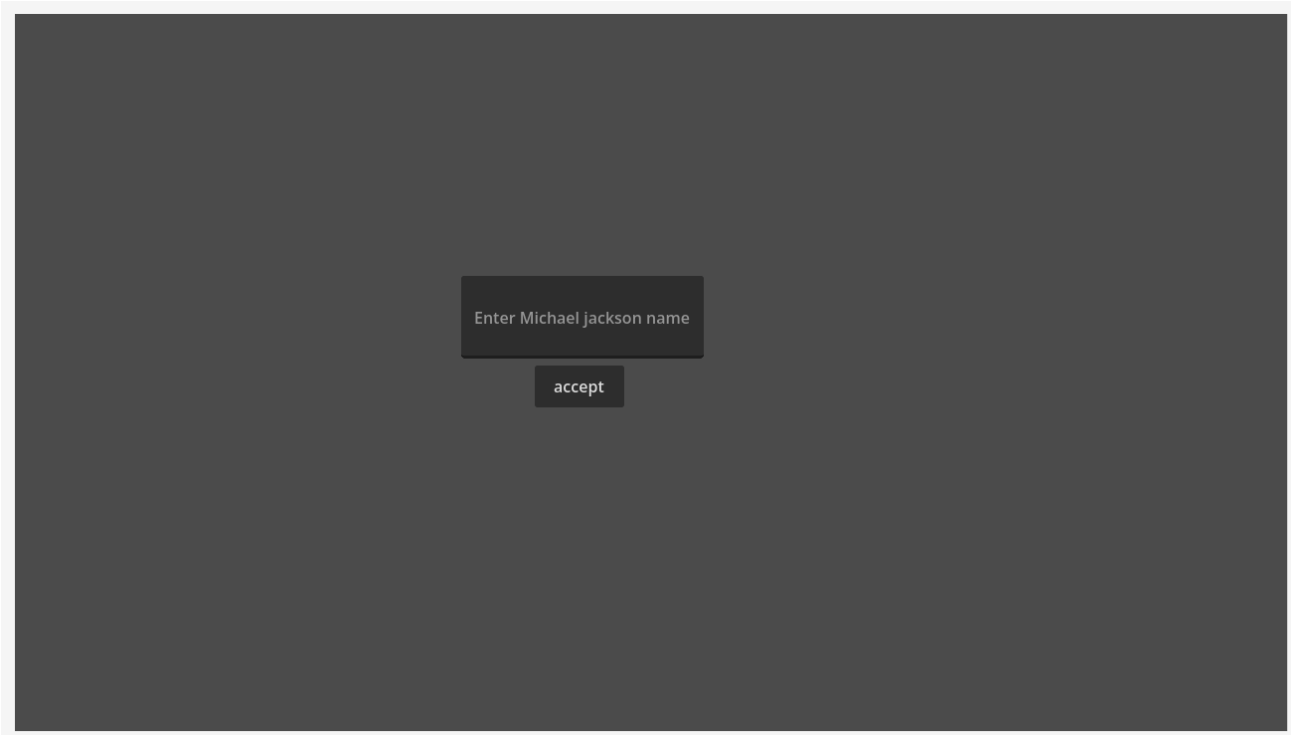

Рис. 3.13 - Екран вводу імені персонажа.

Перехід між сценами в усіх випадках виконано методом get_tree().change_scene_to_file(), як і планувалось у підрозділі 2.4. Загальний потік гри такий: головне меню, екран вводу імені, ігровий рівень. Системи життів, екранів Game Over і перемоги, а також переходів між кількома рівнями, описані у розділі 2, у поточній версії не реалізовані і винесені у перспективи розвитку, оскільки гра побудована навколо одного демонстраційного рівня.

3.7. Тестування розробленої гри

Тестування виконано у три етапи: функціональне (перевірка правильності роботи механік), продуктивне (оцінка кадрової частоти і споживання ресурсів) і стабільне (виявлення збоїв при тривалому використанні).

Функціональне тестування проведено за заздалегідь складеним набором тест-кейсів. Кожен описує дію, очікуваний і фактичний результат та статус. Результати наведено у таблиці 3.1.

Таблиця 3.1 Результати функціонального тестування

№	Тест-кейс	Очікуваний результат	Фактичний результат	Статус
1	Запуск гри з головного меню	Відкривається екран вводу імені	Завантажується перший рівень	Пройдено
2	Підтвердження імені	Завантажується рівень, ім'я над персонажем	Працює коректно	Пройдено
3	Порожнє поле імені	Залишається ім'я за замовчуванням	Залишається за замовчуванням	Пройдено
4	Рух праворуч	Персонаж рухається, анімація move_right	Працює коректно	Пройдено
5	Рух ліворуч	Персонаж рухається, анімація move_left	Працює коректно	Пройдено
6	Стрибок з землі	Персонаж піднімається і падає назад	Працює коректно	Пройдено
7	Змінна висота стрибка	Коротке натискання дає низький стрибок	Працює коректно	Пройдено
8	Coyote time	Стрибок можливий 0.1 с після сходу з краю	Працює коректно	Пройдено
9	Буферизація стрибка	Раннє натискання спрацьовує при приземленні	Працює коректно	Пройдено
10	Збір монети	Монета зникає, лічильник +1	Працює коректно	Пройдено
11	Захист від подвійного збору	Монета не нараховується двічі	Працює коректно	Пройдено
12	Скидання рахунку на старті	Новий запуск починається з нуля	Працює коректно	Пройдено
13	Слідування камери	Камера тримає персонажа у кадрі	Працює коректно	Пройдено
14	Вихід з гри	Кнопка Quit закриває	Працює	Пройдено

		застосунок	коректно	

Усі тест-кейси успішно пройдено. У ході розробки виявлено і усунено кілька проблем: лічильник монет спершу опитував сховище щокадру у _process, що було замінено на подієве оновлення через сигнал; збір монети не мав захисту від повторного спрацювання, через що в окремих випадках можливе було подвійне нарахування, що виправлено прапорцем is_collected; рахунок не скидався між запусками, тому додано виклик reset() на старті гри.

Тестування продуктивності виконано запуском гри на двох пристроях різного класу. Результати наведено у таблиці 3.2.

Таблиця 3.2 Результати тестування продуктивності

Пристрій	Конфігурація	Середній FPS	Просадки FPS
Ноутбук 1	Intel i5-10210U, 8 ГБ RAM, інтегрована Intel UHD	144	Не зафіксовано
Ноутбук 2	Intel i3-7100U, 4 ГБ RAM, інтегрована Intel HD 620	60	Не зафіксовано

Обидві системи стабільно тримали 144 кадрів за секунду. Споживання пам'яті лишалось у межах близько 50-70 МБ, що є дуже скромним показником, зумовленим простотою сцени і невеликою кількістю об'єктів.

Стабільне тестування полягало у тривалій грі з багаторазовим перезапуском (повернення в меню, новий ввід імені, повторне проходження). За цей час не виявлено крашів, втрат продуктивності з часом чи витоків пам'яті: показники споживання залишались стабільними, анімації та лічильник працювали без збоїв.

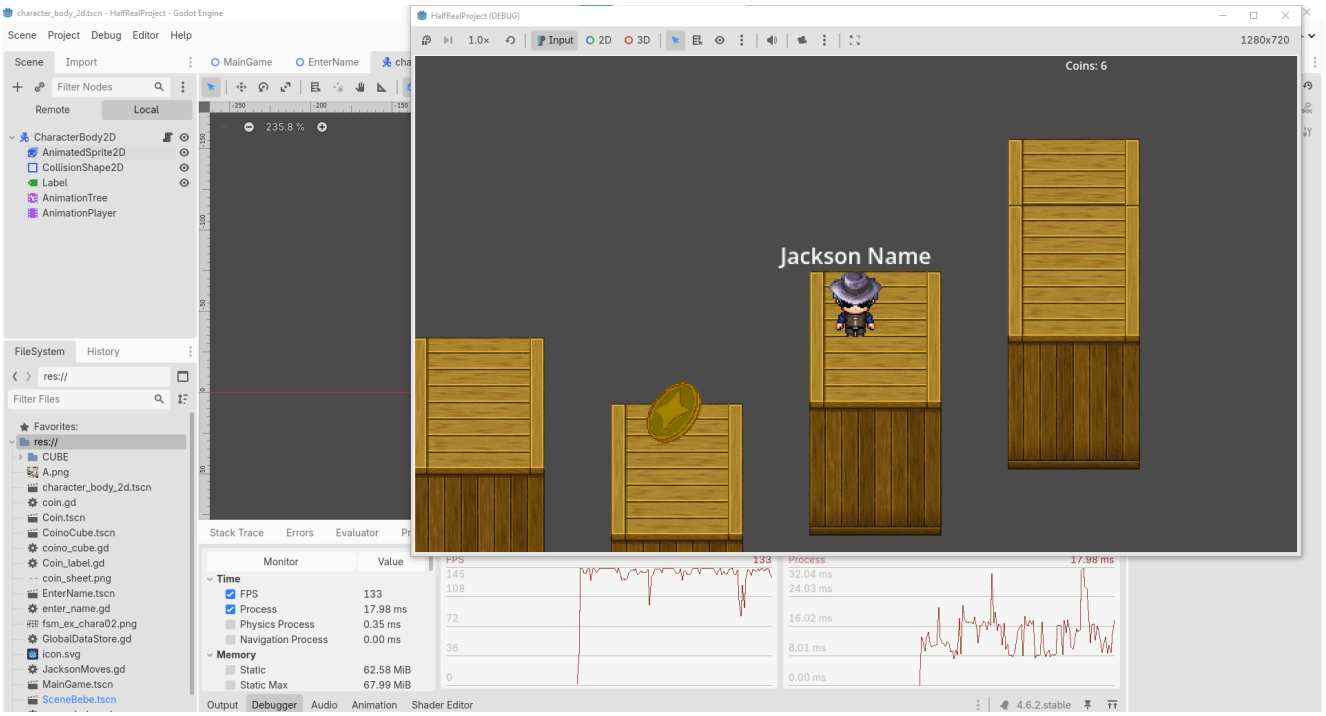

Рис. 3.14 - Скріншот вкладки Monitor у Godot

Підсумовуючи, можна стверджувати, що розроблена гра є функціонально працездатною у межах реалізованого обсягу, стабільно працює на цільових пристроях і коректно виконує всі реалізовані механіки. Виявлені під час тестування дрібні недоліки усунено, а закладена архітектура дозволяє нарощувати функціонал (життя, додаткові рівні, паралакс, звук, бойова система) без переписування наявного коду.

ВИСНОВКИ

У процесі виконання кваліфікаційної роботи поставлену мету досягнуто: розроблено двовимірний платформер з бічним скролінгом засобами ігрового рушія Godot з реалізацією базових механік жанру.

У теоретичній частині досліджено походження і еволюцію жанру платформер, проаналізовано ключові ігрові механіки, серед яких особливу увагу приділено стрибку та допоміжним прийомам на кшталт coyote time. Розглянуто архітектуру Godot, його систему вузлів і сцен, а також мову GDScript. Виконано порівняльний аналіз існуючих платформерів (Super Mario Bros, Celeste, Hollow Knight, Shovel Knight) та інструментів їх розробки, на основі якого обґрунтовано вибір Godot завдяки відкритій ліцензії, нативній підтримці 2D і простоті GDScript.

У практичній частині реалізовано систему керування персонажем на базі CharacterBody2D з аркадною фізичною моделлю та прийомами coyote time, буферизації і змінної висоти стрибка, налаштовано бічний скролінг камери зі згладжуванням, розроблено ігровий рівень засобами редактора TileMap, реалізовано систему анімацій через AnimatedSprite2D, збирання монет через Area2D, а також глобальне сховище стану GlobalDataStore (AutoLoad-сінглтон) і подієвий ігровий інтерфейс з лічильником монет, головним меню та екраном вводу імені персонажа.

Тестування підтвердило коректність роботи реалізованих механік (усі 14 тест-кейсів пройдено), стабільну продуктивність на рівні 60 кадрів за секунду на двох тестових пристроях, відсутність крашів і витоків пам'яті при тривалій грі. Побудована архітектура є модульною: додавання нових предметів чи рівнів не вимагає переписування наявного коду.

Серед свідомих обмежень поточної версії слід зазначити один демонстраційний рівень, відсутність системи життів, паралакс-фону, звукового супроводу та збереження прогресу. Перспективи розвитку охоплюють розширення контенту, додавання нових механік (подвійний стрибок, ривок), паралакс-фону і звуку, реалізацію збереження через ConfigFile, локалізацію та збірку для мобільних платформ.

ПЕРЕЛІК ПОСИЛАНЬ

1. Grand View Research. Video Game Market Size, Share And Growth Report, 2030 [Електронний ресурс]. - Режим доступу: https://www.grandviewresearch.com/industry-analysis/video-game-market
2. History of platform games: 9 steps of genre evolution [Електронний ресурс] / Red Bull. - Режим доступу: https://www.redbull.com/in-en/evolution-of-platformers
3. Altice N. The long shadow of Super Mario Bros. [Електронний ресурс] / N. Altice // Game Developer. - 2015. - Режим доступу: https://www.gamedeveloper.com/design/the-long-shadow-of-i-super-mario-bros-i-
4. Platformer Game Design (Definition, Fundamentals, Mechanics) [Електронний ресурс] / gamedesignskills.com. - Режим доступу: https://gamedesignskills.com/game-design/platformer/
5. Thorson M. Celeste & Forgiveness [Електронний ресурс] / M. Thorson // Medium. - 2018. - Режим доступу: https://maddythorson.medium.com/celeste-forgiveness-31e4a40399f1
6. Thorson M. Celeste and TowerFall Physics [Електронний ресурс] / M. Thorson // Medium. - 2020. - Режим доступу: https://maddythorson.medium.com/celeste-and-towerfall-physics-d24bd2ae0fc5
7. Pichlmair M. Designing Game Feel. A Survey [Електронний ресурс] / M. Pichlmair, M. Johansen // arXiv. - 2020. - Режим доступу: https://arxiv.org/pdf/2011.09201
8. Kotaku. Devs React To Unity's Newly Announced Fee For Game Installs [Електронний ресурс]. - Режим доступу: https://kotaku.com/unity-engine-subscription-cost-unreal-godot-indie-dev-1850831032
9. Godot Engine. Official website [Електронний ресурс]. - Режим доступу: https://godotengine.org

10. Godot Engine. License [Електронний ресурс]. - Режим доступу: https://godotengine.org/license/
11. Linietsky J. A brief introduction to the Godot Engine with Juan Linietsky, Lead Developer [Електронний ресурс] / J. Linietsky // Software Freedom Conservancy Blog. - 2020. - Режим доступу: https://sfconservancy.org/blog/2020/dec/28/Juan-Linietsky-interview/
12. Godot Engine documentation. GDScript reference [Електронний ресурс]. - Режим доступу: https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html
13. MDN Web Docs. Tiles and tilemaps overview - Game development [Електронний ресурс]. - Режим доступу: https://developer.mozilla.org/en-US/docs/Games/Techniques/Tilemaps
14. Godot Engine documentation. Using CharacterBody2D [Електронний ресурс]. - Режим доступу: https://docs.godotengine.org/en/stable/tutorials/physics/using_character_body_2d.html
15. Indie Game Academy. How to Make a Smooth Movement System for a 2D Platformer in Godot [Електронний ресурс]. - Режим доступу: https://indiegameacademy.com/how-to-make-a-smooth-movement-system-for-a-2d-platformer-in-godot/
16. Coomans C. Learn Godot 4 by Making a 2D Platformer - Part 1: Project Editor & Overview [Електронний ресурс] / C. Coomans // DEV Community. - 2023. - Режим доступу: https://dev.to/christinec_dev/learn-godot-4-by-making-a-2d-platformer-part-1-project-editor-overview-1ap4
17. Godot Engine documentation. Singletons (Autoload) [Електронний ресурс]. - Режим доступу: https://docs.godotengine.org/en/stable/tutorials/scripting/singletons_autoload.html

18. GDQuest. The Events bus singleton [Електронний ресурс]. - Режим доступу: https://www.gdquest.com/tutorial/godot/design-patterns/event-bus-singleton/
19. Marcano Vargas C. Building a 2D Platformer Game with Godot. Part 2 [Електронний ресурс] / C. Marcano Vargas // Medium. - 2023. - Режим доступу: https://medium.com/@carlosmarcano2704/building-a-2d-platformer-game-with-godot-part-2-10ad409babdf
20. Jun T. How to design breathtaking 2D platformer levels [Електронний ресурс] / T. Jun. - 2024. - Режим доступу: https://www.tadeasjun.com/blog/2d-level-design/
21. Godot Engine. Godot 4.0 sets sail: all aboard for new horizons [Електронний ресурс]. - 2023. - Режим доступу: https://godotengine.org/article/godot-4-0-sets-sail-all-aboard-for-new-horizons/

ДОДАТОК А
ПРЕЗЕНТАЦІЯ ДО ЗВІТУ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНІ СИСТЕМИ ТА ТЕХНОЛОГІЇ

Платформер з бічним скролінгом на основі ігрового рушія Godot

Виконав студент 4 курсу
групи ІСД-42
Ситник Євген Олегович
Керівник роботи
Кандидат Технічих Наук, Сеньков Олег Вікторович

Київ – 2026

Мета, об'єкт та предмет дослідження

Мета роботи

Розробка двовимірного платформера з бічним скролінгом на основі ігрового рушія Godot із реалізацією повного циклу створення гри: від проєктування архітектури та ігрових рівнів до впровадження системи керування персонажем, анімацій, збирання предметів та ігрового інтерфейсу користувача.

Об'єкт дослідження

Процеси розробки двовимірних ігор жанру платформер з бічним скролінгом засобами сучасних ігрових рушіїв.

Предмет дослідження

Методи та програмні засоби реалізації платформера з бічним скролінгом на основі ігрового рушія Godot із застосуванням мови програмування GDScript.

Завдання роботи

Завдання роботи

1. Дослідити теоретичні основи жанру та ключові ігрові механіки
2. Виконати порівняльний аналіз рушіїв і обґрунтувати вибір Godot
3. Спроєктувати модульну архітектуру гри, рівні та інтерфейс
4. Реалізувати та протестувати платформер з бічним скролінгом

3

Жанр платформер: класифікація

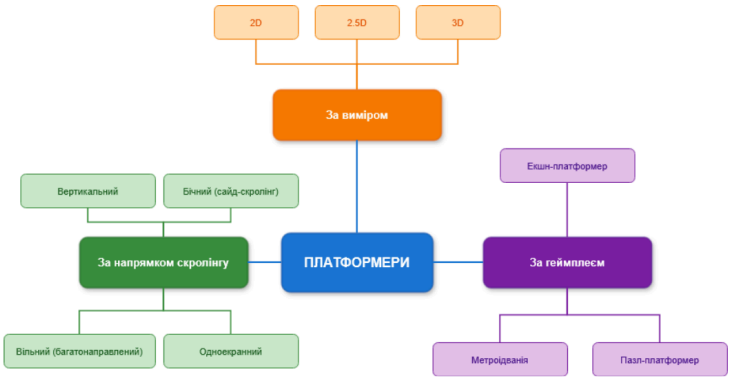

Рис. 1 - Класифікація платформерів за напрямком скролінгу та виміром

- Перший представник — Donkey Kong (1981)
- Канон бічного скролінгу заклав Super Mario Bros (1985)
- Інді-відродження 2010-х повернуло 2D у топ жанрів
- Обраний напрям: 2D, бічний (горизонтальний) скролінг, лінійні рівні

4

Ключові механіки платформера

Coyote time
Стрибок дозволений ще ~0.1 с після сходу з краю платформи

Буферизація вводу
Раннє натискання спрацьовує автоматично при приземленні

Змінна висота стрибка
Чим довше тримати клавішу — тим вищий стрибок

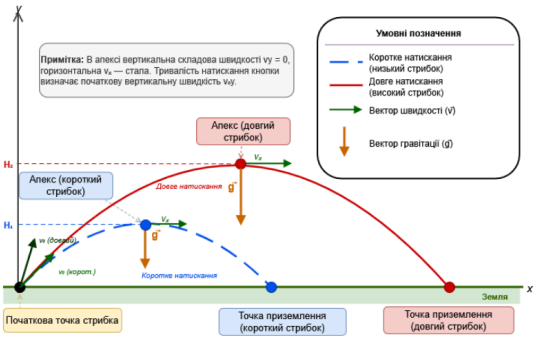

Рис. 2 - Траєкторія стрибка: коротке та довге натискання

5

Порівняння ігрових рушіїв

Рушій	Ліцензія	Мова	Підтримка 2D	Розмір
Unity	Proprietary / Runtime Fee	C#	Повна	~1–2 ГБ
Unreal Engine	Proprietary (Royalty 5%)	C++ / Blueprints	Часткова	~10–50 ГБ
GameMaker	Proprietary	GML	Відмінна	~300 МБ
Construct	Proprietary	JS / No-code	Відмінна	~200 МБ
Phaser	MIT	JS / TS	Відмінна	~1 МБ
Godot	**MIT**	**GDScript / C# / C++**	**Відмінна**	**~65–100 МБ**

6

Обґрунтування вибору Godot

Ліцензія MIT — вільне використання без роялті, проєкт повністю належить автору

Нативний 2D — окремий рендерер і фізика, а не «3D з вимкненою глибиною»

Інструменти «з коробки» — TileMap, Camera2D, CharacterBody2D, Area2D, ParallaxBackground

Простота GDScript — синтаксис, схожий на Python, без компіляції, прямий доступ до API

Легкість рушія — один файл ~100 МБ, без реєстрації та встановлення

Активна спільнота — документація, форуми, зростання після 2023 на тлі суперечок Unity

7

Загальна архітектура гри

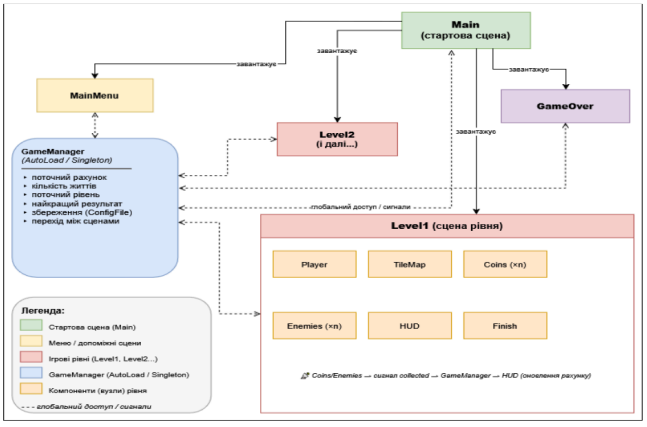

Рис. 3 - Загальна архітектура гри у вигляді діаграми сцен

- Стартова сцена Main → головне меню → ігрові рівні
- Кожен об'єкт (гравець, монета) — окрема багаторазова сцена замість префабів
- Глобальний стан — AutoLoad-сінглтон GlobalDataStore
- Обмін даними — через систему сигналів (слабкий зв'язок компонентів)
- Модульність: нові рівні й механіки додаються без переписування коду

8

Керування персонажем та фізика

- Персонаж — на вузлі CharacterBody2D: повний контроль над рухом і вбудована обробка колізій
- move_and_slide() переміщує тіло та реагує на зіткнення з оточенням
- Input.get_axis() дає плавний рух з невеликою інерцією
- Реалізовано всі три прийоми game feel: coyote time, буферизацію та змінну висоту стрибка
- Камера Camera2D — дочірній вузол гравця, зі згладжуванням і межами Limit

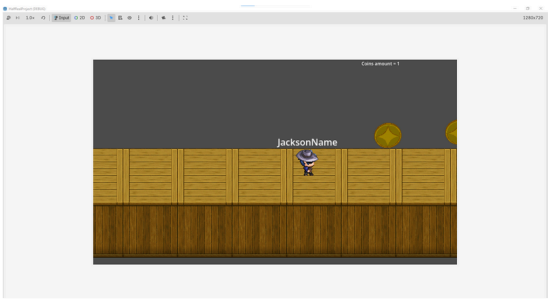

Рис. 4 - Структура сцени персонажа у редакторі Godot

Рівень, тайли та анімації

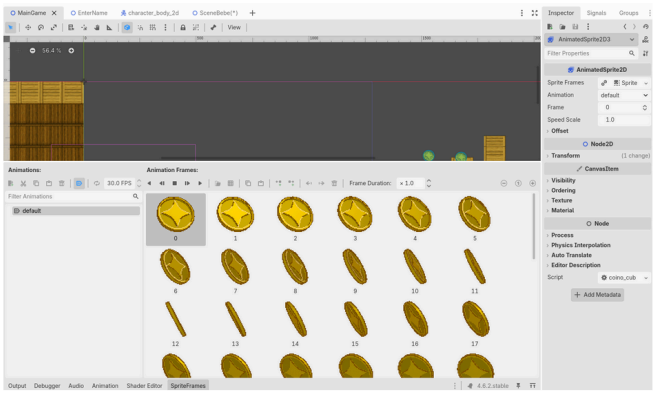

Рис. 5 - Кадри обертання монети у редакторі SpriteFrames

- Рівень побудовано на вузлі TileMapLayer (Godot 4) з колізіями з атласа тайлів
- Анімації персонажа — через AnimatedSprite2D: default, move_left, move_right
- Монета — Area2D з покадровою анімацією обертання та автозапуском
- play() запускає анімацію лише якщо вона ще не йде — без перезапуску щокадру

Збирання предметів та інтерфейс

- Монета фіксує дотик гравця сигналом body_entered; прапорець is_collected захищає від подвійного збору
- GlobalDataStore рахує монети та зберігає ім'я; при зміні випускає сигнал coins_changed
- HUD у CanvasLayer підписаний на сигнал — оновлюється подієво, а не щокадру
- Головне меню та екран вводу імені; переходи через get_tree().change_scene_to_file()

Рис. 6 - Лічильник монет та ім'я персонажа у грі

Слабкий зв'язок: монета не знає про HUD, HUD не знає про монету — обидва спілкуються лише через сінглтон.

11

Тестування розробленої гри

Етапи та результати

Функціональне: усі 14 тест-кейсів пройдено (рух, стрибок, coyote/буфер, збір монет, камера, скидання рахунку)

Продуктивне: стабільні 60–144 FPS на двох ПК, споживання пам'яті 50–70 МБ

Стабільне: крашів і витоків пам'яті при тривалій грі з перезапусками не виявлено

Виявлено та усунено

- Лічильник опитував сховище щокадру → замінено на подієве оновлення через сигнал
- Монета могла нараховуватись двічі → додано прапорець is_collected
- Рахунок не скидався між запусками → додано виклик reset() на старті гри

12

Висновки

- Мету роботи досягнуто: розроблено двовимірний платформер з бічним скролінгом засобами рушія Godot з реалізацією базових механік жанру
- Реалізовано керування персонажем на CharacterBody2D з аркадною фізикою та прийомами coyote time, буферизації і змінної висоти стрибка
- Налаштовано бічний скролінг камери, рівень на TileMap, анімації, збирання монет, глобальне сховище та подієвий інтерфейс
- Тестування підтвердило коректність механік, стабільну продуктивність та відсутність витоків пам'яті
- Архітектура модульна — додавання нових рівнів і механік не вимагає переписування коду

Перспективи розвитку

Система життів та екрани Game Over / перемоги, кілька рівнів і переходи між ними, паралакс-фон і звуковий супровід, нові механіки (подвійний стрибок, ривок), збереження через ConfigFile, локалізація та збірка для мобільних платформ.

Апробація результатів дослідження

Основні положення і результати кваліфікаційної роботи доповідались та були опубліковані у збірниках тез наступних науково-практичних конференцій:

1. Ситник Є.О. - «ЦИФРОВИЙ ДВІЙНИК МІСТА: ВІД РЕАКТИВНОГО ГАСІННЯ ПОЖЕЖ ДО ПРОАКТИВНОГО МОДЕЛЮВАННЯ». Тези доповіді на VII Міжнародній науково-технічній конференції «Сучасний стан та перспективи розвитку IoT»
2. Ситник Є.О. - «ЗАСТОСУВАННЯ ІГРОВОГО РУШІЯ GODOT ENGINE ДЛЯ РОЗРОБКИ 2D-ПЛАТФОРМЕРА З БІЧНИМ СКРОЛІНГОМ». Тези доповіді на VII Всеукраїнській науково-технічній конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях»

Дякую за увагу!