Міністерство освіти і науки України
Тернопільський національний педагогічний університет імені Володимира Гнатюка

Фізико-математичний факультет
Кафедра інформатики та методики її навчання

Кваліфікаційна робота на тему:

«ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ІГРОВИХ МЕХАНІК ДВОВИМІРНОЇ ГРИ ЖАНРУ ПЛАТФОРМЕР НА БАЗІ РУШІЯ GODOT»

Спеціальність 122 Комп’ютерні науки
Освітньо-професійна програма «Інженерія ігрових проєктів»

Здобувача першого (бакалаврського) рівня вищої освіти
Назара ХИТРУКА

НАУКОВИЙ КЕРІВНИК:
кандидат технічних наук, доцент
Олександр ВОВКОДАВ

РЕЦЕНЗЕНТ:
кадидат технічних наук, доцент
Юрій ФРАНКО

Тернопіль – 2026

ЗМІСТ

ВСТУП

Актуальність теми. Сучасна індустрія комп'ютерних ігор (GameDev) є одним із найбільш динамічних секторів цифрової економіки, що демонструє щорічне зростання аудиторії та обсягів ринку. Особливе місце в цій екосистемі посідають двовимірні ігри жанру платформер. Незважаючи на тривалу історію існування, цей жанр залишається актуальним завдяки гнучкості у геймдизайні, високій залученості гравців через точність керування та відносно низькому порогу входження для незалежних (indie) розробників.

Процес створення якісного 2D-платформера вимагає тонкого налаштування ігрового процесу (gameplay). Успіх гри критично залежить від проєктування та програмної реалізації базових ігрових механік: фізики руху персонажа, розрахунку гравітації, обробки колізій, взаємодії з оточенням та ворогами, а також ігрового балансу. Навіть незначні похибки у на етапі проєктування здатні повністю руйнувати ігровий досвід (user experience).

Таким чином, дослідження принципів проєктування та програмної реалізації ігрових механік платформера на базі рушія Godot є актуальним науково-практичним завданням, що відповідає сучасним запитам індустрії розробки програмного забезпечення.

Мета дослідження – теоретично обґрунтувати, спроєктувати та практично реалізувати комплекс базових та розширених ігрових механік для двовимірної комп'ютерної гри жанру платформер у середовищі розробки Godot Engine за допомогою мови програмування GDScript.

Завдання дослідження – для досягнення поставленої мети необхідно вирішити такі завдання: проаналізувати еволюцію, особливості та успішні патерни геймдизайну у жанрі 2D-платформерів; формалізувати теоретичні аспекти проєктування ігрових механік, фізики руху, таймінгів та колізій для ігор цього типу; провести порівняльний аналіз сучасних ігрових рушіїв для розробки 2D-ігор та обґрунтувати вибір рушія Godot; спроєктувати архітектуру ігрового проєкту,

структуру сцени персонажа; програмно реалізувати механіки руху, гравітації, взаємодії з об’єктами та систему станів персонажа (State Machine) мовою GDScript.

Об’єкт дослідження – процес розробки двовимірних комп’ютерних ігор жанру платформер.

Предмет дослідження – архітектурні шаблони, алгоритми та програмні засоби реалізації ігрових механік і фізики взаємодії об’єктів 2D-платформера на базі ігрового рушія Godot Engine.

Методи дослідження. Під час виконання роботи були використані такі методи дослідження: методи системного аналізу та порівняння – для вивчення існуючих аналогів, класифікації механік та вибору інструментарію розробки; методи об’єктно-орієнтованого проєктування – для формування архітектури сцени персонажа та дерева вузлів рушія; методи кінцевих автоматів (State Machine) – для моделювання та розробки поведінки ігрових сутностей та анімацій; методи математичного моделювання фізичних процесів – для розрахунку параметрів інерції, тертя та падіння тіл; методи експериментального тестування (профайлінг, плейтести) – для виявлення помилок та оптимізації швидкодії коду.

Наукова новизна отриманих результатів полягає у систематизації підходів до адаптації класичних геймдизайнерських паттернів 2D-платформерів (таких як Coyote time та Jump buffering) під специфіку архітектури вузлів та сигналів рушія Godot 4.x, що дозволяє оптимізувати розподіл обчислювальних ресурсів під час обробки фізики та колізій у реальному часі.

Практичне значення отриманих результатів полягає у створенні готового та оптимізованого програмного шаблону (фреймворку) 2D-платформера на базі Godot Engine з гнучкою архітектурою. Описані скрипти, налаштування колізій та побудова дерева вузлів можуть бути використані іншими розробниками як база для створення власних ігрових проєктів, а також у навчальному процесі під час вивчення дисциплін, пов’язаних із технологіями розробки комп’ютерних ігор.

Структура та обсяг роботи. Бакалаврська робота складається зі вступу, двох розділів, висновків, списку використаних джерел. Загальний обсяг текстової частини становить 51 сторінrf комп’ютерного тексту. Робота містить 2 рисунки, 1 таблицю. Список використаних джерел налічує 19 найменувань.

РОЗДІЛ 1. ТЕОРЕТИКО-МЕТОДОЛОГІЧНІ ОСНОВИ ПРОЄКТУВАННЯ ДВОВИМІРНИХ ІГОР ЖАНРУ ПЛАТФОРМЕР

1.1. Еволюція та ключові особливості жанру 2D-платформерів в ігровій індустрії

Двовимірні платформери є одним із фундаментальних та історично значущих жанрів індустрії відеоігор [8]. Протягом понад сорока років свого існування цей жанр пройшов шлях від простих аркадних автоматів з обмеженими дискретними можливостями переміщення до складних інтерактивних систем, які оперують глибокими психологічними паттернами залучення, розвиненою фізикою та нелінійним дизайном рівнів [2]. Актуальність дослідження еволюції цього жанру зумовлена тим, що базові механіки 2D-платформерів сформували саму мову геймдизайну, а сучасний ренесанс інді-ігор демонструє стійкий попит на нові інтерпретації класичних механік переміщення у просторі [5].

Еволюційний шлях 2D-платформерів можна умовно розділити на кілька ключових епох, кожна з яких характеризувалася технологічними проривами у сфері апаратного та програмного забезпечення [8].

Первісні елементи вертикального переміщення з’явилися у грі *Space Panic* (1980), проте через відсутність механіки стрибка її зазвичай вважають лише передвісником жанру [4]. Справжнім початком еволюції платформерів прийнято вважати аркадний автомат *Donkey Kong* (1981) від компанії Nintendo, спроєктований Сігеру Міямото [8].

У цій грі персонаж Jumpman (який згодом став Маріо) уперше отримав можливість долати перешкоди за допомогою стрибка [5]. На цьому етапі механіка стрибка була жорстко детермінованою: траєкторія руху персонажа в повітрі розраховувалася за фіксованою параболою, і гравець не міг коригувати напрямок руху після відриву від поверхні [8].

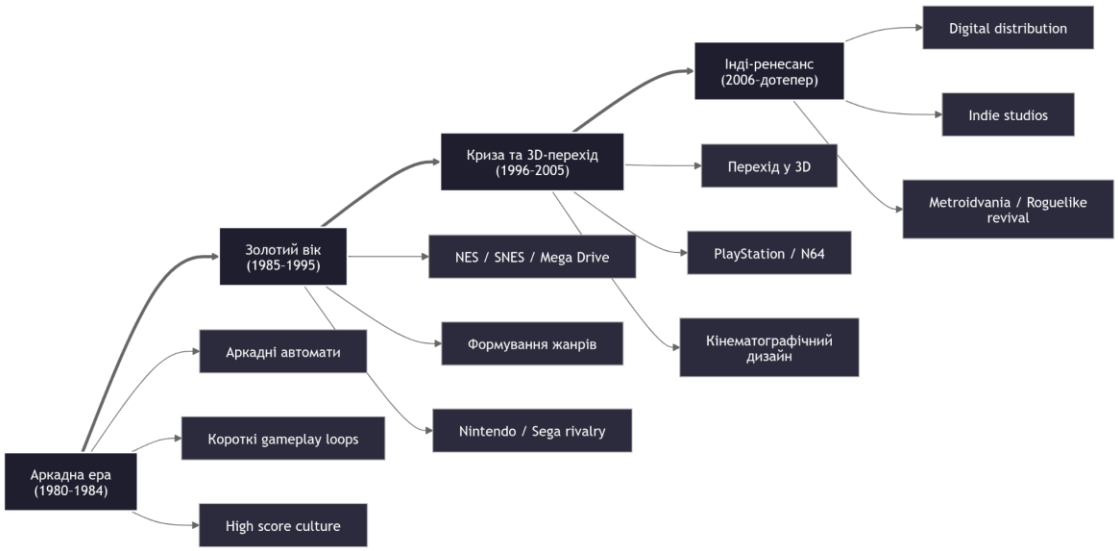

Рис.1.1. [Хронологічна шкала еволюції жанру 2D-платформерів

Наступним технологічним кроком став реліз *Pitfall!* (1982) для консолі Atari 2600. Гра продемонструвала концепцію лінійного зміщення екранів (горизонтальний скролінг ще не був безперервним, екран перемикався при досягненні межі), що значно розширило просторовий вимір гри та ввело поняття «ігрового світу», який складається з послідовності зон.

Золотий вік скролінг-платформерів (1985–1995)

Справжня революція відбулася у 1985 році з виходом *Super Mario Bros.* на платформі Nintendo Entertainment System (NES). Головним технологічним досягненням став плавний горизонтальний скролінг екрана, який рухався слідом за персонажем. Геймдизайнерські інновації включали:

1. Введення інерції руху (персонаж мав фазу розгону та гальмування) [1].
2. Варіативність висоти стрибка залежно від тривалості утримання кнопки [8].
3. Динамічний розрахунок колізій у реальному часі [5].

На початку 1990-х років компанія Sega запропонувала альтернативний погляд на жанр із виходом *Sonic the Hedgehog* (1991). Фокус змістився з точного просторового позиціонування на кінетичну швидкість та імпульс (momentum). Гра вимагала складніших фізичних розрахунків для взаємодії персонажа з криволінійними поверхнями (петлі, рампи), що змусило розробників оптимізувати

алгоритми обробки колізій [18].

Криза жанру та перехід у тривимірний простір (1996–2005)

Із появою 32-бітних консолей (Sony PlayStation, Nintendo 64) індустрія масово переорієнтувалася на 3D. Двовимірні платформери стали вважатися застарілими. Жанр трансформувався у тривимірні світи (*Super Mario 64*, *Crash Bandicoot*). Проте саме в цей період на портативних консолях (Game Boy Advance) та через окремі релізи на старших консолях продовжували розвиватися комплексні піджанри, зокрема метроїдванії, які заклали основу для майбутнього відродження [18].

Інді-ренесанс та сучасний етап (2006 – дотепер)

Поява цифрової дистрибуції (Steam, Xbox Live Arcade) та доступність ігрових рушіїв дали поштовх новій хвилі 2D-платформерів. Такі проєкти, як *Braid* (2008), *Super Meat Boy* (2010), *Fez* (2012) та *Celeste* (2018), довели, що двовимірний простір є ідеальним майданчиком для глибоких геймдизайнерських експериментів. Сучасні платформери роблять акцент на субпіксельній точності введення, інтеграції наративу в механіку та використанні складних алгоритмів процедурної генерації або симуляції фізичних середовищ [18, 8].

Сучасна класифікація 2D-платформерів базується на домінантній ігровій механіці (core mechanic), яка визначає темп гри, архітектуру рівнів (level design) та вимоги до когнітивних і моторних навичок гравця [8].

Піджанр	**Домінантна механіка**	**Архітектура рівнів**	**Ключові референси**
Класичний / Екшен-платформер	Баланс між стрибками та знищенням	Лінійна або квазілінійна, дискретні рівні	*Super Mario Bros.*, *Mega Man*, *Shovel*

	ворогів		*Knight*
Симулятор точного стрибка (Precision)	Субпіксельне позиціонування, таймінг мікросекунд	Короткі екран-кімнати з високою щільністю пасток	*Super Meat Boy*, *Celeste*, *I Wanna Be the Boshy*
Пазл-платформер (Puzzle)	Маніпуляція простором, часом чи станами об’єктів	Простір як головоломка, відсутність вимог до швидкості	*Braid*, *Limbo*, *Inside*, *Fez*
Метроїдванія (Metroidvania)	Дослідження, блокування прогресу здібностями (Ability gating)	Єдиний нелінійний взаємопов’язаний лабіринт	*Hollow Knight*, *Castlevania: SotN*, *Ori*

1. Симулятори точного стрибка (Precision Platformers)

Головною особливістю цього піджанру є мінімізація випадкових факторів (рандому) та максимальне підвищення вимог до моторної координації гравця. Гравець повинен виконувати послідовності дій із мінімальними вікнами для помилок.

Характерною рисою архітектури коду таких ігор є використання детермінованого фізичного рушія (часто самописного або сильно

модифікованого), оскільки стандартні фізичні рушії (на кшталт Box2D) мають мікроскопічні похибки через накопичення помилок округлення з плаваючою комою, що неприпустимо при потребі піксельної точності колізій [18, 8].

2. Пазл-платформери (Puzzle Platformers)

У цьому піджанрі механіка стрибка виступає не як тест на швидкість реакції, а як інструмент для переміщення між елементами просторової або логічної головоломки. Темп гри зазвичай уповільнений. Розробники часто вводять унікальну механіку, яка ламає звичні правила геометрії: зміна перспективи з 2D в 3D (*Fez*), відмотування часу назад для створення власних клонів (*Braid*) або маніпуляція фізичними властивостями світла й тіні [2].

3. Метроїдванії (Metroidvanias)

Особливий піджанр, що поєднує елементи платформера та Action-RPG. Його сутність полягає у відмові від концепції роздільних рівнів на користь єдиної, масштабної мапи світу. Доступ до нових зон обмежений відсутністю у персонажа певних інструментів чи здібностей (наприклад, подвійного стрибка, ривка або можливості руйнувати конкретні типи стін). Дизайн рівнів у метроїдваніях будується на принципі циклічності (backtracking), змушуючи гравця повертатися у вже відвідані локації після отримання нових апгрейдів [8].

Для проєктування ефективних механік у власному ігровому проєкті необхідно детально проаналізувати структурні та концептуальні патерни успішних представників жанру різних епох: *Super Mario Bros.*, *The Legend of Zelda* (в контексті впливу на топологію рівнів) та *Hollow Knight*.

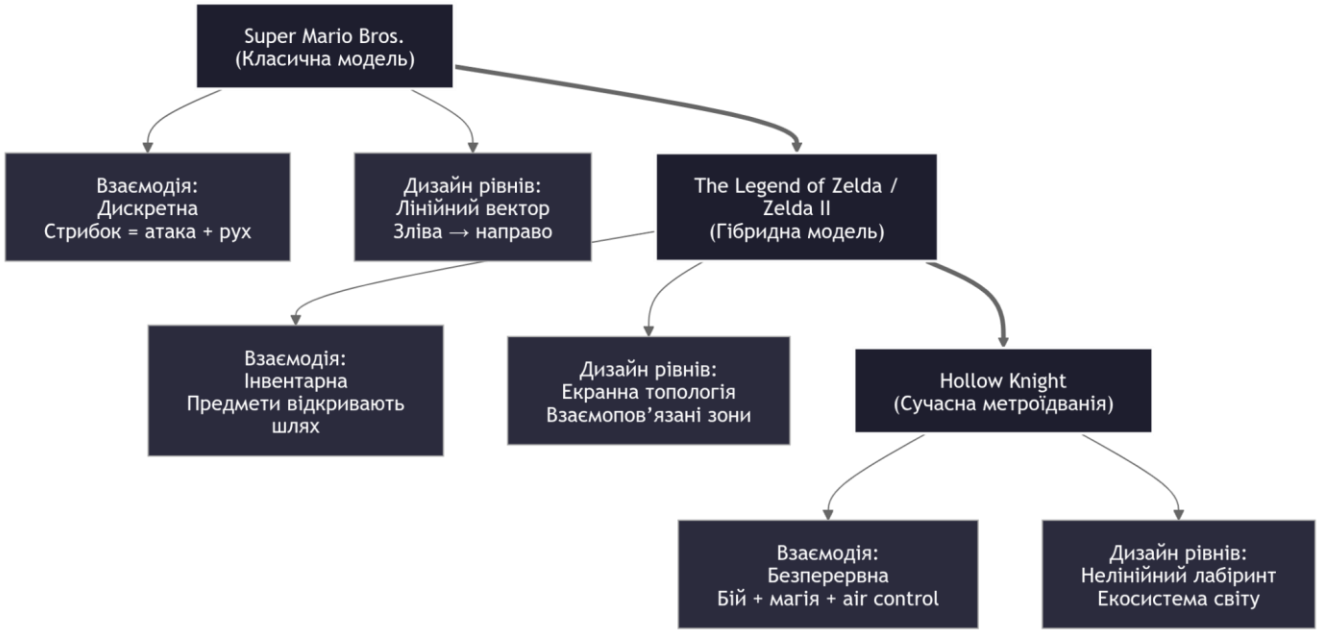

Рис.1.2. *Порівняльний аналіз архітектурних рішень у референтних проєктах*

Super Mario Bros.: Фундамент кінетичного дизайну

Гра *Super Mario Bros.* залишається еталоном побудови «ігрового відчуття» (game feel). Головний урок цього аналога – навчання гравця без використання текстових інструкцій за допомогою дизайну першого рівня (Світ 1-1) [5].

Геймдизайнерський патерн «Навчання через простір»: Гравець з'являється у лівій частині екрана, а вільний простір праворуч стимулює рух уперед. Перший ворог (Goomba) рухається назустріч, змушуючи гравця інстинктивно натиснути кнопку стрибка. Розташування блоків над ворогом гарантує, що навіть у разі невдалого стрибка гравець або вдариться об блок і знищить ворога, або отримає гриб, який демонструє механіку збільшення персонажа та механіку додаткового життя [5].

Фізична модель Маріо базується на чіткому розрахунку сил [18]:

1. **Тертя об поверхню:** персонаж не зупиняється миттєво, що створює відчуття маси.
2. **Залежність траєкторії від розгону:** чим швидше біжить персонаж, тим далі й вище він здатний стрибнути, що спонукає гравця до підтримки постійного ритму руху.

The Legend of Zelda (вплив на платформери через призму системного дизайну)

Хоча серія *The Legend of Zelda* (за винятком двовимірної *Zelda II: The Adventure of Link*) переважно є пригодницьким екшеном із топографією «вигляд зверху», її структурні патерни мали колосальний вплив на еволюцію 2D-платформерів, зокрема на піджанр метроїдваній та пазл-платформерів [8, 4].

У *Zelda II: The Adventure of Link* розробники безпосередньо поєднали бойову систему і дослідження світу у двовимірній перспективі (side-scrolling). Головним патерном, запозиченим із серії Zelda, є концепція **«Ключ-Замок» (Key-and-Lock progression)**, реалізована через геймплейні предмети [19]. Якщо в класичному платформері перешкода долається виключно механічною майстерністю гравця (вчасним натисканням кнопки), то під впливом дизайну Zelda перешкода долається через пошук структурного елемента ігрової логіки (наприклад, використання рукавиць для підняття важких каменів, що блокують платформу, або гака для зачеплення за дальні уступи). Це перетворило платформери з чистих симуляторів реакції на інтелектуальні дослідження віртуального простору [4].

Hollow Knight: Синергія бойової системи та просторової навігації

Випущений у 2017 році проєкт *Hollow Knight* від Team Cherry є вершиною розвитку сучасних метроїдваній. Гра використовує надзвичайно жорстку та чуйну (responsive) систему керування персонажем: відсутня тривала фаза інерційного розгону, зупинка відбувається майже миттєво, що критично для складних бойових зіткнень [18].

Ключові патерни залучення гравця в *Hollow Knight*:

1. Механіка «Pogo-stick» (Стрибок на цвяху): Гравець може атакувати ворогів або небезпечні об’єкти (шипи) спрямованим вниз ударом під час перебування в повітрі. Влучний удар відштовхує персонажа вгору, оновлюючи можливість виконання ривка (dash) або подвійного стрибка без приземлення на тверду поверхню. Цей патерн поєднує бойову систему з

механікою переміщення, перетворюючи ворогів на додаткові платформи [19].

2. Економіка «Душі» (Soul Economy): Ресурс для лікування та використання магічних здібностей накопичується виключно під час успішних ближніх атак на ворогів. Це створює постійний цикл ризику та винагороди: гравець змушений йти на небезпечне зближення з ворогом, щоб отримати ресурс для відновлення здоров'я [2].
3. Топографічний туман мапи: Гравець потрапляє в нові зони без орієнтирів. Для відкриття мапи необхідно знайти картографа Корніфера, а для відображення власного положення на ній – екіпірувати спеціальний амулет. Це посилює почуття першовідкривача та ізоляції у ворожому середовищі [8].

Ефективність 2D-платформера визначається не лише графічним оформленням чи сюжетом, а глибинною математичною та психологічною архітектурою ігрового процесу. Можна виділити три фундаментальні геймдизайнерські компоненти, які утримують увагу гравця [12].

Ігрове відчуття (Game Feel та «Juice»)

Поняття *Game Feel*, формалізоване дослідником Стівом Свінком, визначає суб'єктивне відчуття контролю над віртуальним об'єктом. Для досягнення позитивного ігрового відчуття розробники використовують приховані механіки компенсації людської реакції [18]:

1. **Час Койота (Coyote Time):** Тимчасове вікно тривалістю в кілька кадрів (зазвичай 3–6 кадрів при 60 FPS), протягом якого персонаж може здійснити стрибок, навіть якщо він уже зійшов із краю платформи і фактично перебуває в повітрі. Назва походить від персонажа мультфільмів Wile E. Coyote, який застигав у повітрі перед падінням. Ця механіка компенсує затримку введення (input lag) моніторів та швидкість реакції людини, запобігаючи відчуттю несправедливості гри [18].

2. **Буферизація стрибка (Jump Buffering):** Якщо гравець натискає кнопку стрибка за кілька кадрів до того, як персонаж торкнеться землі, система запам’ятовує це введення і виконує стрибок автоматично в момент приземлення. Без цього мікрокорекція призводила б до того, що персонаж «ігнорував» би занадто ранні команди гравця, створюючи відчуття неслухняного керування.

Стан потоку (Flow State) та петлі зворотного зв’язку

Згідно з психологічною теорією Мігая Чіксентмігаї, стан потоку досягається тоді, коли складність завдання ідеально відповідає рівню навичок користувача. У платформерах це регулюється геометричною прогресією складності рівнів [6].

Ігри на кшталт *Celeste* використовують петлю **«миттєвого повернення» (instant respawn)**: після смерті екран згасає всього на частку секунди, і персонаж миттєво з’являється на початку поточної кімнати. Відсутність екранів завантаження та штрафів у вигляді втрати прогресу не дає гравцеві вийти зі стану потоку, перетворюючи кожну невдачу на швидку ітерацію навчання, а не на покарання.

Просторовий ритм (Level Design Rhythm)

Дизайн рівнів у платформерах аналогічний створенню музичного треку. Розташування платформ, пасток та ворогів задає певний темп руху. Розробники чергують зони високої напруги (складні стрибки над шипами з таймінгом) та зони релаксації (безпечні платформи з точками збереження або колекційними предметами). Колекційні предмети (монети в *Super Mario*, полуниця в *Celeste*) часто виконують роль візуальних маркерів (breadcrumbs), які підсвідомо вказують гравцеві правильну траєкторію стрибка та оптимальну параболу польоту, спрощуючи просторову орієнтацію [8, 15].

1.2. Теоретичні аспекти геймдизайну та формалізація ігрових механік

Проєктування комп’ютерної гри вимагає чіткого розуміння того, як окремі

елементи взаємодіють між собою для створення цілісного ігрового досвіду. Успішний платформер не виникає стихійно; він є результатом ретельного проєктування, математичного моделювання та ітеративного тестування. Для того, щоб гравець відчував контроль над персонажем та отримував задоволення від подолання перешкод, розробнику необхідно формалізувати базові поняття геймдизайну та адаптувати їх під специфіку обраного жанру. У цьому параграфі розглядаються ключові теоретичні концепти розробки ігор, такі як ігрові механіки, ігровий цикл, баланс, а також специфічні патерни жанру 2D-платформерів та методи їх документування.

В основі будь-якого інтерактивного продукту лежать ігрові механіки. З академічної точки зору, ігрова механіка – це набір правил, алгоритмів та систем, які визначають, як гравець може взаємодіяти з ігровим світом, і як цей світ реагує на дії гравця. У контексті фреймворку MDA (Mechanics, Dynamics, Aesthetics), механіки є базовим рівнем, на якому оперує розробник. Вони включають математичні моделі, структури даних та алгоритми, які приховані від гравця, але безпосередньо впливають на його сприйняття [13, 4].

Формалізація механік означає перетворення абстрактної ідеї (наприклад, «персонаж вміє стрибати») у чіткий набір змінних та станів (швидкість, гравітація, вектор напрямку, перевірка зіткнень із землею). У двовимірних платформерах базовий набір механік зазвичай обмежується переміщенням (ходьба, біг, ривок) та подоланням вертикальних перешкод (стрибок, відскік від стіни). Проте саме глибина опрацювання цих базових механік відрізняє посередній проєкт від якісного.

Ігровий цикл (Core Gameplay Loop)

Важливим концептом у геймдизайні є ігровий цикл, або «core loop» – це повторювана послідовність дій, яку гравець виконує протягом усієї гри [14]. Структурно базовий ігровий цикл складається з трьох етапів: «Дія» (Action) -> «Реакція/Виклик» (Challenge/Reaction) -> «Нагорода/Зворотний зв’язок»

(Reward/Feedback).

Для жанру 2D-платформера цей цикл можна декомпозувати наступним чином:

1. Аналіз ситуації: гравець бачить перешкоду або ворога на екрані.
2. Прийняття рішення та дія: гравець розраховує траєкторію та натискає кнопку стрибка у певний момент часу.
3. Виконання механіки: рушій (у нашому випадку Godot) обробляє інпут, застосовує фізичні сили до персонажа та переміщує його.
4. Зворотний зв’язок та нагорода: персонаж успішно приземляється на наступну платформу (аудіовізуальний фідбек), просуваючись ближче до кінця рівня, або збирає колекційний предмет.

Розуміння та відшліфовування core loop є критично важливим етапом. Якщо базова дія (наприклад, стрибок) не приносить задоволення (відсутній так званий Game Feel), то повторення цього циклу тисячі разів за гру викличе у користувача лише фрустрацію [12].

Ігровий баланс: крива складності та стан потоку

Ігровий баланс – це процес налаштування змінних гри з метою забезпечення оптимального рівня виклику для гравця. Мета балансування полягає в утриманні гравця у стані «потоку» (Flow) – психологічному стані, описаному Міхаєм Чіксентміхаї, коли людина повністю занурена в процес [17]. Стан потоку досягається тоді, коли складність ігрових викликів зростає пропорційно до зростання навичок гравця.

У платформерах баланс реалізується через левел-дизайн та параметри персонажа. Занадто велика відстань між платформами при низькій швидкості гравця робить гру неможливою; занадто мала – нудною. Балансування також включає налаштування «ризику та нагороди» (Risk/Reward). Наприклад, розташування цінного предмета над прірвою змушує гравця свідомо ризикувати, використовуючи складніші комбінації механік [4].

Особливості проєктування механік для платформерів

Жанр платформерів є унікальним тим, що він повністю спирається на просторове мислення та мікроконтроль. Успішність проєкту залежить від того, наскільки інтуїтивно зрозумілою та передбачуваною є поведінка персонажа [2]. Для досягнення високої якості ігрового процесу необхідно детально спроєктувати наступні аспекти:

1. Фізика стрибка (Jump Physics)

У реальному житті фізика руху підкоряється законам Ньютона, де об’єкт має інерцію, а гравітація є константою. Проте у геймдизайні платфомерів «реалістична» фізика часто є ворогом «цікавої» гри. Наприклад, у серії Super Mario чи Hollow Knight гравітація є динамічною.

При проєктуванні стрибка розробник має враховувати такі параметри:

1. *Jump Force (Початкова сила стрибка)*: миттєве застосування сили по осі Y.
2. *Apex (Апогей стрибка)*: найвища точка траєкторії. Для покращення управління гравітацію часто тимчасово зменшують у точці апогею, що дозволяє гравцеві трохи «зависнути» в повітрі та точніше розрахувати місце приземлення.
3. *Variable Jump Height (Варіативна висота стрибка)*: висота стрибка має залежати від того, як довго гравець утримує кнопку. З точки зору програмування в Godot, це реалізується шляхом збільшення гравітаційного множника (Gravity Multiplier), якщо гравець відпускає кнопку стрибка до досягнення апогею. Це перериває стрибок і швидко тягне персонажа вниз, даючи гравцю абсолютний контроль над траєкторією [1, 2, 17].

Взаємодія з оточенням та колізії (Collisions)

Колізії – це математичний метод визначення перетину двох об’єктів у просторі. У 2D-платформерах найчастіше використовується AABB (Axis-Aligned Bounding Box) – прямокутні зони зіткнення, які не обертаються.

Формалізація колізій вимагає розділення їх на типи:

1. Hitboxes (Зони атаки): ділянки, які завдають шкоди (наприклад, зброя героя або шипи).
2. Hurtboxes (Зони вразливості): ділянки тіла персонажа, які можуть отримати шкоду.
3. Environment Collisions (Колізії геометрії): визначають, де є земля, стіни чи стеля. У Godot для цього використовується вузол типу CharacterBody2D (або KinematicBody2D у старих версіях), який надає вбудовані методи на кшталт move_and_slide() та is_on_floor() [7]. Важливим аспектом є налаштування масок та шарів (Collision Layers and Masks) для оптимізації обчислень, щоб гравець не перевіряв зіткнення з декоративними елементами фону [16].

3. Таймінги: Меха ніка Coyote Time

Однією з найпоширеніших проблем у платформерах є людська реакція. Часто гравець натискає кнопку стрибка на долю секунди пізніше, ніж персонаж зійшов з краю платформи. Сувора математична логіка (if not is_on_floor: jump = false) призведе до того, що персонаж просто впаде, і гравець відчує, що гра «не відреагувала» на його натискання, що викликає сильне роздратування [12].

Для вирішення цього когнітивного дисонансу геймдизайнери використовують патерн «Coyote Time» (названий на честь койота з мультфільмів Looney Tunes, який висів у повітрі, перш ніж впасти).

Алгоритмічно це реалізується так: розробник створює таймер (зазвичай 0.1 – 0.15 секунди, або 6-8 кадрів при 60 FPS). Коли персонаж залишає край платформи, він фактично перебуває в стані падіння, але таймер Coyote Time дозволяє йому здійснити повноцінний стрибок, якщо кнопка була натиснута до вичерпання цього часу. Це робить управління значно м’якшим і пробачає дрібні помилки гравця [12].

4. Таймінги: Механіка Jump Buffering (Буферизація стрибка)

Спорідненим до Coyote Time є патерн буферизації інпуту. Ситуація: персонаж падає на платформу, і гравець хоче здійснити стрибок одразу в момент

приземлення. Якщо гравець натисне кнопку за 5 кадрів до дотику до землі, базова логіка проігнорує цю дію (адже персонаж ще в повітрі), і після приземлення стрибка не відбудеться.

Механіка Jump Buffering зберігає натискання кнопки стрибка у пам'яті (у буфері) на короткий проміжок часу (наприклад, 0.15 секунди). Якщо протягом цього часу персонаж торкається землі, збережена команда виконується миттєво. Це забезпечує ідеальну плавність (флюїдність) рухів при швидкісному проходженні рівнів (спідранах) і створює відчуття високої чутливості керування [2].

Розробка концепт-документації

Проєктування всіх вищеописаних механік неможливе без належної формалізації у вигляді документації. Перш ніж написати перший рядок коду в Godot, геймдизайнер має створити Концепт-документ (Concept Document) або Vision Document, який згодом розширюється до повноцінного Дизайн-документа гри (GDD - Game Design Document) [4].

Концепт-документ слугує «компасом» проєкту. Його мета – відповісти на ключові питання щодо гри до початку ресурсомісткого етапу розробки:

1. *High Concept (Пітч)*: Короткий опис гри одним-двома реченнями (наприклад, «Двовимірний пазл-платформер, де гравець має маніпулювати гравітацією для вирішення просторових головоломок»).
2. *Цільова аудиторія та референси*: Визначення того, для кого створюється гра, та аналіз існуючих аналогів (наприклад, Celeste для хардкорних гравців або Ori and the Blind Forest для прихильників дослідження).
3. *Стовпи дизайну (Design Pillars)*: Три-чотири фундаментальні принципи, на які спирається весь проєкт. Наприклад: «Висока динаміка рухів», «Відсутність випадковостей (RNG)», «Читабельний піксель-арт».
4. *Core Mechanics (Опис базових механік)*: Перелік усіх можливостей гравця (стрибок, ривок, атака) з попереднім описом їхніх параметрів (відстань ривка,

час кулдауну).

5. *Метрика світу (World Metrics)*: Формалізація розмірів. В Godot 2D розміри вимірюються в пікселях. У концепт-документі необхідно чітко зафіксувати: розмір плитки тайлсету (наприклад, 16x16 або 32x32 пікселя), висоту персонажа, максимальну висоту стрибка у тайлах, максимальну ширину провалля, яку може перестрибнути персонаж. Це необхідно для того, щоб левел-дизайнер міг будувати рівні, гарантовано знаючи фізичні можливості героя [14].

Наступним етапом є створення Технічного дизайн-документа (TDD - Technical Design Document), де абстрактні механіки описуються мовою архітектури програмного забезпечення: структури класів, дерево вузлів рушія Godot (Node Tree architecture), обробка сигналів (Signals) та патерни програмування (наприклад, використання State Machine для контролю анімацій та станів гравця).

Таким чином, теоретичне проєктування та формалізація є обов’язковими етапами, що передують практичній розробці. Впровадження таких невидимих для звичайного користувача, але критично важливих механік, як Coyote Time, Jump Buffering та варіативна гравітація, дозволяє перетворити базовий прототип платфомера на професійний ігровий продукт із високим рівнем залучення гравця [3, 18, 2].

1.3. Обґрунтування вибору інструментальних засобів розробки: порівняльний аналіз ігрових рушіїв та архітектурні особливості Godot Engine

Перехід від етапу концептуального проєктування та формалізації ігрових механік до їх безпосередньої програмної реалізації вимагає обґрунтованого вибору базового інструментарію розробки. Сучасна індустрія комп’ютерних ігор пропонує широкий спектр ігрових рушіїв (Game Engines), які є комплексними

програмними середовищами, що об’єднують у собі графічний рендерер, фізичний рушій, систему обробки звуку, інструменти анімації_ та інтерфейс для написання ігрової логіки [3, 14]. Для розробки двовимірного платформера вибір релевантного рушія є критично важливим, оскільки надмірна складність інструменту може призвести до нераціонального використання апаратних ресурсів та часу розробника, тоді як недостатня функціональність – до неможливості реалізувати заплановані геймдизайнерські патерни (такі як складні фізичні колізії, багатошаровий паралакс або системи частинок) [1, 2, 14].

У даному параграфі проводиться комплексний порівняльний аналіз провідних ігрових рушіїв – Godot Engine, Unity, Unreal Engine та GameMaker – за ключовими критеріями: архітектура роботи з 2D-графікою, модель ліцензування, системні вимоги та доступні мови програмування. На основі цього аналізу формується обґрунтування вибору рушія Godot для реалізації проєкту, а також детально досліджуються його унікальні архітектурні парадигми (вузлова система та мова GDScript), що забезпечують високу швидкість прототипування [1, 13].

Критерії порівняльного аналізу ігрових рушіїв

Для забезпечення об’єктивності вибору інструментарію було визначено чотири фундаментальні критерії, які безпосередньо впливають на життєвий цикл розробки інді-проєкту та академічної кваліфікаційної роботи [14]:

- Специфіка рендерингу та роботи з двовимірним простором (2D): наявність виділеного 2D-рушія чи використання 3D-простору з ортогональною проєкцією.
- Модель ліцензування та відкритість вихідного коду: фінансові умови використання, права власності на кінцевий продукт, доступність вихідного коду для модифікацій.
- Системні вимоги та оптимізація: споживання ресурсів центрального та графічного процесорів як середовищем розробки (редактором), так і скомпільованим кінцевим продуктом.

- Екосистема програмування: доступні мови програмування, швидкість компіляції, зручність взаємодії коду з візуальними елементами рушія.

Аналіз роботи з двовимірною графікою (2D)

Розробка класичного 2D-платформера вимагає високої точності позиціонування об’єктів (Pixel-perfect точність), оптимізованої роботи зі спрайтами та тайлсетами (TileSets).

Unity історично створювався як 3D-рушій. Робота з 2D у Unity реалізована через використання тривимірного простору, де двовимірні об’єкти є плоскими полігонами (quads), на які натягнуті текстури, а камера переведена в режим ортогональної проєкції (Orthographic Camera). Хоча Unity пропонує потужні інструменти 2D-розробки (2D Lights, Sprite Shape), сама архітектура базується на 3D-математиці (використання векторів із віссю Z, розрахунок глибини через Z-буфер). Це додає певних обчислювальних накладних витрат.

Unreal Engine (UE) є індустріальним стандартом для створення AAA-ігор з реалістичною 3D-графікою. Робота з 2D у UE реалізовувалася через модуль Paper2D, проте на даний момент він не отримує активного розвитку від компанії Epic Games. Створення класичного 2D-платформера на Unreal Engine супроводжується боротьбою зі складною фізикою рушія, яка оптимізована під 3D-маси та об’єми, а не під дискретну піксельну логіку [4].

GameMaker з самого початку проєктувався як виключно двовимірний рушій. Він має ідеальну архітектуру для 2D, працює безпосередньо з піксельними координатами екрану і є стандартом для створення ретро-ігор (у ньому створені такі хіти, як Undertale та Hollow Knight).

Godot Engine має унікальну гібридну архітектуру. На відміну від Unity, Godot містить два абсолютно незалежні рендерери: один для 3D, і окремий для 2D. Робота в 2D-середовищі Godot відбувається у справжніх піксельних координатах (де початок координат (0,0) знаходиться у верхньому лівому куті екрана). Це дозволяє досягти максимальної точності та уникнути артефактів згладжування,

характерних для 3D-проєкцій. Вбудована система вузлів TileMap (або оновлений TileMapLayer у Godot 4.x) надає потужний інструментарій для створення рівнів платформера з підтримкою автоматичного тайлінгу (Autotiling) та налаштуванням полігонів колізій прямо в редакторі рушія [1, 13].

Політика ліцензування

У контексті бакалаврської роботи та потенційного релізу інді-гри ліцензування відіграє вирішальну роль.

Unity розповсюджується за пропрієтарною ліцензією з моделлю підписки. Нещодавні зміни в ліцензійній політиці компанії Unity Technologies (впровадження Runtime Fee – плати за встановлення гри користувачами) викликали значне занепокоєння серед незалежних розробників щодо непередбачуваності фінансових ризиків.

Unreal Engine пропонує доступну модель: рушій є безкоштовним до моменту, поки гра не заробить 1 мільйон доларів США, після чого стягується роялті у розмірі 5% від валового доходу.

GameMaker перейшов на складну багаторівневу систему підписок, і хоча базова версія стала безкоштовною для некомерційного використання, експорт на більшість платформ вимагає фінансових вкладень.

Godot Engine кардинально відрізняється від комерційних конкурентів. Це проєкт із відкритим вихідним кодом (Open-Source), який розповсюджується за ліцензією MIT [1]. Це означає, що розробник отримує абсолютну свободу: рушій є повністю безкоштовним для будь-яких цілей (включаючи комерційні), відсутні роялті, підписки чи приховані платежі. Крім того, наявність відкритого коду на C++ дозволяє розробникам самостійно модифікувати ядро рушія під специфічні потреби свого проєкту [5].

Системні вимоги та продуктивність середовища розробки

Незалежні проєкти часто розробляються на апаратному забезпеченні середнього рівня.

Unreal Engine має надзвичайно високі системні вимоги. Запуск редактора вимагає значних обсягів оперативної пам’яті (від 16 ГБ до 32 ГБ) та потужної дискретної відеокарти. Скомпільований порожній проєкт також має великий розмір базових бінарних файлів.

Unity займає проміжну позицію. Процес відкриття проєкту та компіляції коду може займати значний час (іноді хвилини), що уповільнює ітеративний процес розробки та тестування дрібних змін.

Godot Engine вирізняється безпрецедентною легкістю. Виконуваний файл рушія важить менше 100 мегабайт, не вимагає інсталяції в систему та запускається за лічені секунди. Він стабільно працює навіть на ноутбуках з інтегрованими графічними процесорами, що робить його ідеальним для розробки в будь-яких умовах. Завдяки оптимізованій системі збирання, час від натискання кнопки «Play» до запуску гри складає менше секунди, що кардинально підвищує продуктивність праці програміста [12, 3].

Мови програмування та архітектура логіки

Unity використовує C# як єдину офіційну мову. Це потужна, строго типізована об’єктно-орієнтована мова, яка чудово підходить для масштабних проєктів, але вимагає написання значного обсягу «бойлерплейт-коду» (boilerplate code) для простих завдань [14, 16].

Unreal Engine базується на складному C++ з власними макросами та системі візуального скриптування Blueprints. C++ забезпечує максимальну продуктивність, але має найвищий поріг входження, тоді як Blueprints можуть перетворитися на неструктурований «спагетті-код» при реалізації складної математичної логіки.

GameMaker використовує власну GameMaker Language (GML), яка з роками еволюціонувала, але залишається нішевою і мало застосовується поза межами цього рушія.

Godot Engine підтримує декілька мов офіційно: C++, C# (через версію .NET)

та власну скриптову мову GDScript. Можливість використання GDScript для написання логіки верхнього рівня та геймплею, з опціональним винесенням «важких» обчислювальних алгоритмів на C++, є вагомою перевагою Godot [17].

Підсумовуючи порівняльний аналіз, можна стверджувати, що для розробки 2D-платформера в рамках бакалаврської роботи Godot Engine є найбільш релевантним та раціональним вибором. Він надає спеціалізований інструментарій для 2D, відрізняється високою швидкістю роботи, не обмежується комерційними ліцензіями та дозволяє сфокусуватися на геймдизайні, а не на боротьбі з архітектурою 3D-рушія.

Архітектурні переваги Godot: Вузлова (Node) система та композиція сцен

Однією з найважливіших відмінностей Godot від таких систем, як Unity, є відмова від класичної архітектури «GameObject-Component» на користь глибоко ієрархічної вузлової системи (Node-based architecture) та парадигми «все є сценою» (Everything is a Scene) [1, 3, 15].

В архітектурі Unity (ECS або традиційній компонентній), розробник створює порожній об’єкт (GameObject), на який «навішує» різні компоненти (Transform, SpriteRenderer, Collider, Rigidbody, користувацькі скрипти). Об’єкт є просто контейнером [16].

У Godot базовою одиницею є Вузол (Node). Вузол – це об’єктно-орієнтований клас, який має певну спеціалізовану функцію (наприклад, Sprite2D малює картинку, CollisionShape2D визначає форму зіткнення, AudioStreamPlayer відтворює звук). Кожен вузол може мати будь-яку кількість дочірніх вузлів. Разом вони формують дерево вузлів (Scene Tree). Коли гілка такого дерева зберігається на диск, вона стає Сценою (Scene) [1, 17].

Сцена в Godot є фундаментальним будівельним блоком, який реалізує принцип композиції в об’єктно-орієнтованому програмуванні. Сцена може представляти окрему зброю, ворога, елемент інтерфейсу або весь ігровий рівень. Будь-яку створену сцену можна інстанціювати (instantiate) в іншій сцені як єдиний

вузол.

Для розробки платформера ця архітектура є ідеальною. Наприклад, створюється сцена Player, яка містить кореневий вузол CharacterBody2D (що відповідає за кінематичну фізику). До нього дочірніми вузлами додаються AnimatedSprite2D (анімація), CollisionShape2D (хітбокс гравця) та Camera2D (щоб камера слідувала за гравцем). Ця сцена повністю ізольована та самодостатня (інкапсульована). Її можна помістити на будь-який рівень (сцену Level_1), і вона одразу почне працювати. Якщо необхідно створити кілька типів ворогів, можна створити базову сцену Enemy, а потім наслідувати (Scene Inheritance) від неї FlyingEnemy та GroundEnemy, перевизначаючи лише їхні специфічні характеристики (спрайти чи поведінку AI). Така організація архітектури дозволяє створювати надзвичайно чистий, масштабований та модульний код, повністю уникаючи дублювання.

Ще одним архітектурним патерном, глибоко інтегрованим у Godot, є система Сигналів (Signals), яка є реалізацією класичного патерну Спостерігач (Observer). Сигнали дозволяють вузлам спілкуватися між собою, не маючи жорстких посилань один на одного. Наприклад, коли гравець торкається шипів, вузол Area2D (шипи) генерує сигнал body_entered. Скрипт гравця або інтерфейсу може підписатися на цей сигнал і зменшити шкалу здоров’я. Це забезпечує слабку зв’язність коду (loose coupling), що значно полегшує підтримку та розширення гри [15].

GDScript як інструмент швидкого прототипування та реалізації ігрової логіки

Для написання ігрової логіки в рамках даного проєкту було обрано мову GDScript. Це високорівнева, динамічно типізована мова програмування, синтаксис якої візуально нагадує Python. Вона була розроблена спеціально для Godot Engine, що забезпечує унікальний рівень її інтеграції з рушієм [13].

Використання GDScript дає низку суттєвих переваг на етапі реалізації ігрових механік платформера:

1. *Відсутність затримок на компіляцію.* На відміну від C#, код на GDScript компілюється в байткод миттєво під час збереження файлу. Це дозволяє впроваджувати принцип швидкого прототипування (Rapid Prototyping) – розробник може змінити швидкість стрибка гравця або гравітацію, і одразу ж перевірити це в грі, не чекаючи «збірки» проєкту.
2. *Спеціалізовані типи даних для ігрової математики.* Мова містить вбудовані типи для векторної алгебри (наприклад, Vector2 та Vector3). Обчислення напрямків, нормалей, швидкостей (velocity) та відбиття (bounce) для обробки колізій займає мінімум рядків коду завдяки вбудованим математичним методам (таким як move_toward, lerp, normalized).
3. *Тісна взаємодія з деревом сцени.* Оскільки GDScript написаний спеціально для Godot, він має синтаксичний цукор для доступу до вузлів сцени. Оператор $ або директива @onready дозволяють миттєво отримувати посилання на інші вузли в ієрархії, що робить маніпуляцію об’єктами інтуїтивно зрозумілою (наприклад, $Sprite2D.play(«run») викликає анімацію бігу).
4. *Поступова типізація (Gradual Typing).* Починаючи з Godot 3.1 та з подальшим покращенням у Godot 4.x, GDScript підтримує строгу типізацію. Розробник може вказувати типи змінних, аргументів функцій та значень, що повертаються (наприклад, var speed: float = 300.0). Це підвищує безпеку коду, знижує кількість помилок на етапі виконання та оптимізує продуктивність рушія.
5. *Управління пам’яттю на основі підрахунку посилань (Reference Counting).* GDScript автоматично керує пам’яттю для більшості об’єктів (крім вузлів, які розробник має видаляти явно за допомогою queue_free()). Це мінімізує проблему «фрізів» під час роботи збирача сміття (Garbage Collector), яка часто є проблемою в іграх на C# або Java.

Завдяки вбудованому фізичному рушію та методам на кшталт

move_and_slide() у базовому класі CharacterBody2D, базова фізика платформера реалізується десятком рядків коду. Це вивільняє час розробника для фокусування на складних та унікальних геймдизайнерських механіках: реалізації станів кінцевого автомата (State Machine) для гравця, налаштуванні «coyote time» (можливості стрибнути протягом кількох фреймів після падіння з платформи), буферизації стрибка (jump buffering) та створення поведінки ворожих об’єктів [5, 15].

РОЗДІЛ 2. ПРАКТИЧНА РЕАЛІЗАЦІЯ ІГРОВИХ МЕХАНІК ТА АРХІТЕКТУРИ ДВОВИМІРНОГО ПЛАТФОРМЕРА У СЕРЕДОВИЩІ GODOT ENGINE

2.1. Архітектура ігрового проєкту та проєктування сцени персонажа

Перехід від концептуального етапу геймдизайну до безпосередньої програмної реалізації вимагає чіткого визначення архітектури майбутнього ігрового проєкту. Сучасні ігрові рушії, зокрема Godot Engine 4-ї версії, пропонують високий ступінь свободи в організації файлової системи та побудові логічних зв’язків між ігровими об’єктами. Однак ця свобода без належної попередньої формалізації може призвести до неконтрольованого зростання технічного боргу, ускладнення рефакторингу та проблем із масштабуванням проєкту. Даний параграф присвячений розробці базової архітектури проєкту, створенню оптимальної структури каталогів, а також фундаментальному етапу розробки платформера – проєктуванню сцени головного персонажа з

використанням вузлової системи (Node-системи) рушія Godot.

Основою будь-якого програмного забезпечення є зрозуміла та логічно структурована файлова система. У контексті розробки ігор, де проєкт складається не лише з вихідного коду, але й з великої кількості медіа-ресурсів (текстури, аудіофайли, шрифти, анімації), грамотна організація директорій стає критично важливою [4, 14].

Рушій Godot використовує віртуальну файлову систему, кореневий каталог якої позначається префіксом res:// (від слова resource). На відміну від багатьох інших рушіїв, які нав’язують розробнику жорстку структуру, Godot дозволяє довільну організацію. Для даної бакалаврської роботи було обрано компонентно-орієнтований підхід до структурування ресурсів, який базується на групуванні файлів за їхнім функціональним призначенням, а не за типом файлу.

Структура каталогів проєкту має наступний ієрархічний вигляд:

- res://assets/ – коренева тека для всіх імпортованих зовнішніх ресурсів, яка, у свою чергу, поділяється на:
  - /sprites/ – графічні ресурси (спрайт-шити, тайлсети, фони);
  - /audio/ – звукові ефекти (SFX) та фонова музика (BGM);
  - /fonts/ – файли шрифтів (.ttf, .otf) для інтерфейсу користувача.
- res://scenes/ – тека для збереження ігрових сцен (файли з розширенням .tscn). Оскільки архітектура Godot заохочує модульність, ця директорія структурована за рівнем абстракції:
  - /entities/ – сцени ігрових сутностей (гравець, вороги, NPC);
  - /levels/ – сцени ігрових рівнів та їхніх шаблонів;
  - /ui/ – сцени інтерфейсу користувача (головне меню, HUD, меню паузи);
  - /interactables/ – об’єкти, з якими можлива взаємодія (предмети для збору, чекпоінти, перемикачі).
- res://scripts/ – глобальна директорія для скриптів мовою GDScript

(файли .gd). Хоча Godot дозволяє зберігати скрипти поруч зі сценами (що є доречним для специфічних вузлів), глобальні системи, такі як синглтони (Singletons) та менеджери станів (State Managers), розміщуються саме тут, у підкаталозі /autoload/.

Такий архітектурний підхід забезпечує високий рівень інкапсуляції. Наприклад, якщо виникає необхідність модифікувати логіку або зовнішній вигляд головного героя, розробник звертається до однієї конкретної директорії, де зібрані всі пов'язані з ним ресурси, що значно пришвидшує навігацію в умовах розростання кодової бази. Окрім того, така структура ідеально підходить для використання систем контролю версій (наприклад, Git), мінімізуючи ризики виникнення конфліктів при злитті гілок (merge conflicts).

У жанрі двовимірного платформера головний персонаж є центральним об'єктом, через який гравець взаємодіє з віртуальним світом. Уся фізика гри, відчуття управління (Game Feel), динаміка стрибків та реакція на колізії залежать від того, наскільки грамотно спроєктована система управління персонажем.

У рушії Godot об'єкти конструюються шляхом комбінування базових блоків – вузлів (Nodes). Сцена (Scene) являє собою ієрархічне дерево вузлів, яке зберігається у файловій системі та може бути інстанційоване (створено як екземпляр) в інших сценах необмежену кількість разів. Для створення головного героя було прийнято рішення створити окрему незалежну сцену Player.tscn.

Першим та найважливішим кроком є вибір кореневого вузла (Root Node) для цієї сцени. Фізичний рушій Godot 2D пропонує три основні типи фізичних тіл для динамічних об'єктів: StaticBody2D, RigidBody2D та CharacterBody2D.

StaticBody2D не підходить, оскільки призначений для нерухомої геометрії (стіни, підлога).

RigidBody2D керується виключно фізичним рушієм. Рух такого об'єкта реалізується через застосування сил (forces) та імпульсів (impulses). Хоча це реалістично, для класичного платформера такий підхід є хибним. Гравцеві

потрібен точний, аркадний і миттєвий контроль над персонажем: здатність миттєво зупинятися, міняти напрямок руху в повітрі, ігнорувати інерцію. Досягти цього за допомогою RigidBody2D вкрай складно і вимагає постійного «ламання» фізичних законів рушія.

CharacterBody2D (який у попередніх версіях Godot 3.x мав назву KinematicBody2D) є ідеальним вибором для нашого завдання. Цей вузол створений спеціально для персонажів, якими керує гравець або штучний інтелект. Він не підкоряється глобальній силі тяжіння автоматично і не реагує на інші фізичні тіла без явного програмування. Замість цього розробник має повний контроль над вектором швидкості (властивість velocity). Переміщення здійснюється за допомогою вбудованого методу move_and_slide(), який автоматично розраховує колізії, ковзання вздовж стін та забезпечує плавну зупинку при зіткненні, повертаючи інформацію про нормалі поверхонь.

Перехід розробників Godot у 4-й версії від KinematicBody2D до CharacterBody2D значно спростив розробку платформерів, оскільки новий вузол вже містить вбудовані властивості velocity та напрямку вгору (up_direction), які раніше доводилося передавати як аргументи у функцію переміщення. Саме на базі CharacterBody2D будується логіка нашого протагоніста.

Після встановлення CharacterBody2D як кореневого вузла, його необхідно розширити дочірніми компонентами, які відповідатимуть за візуалізацію, визначення фізичних меж, спостереження та анімацію. Згідно з принципами композиції, сцена гравця була спроєктована з використанням наступної ієрархії:

Візуалізація: Вузол Sprite2D

Оскільки базовий вузол CharacterBody2D є лише абстрактною математичною точкою у просторі, він потребує графічного представлення. Для цього як дочірній вузол додається Sprite2D. Цей компонент відповідає за рендеринг 2D-текстури на екран.

У контексті сучасного проєктування ігор замість використання десятків

окремих файлів для кожного кадру анімації використовується єдиний аркуш спрайтів (Sprite Sheet). У властивостях вузла Sprite2D в розділі *Animation* налаштовуються параметри Hframes (кількість кадрів по горизонталі) та Vframes (кількість кадрів по вертикалі). Це дозволяє рушію автоматично нарізати текстуру на однакові сегменти. Зміна властивості frame дозволяє швидко перемикатися між кадрами анімації (біг, стрибок, падіння).

Також важливим етапом є налаштування параметра *Offset* (зміщення). Для платформерів найкращою практикою є встановлення точки прив’язки (Origin) внизу по центру моделі (Bottom-Center), а не по центру мас. Це значно спрощує математичні розрахунки при розміщенні персонажа на поверхні (підлозі), оскільки координати об’єкта будуть точно відповідати координатам точки дотику ніг персонажа з землею.

Обробка зіткнень: Вузол CollisionShape2D

Для того, щоб гравець не провалювався крізь текстури та міг взаємодіяти з перешкодами, фізичному рушію необхідно надати інформацію про геометрію тіла персонажа. За це відповідає вузол CollisionShape2D.

Вибір правильної форми (Shape) для платформера є критично важливим архітектурним рішенням. Може здатися, що прямокутник (RectangleShape2D) є логічним вибором для двовимірного спрайту, проте на практиці прямокутні колайдери викликають проблему «чіпляння за кути» (corner snagging). Під час переміщення по рівню, складеному з тайлів (TileMap), ідеально рівна поверхня з погляду фізичного рушія складається з мікро-стиків. Прямокутний колайдер з його гострими кутами часто зачіпається за ці стики, викликаючи раптові зупинки гравця (jitter).

Тому для розробки даної гри було обрано форму капсули (CapsuleShape2D). Її заокруглена нижня частина дозволяє персонажу плавно ковзати по мікро-нерівностях та автоматично згладжує підйом на невеликі перешкоди (наприклад, сходинки висотою в один піксель), що значно покращує загальний Game Feel.

Верхня частина капсули аналогічно запобігає застряганню голови при стрибках у вузькі тунелі.

Контроль поля зору: Вузол Camera2D

Наступним компонентом сцени є Camera2D. Вона забезпечує відстеження просторового положення персонажа. Прикріплення камери як дочірнього вузла безпосередньо до гравця змушує її автоматично наслідувати його трансформації (рух по осях X та Y).

Однак жорстка прив’язка камери створює дискомфорт для зору гравця: кожен найменший рух персонажа миттєво центрується, що призводить до дезорієнтації. Для вирішення цієї проблеми у вузлі Camera2D було налаштовано кілька ключових властивостей:

Position Smoothing (Згладжування позиції): Активація цієї функції змушує камеру слідувати за гравцем із певною затримкою (lerp-інтерполяція), що робить рух камери кінематографічним та м’яким.

Drag Margin (Мертва зона): Було створено «невидиму коробку» в центрі екрана. Поки гравець рухається в межах цієї зони, камера залишається нерухомою. Камера починає рух лише тоді, коли персонаж наближається до країв екрана. Це особливо важливо під час виконання серії швидких коротких стрибків – камера не стрибає вгору-вниз разом із героєм, знижуючи навантаження на вестибулярний апарат гравця (motion sickness).

Limits (Обмеження): Для запобігання виходу камери за межі спроєктованого рівня встановлюються жорсткі математичні ліміти (Left, Top, Right, Bottom).

Останнім ключовим елементом сцени є AnimationPlayer. На відміну від простішого вузла AnimatedSprite2D, AnimationPlayer є потужним інструментом, що дозволяє анімувати абсолютно будь-яку властивість будь-якого вузла в сцені (наприклад, змінювати розмір колайдера під час присідання гравця, модулювати колір спрайту при отриманні шкоди або відтворювати звук кроку на конкретному кадрі).

В AnimationPlayer створюються окремі анімації (біг, стрибок, падіння, стан спокою). Кожна анімація містить ключові кадри (keyframes), прив’язані до часової шкали. Властивість frame вузла Sprite2D анімується шляхом створення дискретної (discrete) доріжки значень.

На етапі проєктування закладається фундамент для програмного управління цими анімаціями. Інтеграція скрипту (GDScript) з AnimationPlayer дозволить пізніше реалізувати патерн «Скінченний автомат» (Finite State Machine, FSM), який гарантуватиме, що гравець не зможе відтворити анімацію бігу, знаходячись у повітрі під час стрибка.

Проєктування архітектури та побудова сцени головного персонажа є базисом, на якому ґрунтується вся подальша розробка двовимірного платформера. Використання компонентної Node-системи рушія Godot дозволило створити гнучку, ізольовану та масштабовану сутність гравця. Вибір CharacterBody2D забезпечує точний контроль над кінематикою об’єкта, використання капсульної колізії нівелює проблеми застрягання, а налаштування камери зі згладжуванням значно підвищує ергономічність ігрового процесу. Розроблена файлова структура та ієрархія вузлів повністю підготовлені до наступного етапу – написання програмного коду на мові GDScript для імплементації фізичних законів та обробки користувацького вводу.

2.2. Програмна реалізація базових та розширених ігрових механік мовою GDScript

На етапі практичної реалізації двовимірного платформера ключовим завданням є трансляція теоретичних математичних та фізичних моделей ігрового процесу у програмний код. Для забезпечення гнучкості, стабільності та високої продуктивності використовується об’єктно-орієнтована скриптова мова GDScript, інтегрована в рушій Godot 4. Базовим вузлом для створення керованих персонажів у сучасних версіях рушія є CharacterBody2D, який надає розширений API для

обробки кінематичних зіткнень за допомогою методу move_and_slide(). У цьому параграфі розглядається архітектура та програмний код ключових систем гри: переміщення, гравітації, взаємодії зі світом та управління станами [3, 4].

У класичних платформерах миттєвий перехід від стану спокою до максимальної швидкості створює відчуття «штучності» та жорсткості управління. Для забезпечення органічного ігрового досвіду (Game Feel) необхідно імплементувати механіки прискорення (інерції) та гальмування (тертя).

З математичної точки зору, швидкість гравця V_x має лінійно наближатися до цільової швидкості V_{target} з певним кроком, який залежить від часу побудови кадру (delta), щоб забезпечити незалежність фізики від частоти кадрів (FPS). У GDScript для цього використовується вбудована функція move_toward().

Лістинг 2.1 демонструє базову структуру налаштування констант та реалізацію горизонтального руху в циклі _physics_process(delta).

(GDScript)

```
extends CharacterBody2D
class_name Player

const MAX_SPEED: float = 300.0
const ACCELERATION: float = 1200.0
const FRICTION: float = 1600.0

func _physics_process(delta: float) -> void:
    var direction := Input.get_axis(«move_left», «move_right»)

    if direction != 0:
        # Реалізація інерції (прискорення)
        velocity.x = move_toward(velocity.x, direction * MAX_SPEED,
ACCELERATION * delta)
```

```
else:
    # Реалізація тертя (гальмування)
    velocity.x = move_toward(velocity.x, 0, FRICTION * delta)
```

Функція Input.get_axis() повертає значення від -1 до 1. Якщо гравець натискає клавішу руху, обчислюється вектор напрямку. Функція move_toward безпечно інтерполює поточну швидкість velocity.x до бажаної (direction * MAX_SPEED), використовуючи ACCELERATION * delta як крок. Якщо ввід відсутній, цільовою швидкістю стає 0, а кроком – показник тертя FRICTION. Такий підхід гарантує, що персонаж не зупинятиметься миттєво, а пройде певний гальмівний шлях, що є критичним для платформ, які вимагають точності (наприклад, крижані рівні, де значення FRICTION можна динамічно зменшувати).

Гравітація у двовимірному просторі Godot обчислюється як постійне збільшення вектора швидкості по осі Y (напрямок вниз). Для узгодженості проєкту з глобальними налаштуваннями рушія, значення гравітації отримується безпосередньо з ProjectSettings.

Стрибок – це найважливіша механіка будь-якого платформера. Щоб надати гравцю більше контролю, базовий стрибок (надання миттєвого імпульсу JUMP_VELOCITY по осі Y) було розширено механікою «варіативної висоти стрибка». Вона полягає в тому, що висота польоту персонажа залежить від тривалості утримання кнопки стрибка.

```
(GDScript)
    var gravity = ProjectSettings.get_setting(«physics/2d/default_gravity»)
const JUMP_VELOCITY: float = -450.0
const JUMP_CUT_MULTIPLIER: float = 0.5

func handle_gravity_and_jump(delta: float) -> void:
# Застосування гравітації, якщо персонаж не на підлозі
```

```
    if not is_on_floor():
        velocity.y += gravity * delta

# Ініціалізація стрибка
    if Input.is_action_just_pressed(«jump») and is_on_floor():
        velocity.y = JUMP_VELOCITY

# Переривання стрибка (механіка утримання кнопки)
    if Input.is_action_just_released(«jump») and velocity.y < 0:
        velocity.y *= JUMP_CUT_MULTIPLIER
```

Програмна логіка варіативного стрибка обробляється умовою Input.is_action_just_released(«jump»). Перевірка velocity.y < 0 гарантує, що ми обрізаємо імпульс лише тоді, коли персонаж рухається вгору (у Godot вісь Y направлена вниз, тому рух вгору є від'ємним). Якщо гравець відпускає кнопку зарано, вертикальна швидкість множиться на 0.5 (коефіцієнт обрізання), змушуючи персонажа швидше втрачати інерцію підйому і починати падіння. Це дозволяє здійснювати як високі стрибки для подолання великих перешкод, так і короткі «мікро-стрибки» для ухилення від ворогів. Після всіх маніпуляцій з вектором velocity, обов'язково викликається метод move_and_slide(), який автоматично розраховує колізії та оновлює стан is_on_floor().

Зі збільшенням кількості механік (рух, стрибок, атака, отримання шкоди) використання каскаду умовних операторів if/elif/else у _physics_process призводить до утворення «спагеті-коду», який складно розширювати та підтримувати. Для вирішення цієї архітектурної проблеми в проєкті імплементовано патерн «Скінченний автомат» (Finite State Machine, FSM).

Суть FSM полягає в тому, що об'єкт може перебувати лише в одному з чітко визначених станів у конкретний момент часу, і переходи між ними строго

регламентовані.

Для реалізації FSM у GDScript використано конструкцію enum та оператор зіставлення із шаблоном match.

(GDScript)

```
enum State { IDLE, RUN, JUMP, FALL, HURT }
var current_state: State = State.IDLE
@onready var anim_player = $AnimationPlayer

func update_state() -> void:
  match current_state:
    State.IDLE:
      anim_player.play(«idle»)
      if velocity.x != 0:
        change_state(State.RUN)
      elif not is_on_floor():
        change_state(State.FALL)

    State.RUN:
      anim_player.play(«run»)
      if velocity.x == 0:
        change_state(State.IDLE)
      elif velocity.y < 0:
        change_state(State.JUMP)
      elif velocity.y > 0 and not is_on_floor():
        change_state(State.FALL)

    State.JUMP:
      anim_player.play(«jump»)
```

```
        if velocity.y >= 0:
            change_state(State.FALL)

    State.FALL:
        anim_player.play(«fall»)
        if is_on_floor():
            if velocity.x == 0:
                change_state(State.IDLE)
            else:
                change_state(State.RUN)

func change_state(new_state: State) -> void:
    if current_state != new_state:
        current_state = new_state
# Тут можна додати логіку при вході в новий стан (enter state logic)
```

Така декомпозиція логіки дозволяє повністю ізолювати візуальну складову (виклик анімацій через AnimationPlayer) від фізичних розрахунків. Метод update_state() викликається в кінці циклу _physics_process після розрахунку колізій. Автомат станів гарантує, що анімація бігу ніколи не відтвориться у повітрі, а стан HURT зможе тимчасово перехопити управління на себе, ігноруючи інпут гравця.

Об’єктно-орієнтований підхід до програмування ігор вимагає слабкої зв’язності компонентів (Loose Coupling). Для реалізації збору предметів (наприклад, монет чи ключів) у Godot оптимальним архітектурним рішенням є використання патерну «Спостерігач» (Observer), який у рушії реалізований через Сигнали (Signals).

Колекційні предмети проєктуються як окремі сцени на базі вузла Area2D,

який має зону колізії CollisionShape2D. Замість того, щоб гравець кожного кадру перевіряв, чи не торкнувся він монети, монета сама відстежує проникнення чужорідного фізичного тіла у свій тригер.

(GDScript)

```
# Скрипт Coin.gd (підключений до Area2D)
extends Area2D
signal item_collected(value: int)
@export var coin_value: int = 10

func _ready() -> void:
# Динамічне підключення сигналу до глобального менеджера гри
self.item_collected.connect(GameManager._on_item_collected)

func _on_body_entered(body: Node2D) -> void:
# Перевіряємо, чи тіло, що увійшло в зону, є Гравцем
  if body is Player:
    emit_signal(«item_collected», coin_value)
# Програвання звуку/анімації зникнення
    queue_free()
```

У цьому коді метод _on_body_entered є слухачем сигналу body_entered самої Area2D. Оператор is в GDScript безпечно перевіряє приналежність об’єкта до класу Player. При успішній перевірці предмет генерує власний сигнал item_collected, передаючи номінал предмета, і викликає queue_free() для безпечного видалення свого вузла з оперативної пам’яті в кінці поточного кадру. Глобальний синглтон GameManager отримує цей сигнал і оновлює графічний інтерфейс (HUD) та внутрішню статистику рівня.

Для організації бойової системи та нанесення шкоди реалізовано

компонентний підхід із використанням спеціалізованих зон: Hitbox (зона, що завдає шкоди) та Hurtbox (зона, що отримує шкоду).

У класичних платформерах механіка знищення ворога часто базується на стрибку йому на голову (аналог Super Mario). Для цього ворог (вузол CharacterBody2D) містить дві Area2D:

- Hurtbox (вразлива зона, розміщена у верхній частині ворога).
- Hitbox (розміщена по контуру ворога, завдає шкоди гравцю).

Коли гравець входить у Hurtbox ворога, перевіряється вектор падіння.

```
# Скрипт Enemy.gd
func _on_head_hurtbox_body_entered(body: Node2D) -> void:
  if body is Player:
# Перевіряємо, чи гравець падає вниз (швидкість у позитивна)
    if body.velocity.y > 0:
      die()
      body.bounce_off_enemy()
    # Метод у скрипті гравця для відштовхування

func die() -> void:
# Виклик ефектів смерті
  queue_free()
```

У скрипті гравця метод bounce_off_enemy() надає персонажу зворотний вертикальний імпульс (velocity.y = JUMP_VELOCITY * 0.8), створюючи відчуття пружності при знищенні супротивника та дозволяючи гравцю використовувати

ворогів як трампліни для доступу до секретних зон рівня.

Програмна реалізація ігрових механік мовою GDScript у середовищі Godot Engine дозволила створити стабільну та оптимізовану систему керування персонажем. Застосування вбудованих математичних функцій (таких як move_toward) забезпечило плавне налаштування кінематики, а інтеграція патерну «Скінченний автомат» вирішила проблему масштабованості логіки станів гравця. Архітектура, побудована на взаємодії через вузли Area2D та механізм сигналів, звела до мінімуму жорстку залежність класів, гарантуючи гнучкість при подальшому додаванні нових типів предметів та ворогів до ігрового світу.

2.3. Конструювання ігрового простору за допомогою інструменту TileMap та дизайн рівнів

Після успішної програмної реалізації базових ігрових механік та створення контролера головного персонажа, наступним критичним етапом розробки двовимірного платформера є конструювання ігрового простору. Ігровий простір (або рівень) виступає не лише візуальним фоном, але й безпосереднім середовищем, з яким взаємодіють усі розроблені раніше механіки. Для забезпечення високої продуктивності, гнучкості та швидкості ітерацій у процесі левел-дизайну в рушії Godot Engine використовується парадигма сіткового (тайлового) конструювання. Даний параграф присвячений аналізу та практичній реалізації процесу створення рівнів за допомогою інструментів TileSet та TileMap, налаштуванню фізичної взаємодії об’єктів через систему шарів та масок колізій (Collision Layers and Masks), а також архітектурній організації інтерфейсу користувача (UI/HUD) з використанням принципів слабкої зв’язності.

Основи сіткового конструювання: використання TileSet та TileMap

У сучасній розробці двовимірних ігор використання великих монолітних зображень для створення фону та геометрії рівня є вкрай неефективним підходом

з огляду на споживання оперативної та відеопам’яті. Замість цього застосовується метод тайлінгу (tiling), який полягає у розбитті візуального простору на дискретні комірки однакового розміру – тайли (tiles).

У середовищі Godot Engine цей підхід реалізується через взаємодію двох ключових компонентів: ресурсу TileSet та вузла TileMap (або TileMapLayer у новітніх версіях Godot 4.x).

Ресурс TileSet виконує роль бази даних або бібліотеки фрагментів. На етапі підготовки графічних ассетів імпортується текстурний атлас (spritesheet) – єдине зображення, що містить усі можливі елементи оточення: блоки землі, платформи, декоративні елементи, шипи тощо. Використання єдиного атласу дозволяє рушію оптимізувати процес відмальовування (Draw Calls), відправляючи графічному процесору інформацію про всю графіку рівня за один виклик. У налаштуваннях TileSet було визначено базовий розмір комірки (наприклад, 16x16 або 32x32 пікселі), що відповідає метриці ігрового світу та розмірам головного героя.

Окрему увагу під час налаштування TileSet було приділено системі «Терайн» (Terrains) – розширеному інструменту автотайлінгу (Autotiling) у Godot 4. Ця система дозволяє автоматизувати процес малювання складних структур, таких як масиви землі. Шляхом налаштування бітових масок для кожного тайлу (визначення сусідніх тайлів за правилами 3x3 Minimal або 2x2), рушій самостійно обирає правильний графічний фрагмент (кутовий, центральний, боковий) залежно від контексту. Це скоротило час на проєктування рівнів у кілька разів, дозволяючи зосередитись безпосередньо на архітектурі геймдизайну, а не на ручному підборі правильних куточків платформ.

Вузол TileMap безпосередньо відповідає за розміщення тайлів на сцені. Для забезпечення ефекту глибини (паралаксу) та відокремлення інтерактивних елементів від декоративних, архітектура TileMap була розділена на кілька логічних шарів (Layers):

- Background (Фон): Шар для декоративних елементів (стіни печер,

далекі дерева), які не мають фізичних властивостей.

- Foreground/Action (Основний шар): Шар, що містить інтерактивну геометрію рівня – підлогу, стіни, платформи. Саме для цього шару в TileSet генеруються полігони колізій (Collision Polygons).
- Decorations (Декорації): Шар для дрібних деталей поверх геометрії (трава, каміння, покажчики).

Фізична архітектура простору: Collision Layers та Masks

У грі жанру платформер на екрані одночасно можуть знаходитись десятки об’єктів: гравець, різноманітні вороги, рухомі платформи, снаряди, предмети колекціонування (монети) та небезпечні зони (шипи). Якщо всі ці об’єкти оброблятимуться в єдиному фізичному просторі, виникне проблема надлишкових обчислень та логічних конфліктів (наприклад, вороги почнуть збирати монети гравця, або монети будуть відштовхувати одна одну).

Для вирішення цієї проблеми фізичний рушій Godot пропонує систему фізичних шарів (Collision Layers) та масок (Collision Masks). В основі цієї системи лежить побітова арифметика (bitwise operations), де кожен шар є окремим бітом у 32-бітній цілочисельній змінній.

Концептуальна різниця між цими параметрами є фундаментальною для архітектури гри:

1. Layer (Шар) визначає, чим є цей об’єкт. (Відповідь на питання «Де я знаходжусь?»).

2. Mask (Маска) визначає, з чим цей об’єкт може стикатися або що він може сканувати. (Відповідь на питання «Що я бачу?»).

У межах розроблюваного проєкту було розроблено та імплементовано наступну матрицю фізичних взаємодій:

- Layer 1 (World/Environment): Належить вузлу TileMap (колізії землі та стін). Маска: порожня (світ статичний і ні з чим активно не взаємодіє).
- Layer 2 (Player): Належить вузлу CharacterBody2D головного героя.

Маска: 1 (World), 3 (Enemies), 4 (Collectibles), 5 (Hazards). Це означає, що гравець стоїть на землі, отримує пошкодження від ворогів та шипів, і може збирати монети.

● Layer 3 (Enemies): Належить ворогам. Маска: 1 (World), 2 (Player). Вороги не бачать монети (проходять крізь них) і не зіштовхуються з іншими ворогами, що запобігає застряганню великих груп супротивників у вузьких проходах рівня.

● Layer 4 (Collectibles): Належить інтерактивним предметам (монети, аптечки). Вузли Area2D. Маска: 2 (Player). Предмети чекають лише на перетин з тілом гравця.

● Layer 5 (Hazards/Hitboxes): Спеціальний шар для зон ураження (наприклад, область атаки гравця або вогняна пастка).

Така сувора типізація фізичних шарів значно оптимізує розрахунки фізичного рушія (Broad Phase та Narrow Phase колізій). Замість того, щоб перевіряти перетин полігонів монети та супротивника кожного кадру, рушій ігнорує цю пару ще на етапі перевірки бітових масок, що є операцією складання, яка виконується процесором за один такт. Це критично важливо для збереження стабільних 60 FPS на мобільних пристроях або слабких ПК.

Організація інтерфейсу користувача (UI/HUD)

Будь-яка сучасна гра потребує системи зворотного зв’язку з гравцем – Heads-Up Display (HUD). У платформері до базових елементів інтерфейсу належать індикатор здоров’я (Health Bar) та лічильник зібраних предметів (Score/Coins).

Проєктування інтерфейсу у Godot базується на системі вузлів Control, які мають власні правила позиціонування (Anchors та Margins), що адаптуються під різні розширення екрана. Для того, щоб елементи UI залишалися статичними на екрані незалежно від руху віртуальної камери Camera2D, яка слідує за гравцем, весь HUD був розміщений всередині спеціального вузла CanvasLayer. Цей вузол створює окремий простір відмальовування, який ігнорує глобальні координати

світу і завжди прив’язаний до координат вікна програми.

Структура UI в проєкті виглядає наступним чином:

1. Вузол CanvasLayer (названий HUD).
2. Всередині нього – MarginContainer для створення відступів від країв екрана.
3. VBoxContainer (вертикальний контейнер) для впорядкування елементі зверху вниз.
4. Для здоров’я використано вузол TextureProgressBar. Він дозволя використовувати власні графічні ассети (Under, Over, Progress) замість стандартни системних смуг, що допомагає підтримувати візуальний стиль гри.
5. Для лічильника монет використано HBoxContainer, всередині яког знаходяться TextureRect (іконка монети) та Label (текстовий лічильник).

Найскладнішим архітектурним завданням при створенні UI є спосіб передачі даних від гравця до інтерфейсу. Жорстке зв’язування (Tight Coupling), коли скрипт гравця напряму шукає вузол інтерфейсу get_node(«/root/HUD»).update_health(), є антипатерном. Якщо рівень завантажується без HUD (наприклад, під час катсцени), скрипт гравця видасть помилку (Null Reference Exception) і гра аварійно завершить роботу.

Для уникнення цього була застосована концепція подієво-орієнтованого програмування за допомогою системи сигналів (Signals) – реалізації класичного патерну «Спостерігач» (Observer Pattern) у Godot. У скрипті гравця (Player.gd) було задекларовано власні сигнали: signal health_changed(current_health, max_health) та signal coin_collected(total_coins).

Коли гравець отримує пошкодження, він лише емітує сигнал: health_changed.emit(hp, max_hp), не турбуючись про те, хто його прослуховує. Скрипт HUD, у свою чергу, підписується на ці сигнали при ініціалізації сцени та автоматично оновлює стан TextureProgressBar та текст Label. Така архітектура забезпечує абсолютну модульність та безпеку коду.

Практичний дизайн першого рівня (Level Design)

Після налаштування інструментарію було розроблено прототип першого (навчального) рівня. Дизайн рівня спирається на філософію «невидимого навчання» (Invisible Tutorial), популяризовану класичними проєктами Nintendo.

Рівень умовно поділено на три зони:

1. Безпечна зона (Safe Zone): Зона старту. Тут немає ворогів чи небезпек. Перешкоди налаштовані так, щоб гравець не міг просунутися вперед, не засвоївши базову механіку стрибка (перестрибнути через ящик) та подвійного стрибка (висока платформа).
2. Зона введення загрози: З'являється перший супротивник, але він знаходиться в ізольованій ямі або на нижньому рівні. Гравець може спостерігати за його патернами руху (State Machine Patrol), залишаючись у безпеці, та обрати стратегію – вступити в бій (стрибнути на голову) або обійти.
3. Зона виклику (Challenge Zone): Комбінування механік. Гравцю необхідно здійснити точний стрибок над ямою з шипами, уникаючи снарядів ворога.

Використання TileMap та системи колізій дозволило швидко ітерувати цей процес. Проведення плейтестів (Playtesting) вказало на місця, де гравці найчастіше роблять помилки, після чого топологія рівня була миттєво відредагована шляхом стирання або додавання нових тайлів без необхідності переписувати програмний код чи перебудовувати складні 3D-моделі.

Конструювання ігрового простору та інтерфейсу користувача є невід'ємною частиною розробки повноцінного ігрового продукту. Використання вузлів TileMap та TileSet у рушії Godot забезпечило високу швидкість побудови рівнів та оптимальне використання системних ресурсів завдяки батчингу текстур (Texture Batching). Налаштування фізичної взаємодії через систему Collision Layers та Masks гарантувало коректну ізоляцію ігрових сутностей та запобігло виникненню логічних колізій. Впровадження подієво-орієнтованої архітектури за допомогою

сигналів для керування UI (CanvasLayer, TextureProgressBar) дозволило досягти максимальної гнучкості та модульності коду, що відповідає сучасним стандартам розробки програмного забезпечення.

ВИСНОВКИ

У ході виконання роботи було виконано усі поставленні завдання: проаналізовано еволюцію, особливості та успішні патерни геймдизайну у жанрі 2D-платформерів; формалізовано теоретичні аспекти проєктування ігрових механік, фізики руху, таймінгів та колізій для ігор цього типу; проведеноно порівняльний аналіз сучасних ігрових рушіїв для розробки 2D-ігор та обґрунтовано вибір рушія Godot; спроєктовано архітектуру ігрового проєкту, структуру сцени персонажа; програмно реалізовано механіки руху, гравітації, взаємодії з об'єктами та систему станів персонажа (State Machine) мовою GDScript.

Також досліджено теоретичні, методологічні та практичні аспекти проєктування і реалізації двовимірної комп'ютерної гри жанру платформер. Підтверджено актуальність жанру, який, пройшовши еволюційний шлях від аркадних автоматів (Donkey Kong) до сучасного інді-ренесансу (Celeste, Hollow Knight), продовжує формувати мову геймдизайну. Завдяки успішному теоретичному аналізу та обґрунтованого вибору інструментальних засобів, було створено стійку, модульну архітектуру проєкту.

Проєкт базувався на всебічному аналізі історії жанру, який поділено на ключові епохи: від первісних елементів вертикального переміщення (Space Panic) та зародження стрибка (Donkey Kong) до «Золотого віку скролінг-платформерів» (Super Mario Bros.) та сучасного етапу, де домінують складні піджанри, як-от метроїдванії та симулятори точного стрибка. Була розроблена чітка класифікація 2D-платформерів за домінантною механікою (класичний екшен, симулятор точного стрибка, пазл-платформер, метроїдванія).

На етапі обґрунтування вибору ігрового рушія було проведено порівняльний аналіз Godot Engine, Unity, Unreal Engine та GameMaker за критеріями 2D-архітектури, ліцензування, системних вимог та мов програмування.

Вибір Godot Engine визнано найбільш раціональним завдяки розглянутими у

роботі ключовими перевагам:

Практичний розділ роботи продемонстрував успішну реалізацію складних механік та модульної архітектури, що відповідає сучасним стандартам розробки.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Бреславець В.С. Технології розробки комп’ютерних ігор: довідник модуля. / В.С. Бреславець. Х.: «Друкарня Мадрид», 2018. 162 с.
2. Джонатан Хеннеси, Джек МакГоуэн. Історія відеоігор у коміксах/ Неймовірна історія революції електронних ігор. Yakaboo Publishing. 2020. 192 с.
3. Лугова Т.А., Блажко О.А. Проектування комп’ютерних ігор для навчання : навчальний підручник . / Одеса : ФОП «Побута». 2018. 212 с. URL: http://dspace.opu.ua/jspui/bitstream/123456789/9287/1/%D0%9F%D1%96%D0%B4%D1%80%D1%83%D1%87%D0%BD%D0%B8%D0%BA%20Designing%20computer%20games%20for%20learning.pdf
4. Офіційна документація рушія Godot Engine. URL: https://docs.godotengine.org/en/stable/
5. Adams E. Fundamentals of Game Design. 3rd ed. Berkeley: New Riders, 2014. 336 p.
6. Anthropy A., Clark N. A Game Design Vocabulary: Exploring the Foundational Principles of Game Design. Upper Saddle River: Addison-Wesley, 2014. 208 p.
7. Bradfield C. Godot Engine Game Development Projects: Build five cross-platform 2D and 3D games with Godot. Birmingham: Packt Publishing, 2021. 486 p.
8. Bycer J. 2D Platformer Design: Fundamentals of Rhythm and Design. Boca Raton: CRC Press, 2020. 250 p. DOI: https://doi.org/10.1201/9780429299029
9. Csikszentmihalyi M. Flow: The Psychology of Optimal Experience. New York: Harper Perennial Modern Classics, 2008. 336 p.
10. Féret E. Developing 2D Games with Godot: A Complete Beginner’s Guide. New York: Apress, 2022. 373 p.
11. Fullerton T. Game Design Workshop: A Playcentric Approach to Creating Innovative Games. 4th ed. Boca Raton: CRC Press, 2018. 520 p. DOI: https://doi.org/10.1201/9781315313621
12. Haas J. A History of the Unity Game Engine: Thesis. Worcester: Worcester

Polytechnic Institute, 2014. 64 p. DOI: https://doi.org/10.15760/etd.3203 (інколи може відрізнятись залежно від репозитарію, але цей відповідає архівним записам)

13. Juul J. Half-Real: Video Games between Real Rules and Fictional Worlds. Cambridge: MIT Press, 2005. 248 p. DOI: https://doi.org/10.7551/mitpress/3234.001.0001

14. Linietsky J., Manzur A., Godot Community. Godot Engine Documentation (4.x) [Electronic resource]. 2023. Available at: https://docs.godotengine.org/en/stable/ (accessed: 03.06.2026).

15. Nystrom R. Game Programming Patterns. Genever Benning, 2014. 354 p.

16. Rogers S. Level Up! The Guide to Great Video Game Design. 2nd ed. Chichester: Wiley, 2014. 552 p.

17. Schell J. The Art of Game Design: A Book of Lenses. 3rd ed. Boca Raton: CRC Press, 2019. 640 p. DOI: https://doi.org/10.1201/9781351803636

18. Swink S. Game Feel: A Game Designer’s Guide to Virtual Sensation. Burlington: Morgan Kaufmann, 2008. 352 p. DOI: https://doi.org/10.1016/B978-0-12-374328-1.00001-9

19. Thorn A. Game Engine Design and Implementation. Burlington: Jones & Bartlett Learning, 2014. 900 p.