МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет імені В.Н. Каразіна

Факультет: **ННІ Каразінський банківський інститут**
Кафедра: **Інформаційних технологій та математичного моделювання**
Спеціальність: **122 Комп'ютерні науки**
Освітня програма: **Комп'ютерні науки та інформаційні технології в бізнесі**
Група: АК-21М **денна форма навчання**

КВАЛІФІКАЦІЙНА МАГІСТЕРСЬКА РОБОТА

на тему:

РОЗРОБКА 2D-ГРИ З ІНТЕРАКТИВНИМИ МЕХАНІКАМИ СТВОРЕННЯ ТА РУЙНУВАННЯ ІГРОВОГО СЕРЕДОВИЩА

ЗА НАКАЗОМ № 4601-5 3262 ВІД 15 вересня 2025 РОКУ

Здобувача вищої освіти **Фаткуліна Валерія Валерійовича**

Робота допущена до захисту в ЕК
протокол кафедри ІТММ № від _____2025р.

Завідувач кафедри ІТММ
к.п.н., доцент
_____ Н.І. Стяглик

Науковий керівник
к.т.н.
_____ А.В. Рогов

м. Харків 2025 р.

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет імені В. Н. Каразіна

Факультет навчально-науковий інститут "Каразінський банківський інститут"
Кафедра інформаційних технологій та математичного моделювання
Рівень вищої освіти другий (Магістерський)
Спеціальність 122 Комп'ютерні науки
Освітня програма Комп'ютерні науки та інформаційні технології в бізнесі

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ **Н. І. Стяглик**
Підпис ініціали, прізвище

"15" вересня 2025 року

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ (ПРОЄКТ)

Фаткуліну Валерію Валерійовичу
(прізвище, ім'я, по батькові студента)

1. Тема роботи: Розробка 2D-гри з інтерактивними механіками створення та руйнування ігрового середовища

керівник роботи Рогов Андрій Володимирович, к.т.н.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від "15" вересня 2025 року № 4601-5_3262

2. Строк подання студентом роботи 24 листопада 2025 року

3. Перелік питань, які потрібно розробити:

У розділі 1: Провести аналіз предметної області гібридних 2D-ігор, виконати порівняння аналогів та сформулювати функціональні й нефункціональні вимоги до прототипу.

У розділі 2: Обґрунтувати вибір технологічного стеку шляхом порівняння з альтернативами, а також спроєктувати об'єктно-орієнтовану архітектуру проєкту з використанням патернів проєктування.

У розділі 3: Описати практичну реалізацію ключових програмних модулів.

У розділі 4: Провести модульне та інтеграційне тестування розробленого прототипу, проаналізувати його продуктивність та сформулювати висновки щодо досягнення мети роботи і напрямків подальшого розвитку.

4.План роботи

№ з/п	Назви етапів роботи
1	Вибір здобувачем теми кваліфікаційної магістерської роботи
2	Затвердження плану і завдання кваліфікаційної магістерської роботи
3	Здача кваліфікаційної магістерської роботи керівнику
4	Підпис кваліфікаційної магістерської роботи керівника
5	Підпис кваліфікаційної магістерської роботи у нормоконтролера
6	Допуск завідувачем кафедри до захисту кваліфікаційної магістерської роботи
7	Захист кваліфікаційної магістерської роботи

5. Дата видачі завдання 15 вересня 2025 року

Студент _____ Фаткулін В.В.
підпис ініціали, прізвище

Керівник роботи _____ Рогов А.В.
підпис ініціали, прізвище

РЕФЕРАТ

НА КВАЛІФІКАЦІЙНУ МАГІСТЕРСЬКУ РОБОТУ

«РОЗРОБКА 2D-ГРИ З ІНТЕРАКТИВНИМИ МЕХАНІКАМИ СТВОРЕННЯ ТА РУЙНУВАННЯ ІГРОВОГО СЕРЕДОВИЩА»

Фаткуліна Валерія Валерійовича

Обсяг: 62 сторінки, 12 рисунків, 2 таблиці, 27 джерел у списку літератури.

Об'єктом дослідження є процес розробки інтерактивних 2D-ігор з використанням мови програмування Java.

Предметом дослідження є методи програмної реалізації ігрових механік, зокрема фізики 2D-платформера, системи анімації, обробки вводу та управління станами гри за допомогою стандартної бібліотеки Java Swing.

Метою роботи є дослідження, проєктування та розробка програмного прототипу 2D PvP гри з елементами будівництва та захоплення точок.

Для досягнення поставленої мети в роботі вирішено такі завдання:

1. Проведено аналіз предметної області та існуючих ігор-аналогів у жанрі 2D PvP платформерів для визначення ключових вимог до проєкту.
2. Обґрунтовано вибір технологічного стеку (Java, Swing) та спроєктовано об'єктно-орієнтовану архітектуру гри з використанням патернів проєктування.
3. Розроблено програмну реалізацію ігрового циклу та системи управління станами гри.
4. Реалізовано фізичну модель руху персонажів, систему виявлення зіткнень та механізми обробки введення для двох гравців.
5. Впроваджено специфічні ігрові механіки: фазу будівництва руйнівних платформ, систему стрільби та логіку захоплення контрольних точок.
6. Інтегровано графічні ресурси та розроблено систему спрайтової анімації персонажів.

7. Виконано тестування розробленого прототипу.

У роботі обґрунтовано вибір технологічного стеку Java 21 та Swing, проаналізовано існуючі ігри-аналоги (Таблиця 1.1) та спроєктовано об'єктно-орієнтовану архітектуру (Рис. 2.1) з використанням патернів проєктування, таких як «Стан» (State) та «Фабричний метод» (Factory Method).

Практична частина демонструє ручну реалізацію 2D-фізики платформера, включаючи метод розділення осей (AABB Separation) та покращення керування («Coyote Time», «Jump Buffering»). Розроблено надійну систему обробки одночасного вводу для двох гравців («Stateful Input»).

Реалізовано ключові ігрові механіки: фазу будівництва (з процедурною генерацією «скляних» платформ), систему стрільби та руйнування, логіку захоплення точок та підрахунок очок. Інтегровано графічні ресурси, включаючи систему анімації зі спрайт-смуг та завантаження рівнів з текстового файлу.

В результаті створено повноцінний ігровий цикл (меню, фази гри, екран завершення). У Розділі 4 проведено тестування функціоналу та аналіз продуктивності. Робота підтверджує, що бібліотека Swing є дієвим інструментом для швидкого прототипування ігрових механік з нуля.

КЛЮЧОВІ СЛОВА: JAVA, SWING, 2D ГРА, ПЛАТФОРМЕР, PVP, ІГРОВИЙ РУШІЙ, ФІЗИКА, AABB, РОЗДІЛЕННЯ ОСЕЙ, АНІМАЦІЯ, ПАТЕРНИ ПРОЄКТУВАННЯ, МАШИНА СКІНЧЕННИХ СТАНІВ (FSM), БУДІВНИЦТВО, ЗАХОПЛЕННЯ ТОЧОК, ОБРОБКА ВВОДУ.

ABSTRACT

FOR A QUALIFICATION MASTER'S THESIS

«DEVELOPMENT OF A 2D GAME WITH INTERACTIVE MECHANICS FOR CREATING AND DESTROYING THE GAME ENVIRONMENT»

Fatkulin Valerii

Volume: 62 pages, 12 figures, 2 tables, 27 sources in the bibliography.

The object of the study is the process of developing interactive 2D games using the Java programming language.

The subject of the study is the methods of software implementation of game mechanics, in particular the physics of a 2D platformer, animation systems, input processing, and game state management using the standard Java Swing library.

The aim of the work is to research, design and develop a software prototype of a 2D PvP game with elements of construction and point capture.

To achieve the set goal, the following tasks were solved in the work:

1. An analysis of the subject area and existing analogue games in the 2D PvP platformer genre was carried out to determine the key requirements for the project.
2. The choice of technology stack (Java, Swing) was justified and an object-oriented game architecture was designed using design patterns.
3. The software implementation of the game cycle and the game state management system was developed.
4. A physical model of character movement, a collision detection system, and input processing mechanisms for two players were implemented.
5. Specific game mechanics were implemented: a destructive platform construction phase, a shooting system, and logic for capturing control points.
6. Graphic resources were integrated and a character sprite animation system was developed.

7. The developed prototype was tested.

The work justifies the choice of the Java 21 and Swing technology stack, analyses existing similar games (Table 1.1) are analysed, and an object-oriented architecture (Figure 2.1) is designed using design patterns such as State and Factory Method.

The practical part demonstrates the manual implementation of 2D platformer physics, including the AABB Separation method and control improvements (‘Coyote Time’, ‘Jump Buffering’). A reliable system for processing simultaneous input for two players (‘Stateful Input’) has been developed.

Key game mechanics have been implemented: the construction phase (with procedural generation of ‘glass’ platforms), the shooting and destruction system, the logic of capturing points, and scoring. Graphic resources have been integrated, including an animation system with sprite strips and level loading from a text file.

As a result, a complete game cycle (menu, game phases, completion screen) has been created. In Section 4, functionality testing and performance analysis were carried out. The work confirms that the Swing library is an effective tool for rapid prototyping of game mechanics from scratch.

KEYWORDS: JAVA, SWING, 2D GAME, PLATFORMER, PVP, GAME ENGINE, PHYSICS, AABB, AXIS SEPARATION, ANIMATION, DESIGN PATTERNS, FINITE STATE MACHINE (FSM), CONSTRUCTION, POINT CAPTURE, INPUT PROCESSING.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧОК, СИМВОЛІВ І ТЕРМІНІВ

AABB (Axis-Aligned Bounding Box) — осьовирівняні обмежувальні прямокутники; метод виявлення зіткнень, що використовується в грі.

API (Application Programming Interface) — прикладний програмний інтерфейс; набір інструментів для створення програмного забезпечення.

BGM (Background Music) — фонова музика.

CPU (Central Processing Unit) — центральний процесор; апаратний компонент, на якому виконуються обчислення та рендеринг графіки Swing.

DRY (Don't Repeat Yourself) — принцип розробки програмного забезпечення, спрямований на уникнення дублювання інформації та коду.

EDT (Event Dispatch Thread) — потік обробки подій; основний потік у бібліотеці Swing, який відповідає за обробку подій графічного інтерфейсу.

FPS (Frames Per Second) — кадрова частота; кількість кадрів, що відображаються на екрані за одну секунду.

FR (Functional Requirements) — функціональні вимоги; перелік функцій, які повинна виконувати система.

GUI (Graphical User Interface) — графічний інтерфейс користувача.

HUD (Heads-Up Display) — елемент інтерфейсу гри, що відображається поверх ігрового процесу (рахунок, таймер).

IDE (Integrated Development Environment) — інтегроване середовище розробки програмного забезпечення.

JDK (Java Development Kit) — комплект розробника застосунків мовою Java, що включає компілятор та стандартні бібліотеки.

JVM (Java Virtual Machine) — віртуальна машина Java; середовище, що забезпечує виконання Java-коду на різних операційних системах.

KISS (Keep It Simple, Stupid) — принцип проєктування, який стверджує, що більшість систем працюють найкраще, якщо вони залишаються простими,

а не ускладненими.

LTS (Long-Term Support) — версія програмного забезпечення з довгостроковою підтримкою (використано Java 21 LTS).

NFR (Non-functional Requirements) — нефункціональні вимоги; вимоги до якості роботи системи (продуктивність, зручність, надійність).

OOP (Object-Oriented Programming, ООП) — об'єктно-орієнтоване програмування; парадигма програмування, заснована на концепції об'єктів.

PvP (Player-versus-Player) — режим гри «гравець проти гравця», в якому учасники змагаються один з одним.

SFX (Sound Effects) — звукові ефекти.

Swing — бібліотека для створення графічного інтерфейсу користувача для програм на мові Java.

Буферизація стрибка (Jump Buffering) — техніка покращення керування, яка реєструє натискання клавіші стрибка незадовго до приземлення персонажа.

Геймплей (Gameplay loop) — ігровий цикл; повторюваний набір дій, які виконує гравець для досягнення цілей гри.

Машина скінченних станів (Finite State Machine, FSM) — математична модель обчислень та патерн проєктування, використаний для керування станами гри (меню, гра, кінець гри).

Спрайт (Sprite) — двовимірне растрове зображення, що є частиною графічного об'єкта в грі.

Час Койота (Coyote Time) — ігрова механіка, що дає гравцеві можливість виконати стрибок протягом короткого часу після того, як він зійшов з платформи.

ВСТУП

Актуальність даної роботи зумовлена постійним зростанням ігрової індустрії та попитом на інтерактивні розважальні продукти. Ринок казуальних та "гіпер-казуальних" ігор демонструє стабільне зростання, а локальні PvP (Player-versus-Player) ігри залишаються популярним способом соціальної взаємодії. Метою даної дипломної роботи є дослідження, проєктування та розробка програмного прототипу комп'ютерної гри у жанрі 2D PvP платформера з елементами будівництва та захоплення точок. Цей вибір дозволяє застосувати широкий спектр знань з алгоритмів, структур даних, принципів об'єктно-орієнтованого проєктування та програмної інженерії [1].

Об'єктом дослідження є процес розробки інтерактивних графічних додатків, зокрема 2D-ігор, з використанням мови програмування Java.

Предметом дослідження є конкретні ігрові механіки (фізика 2D-платформера, система анімації, обробка вводу, управління станами гри) та методи їх програмної реалізації за допомогою стандартної бібліотеки Java Swing[9].

Проєкт отримав робочу назву «Нагодуй улюбленця!» та використовує тематику протистояння домашніх тварин (кота та собаки), які змагаються за їжу, представлену у вигляді точок захоплення. Окрім стандартних для платформера механік руху, стрибків та стрільби, гра включає унікальні елементи: фазу будівництва тимчасових платформ та можливість їх руйнування.

Основною метою було створення функціонального прототипу на Java та Swing, що дозволило зосередитись на реалізації ігрової логіки з нуля, не заглиблюючись у вивчення складних сторонніх ігрових рушіїв, таких як Unity чи Godot.

У даній роботі детально описано процес розробки гри, починаючи з

аналізу предметної області та вибору технологій, і закінчуючи реалізацією конкретних ігрових механік та інтеграцією графічних ресурсів. Структура звіту включає: аналіз предметної області (Розділ 1), опис практичних аспектів та архітектури (Розділ 2), опис реалізації (Розділ 3), висновки та список використаних джерел.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

Цей розділ присвячено аналізу предметної області проєкту, дослідженню ринку та існуючих рішень, а також формулюванню технічних та функціональних вимог до програмного продукту, що розробляється.

1.1 Історія розвитку та сучасний стан жанру 2D PvP платформерів

Жанр, до якого належить розроблювана гра «Нагодуй улюбленця!», є результатом поєднання двох фундаментальних напрямків ігрової індустрії: класичних платформерів та змагальних багатокористувацьких ігор (PvP).

Історія платформерів бере свій початок на початку 1980-х років. Перші представники жанру, такі як Space Panic (1980) та Donkey Kong (1981), заклали основи механіки: переміщення між платформами різної висоти та уникнення перешкод. Революційним кроком став вихід Super Mario Bros. (1985), який впровадив інерційну фізику руху, прокручування екрану (scrolling) та складний дизайн рівнів.

У 1990-х роках, з розвитком 16-бітних консолей, жанр досяг свого розквіту , еволюціонуючи від простих аркад до складних пригодницьких ігор з елементами дослідження (Metroidvania). Однак, домінування 3D-графіки наприкінці 90-х (поява PlayStation та N64) призвело до тимчасового спаду популярності класичних 2D-ігор.

Паралельно з розвитком платформерів еволюціонували файтинги (Street Fighter II, Mortal Kombat). Наприкінці 90-х ці два напрямки об'єдналися у піджанр, відомий як Platform Fighter або Arena Platformer.

Основоположником цього напрямку вважається серія Super Smash Bros. (1999), яка замінила класичну для файтингів шкалу здоров'я на відсотки

пошкодження та механіку вибивання суперника за межі арени. Це кардинально змінило фокус гри: від заучування складних комбінацій ударів до позиціонування, контролю простору та використання вертикальності рівня.

З середини 2000-х років, завдяки поширенню цифрової дистрибуції (Steam) та доступності ігрових рушіїв, почався так званий "інді-ренесанс". Незалежні розробники повернули популярність піксель-арту та 2D-механікам, переосмисливши їх для сучасної аудиторії.

Ключові ігри, що вплинули на формування сучасного вигляду жанру локальних PvP платформерів:

- TowerFall Ascension (2013): Продемонструвала, що проста механіка стрільби з лука в обмеженому 2D-просторі може забезпечити глибокий та динамічний змагальний процес. Гра зробила акцент на швидких раундах ("one-hit kill") та локальному мультиплеєрі.
- Duck Game (2014): Додала елементи хаосу, гумору та величезний арсенал зброї, закріпивши популярність "паті-геймів" (ігор для вечірок).
- Nidhogg (2014): Експериментувала з метою гри, замінивши просте знищення на необхідність дістатися до протилежного краю карти.
- Terraria (2011): Хоча це переважно PvE гра, вона популяризувала поєднання платформінгу з механіками будівництва та руйнування світу в реальному часі, що стало безпосереднім натхненням для механік проєкту «Нагодуй улюбленця!».

Сьогодні жанр 2D PvP платформерів залишається актуальним, особливо в ніші локальних розваг. Сучасні тенденції включають:

1. Асиметричний геймплей: Гравці можуть мати різні цілі або здібності (наприклад, один будує пастки, інший намагається вижити).
2. Процедурна генерація: Створення унікальних арен для кожного матчу для підвищення реіграбельності.
3. Гібридизація: Додавання елементів інших жанрів, таких як MOBA

(захоплення точок, класи персонажів) або "королівська битва" (звуження зони гри).

Розроблюваний проєкт «Нагодуй улюбленця!» слідує сучасним тенденціям, поєднуючи класичну механіку платформінгу з тактичними елементами будівництва та метою "King of the Hill" (захоплення точок), що виділяє його на фоні традиційних "deathmatch" арен.

1.2. Огляд жанру 2D PvP платформерів з елементами будівництва

Розроблювана гра «Нагодуй улюбленця!» належить до гібридного жанру, що поєднує елементи класичного 2D платформера, PvP (Player versus Player) арени та механік будівництва. Кожен з цих елементів має фундаментальний вплив на ігровий процес та архітектуру програмного забезпечення.

2D платформери є одним із найстаріших жанрів, що бере свій початок з аркадних автоматів. Його ключові риси – це навігація персонажа у двовимірному просторі (зазвичай по осях X та Y), взаємодія з платформами та реалізація фізичної моделі, що включає гравітацію, інерцію та обробку зіткнень. Успіх жанру, закладений такими іграми як Super Mario Bros., залежить від якості керування персонажем та дизайну рівнів [4]. У контексті даної роботи, ключовим завданням була точна реалізація фізики руху та надійної системи виявлення зіткнень з нуля, без використання сторонніх фізичних рушіїв.

PvP арена – це ігровий режим, де два або більше гравців змагаються на обмеженій локації. На відміну від класичних файтингів (Mortal Kombat, Street Fighter) або складних у реалізації MOBA[15] (Dota 2, League of Legends), для даного проєкту обрано модель змагання за очки шляхом захоплення контрольних точок. Цей підхід є менш агресивним і дозволяє сфокусуватися

на стратегії переміщення та контролю території.

Елементи будівництва – механіка, що дозволяє гравцям динамічно змінювати ігровий світ шляхом створення нових об'єктів. Ігри як Minecraft та Terraria продемонстрували гнучкість цієї механіки. У даному проєкті будівництво обмежене окремою тактичною фазою перед раундом, а можливість руйнування споруд додає необхідний баланс та контр-стратегії.

Гібридний жанр поєднує ці елементи, вимагаючи від гравців не лише навичок платформінгу, але й стратегічного мислення під час фази будівництва та активної боротьби за контрольні точки під час ігрової фази.

Для визначення позиціонування проєкту «Нагодуй улюбленця!» було проведено аналіз існуючих ігор, які використовують схожі механіки.

Таблиця 1.1

Порівняльний аналіз ігор-аналогів

Критерій	Super Smash Bros.	TowerFall Ascension	Terraria (PvP)	«Нагодуй улюбленця!»
Жанр	2D Арена-файтинг, Платформер	2D PvP Арена, Платформер-шутер	2D Пісочниця, Action-RPG, Платформер	2D PvP Арена, Платформер з елементами будівництва
Мета гри	Вибити суперників за межі арени	Залишитися останнім живим, вбиваючи суперників	Знищення суперника (гнучко)	Набрати більше очок за захоплення контрольних точок (мисок з їжею)
Ключові механіки	2D платформін	2D платформі	2D платформі	2D платформінг, стрільба

Критерій	Super Smash Bros.	TowerFall Ascension	Terraria (PvP)	«Нагодуй улюбленця!»
	г, унікальні атаки персонажів, предмети, щити, ухилення.	нг, швидкі рухи, стрільба з лука, підбирання стріл, ухилення.	нг, крафтинг, збір ресурсів, бої (зброя, магія), дослідження.	снарядами, захоплення точок.
Система будівництва	Відсутня (окрім окремого редактора рівнів)	Відсутня	Дуже глибока, основна механіка. Будівництво з блоків у реальному часі.	Обмежена тактичною фазою перед раундом, створення тимчасових платформ.
Руйнування	Руйнування тимчасових ящиків та деяких елементів арени.	Руйнування ящиків зі скарбами.	Повне руйнування ландшафту та будь-яких споруд.	Руйнування лише платформ, створених гравцями.

Аналіз показує, що прототип «Нагодуй улюбленця!» займає унікальну нішу, поєднуючи швидкий темп PvP арени з тактичною фазою будівництва та метою, орієнтованою на захоплення точок, що є рідкісною комбінацією в цьому жанрі.

1.3. Аналіз цільової аудиторії та вибір тематики проєкту

Вибір тематики є ключовим фактором, що визначає привабливість гри. Було обрано легку, гумористичну тематику протистояння домашніх тварин

(кота та собаки) за їжу.

Обґрунтування вибору тематики:

1. Широка впізнаваність: Тема домашніх тварин є універсальною та викликає позитивні емоції. Класичне протистояння "кіт проти собаки" є поширеним культурним тропом.

2. Ненасильницький конфлікт: Змагання за їжу (захоплення мисок) замість прямої агресії робить гру придатною для ширшої аудиторії, включаючи сім'ї та дітей. Стрільба адаптована під кидання іграшками.

3. Візуальний потенціал: Тема відкриває простір для створення милих персонажів. Обраний піксель-арт [14] стиль добре пасує до цієї тематики.

Цільова аудиторія: Враховуючи тематику та механіки, цільовою аудиторією гри є:

- Казуальні гравці: Цінують простий геймплей та приємну атмосферу.
- Сім'ї та діти: Приваблює ненасильницький характер гри.
- Шанувальники локального мультиплеєру.

Хоча обрана тематика може не зацікавити гравців, що шукають складних PvP змагань, для прототипу такий фокус є виправданим.

1.4. Постановка задачі та проєктування ігрового циклу

Задача полягає у створенні гри з чітким ігровим циклом (gameplay loop), керованим станами. Для керування станами гри була обрана архітектура машини скінченних станів (Finite State Machine) [13], яка є стандартним патерном в розробці ігор [6]. Цей підхід дозволяє уникнути надмірної складності умовних операторів та чітко розділити логіку для кожного стану гри.

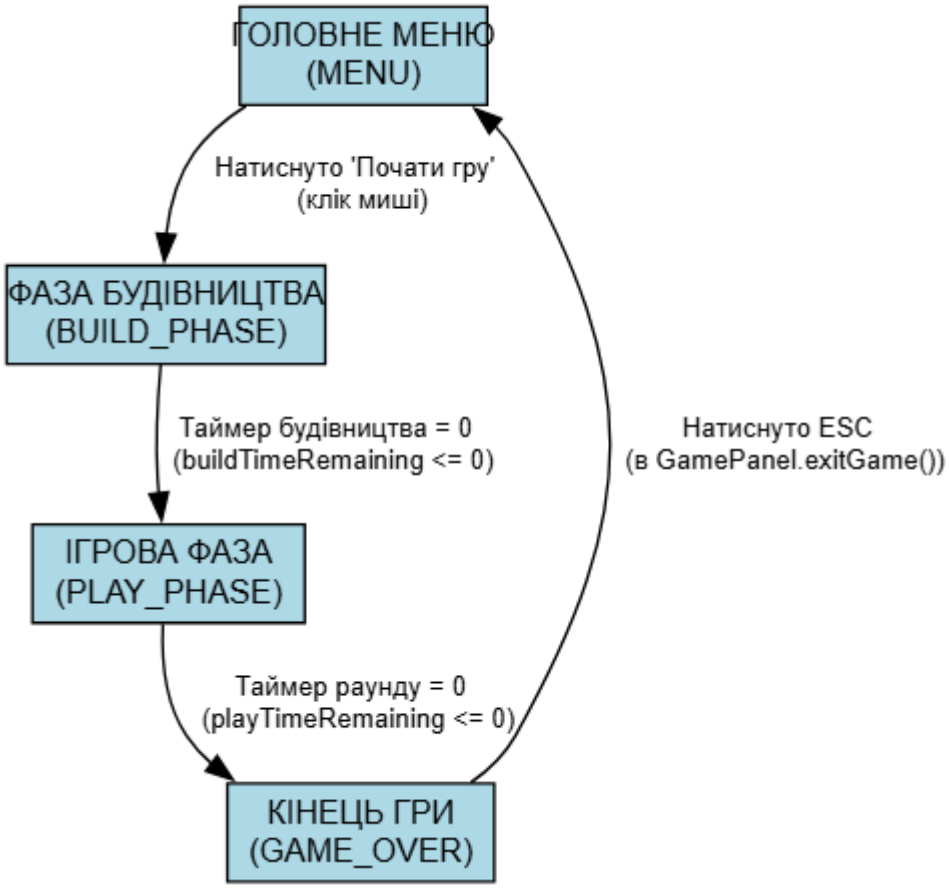

Рис. 1.1 — Діаграма станів ігрового циклу

Формалізація вимог до програмного продукту:

Функціональні вимоги (FR):

1. (FR 1) Система повинна відображати головне меню з кнопками "Почати гру" та "Вийти".
2. (FR 2) Гра повинна мати фазу будівництва, обмежену 15 секундами.
3. (FR 3) У фазі будівництва гравці повинні мати можливість створювати, переміщувати та повертати "примарну" платформу на 90 градусів.
4. (FR 4) Платформи, створені гравцями, повинні бути руйнівними.
5. (FR 5) Гра повинна мати ігрову фазу, обмежену 45 секундами.
6. (FR 6) Гравці повинні мати можливість рухатись (вліво, вправо) та стрибати.
7. (FR 7) Гравці повинні мати можливість стріляти снарядами.

8. (FR 8) Снаряди повинні руйнувати платформи гравців.
9. (FR 9) Гравці повинні мати можливість захоплювати контрольні точки при контакті.
10. (FR 10) Система повинна нараховувати очки за час утримання точок.
11. (FR 11) Гра повинна відображати HUD з рахунком та таймером.
12. (FR 12) Гра повинна переходити у стан GAME_OVER після завершення таймера ігрової фази.
13. (FR 13) Система повинна відображати екран переможця на основі очок.
14. (FR 14) Гравець повинен мати можливість вийти з гри через меню або натисканням ESC.

Нефункціональні вимоги (NFR):

1. (NFR 1) Гра повинна підтримувати двох гравців одночасно за однією клавіатурою.
2. (NFR 2) Гра повинна працювати з частотою оновлення не менше 60 кадрів на секунду (впроваджено 120 FPS).
3. (NFR 3) Управління персонажем повинно бути "відзивчивим" (вирішено через "Coyote Time", "Jump Buffering").
4. (NFR 4) Графічні ресурси мають завантажуватися ефективно, без повторного читання з диска під час гри.
5. (NFR 5) Код має бути написаний мовою Java 21 та використовувати виключно бібліотеки Swing/AWT.
6. (NFR 6) Код має бути структурований відповідно до принципів ООП, DRY[8] та KISS[9].

РОЗДІЛ 2
ТЕХНОЛОГІЇ ТА АРХІТЕКТУРА ПРОЄКТУ

2.1. Обґрунтування вибору технологічного стеку

Для реалізації проєкту «Нагодуй улюбленця!» було обрано наступний стек технологій: Java 21 та Java Swing.

Мова програмування: Java (версія 21) [1] Вибір Java як основної мови був зумовлений її ключовими перевагами:

1. Об'єктно-орієнтований підхід: ООП [3] є фундаментом мови, що дозволило спроєктувати чисту архітектуру гри, де кожен елемент (Гравець, Платформа, Снаряд) є окремим класом зі своїми даними та поведінкою (інкапсуляція).
2. Платформонезалежність: Завдяки віртуальній машині Java (JVM)[16], скомпільований код може працювати на будь-якій ОС (Windows, macOS, Linux) без модифікацій.
3. Багата стандартна бібліотека (JDK): Надає потужні вбудовані інструменти для роботи з колекціями (java.util), введенням/виведенням (java.io) та графікою (java.awt, javax.swing).
4. Сучасні можливості (Java 21): Використання актуальної LTS (Long-Term Support) [17] версії Java надає доступ до сучасних оптимізацій JVM та синтаксичних покращень. Це також закладає фундамент для майбутніх розширень, наприклад, використання віртуальних потоків Project Loom [5] для реалізації мережевого мультиплеєру.

Бібліотека Swing [9], обрана для даного проєкту, базується на архітектурі "Model-View-Controller" (MVC), але зі спрощеною реалізацією (Model-Delegate). Ключовою особливістю Swing є те, що всі її компоненти є "легковагими" (lightweight). Це означає, що вони малюються засобами Java на

спільному полотні (вікні), не використовуючи нативних вікон операційної системи для кожного елемента. Рендеринг у Swing відбувається переважно за допомогою CPU (Central Processing Unit) через API Java 2D. Хоча сучасні реалізації JVM намагаються використовувати апаратне прискорення (Direct3D на Windows або OpenGL на Linux/macOS) для деяких операцій (наприклад, копіювання буферів зображень), основна логіка відмальовки складних сцен залишається програмною. Це створює певні обмеження для високопродуктивних ігор з великою кількістю частинок або складною геометрією, проте для тайлового 2D-платформера продуктивність є цілком достатньою. Критичним аспектом архітектури Swing є однопоточність. Всі операції з малювання та обробки подій відбуваються в єдиному потоці — Event Dispatch Thread (EDT). Це вимагає від розробника суворої дисципліни: будь-які "важкі" обчислення мають бути винесені в окремі потоки (Worker Threads), інакше інтерфейс гри "зависне". У розробленому проєкті цю проблему вирішено шляхом оптимізації ігрового циклу, де оновлення логіки та рендеринг не перевищують часовий слот у 8 мс. Вибір Swing для візуалізації був прагматичним рішенням для прототипування :

1. Вбудована у JDK: Swing є частиною стандартної бібліотеки, що виключає необхідність налаштування сторонніх залежностей.

2. Достатній функціонал для 2D: Надає весь необхідний інструментарій: JFrame (вікно), JPanel (полотно), Timer (ігровий цикл), Graphics2D (малювання), KeyListener (введення) та JOptionPane (діалоги).

3. Повний контроль: На відміну від ігрових рушіїв, Swing дозволяє повністю контролювати цикл малювання (paintComponent) та оновлення (actionPerformed), що було однією з цілей практики.

Порівняльний аналіз альтернативних технологій Вибір Swing був зроблений після аналізу альтернатив.

Таблиця 2.1

Порівняльний аналіз технологій

Критерій	Java Swing	JavaFX	LibGDX
Продуктивність (Апаратне прискорення)	Низька. Графіка малюється переважно силами CPU. Апаратне прискорення відсутнє або дуже обмежене, що може призводити до падіння FPS при великій кількості об'єктів.	Середня / Висока. Сучасна бібліотека, що використовує апаратне прискорення (DirectX, OpenGL). Добре підходить для 2D-графіки та UI.	Висока / Дуже висока. Повноцінний ігровий рушій, що працює безпосередньо з [OpenGL] [14] / WebGL. Створений спеціально для ігор, забезпечує максимальну продуктивність3.
Поріг входження	Низький. Проста для розуміння, якщо ви знаєте Java. Ігровий цикл (Timer) та малювання (paintComponent) реалізуються вручну, що дає повний контроль.	Середній. Вимагає вивчення нової архітектури (Scene Graph) та підходів до малювання. Ігровий цикл менш очевидний (використовуєтьс я AnimationTimer).	Високий. Це повноцінний фреймворк зі своїм API, системою завантаження асетів, керуванням станами та життєвим циклом. Вимагає значного часу на вивчення.
Наявність ігрового API	Відсутній. Усі механіки (фізика, анімація, зіткнення, звук) потрібно реалізовувати з	Базовий. Містить класи для простих анімацій (Transition), медіа (AudioClip), але не має вбудованої фізики чи	Повноцінний. Має все необхідне для розробки ігор: 2D/3D рендеринг, API для спрайтів, аудіосистему,

Критерій	Java Swing	JavaFX	LibGDX
	нуля, використовуючи базові інструменти AWT/Swing.	системи керування спрайтами.	обробку вводу, інтеграцію з фізичним рушієм (Box2D).
Залежності	Жодних. Вбудована у стандартний JDK, що забезпечує максимальну простоту розгортання.	Так. З часів Java 11, JavaFX не входить до складу JDK і має підключатися як окрема стороння бібліотека (наприклад, через Maven/Gradle).	Так. Є великою сторонньою залежністю. Вимагає налаштування проєкту через власний інструмент gdx-setup.jar для керування бібліотеками під різні платформи (Desktop, Android).

- Ігрові рушії (Unity, Godot): Надають готові фізичні рушії та редактори, але вимагають вивчення нових мов (C#, GDScript) і приховують реалізацію базових алгоритмів.
- JavaFX: Сучасніша бібліотека з апаратним прискоренням [11]. На відміну від Swing, JavaFX використовує граф сцени (Scene Graph) — ієрархічну структуру об'єктів, яка повністю відокремлена від механізму рендерингу. Рендеринг у JavaFX виконується окремим потоком (Prism Renderer), який активно використовує апаратне прискорення графічного процесора (GPU). Це забезпечує значно вищу продуктивність для анімацій та ефектів прозорості. Проте, JavaFX має вищий поріг входження та вимагає підключення зовнішніх модулів (починаючи з JDK 11), що ускладнює налаштування середовища для початкового прототипування порівняно з

Swing.

- LibGDX/LWJGL: Високопродуктивні ігрові бібліотеки, що надають прямий доступ до OpenGL [10]. Були відкинуті через значно вищий поріг входження та складність налаштування, що є надлишковим для поточного проєкту. LibGDX — це не просто графічна бібліотека, а повноцінний фреймворк, що надає низькорівневий доступ до OpenGL (через легку обгортку LWJGL). Це дозволяє досягти максимальної продуктивності, оскільки розробник може керувати геометрією, шейдерами та текстурами безпосередньо в пам'яті відеокарти. Однак архітектура LibGDX вимагає керування життєвим циклом додатку (create, render, dispose), ручного управління пам'яттю для нативних ресурсів та розуміння принципів роботи графічного конвеєра. Для задачі створення з фокусом на ігрові механіки, а не на графічні технології, такий рівень складності було визнано надлишковим.

Таким чином, стек Java + Swing є оптимальним балансом між простотою реалізації та можливістю глибокого контролю над ігровою логікою.

2.2. Налаштування середовища та використані API

Розробка велася у IntelliJ IDEA Community Edition [18], потужному безкоштовному IDE. Графічні ресурси завантажувалися з порталу itch.io [19].

Проєкт використовує виключно стандартні API JDK:

- javax.swing (GUI): JFrame, JPanel, Timer, ImageIcon, JOptionPane.
- java.awt (Графіка): Graphics2D (для малювання з прозорістю), Color, Font, BufferedImage (для анімації), Rectangle (для зіткнень).
- java.awt.event (Введення): KeyListener, MouseListener, KeyEvent (для getKeyCode та getKeyLocation).
- javax.imageio.ImageIO: ImageIO.read() для завантаження спрайт-смуг.

- java.util (Колекції): ArrayList, HashSet (для "Stateful Input"), Iterator (для безпечного видалення об'єктів з циклу).

2.3. Архітектура програмного проєкту та структура класів

Проєкт побудований за принципами ООП [3] з чітким розподілом відповідальності (Single Responsibility Principle).

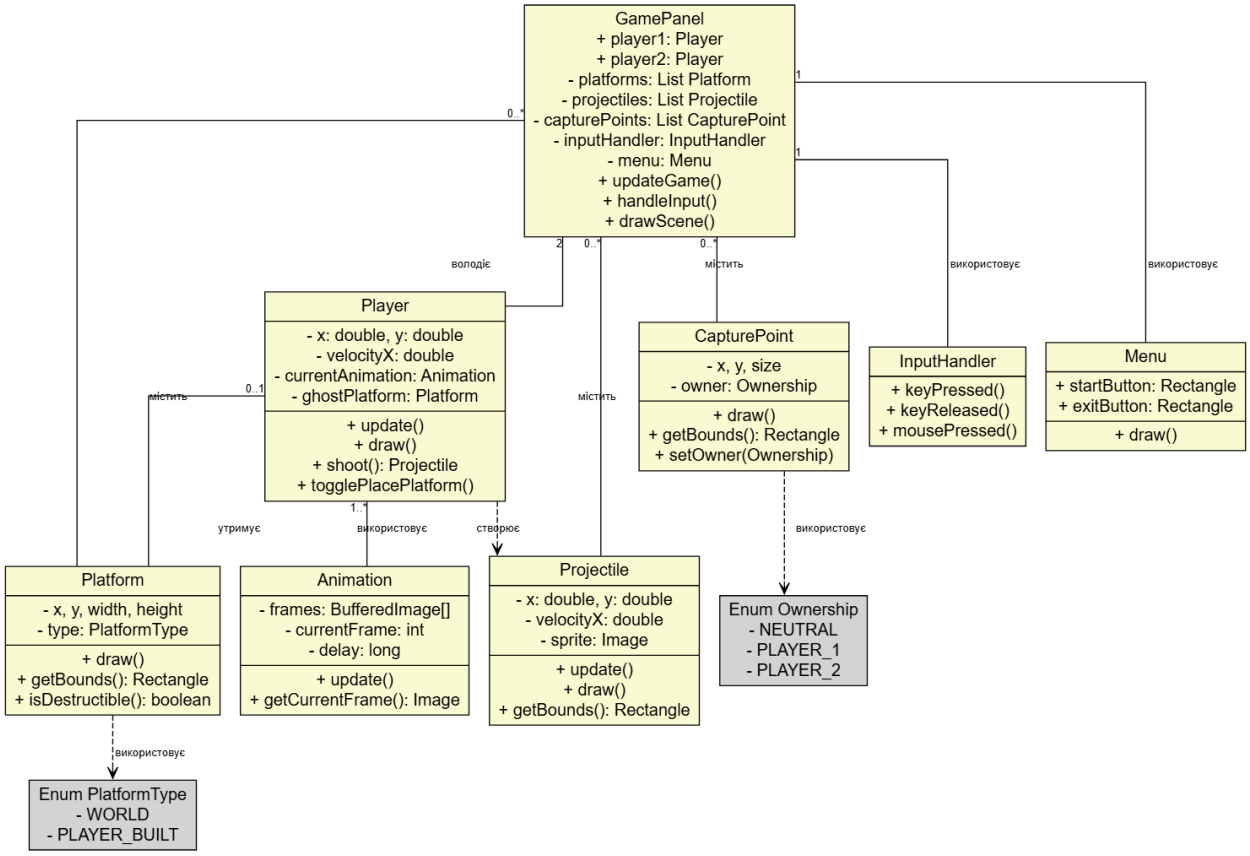

Рис. 2.1 — UML-діаграма основних класів проєкту

Основні класи та їхня відповідальність:

- Main: Точка входу, створює JFrame та GamePanel.
- GamePanel: "Диригент" гри. Успадковує JPanel. Містить ігровий цикл (Timer) та головний switch по GameState. Керує списками об'єктів. Делегує обробку вводу InputHandler, але сам містить логіку реакції на ввід.
- Player: Інкапсулює всю логіку персонажа. Відповідає за власну

фізику (update), анімацію (loadAnimations), малювання (draw), та дії (shoot, togglePlacePlatform).

- Platform, Projectile, CapturePoint: Класи-моделі. Зберігають свій стан та реалізують власну логіку малювання (draw).
- Animation: Допоміжний клас, що керує логікою зміни кадрів анімації.
- InputHandler: Єдиний клас для обробки вводу (миша + клавіатура).
- LevelLoader: Утилітарний клас для завантаження рівня з .txt файлу.
- Menu: Відповідає виключно за малювання головного меню.

Застосовані патерни проєктування: Архітектура проєкту спирається на кілька фундаментальних патернів проєктування [12]:

- State (Стан) [13]: Ключовий патерн для керування грою. Замість великих if-else конструкцій, логіка гри (в updateGame та drawScene) делегується залежно від поточного стану (currentState типу GameState). Це робить код чистим, легко керованим та розширюваним (напр., для додавання стану PAUSED). У контексті гри цей патерн реалізовано для керування глобальними фазами ігрового процесу. Клас GamePanel діє як "Контекст", а enum GameState визначає можливі стани (MENU, BUILD_PHASE, PLAY_PHASE, GAME_OVER). Це дозволило уникнути використання громіздких конструкцій if-else та численних булевих прапорців (наприклад, isGameRunning, isMenuOpen). Замість цього, метод updateGame() делегує виконання конкретної логіки залежно від поточного значення змінної currentState. Такий підхід дозволяє легко додавати нові режими (наприклад, "Пауза" або "Магазин"), просто додавши новий елемент в enum та відповідну гілку в switch-вираз.
- Factory Method (Фабричний метод) [12]: У спрощеному вигляді застосований у класі Player. Метод shoot() є "фабрикою" для об'єктів Projectile. Він інкапсулює логіку створення снаряда, визначаючи його стартову позицію,

швидкість та спрайт залежно від типу гравця. Це відокремлює клас Player від конкретної реалізації Projectile. Цей патерн застосовано для інкапсуляції логіки створення ігрових об'єктів, зокрема снарядів. Замість того, щоб GamePanel самостійно створював об'єкт Projectile, визначаючи його координати та тип, він запитує це у гравця. Клас Player самостійно розраховує початкову позицію снаряда (відносно свого центру та напрямку погляду), обирає відповідний спрайт та повертає готовий об'єкт. Це зменшує зв'язність коду (coupling): GamePanel не потребує знати деталі реалізації пострілу, він лише оперує абстрактним поняттям Projectile.

- Observer (Спостерігач) [21]: Бібліотека Swing інтенсивно використовує цей патерн для обробки подій, і він був інтегрований в архітектуру гри. Клас InputHandler виступає як "Спостерігач" (Observer) на події клавіатури та миші, які генерує вікно гри (Subject). Однак, стандартна реалізація Observer у Swing працює за принципом "push" (подія надсилається в момент виникнення). Для потреб гри (де важливо знати, чи *утримується* клавіша натиснутою) цей патерн було модифіковано: InputHandler накопичує стан клавіш у колекції Set, а ігровий цикл опитує цей стан (polling) на кожному кадрі. Це гібридний підхід, що поєднує подієву модель Swing з потребами ігрового циклу реального часу.

2.4. Система завантаження рівнів на основі файлів

Одним з ключових архітектурних рішень, прийнятих на ранньому етапі проєктування, було відокремлення даних рівня (дизайну ігрового світу) від ігрової логіки (рушія гри). Було вирішено відмовитися від "жорсткого" кодування (hard-coding) координат платформ безпосередньо у коді GamePanel на користь динамічного завантаження рівня з зовнішнього текстового файлу.

Такий підхід має низку фундаментальних переваг, що відповідають

принципам гнучкої розробки:

1. Гнучкість та швидке прототипування: Дизайнер рівнів (або сам розробник) може створювати, тестувати та модифікувати десятки варіантів ігрових арен, просто редагуючи текстовий файл, без необхідності перекомпіляції всього .java проєкту.

2. Поділ відповідальності (Separation of Concerns): Цей принцип [8] реалізовано шляхом винесення даних (рівня) в окремий файл, а логіки (рушій) — в класи Java. GamePanel не знає, який рівень він завантажує, він лише знає, *як* малювати об'єкти зі списків.

3. Масштабованість: Система дозволяє легко додавати нові типи об'єктів (наприклад, шипи, батути), просто додавши новий символ у switch-оператор завантажувача.

4. Простота: Використання текстового файлу (.txt) є реалізацією принципу KISS (Keep It Simple, Stupid) [9], оскільки такий рівень можна переглядати та редагувати у будь-якому, навіть найпростішому, текстовому редакторі.

Реалізація модуля LevelLoader

Для реалізації цього функціоналу було створено спеціалізований утилітарний клас LevelLoader. Його єдиним завданням є читання файлу рівня та заповнення списків ігрових об'єктів, які йому передає GamePanel.

Рівень зберігається у текстовому файлі (напр., level1.txt), де кожен символ представляє одну ігрову “плитку”. Розмір цієї плитки (TILE_SIZE) визначено у LevelLoader як константа (24x24 пікселі).

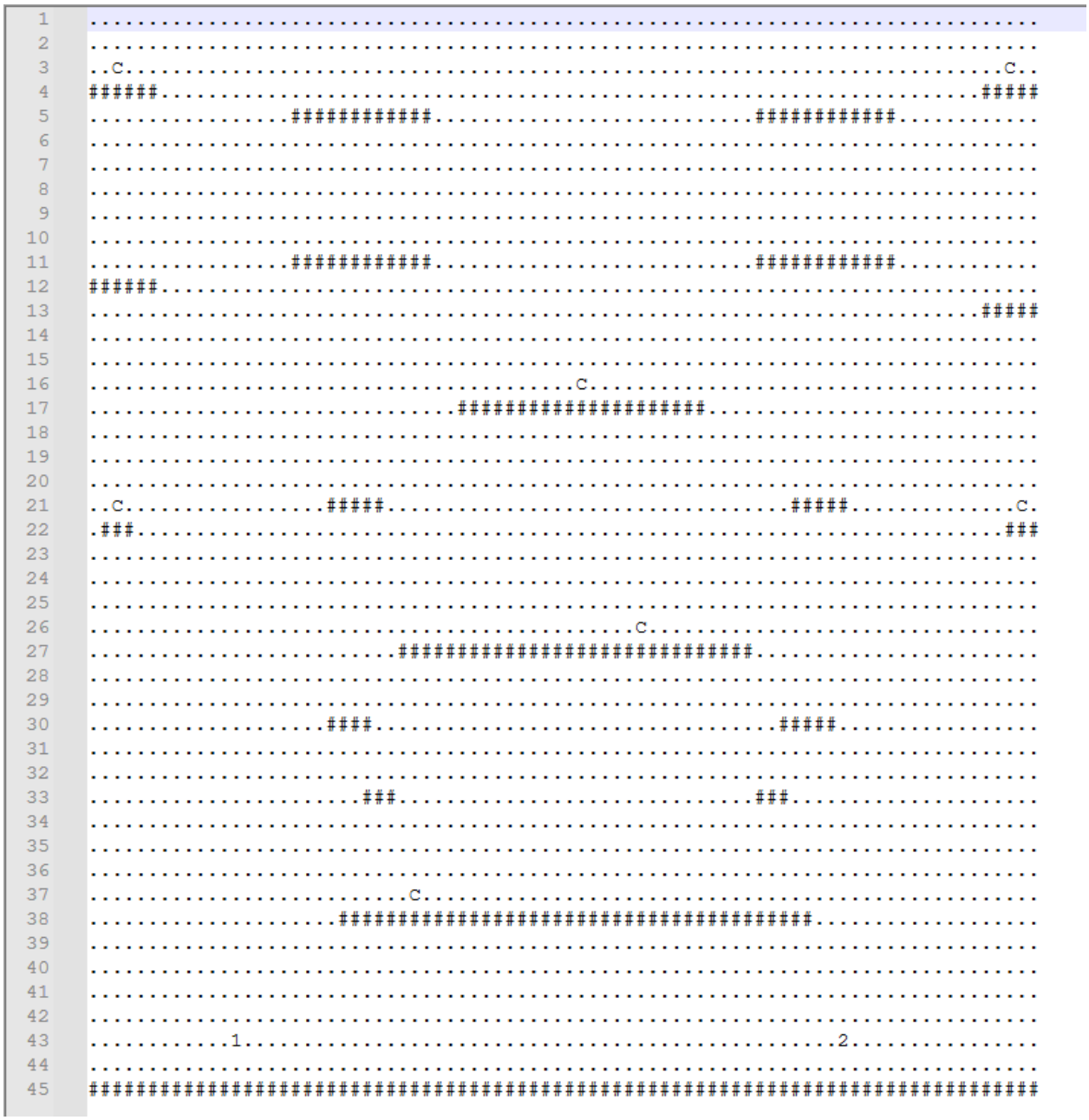

Рис. 2.2 – Приклад файлу конфігурації рівня (level1.txt)

На Рис. 2.2 продемонстровано текстовий файл, що використовується для завантаження ігрового рівня. Цей підхід, реалізований у класі LevelLoader, дозволяє гнучко та швидко створювати нові арени без необхідності перекомпіляції коду.

Під час завантаження, метод LevelLoader.load() читає файл рядок за рядком та, на основі кожного символу (char), створює відповідний ігровий

об'єкт у розрахованих координатах (worldX = x * TILE_SIZE):

- . (крапка): Порожній простір, ігнорується.
- #: Створює нову Platform з типом PlatformType.WORLD.
- C: Створює новий CapturePoint (миску [26]).
- 1: Встановлює стартову позицію для player1.
- 2: Встановлює стартову позицію для player2.

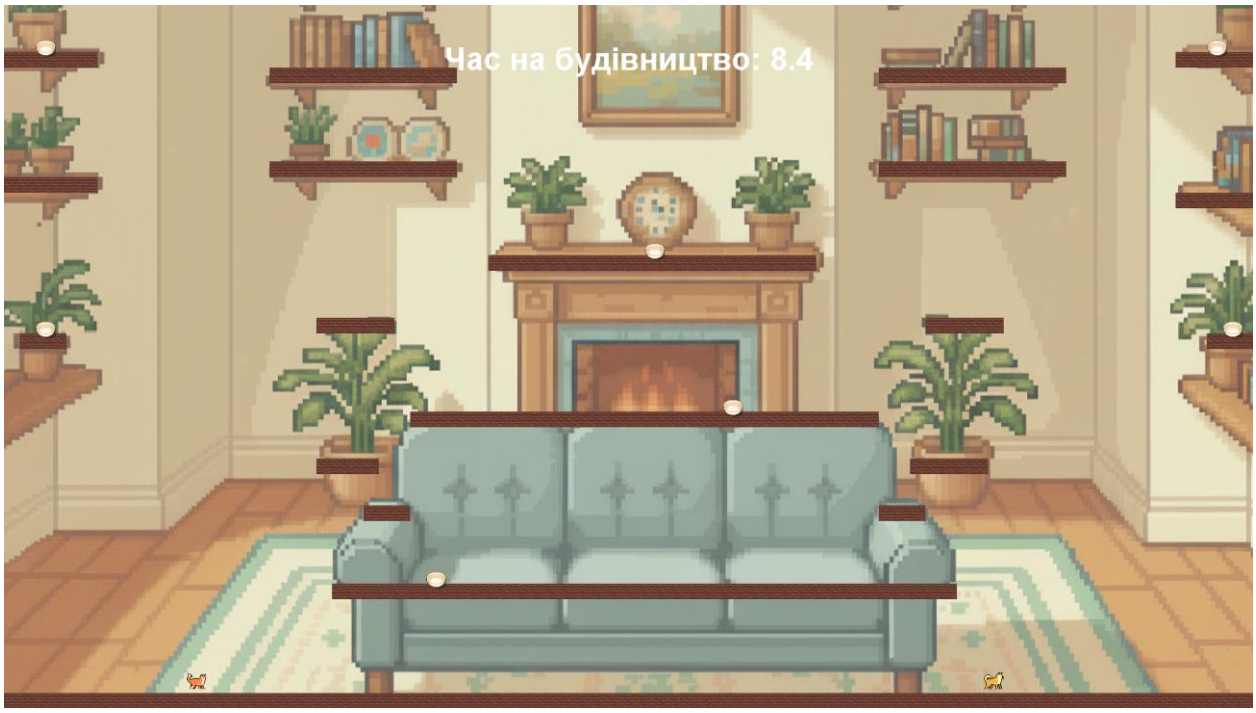

Рис. 2.3 – Візуальна реалізація рівня в грі (скріншот ігрового процесу)

На Рис. 2.3 показано результат роботи LevelLoader під час активної ігрової фази. Можна побачити, як символи з файлу конфігурації (показаного на Рис. 2.2) були трансформовані у візуальні об'єкти:

- Символи # перетворилися на статичні платформи зі спрайтом дерева (platform_wood.png) [25].
- Символи C стали інтерактивними точками захоплення.
- Символи 1 та 2 визначили стартові позиції гравців (кота та собаки).

Такий підхід повністю відокремлює дизайн рівнів від ігрової логіки, що відповідає принципам гнучкої розробки.

Процес завантаження інкапсульовано у методі load().

```
// (З файлу LevelLoader.java)
public class LevelLoader {
    private final String filePath;
    private final int TILE_SIZE = 24; // Розмір одного "символу" рівня в пікселях

    public LevelLoader(String filePath) {
        this.filePath = filePath;
    }

    public void load(List<Platform> platforms, List<CapturePoint> capturePoints, Player p1, Player p2) {
        platforms.clear();
        capturePoints.clear();

        try {
            // 1. Отримання ресурсу як потоку
            InputStream is = getClass().getResourceAsStream(filePath);
            if (is == null) {
                System.err.println("Не вдалося знайти файл рівня: " + filePath);
                return;
            }
            // 2. Створення буферизованого читача
            BufferedReader reader = new BufferedReader(new InputStreamReader(is));
            String line;
            int y = 0; // Поточна координата Y (номер рядка)

            // 3. Читання файлу рядок за рядком
            while ((line = reader.readLine()) != null) {
                // 4. Ітерація по кожному символу в рядку
                for (int x = 0; x < line.length(); x++) {
                    char symbol = line.charAt(x);

                    // 5. Перетворення координат сітки на світові координати
                    int worldX = x * TILE_SIZE;
                    int worldY = y * TILE_SIZE;

                    // 6. Створення об'єктів на основі символу
                    switch (symbol) {
                        case '#':
                            platforms.add(new Platform(worldX,
```

```
worldY, TILE_SIZE, TILE_SIZE, PlatformType.WORLD));
                            break;
                        case 'C':
                            capturePoints.add(new
CapturePoint(worldX, worldY));
                            break;
                        case '1':
                            p1.x = worldX;
                            p1.y = worldY;
                            break;
                        case '2':
                            p2.x = worldX;
                            p2.y = worldY;
                            break;
                    }
                }
                y++;
            }
            reader.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Лістинг 2.1 – Фрагмент коду LevelLoader.java (Метод завантаження)

Аналіз лістингу 2.1:

1. Використання getClass().getResourceAsStream(filePath) є найнадійнішим способом завантаження ресурсів у Java. Це дозволяє знаходити файл level1.txt незалежно від того, чи запускається гра з IDE, чи як готовий .jar файл, за умови, що папка res коректно налаштована як "Resources Root".
2. Використання BufferedReader є стандартним та ефективним способом читання текстових файлів рядок за рядком.
3. Вкладені цикли while та for реалізують ітерацію по двовимірній сітці (Y та X).
4. Розрахунок worldX = x * TILE_SIZE та worldY = y * TILE_SIZE є ключовим моментом, що трансформує індекс символу в текстовому файлі (наприклад, x = 5) у реальну піксельну координату на екрані

(напр., 5 * 24 = 120).

5. Оператор switch(symbol) діє як проста "фабрика" об'єктів. Він читає символ і вирішує, який саме об'єкт (Platform, CapturePoint) потрібно створити в цих координатах.

Інтеграція в GamePanel: Завантажувач викликається один раз у методі GamePanel.resetGame(). Це гарантує, що при кожному поверненні в меню та початку нової гри рівень буде "перезавантажено" до свого початкового стану (всі зруйновані платформи відновляться).

```
// (З файлу GamePanel.java)
private void resetGame() {
    //... (скидання стану гри, таймерів, очок)...

    player1 = new Player(0, 0, Color.CYAN);
    player2 = new Player(0, 0, Color.MAGENTA);

    // Створюємо завантажувач та передаємо йому посилання на списки об'єктів
    LevelLoader loader = new LevelLoader("/level1.txt"); // Шлях до файлу в папці res
    loader.load(platforms, capturePoints, player1, player2);

    if (projectiles != null) {
        projectiles.clear();
    }
}
```

Лістинг 2.2 – Фрагмент коду GamePanel.java (Виклик завантажувача)

Як видно з Лістингу 2.2, GamePanel передає свої списки platforms та capturePoints, а також об'єкти player1 та player2 безпосередньо в метод load(). LevelLoader наповнює ці списки та змінює координати гравців, що є ефективним способом ініціалізації ігрового світу.

2.5. Робота з графічними ресурсами

Усі графічні ресурси проєкту реалізовано у форматі .png із підтримкою прозорості. Їхні розміри адаптовано під базовий розмір ігрової клітинки (TILE_SIZE), що становить 24x24 пікселі, а також під загальну роздільну здатність екрану 1920x1080. Важливим архітектурним рішенням стало розмежування візуальної та фізичної складових для персонажів: при розмірі спрайтів 48x48 пікселів, їхні фізичні межі (hitbox) зменшено до 26x42 пікселів. Це дозволило покращити візуальне сприйняття та забезпечити більш точну обробку зіткнень під час гри.

Програмна реалізація завантаження ресурсів здійснюється через стандартний механізм getClass().getResource(). Для підвищення продуктивності застосовано оптимізацію за допомогою статичних блоків ініціалізації, що дозволяє завантажувати ресурси для масових об'єктів (таких як снаряди або точки захоплення) лише один раз, уникаючи зайвих операцій з пам'яттю.

Система анімації базується на роботі класу Animation, який керує перемиканням кадрів, представлених масивом BufferedImage[]. Формування кадрів відбувається шляхом програмного розрізання спрайт-смуг за допомогою методу BufferedImage.getSubimage(). Окрім того, для реалізації руху вліво (facingDirection = -1) використано технічний прийом із від'ємною шириною у методі Graphics.drawImage(), що ефективно віддзеркалює спрайт по горизонталі без необхідності створення та зберігання додаткових графічних файлів.

РОЗДІЛ 3
КІНЦЕВА РЕАЛІЗАЦІЯ ІГРОВИХ МОДУЛІВ

У цьому розділі детально описано процес програмної реалізації ключових функціональних модулів гри «Нагодуй улюбленця!». Розглядаються використані алгоритми, структури даних та конкретні фрагменти коду, що відповідають за втілення спроєктованих ігрових механік.

3.1. Реалізація ігрового циклу та управління станами

Ядром будь-якої інтерактивної програми, особливо гри, є ігровий цикл (Game Loop). Він відповідає за послідовне оновлення логіки гри та перемальовку екрану [4]. У середовищі Java Swing, яке є однопоточним та керується подіями (Event-Driven), реалізація ігрового циклу через while(true) у головному потоці є неможливою, оскільки це заблокує Потік обробки подій (Event Dispatch Thread - EDT) [10].

Тому, для коректної інтеграції зі Swing, було обрано клас javax.swing.Timer. Він дозволяє виконувати код періодично, гарантуючи, що всі оновлення логіки (updateGame) та виклики перемальовки (repaint) відбуваються безпечно в EDT.

```
// (З файлу GamePanel.java)
public class GamePanel extends JPanel implements ActionListener {
    private Timer timer;

    public GamePanel() {
        //... (інші налаштування)
        setFocusable(true); // Критично важливо для отримання подій
клавіатури
    }

    public void startGame() {
        // Налаштування на ~120 кадрів на секунду (1000 / 120 ≈
8.33 мс)
```

```
        timer = new Timer(1000 / 120, this);
        timer.start();
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        updateGame(); // Оновлення логіки
        repaint();    // Запит на перемальовку
    }

//...
}
```

Лістинг 3.1 – Фрагмент коду GamePanel.java (Ініціалізація та запуск таймера)

Як показано в Лістингу 3.1, клас GamePanel реалізує інтерфейс ActionListener. Його метод actionPerformed стає серцем ігрового циклу, який спрацьовує кожні ~8.33 мс (120 разів на секунду). Це забезпечує високу плавність гри та коректну роботу в екосистемі Swing.

Керування логікою гри реалізовано через патерн State (Стан) [13]. Цей патерн дозволяє об'єкту (в нашому випадку GamePanel) змінювати свою поведінку залежно від внутрішнього стану [6]. Було створено enum GameState (MENU, BUILD_PHASE, PLAY_PHASE, GAME_OVER), який інкапсулює всі можливі стани гри.

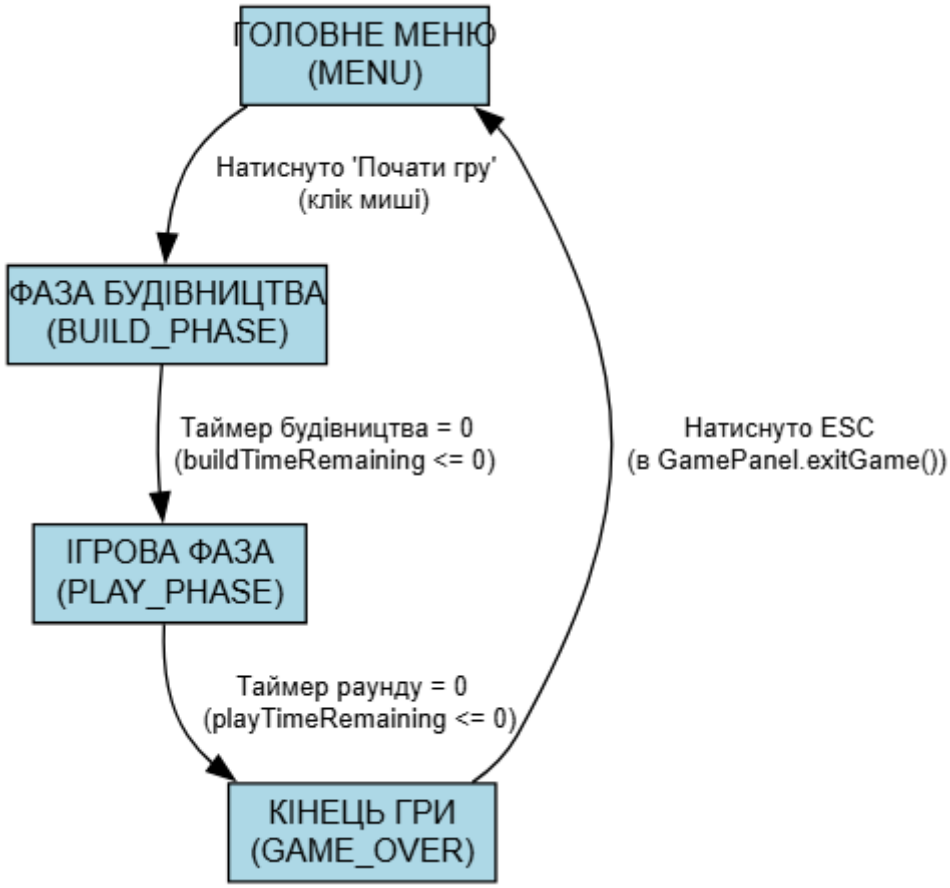

Рис. 3.1 — Діаграма станів гри

Основні методи updateGame() та drawScene() містять switch (currentState), що делегує виконання коду лише тій логіці, яка є релевантною для поточного стану.

```
// (З файлу GamePanel.java)
private void updateGame() {
    handleInput(); // Обробка вводу
    final double DELTA_TIME = 1.0 / 120.0;

    switch (currentState) {
        case MENU:
            // Логіка не оновлюється
            break;
        case BUILD_PHASE:
            buildTimeRemaining -= DELTA_TIME;
            if (buildTimeRemaining <= 0) {
                currentState = GameState.PLAY_PHASE; // Перехід
стану
```

```
                //... (очищення "примарних" платформ)
            }
            break;
        case PLAY_PHASE:
            player1.update(platforms);
            player2.update(platforms);
            updateProjectiles();
            updateCapturePoints();
            updateScores(DELTA_TIME);
            playTimeRemaining -= DELTA_TIME;
            if (playTimeRemaining <= 0) {
                currentState = GameState.GAME_OVER; // Перехід
стану
            }
            break;
        case GAME_OVER:
            // Логіка гри зупинена
            break;
    }
}
```

Лістинг 3.2 – Фрагмент коду GamePanel.java (Керування станами)

Аналіз Лістингу 3.2 показує, що такий підхід (використання Timer + switch за станом) є оптимальним для Swing. Він забезпечує чіткий, керований та безпечний ігровий цикл, де логіка кожного стану ізольована (наприклад, у MENU та GAME_OVER оновлення ігрових об'єктів не відбувається, що економить ресурси).

3.2. Розробка модуля фізики персонажа та зіткнень

Реалізація переконливої фізики є ядром будь-якого платформера. У даній роботі фізична модель була реалізована вручну у класі Player.

Обробка зіткнень (Collision Detection): Для виявлення зіткнень було обрано найпоширеніший та найшвидший метод для 2D-ігор – AABB (Axis-Aligned Bounding Box), або осьовирівняні обмежувальні прямокутники[7]. Кожен ігровий об'єкт (Player, Platform) має метод getBounds(), що повертає java.awt.Rectangle, який описує його фізичні межі (hitbox). Зіткнення

реєструється методом Rectangle.intersects().

Для забезпечення стабільної роботи фізичного рушія було відмовлено від підходу перевірки зіткнень коли нова позиція об'єкта перевіряється відразу по обох координатах (X та Y). Такий підхід часто призводить до помилок "застрягання" у кутах блоків або некоректного визначення сторони зіткнення (наприклад, гра вважає, що персонаж вдарився об стіну, хоча він приземлився на край платформи).

Алгоритм працює у два проходи на кожному кадрі:

Прохід по осі X (Горизонталь):

- До поточної координати X додається вектор швидкості velocityX.
- Одразу виконується перевірка на перетин (intersects) з усіма платформами на рівні.
- У разі виявлення колізії, система аналізує напрямок руху. Якщо velocityX > 0 (рух вправо), гравець "виштовхується" вліво до лівої межі перешкоди (rect.x - player.width). Якщо velocityX < 0 (рух вліво), гравець зміщується до правої межі перешкоди.
- Швидкість velocityX може бути обнулена (опціонально), або збережена для ефекту ковзання.

Прохід по осі Y (Вертикаль):

- Після вирішення всіх горизонтальних конфліктів, до координати Y додається вектор швидкості velocityY.
- Знову виконується перевірка колізій.
- Якщо виявлено перетин при velocityY > 0 (падіння вниз), це трактується як приземлення. Гравець ставиться на верхню грань платформи, вертикальна швидкість обнуляється, а прапорець onGround встановлюється в true. Це критичний момент для роботи механіки стрибка.
- Якщо перетин при velocityY < 0 (рух вгору), це стрибок. Гравець

підіймається по осі Y до моменту зіткнення з платформою, після чого швидкість стає нульовою, імітуючи удар і під дією “гравітації” гравець спускається на платформу що знаходиться нижче.

Така послідовність гарантує, що гра завжди може однозначно визначити, з якого боку відбулося зіткнення, навіть якщо гравець рухається по діагоналі з великою швидкістю.

Для коректної обробки реакції на зіткнення була застосована техніка розділення осей (axis separation).

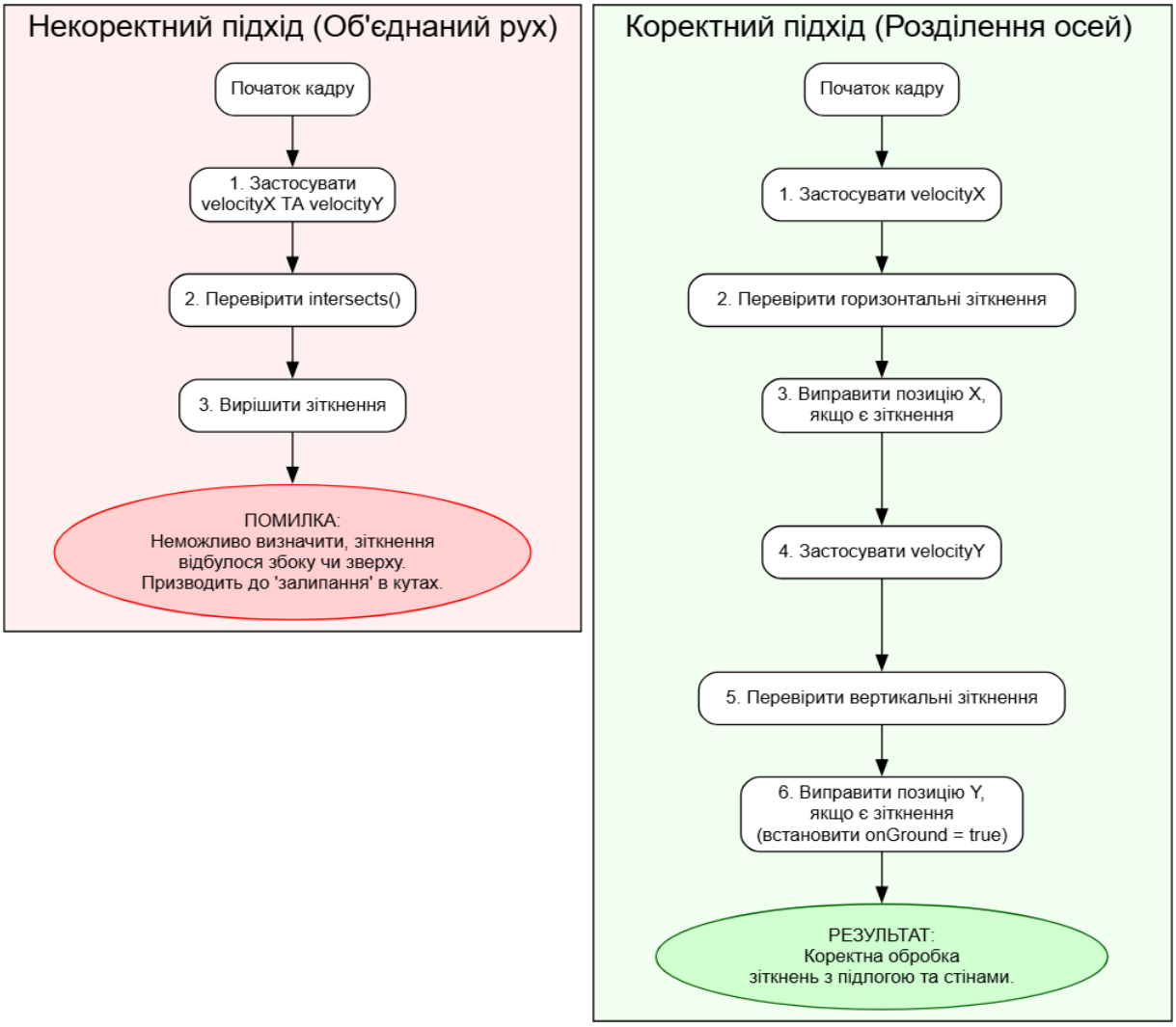

Рис. 3.2 — Схематичне зображення алгоритму розділення осей (AABB Separation)

```
// (З файлу Player.java)
public void update(List<Platform> platforms) {
    // --- 1. Горизонтальний рух ---
    x += velocityX;
    handleHorizontalCollisions(platforms);

    // --- 2. Вертикальний рух (зіткнення) ---
    y += velocityY;
    handleVerticalCollisions(platforms);

    // --- 3. Гравітація та Стрибок (після визначення опори) ---
    applyGravityAndJumpLogic();
    //...
}

private void handleHorizontalCollisions(List<Platform> platforms)
{
    Rectangle playerBounds = getBounds();
    for (Platform platform : platforms) {
        if (playerBounds.intersects(platform.getBounds())) {
            // "Виштовхування" гравця з платформи по осі X
            if (velocityX > 0) x = platform.x - hitboxWidth;
            else if (velocityX < 0) x = platform.x + platform.width;
        }
    }
}

private void handleVerticalCollisions(List<Platform> platforms) {
    Rectangle playerBounds = getBounds();
    boolean currentlyOnGround = false;
    for (Platform platform : platforms) {
        if (playerBounds.intersects(platform.getBounds())) {
            if (velocityY > 0) { // Падіння на платформу
                y = platform.y - hitboxHeight;
                velocityY = 0;
                currentlyOnGround = true;
            } else if (velocityY < 0) { // Удар головою об платформу
                y = platform.y + platform.height;
                velocityY = 0;
            }
        }
    }
    this.onGround = currentlyOnGround;
}
```

Лістинг 3.3 – Фрагмент коду Player.java (Обробка зіткнень)

Як видно з Лістингу 3.3, рух та перевірка зіткнень по осях X та Y

відбуваються послідовно. Це запобігає "залипанню" в стінах при одночасному русі та падінні. Спочатку застосовується горизонтальний рух, і якщо виникає зіткнення, позиція x гравця коригується до межі платформи. Тільки після цього застосовується вертикальний рух та аналогічна перевірка по осі Y.

Покращення керування ("Game Feel"): Для того, щоб керування персонажем відчувалося "відзивчивим", було реалізовано дві поширені техніки [6]:

"Час Койота" (Coyote Time): У грі ця механіка дає гравцеві невелике часове вікно (у нашому випадку 0.1 секунди або ~12 кадрів) для виконання стрибка після того, як персонаж фізично втратив контакт з поверхнею. Тобто замість миттєвої заборони стрибка при onGround = false, введено таймер coyoteTimeCounter.

Алгоритм реалізації:

- Поки гравець стоїть на землі (onGround == true), таймер постійно скидається до свого максимального значення (COYOTE_TIME).
- Коли гравець сходить з платформи, таймер починає зменшуватися на величину DELTA_TIME кожного кадру.
- Умова перевірки можливості стрибка змінюється з if (onGround) на if (coyoteTimeCounter > 0).

Це дозволяє нівелювати затримку реакції людини та зробити переміщення по платформах більш плавним і пробачаючим до помилок.

"Буферизація стрибка" (Jump Buffering): Ця механіка вирішує проблему натискання кнопки стрибка передчасно, коли персонаж ще знаходиться у повітрі, але вже дуже близько до землі, що може сприйматися візуально наче гравець вже знаходиться на землі. Без буферизації таке натискання було б проігноровано, і гравцеві довелося б натискати кнопку ще раз, що порушує потік гри та може сприйматися як не зчитування натискання кнопки стрибку.

Алгоритм реалізації:

- При натисканні клавіші стрибка, замість миттєвої перевірки умов, ми встановлюємо jumpBufferCounter на певне значення (наприклад, 0.1 секунди). Це означає: "Гравець хоче стрибнути".
- Таймер зменшується з часом.
- На кожному кадрі перевіряється умова: якщо jumpBufferCounter > 0 (гравець нещодавно натиснув кнопку) і coyoteTimeCounter > 0 (гравець може відштовхнутися), то виконується стрибок.

```
// (З файлу Player.java)
private void applyGravityAndJumpLogic() {
    // 1. Застосовуємо гравітацію, тільки якщо в повітрі
    if (!this.onGround) {
        velocityY += gravity;
    }

    // 2. Оновлюємо лічильники Coyote Time та Jump Buffer
    final double DELTA_TIME = 1.0 / 120.0;
    if (onGround) coyoteTimeCounter = COYOTE_TIME;
    else coyoteTimeCounter -= DELTA_TIME;

    if (jumpBufferCounter > 0) jumpBufferCounter -= DELTA_TIME;

    // 3. Виконуємо стрибок, якщо обидва лічильники дозволяють
    if (jumpBufferCounter > 0 && coyoteTimeCounter > 0) {
        velocityY = jumpStrength;
        jumpBufferCounter = 0;
        coyoteTimeCounter = 0;
        this.onGround = false; // Примусовий відрив від землі
    }
}
```

Лістинг 3.4 – Фрагмент коду Player.java (Логіка стрибка та гравітації)

Як видно з Лістингу 3.4, критичним виправленням стало застосування гравітації (рядок 1) тільки якщо гравець не на землі (!this.onGround). Це запобігло "тремтінню" персонажа на платформі (постійному перемиканню onGround між true та false), що, у свою чергу, стабілізувало роботу анімації.

3.3. Реалізація системи обробки вводу для двох гравців

Стандартний KeyListener у Swing є проблематичним для ігор, оскільки він базується на подіях та погано обробляє одночасні натискання. Було реалізовано більш надійний підхід "Stateful Input" (Опитування стану).

Клас InputHandler реалізує KeyListener.

Set<Integer> pressedKeys зберігає коди всіх утримуваних клавіш.

GamePanel.handleContinuousInput() на кожному кадрі опитує цей Set та застосовує тривалі дії (рух).

```
// (З файлу InputHandler.java)
public class InputHandler implements KeyListener, MouseListener {
    private final Set<Integer> pressedKeys = new HashSet<>();
    //...
    @Override
    public void keyPressed(KeyEvent e) {
        pressedKeys.add(e.getKeyCode()); // Додаємо клавішу до
колекції
        panel.handleSingleKeyPress(e); // Обробляємо одноразові
дії
    }
    @Override
    public void keyReleased(KeyEvent e) {
        pressedKeys.remove(e.getKeyCode()); // Видаляємо клавішу
    }
    //...
}
```

Лістинг 3.5 – Фрагмент коду InputHandler.java (Збереження стану клавіш)

```
// (З файлу GamePanel.java)
private void handleContinuousInput() {
    Set<Integer> keys = inputHandler.getPressedKeys();
    if (currentState == GameState.PLAY_PHASE) {
        // Гравець 1: умовний оператор 'if-else if-else' гарантує,
        // що 'stopMoving()' викликається, коли ні 'A', ні 'D' не
натиснуті.
        if            (keys.contains(KeyEvent.VK_A)            &&
!keys.contains(KeyEvent.VK_D)) player1.moveLeft();
        else      if      (keys.contains(KeyEvent.VK_D)      &&
```

```
!keys.contains(KeyEvent.VK_A)) player1.moveRight();
        else player1.stopMoving();

        //... (Аналогічно для Гравця 2)
    }
    //... (Аналогічно для BUILD_PHASE)
}
```

Лістинг 3.6 – Фрагмент коду GamePanel.java (Обробка тривалого руху)

Цей підхід вирішив проблему "залипання" руху та дозволив обом гравцям рухатися одночасно. Одноразові дії (стрибок, постріл) обробляються окремо у handleSingleKeyPress. Для розрізнення клавіш, що дублюються (Shift, Control), використовується KeyEvent.getKeyLocation().

3.4. Реалізація модуля анімації

Система анімації базується на техніці "Sprite Sheet Blitting" (перенесення блоків зображення). Замість завантаження сотень окремих файлів для кожного кадру (що є неефективним з точки зору I/O операцій та використання пам'яті), всі кадри однієї анімації зберігаються в одному зображенні — спрайт-смузі (Sprite Strip).

Клас Animation працює як скінченний автомат, що керує часом.

- Ініціалізація: При створенні об'єкта Animation у пам'ять завантажується BufferedImage, що містить всю смугу. За допомогою методу getSubimage(x, y, w, h) смуга програмно "нарізається" на масив окремих кадрів. Це відбувається лише один раз при старті рівня, що мінімізує навантаження на процесор під час гри.
- Таймінг: Метод update() використовує системний час (System.currentTimeMillis()) для відстеження дельти часу. Змінна lastFrameTime зберігає час останнього перемикання кадру.
- Перемикання: Коли різниця між поточним часом і lastFrameTime перевищує поріг delay (наприклад, 100 мс для бігу), індекс поточного

кадру інкрементується.

- Зациклення: Використовується оператор модуля або проста перевірка if (index >= frames.length) index = 0, що забезпечує безперервність анімації (looping).

Такий підхід дозволяє гнучко налаштовувати швидкість кожної анімації окремо, не прив'язуючись до частоти кадрів (FPS) самої гри, що робить візуальну складову незалежною від продуктивності комп'ютера.

Створено допоміжний клас Animation, що інкапсулює логіку анімації. Він зберігає масив кадрів BufferedImage[] та затримку delay.

У Player.loadAnimations() спрайт-смуги завантажуються як BufferedImage. Кадри "нарізаються" за допомогою sheet.getSubimage().

Рис. 3.3 – “Приклад спрайт-смуги” [24]

```
// (З файлу Player.java)
private   BufferedImage[]   loadAnimationFromSheet(BufferedImage
sheet, int startX, int rowY, int frameCount, int width, int height)
{
    if (sheet == null) return new BufferedImage[0];
    BufferedImage[] frames = new BufferedImage[frameCount];
    for (int i = 0; i < frameCount; i++) {
        frames[i] = sheet.getSubimage((startX + i) * width, rowY
* height, width, height);
    }
    return frames;
}

//... всередині public void update(List<Platform> platforms)
if (currentAnimation != null) {
    // Спрощена логіка через відсутність спрайту стрибка
    if (velocityX != 0) {
        currentAnimation = runAnimation;
    } else {
        currentAnimation = idleAnimation;
    }
```

```
        currentAnimation.update();
    }
```

Лістинг 3.7 – Фрагмент коду Player.java (Логіка анімації)

Розділення візуального та фізичного розміру: Для точних зіткнень розміри гравця було розділено: visualSize (50x50) для малювання та hitboxWidth/hitboxHeight (26x42) для фізики.

Віддзеркалення: Рух вліво (facingDirection = -1) реалізовано через трюк з малюванням drawImage з від'ємною шириною, що миттєво віддзеркалює спрайт.

3.5. Реалізація ключових ігрових механік

Будівництво**:** Player має метод togglePlacePlatform(), який створює Platform типу PLAYER_BUILT та зберігає її в ghostPlatform. GamePanel.confirmPlacement() переносить її у загальний список platforms. Платформи гравців малюються процедурно як "скло" (Platform.drawGlass()) з використанням java.awt.Color з альфа-каналом (прозорістю).

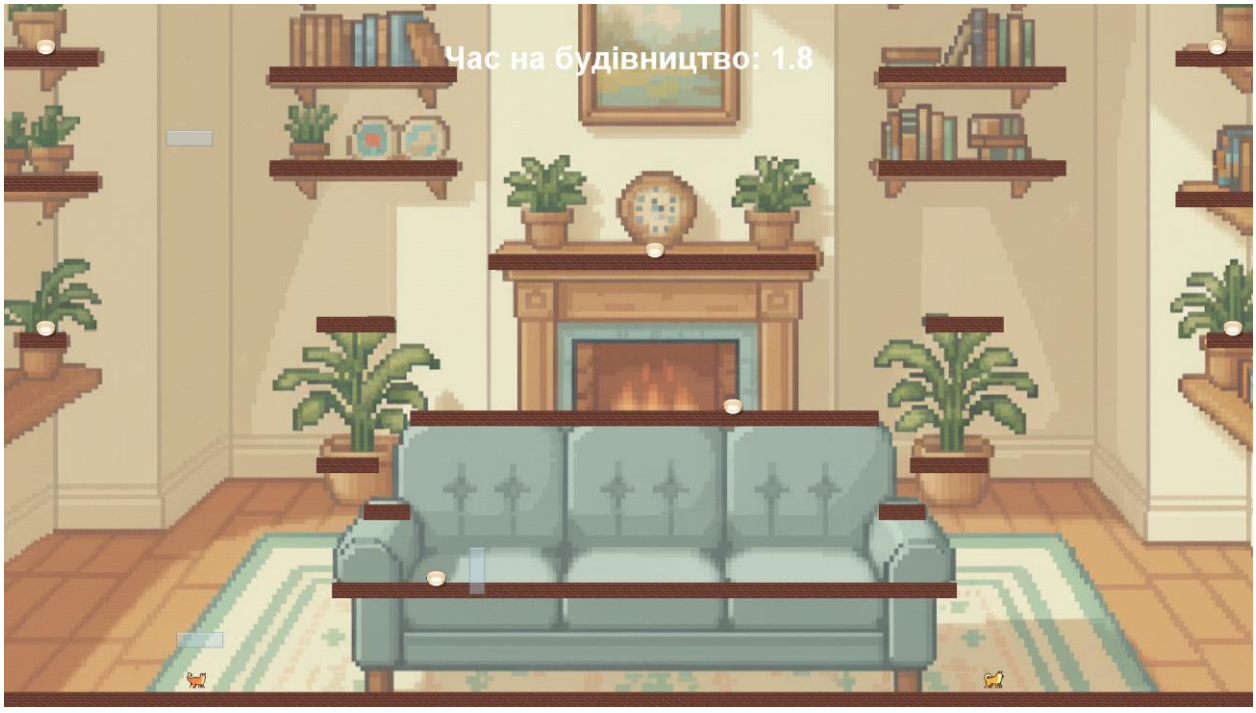

Рис. 3.4 – "Скріншот фази будівництва"

Стрільба та Руйнування: Player.shoot() створює Projectile, обираючи спрайт (golf.png або tennis.png [27]) залежно від типу гравця. GamePanel.updateProjectiles() перевіряє projectile.getBounds(). intersects(platform.getBounds()). Якщо platform.isDestructible() (тобто тип PLAYER_BUILT) повертає true, платформа видаляється з ArrayList за допомогою Iterator.remove().

Рис. 3.5 – "Скріншот пострілу"

3.6. Реалізація графічного інтерфейсу (GUI)

Головне меню (Menu.java): Малює фон (menu_background.png), назву, опис гри та інструкції з керування. Кнопки реалізовані як Rectangle, кліки по яких відстежуються в InputHandler.mousePressed().

Ігровий HUD (GamePanel.drawHUD()): Малює рахунок гравців (з тематичним текстом "грам лососю" / "грам стейку") та таймер ігрової фази.

Екран завершення гри (GamePanel.drawGameOverScreen()): Малює

напівпрозорий темний шар та текст з переможцем залежно від порівняння player1Score та player2Score.

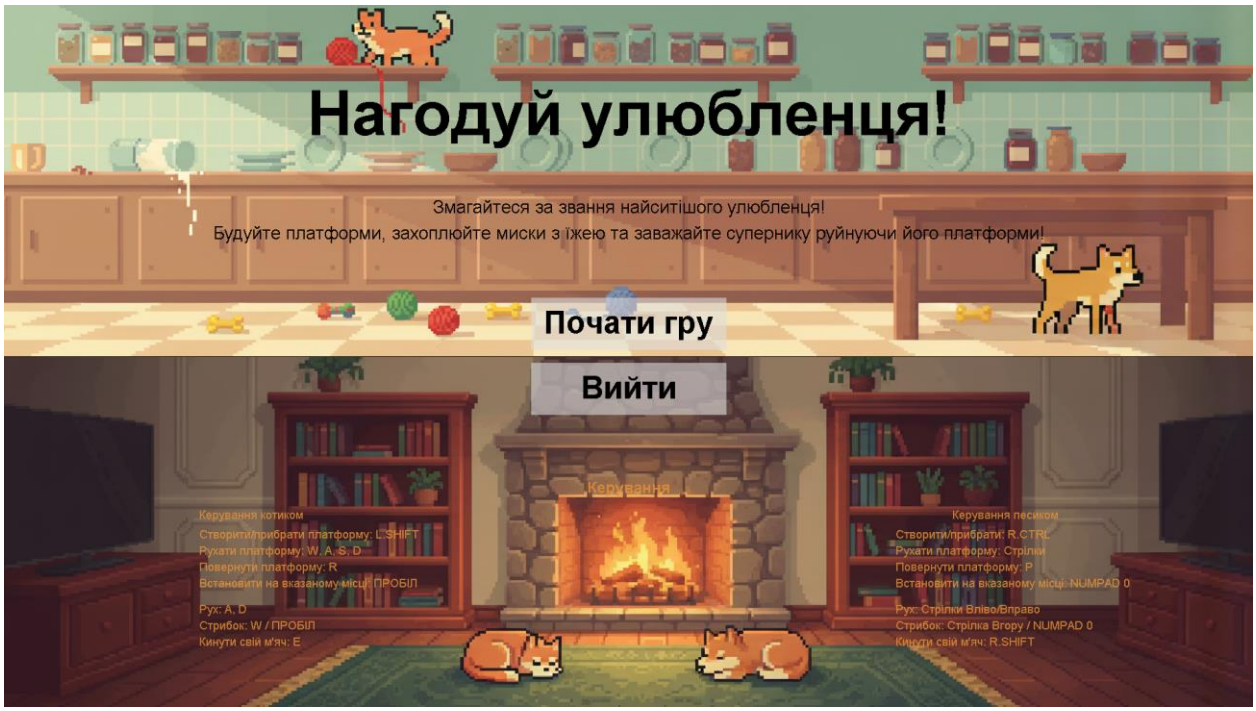

Рис. 3.6 – "Скріншот головного меню"

Рис. 3.7 – "Скріншот ігрового процесу з HUD"

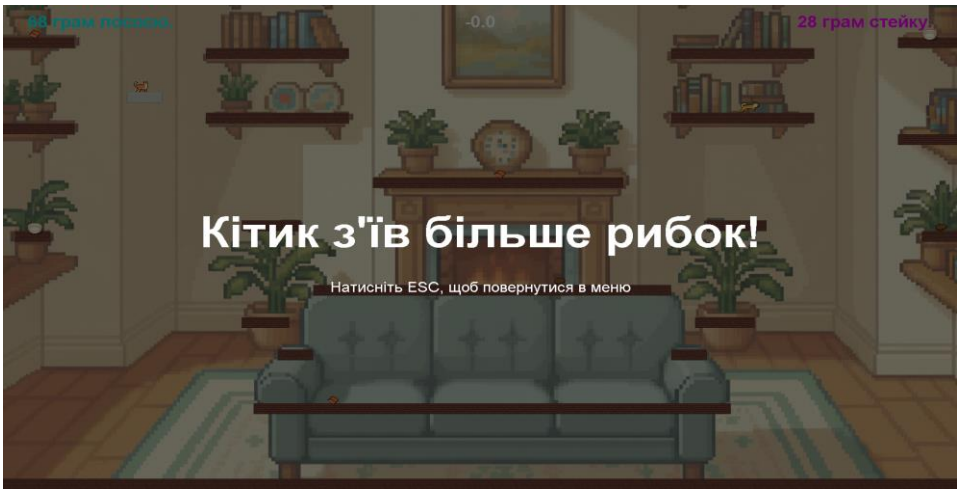

Рис. 3.8 – "Скріншот екрану завершення гри"

3.7. Рефакторинг та документування коду

На завершальному етапі розробки було проведено комплексний рефакторинг програмного коду. Основною метою цього процесу було покращення нефункціональних характеристик системи: читабельності, супроводжуваності та розширюваності, без зміни зовнішньої поведінки програми, дотримуючись принципів DRY (Don't Repeat Yourself) [7] та KISS (Keep It Simple, Stupid) [8].

Інкапсуляція (Encapsulation): Було проведено аудит модифікаторів доступу. Поля класів, які раніше мали пакетний або публічний доступ (наприклад, координати x, y у класі Player або Projectile), було закрито модифікатором private. Для доступу до них з боку інших класів (зокрема, GamePanel для перевірки колізій) було реалізовано публічні методи-геттери (getBounds(), getHitboxWidth(), getType()). Це захищає внутрішній стан об'єктів від некоректних змін ззовні.

Принцип DRY (Don't Repeat Yourself): Було виявлено та усунуто дублювання коду.

- Логіка завантаження ресурсів: Раніше завантаження спрайтів відбувалося в конструкторах кожного екземпляра, що призводило до

надлишкових операцій I/O. Цю логіку було винесено у статичні блоки ініціалізації (static {... }) у класах Platform, CapturePoint та Player. Це гарантує, що ресурси завантажуються в пам'ять лише один раз при запуску програми.

- Логіка малювання: У класі Platform повторюваний код вибору кольору або спрайту було оптимізовано через використання допоміжних методів (drawGlass()).

Принцип KISS (Keep It Simple, Stupid): Складні методи були декомпозовані на менші, більш спеціалізовані.

- Метод update() у класі Player, який відповідав за всю фізику, було розділено на окремі логічні блоки: handleHorizontalCollisions(), handleVerticalCollisions() та applyGravityAndJumpLogic(). Це значно спростило налагодження фізики стрибка та колізій.
- Метод drawScene() у GamePanel було розділено на drawGameWorld() (відмальовка ігрових об'єктів) та drawUI() (відмальовка інтерфейсу), що покращило структуру коду візуалізації.

Розподіл відповідальності (Separation of Concerns): Логіку керування "примарною" платформою під час фази будівництва (створення, переміщення, поворот) було перенесено з класу-контролера GamePanel безпосередньо до класу Player.

Документування: Додано Javadoc-коментарі [2] до всіх класів та публічних методів.

За допомогою інструментів IDE IntelliJ IDEA було проведено автоматичну оптимізацію імпортів (видалення невикористовуваних класів) та форматування коду згідно зі стилем Java Code Conventions (відступи, розташування дужок, іменування змінних). Це забезпечило єдиний стиль коду в усьому проєкті.

3.8. Теоретичне обґрунтування архітектури для майбутнього мультиплеєру

Хоча поточна версія прототипу реалізує локальний мультиплеєр (Hotseat), архітектура проєкту була розроблена з урахуванням потенційного масштабування до мережевої гри. Для реалізації мережевої взаємодії в середовищі Java найбільш перспективним є використання клієнт-серверної моделі з авторитетним сервером.

1. Синхронізація стану: Замість передачі зображення (стримінгу), по мережі мають передаватися лише вхідні дані гравців (Input Prediction) або координати об'єктів (Snapshot Interpolation). Враховуючи динаміку платформера, оптимальним є підхід, де клієнт надсилає серверу натиснуті клавіші, а сервер проводить симуляцію фізики та повертає "істинні" координати.
2. Протокол: Для передавання координат (швидкі, часті оновлення) доцільно використовувати протокол UDP (User Datagram Protocol), який не гарантує доставки, але має мінімальну затримку. Для критичних подій (початок гри, будівництво платформи, зміна рахунку) краще використовувати TCP (Transmission Control Protocol), він гарантовано доставить інформацію.
3. Віртуальні потоки (Java 21): Використання новітньої можливості Java 21 — Project Loom (Virtual Threads) — дозволило б створити високоефективний сервер, де кожне з'єднання обробляється окремим віртуальним потоком. Це значно спрощує написання коду порівняно з класичним неблокуючим I/O (NIO), зберігаючи при цьому високу продуктивність та здатність обслуговувати тисячі одночасних підключень.

РОЗДІЛ 4
ТЕСТУВАННЯ ТА ВИСНОВКИ

4.1. План та результати тестування

Тестування проєкту проводилося вручну (manual testing) протягом усього циклу розробки.

Модульне тестування (Unit Testing):

1. Фізика: Перевірялася коректність стрибків, падіння, відсутність "залипання" на стінах. Особлива увага приділялася тестуванню "Coyote Time" та "Jump Buffering".
2. Зіткнення: Перевірялися всі 8 напрямків зіткнення гравця з платформами.
3. Анімація: Перевірялося коректне перемикання станів IDLE/RUN та віддзеркалення спрайту.
4. Введення: Тестувалися всі клавіші керування, а також одночасне натискання (напр., рух по діагоналі у фазі будівництва, стрибок під час бігу, одночасні дії декількох гравців).

Інтеграційне тестування:

1. Будівництво та Руйнування: Перевірявся повний цикл: гравець 1 будує платформу, гравець 2 її руйнує.
2. Захоплення точки інтересу, зміна рахунку та завершення гри: Перевірявся сценарій, коли гравець утримує точку, рахунок коректно нараховується, таймер завершується, і на екрані GAME_OVER відображається правильний переможець.

Тестування GUI:

1. Перевірялася робота кнопок меню, діалогу виходу (JOptionPane), коректність повернення фокусу на панель.

2. Перевірялося відображення всіх елементів HUD та тексту.

Результати: Під час тестування було виявлено та виправлено низку критичних помилок:

1. Нестабільна анімація (IDLE/RUN): Виправлено шляхом зміни порядку розрахунків фізики в Player.update().

2. Відсутність руху (VelocityX = 0.0): Виправлено шляхом рефакторингу логіки в GamePanel.handleContinuousInput().

3. Втрата фокусу: Виправлено примусовим викликом requestFocusInWindow().

4. Помилка ConcurrentModificationException: Виправлено використанням Iterator при видаленні об'єктів.

4.2. Аналіз продуктивності

Прототип було протестовано на цільовій машині з процесором Intel(R) Core(TM) i3-9100 CPU @ 3.60GHz (3.60 GHz) та 16.0 Гб ОЗП. Гра стабільно працювала без помітних падінь FPS.

- Обмеження Swing: Java Swing [9] не використовує DirectX [22] або OpenGL [10] для рендерингу, покладаючись на CPU та Java 2D API. Це є вузьким місцем. При значному збільшенні кількості об'єктів (сотні снарядів, тисячі платформ) продуктивність почне падати.

Застосовані оптимізації:

- Завантаження ресурсів: Усі спрайти завантажуються один раз при старті гри (static блоки), що усуває затримки під час гри.
- Видалення об'єктів: Снаряди, що вилетіли за екран, видаляються зі списку projectiles, щоб не навантажувати цикл update.

Для комерційного продукту рекомендується перехід на LibGDX [23] або JavaFX [11].

4.3. Напрямки подальшого розвитку проекту

1. Впровадження аудіоряду: Додавання javax.sound.sampled для завантаження та програвання .wav файлів (SFX та BGM).
2. Розширення системи анімації: Додавання анімацій стрибку, отримання пошкоджень, пострілу.
3. Нові типи платформ та ШІ: Створення "слизьких" (лід) або "стрибучих" платформ. Розробка ШІ для однокористувацького режиму.
4. Мережевий мультиплеєр: Рефакторинг архітектури під клієнт-серверний модель з використанням java.net.Socket та віртуальних потоків (Project Loom) [5] Java 21.

ВИСНОВКИ

За результатами дипломної роботи було успішно виконано поставлене завдання: досліджено, спроєктовано та розроблено функціональний прототип 2D PvP гри мовою Java з використанням бібліотеки Swing.

Основні досягнення та реалізований функціонал:

1. Створено ігровий рушій з нуля, що включає ігровий цикл на базі javax.swing.Timer, систему управління станами (GameState) та надійну систему обробки одночасного вводу "Stateful Input".

2. Реалізовано фізичну модель 2D платформера, включаючи гравітацію, зіткнення AABB з розділенням осей, та покращення керування "Coyote Time" і "Jump Buffering".

3. Впроваджено унікальні ігрові механіки: тактичну фазу будівництва руйнівних платформ та змагальний режим захоплення контрольних точок з підрахунком очок.

4. Розроблено повний ігровий цикл: головне меню, фази гри (будівництво, бій), екран завершення з визначенням переможця.

5. Інтегровано графічні ресурси: завантаження спрайтів (.png), реалізовано систему анімації (спокій, біг) зі спрайт-смуг та завантаження рівнів з текстового файлу.

6. Код проєкту структуровано відповідно до принципів ООП (інкапсуляція, розподіл відповідальності) та рефакторинговано згідно з принципами DRY та KISS.

Виклики та їх вирішення: Під час розробки було вирішено низку нетривіальних технічних завдань: налагодження фізики стрибка та запобігання "тремтінню" анімації на платформах; коректна обробка одночасного вводу від двох гравців; вирішення проблем із втратою фокусу введення у Swing; ефективне завантаження та "нарізка" спрайтів для анімації.

Ці проблеми були успішно вирішені шляхом аналізу та застосування відповідних програмних рішень.

Практичне значення: Виконана робота демонструє володіння мовою Java та принципами ООП для розробки складного інтерактивного додатку з нуля. Створений прототип може бути використаний як основа для подальшого розвитку проєкту.

Загалом, мета дипломної роботи була досягнута. Проведено дослідження та реалізовано програмний продукт, що демонструє вирішення поставлених технічних та аналітичних завдань.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Шилдт Г. Java. Посібник для початківців / Герберт Шилдт. – 8-е вид. – К. : Діалектика, 2022. – 688 с.

2. Oracle. (2024). Javadoc Tool. – [Електронний ресурс]. – Режим доступу: https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html

3. Об'єктно-орієнтоване програмування. (2023). Вікіпедія. – [Електронний ресурс].: https://uk.wikipedia.org/wiki/Об'єктно-орієнтоване_програмування

4. Schell, J. The Art of Game Design: A Book of Lenses / Jesse Schell. – 3rd ed. – CRC Press, 2019. – 648 p.

5. Oracle. (2024). Project Loom. – [Електронний ресурс]. – Режим доступу: https://openjdk.org/projects/loom/

6. Nystrom, R. Game Programming Patterns / Robert Nystrom. – Genever Benning, 2014. – 350 p. – [Електронний ресурс]. – Режим доступу: https://www.gameprogrammingpatterns.com/state.html

7. Don't repeat yourself. (2023). Вікіпедія. – [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/wiki/Don't_repeat_yourself

8. Принцип KISS. (2023). Вікіпедія. – [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/wiki/Принцип_KISS

9. Oracle. (2024). Java Swing Tutorial. – [Електронний ресурс]. – Режим доступу: https://docs.oracle.com/javase/tutorial/uiswing/

10. OpenGL. (2024). Khronos Group. – [Електронний ресурс]. – Режим доступу: https://www.opengl.org/

11. Horstmann, C. S. Core Java Volume I – Fundamentals / Cay S. Horstmann. – 12th ed. – Prentice Hall, 2021. – 928 p.

12. Гамма Е., Хелм Р., Джонсон Р., Вліссідес Д. Патерни проєктування.

Елементи повторно використовуваного об'єктно-орієнтованого програмного забезпечення / Е. Гамма, Р. Хелм, Р. Джонсон, Д. Вліссідес. – Addison-Wesley, 1994. – 416 с.

13. State pattern. (2024). Refactoring.Guru. – [Електронний ресурс]. – Режим доступу: https://refactoring.guru/design-patterns/state

14. Pixel art. – [Електронний ресурс].: https://en.wikipedia.org/wiki/Pixel_art

15. MOBA (Multiplayer Online Battle Arena). (2024). Wikipedia. – [Електронний ресурс].: https://en.wikipedia.org/wiki/Multiplayer_online_battle_arena

16. Java Virtual Machine (JVM) Specification. (2024). Oracle. – [Електронний ресурс]. – Режим доступу: https://docs.oracle.com/javase/specs/jvms/se21/html/

17. Oracle Java SE Support Roadmap. (2024). Oracle. – [Електронний ресурс]. – Режим доступу: https://www.oracle.com/java/technologies/java-se-support-roadmap.html

18. JetBrains. (2024). IntelliJ IDEA. – [Електронний ресурс]. – Режим доступу: https://www.jetbrains.com/idea/

19. itch.io - Asset Market. (2024). – [Електронний ресурс]. – Режим доступу: https://itch.io/game-assets

20. Oracle. (2024). Focus Subsystem. – [Електронний ресурс]. – Режим доступу: https://docs.oracle.com/javase/tutorial/uiswing/misc/focus.html

21. Observer pattern. (2024). Refactoring.Guru. – [Електронний ресурс]. – Режим доступу: https://refactoring.guru/design-patterns/observer

22. DirectX. (2024). Microsoft. – [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/en-us/windows/win32/directx

23. LibGDX (2024). – [Електронний ресурс]. – Режим доступу: https://libgdx.com/

24. maccari, d. (2021). Free Street Animal Pixel Art Asset Pack. https://free-game-assets.itch.io/free-street-animal-pixel-art-asset-pack

25. Pixelkitty Art. (2021). Pixel Wood Tileset. https://pixelkitty-art.itch.io/pixel-wood-tileset

26. Ghostpixxells. (2020). PixelFood - Icon Pack [16x16]. https://ghostpixxells.itch.io/pixelfood

27. BeemaxStudio. (2020). Sport Balls Pixel Pack. https://beemaxstudio.itch.io/sport-balls-pixel-pack