МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет радіоелектроніки

Факультет комп’ютерних наук

(повна назва)

Кафедра програмної інженерії

(повна назва)

**КОМПЛЕКСНИЙ КУРСОВИЙ ПРОЄКТ**

**Пояснювальна записка**

рівень вищої освіти перший (бакалаврський)

Ігровий програмний застосунок в жанрі Roguelike. Backend

(тема)

Виконав:

здобувач (ка) 3 курсу, групи ПЗПІ–22–4

Станіслав ЛОГВІН

(Власне ім’я, ПРІЗВИЩЕ)

Спеціальність 121 – Інженерія програмного\_ \_\_

забезпечення

(код і повна назва спеціальності)

Тип програми освітньо–професійна

Освітня програма Програмна інженерія

(повна назва освітньої програми)

Керівник доцент кафедри ПІ Наталя ВАЛЕНДА

(посада, Власне ім’я, ПРІЗВИЩЕ)

|  |
| --- |
| Члени комісії (Власне ім’я, ПРІЗВИЩЕ, підпис) |
| Доц. Побіженко І.О. |
| Ст. викл. Широкопетлєва М.С. |
| Ст. викл. Новіков Ю.С. |

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерних наук

Кафедра програмної інженерії

Рівень вищої освіти перший (бакалаврський)

Спеціальність 121 – Інженерія програмного забезпечення

Тип програми Освітньо–професійна

Освітня програма Програмна Інженерія

(шифр і назва)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Курс | 3 | Група | ПЗПІ–22–4 | Семестр | 6 |

**ЗАВДАННЯ**

***на курсовий проект(роботу) студента***

здобувачеві Логвіну Станіславу Юрійовичу

(прізвище, ім’я, по батькові)

1. Тема роботи Ігровий програмний застосунок в жанрі Roguelike

2. Термін здачі студентом закінченої роботи “ 21 ” 06 2025 р.\_

3. Вихідні дані до проєкту створення гри в середовищі Unity, програмування на мові C#, застосування UI/UX елементів для інтерфейсу, розробка ігрової логіки, керування гравцем, об’єктами, ворогами, створення веб–сторінки з описом гри, додавання функції завантаження

4. Перелік питань, що потрібно опрацювати в роботі

Розробка внутрішньої архітектури гри з використанням UML–діаграм, реалізація ігрової логіки та поведінки об’єктів, програмування механік керування гравцем і взаємодії з ігровим середовищем, обробка подій і станів гри, налаштування сценаріїв для ворогів та NPC, забезпечення стабільності та тестування функціоналу.

**КАЛЕНДАРНИЙ ПЛАН**

|  |  |  |  |
| --- | --- | --- | --- |
| № | Назва етапів роботи | Термін виконання етапів роботи | Примітка |
| 1 | Аналіз предметної галузі | 23.03 – 25.03 | виконано |
| 2 | Розробка постановки задачі | 26.03 – 27.03 | виконано |
| 3 | Проектування ПЗ | 01.04 – 04.03 | виконано |
| 4 | Програмна реалізація | 14.03 – 02.06 | виконано |
| 5 | Аналіз результатів | 02.06 – 04.06 | виконано |
| 6 | Підготовка пояснювальної записки. | 05.06 – 18.06 | виконано |
| 7 | Перевірка на наявність ознак академічного плагіату | 19.06 – 20.06 | виконано |
| 8 | Захист роботи | 22.06 | виконано |

Дата видачі завдання “ 21 ” 03 2025р.

Здобувач \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис)

Керівник роботи доцент кафедри ПІ Наталя ВАЛЕНДА

(підпис) (посада, Власне ім’я, ПРІЗВИЩЕ)

**РЕФЕРАТ / ABSTRACT**

Пояснювальна записка містить: 48 с., 22 рис., 10 джерел.

РОГУЛАЙК, МОБІЛЬНА ГРА, ІГРОВА ЛОГІКА, UNITY, C#, BACKEND, GAME MANAGER, ПОДІЙНА СИСТЕМА, АРХІТЕКТУРА, ПРОЄКТУВАННЯ, ТЕСТУВАННЯ

Об’єктом розробки є мобільний ігровий застосунок жанру Roguelike з процедурною генерацією та ігровими механіками з використанням цеглин як основного інструменту взаємодії.

Метою проєкту є проєктування і реалізація серверної логіки мобільної гри з використанням рушія Unity, з акцентом на структуровану архітектуру компонентів, ефективну обробку ігрових подій та гнучке управління станом гри.

Методи реалізації логіки включають використання мови програмування C#, патернів проєктування (Singleton, Observer), обробку подій через делегати та інтерфейси, структурування коду за допомогою компонентної моделі Unity. Основну логіку гри реалізовано у вигляді окремих модулів: GameManager, BrickManager, EnemyController, LevelManager, які взаємодіють між собою через чітко визначені інтерфейси. Для тестування використовувались вбудовані засоби Unity Editor та модульні перевірки поведінки класів.

У результаті розробки створено архітектурно чіткий ігровий застосунок, що має масштабовану логіку, підтримку ігрових подій, систему рівнів та можливість подальшого розвитку (мережеві функції, прогрес, монетизація).

ROGUELIKE, MOBILE GAME, GAME LOGIC, UNITY, C#, BACKEND, GAME MANAGER, EVENT SYSTEM, ARCHITECTURE, DESIGN, TESTING

The object of development is a mobile Roguelike game application featuring procedural generation and core gameplay based on interactive bricks.

The goal of the project is to design and implement the backend logic of a mobile game using the Unity engine, focusing on modular architecture, efficient game state management, and scalable event handling.

The development methods included the use of the C# programming language, design patterns (Singleton, Observer), delegate–based event management, and Unity's component system. Core gameplay logic was divided into modules: GameManager, BrickManager, EnemyController, LevelManager – each interacting via well–defined interfaces. Unity Editor tools and basic unit testing were used for validation and debugging.

As a result, an architecturally consistent mobile game application was developed, featuring modular gameplay logic, level system, event–driven interactions, and scalability for future improvements (e.g., networking, progress saving, monetization).

**ЗМІСТ**

[**Вступ** 9](#_Toc201506757)

[**1 Аналіз предметної галузі** 11](#_Toc201506758)

[1.1 Аналіз предметної галузі 11](#_Toc201506759)

[1.2 Виявлення проблем та актуалізація рішень 15](#_Toc201506760)

[1.3 Постановка задачі 17](#_Toc201506761)

[**2 Формування вимог до ігрової логіки та бекенд систем** 19](#_Toc201506762)

[2.1 Аналіз ігрових механік та системних вимог 19](#_Toc201506763)

[2.2 Функціональні вимоги до ігрової логіки 23](#_Toc201506764)

[2.2.1 Вимоги до системи процедурної генерації карт 23](#_Toc201506765)

[2.2.2 Вимоги до фізичного рушія та системи взаємодій 24](#_Toc201506766)

[2.2.3 Вимоги до системи штучного інтелекту 25](#_Toc201506767)

[2.2.4 Вимоги до системи керування ігровими об'єктами 26](#_Toc201506768)

[2.3 Нефункціональні вимоги та обмеження системи 28](#_Toc201506769)

[2.3.1 Вимоги до продуктивності та оптимізації 28](#_Toc201506770)

[2.3.2 Вимоги до масштабованості та архітектурної гнучкості 29](#_Toc201506771)

[2.3.3 Вимоги до стабільності та надійності 30](#_Toc201506772)

[2.3.4 Вимоги до ефективності використання ресурсів 30](#_Toc201506773)

[**3 Архітектура ігрової логіки та бекенд систем** 32](#_Toc201506774)

[3.1 Проєктування основних ігрових систем 32](#_Toc201506775)

[3.1.1 Архітектура системи керування об'єктами 34](#_Toc201506776)

[3.1.2 Фізична підсистема та система колізій 35](#_Toc201506777)

[3.1.3 Система управління станами гри 36](#_Toc201506778)

[3.2 Архітектура системи штучного інтелекту 37](#_Toc201506779)

[3.2.1 Базова архітектура AI–агентів 38](#_Toc201506780)

[3.2.2 Система поведінкових патернів 39](#_Toc201506781)

[3.2.3 Система координації та синхронізації 40](#_Toc201506782)

[3.3 Система процедурної генерації контенту 41](#_Toc201506783)

[3.3.1 Математична основа генерації 41](#_Toc201506784)

[3.3.2 Алгоритм процедурної генерації 42](#_Toc201506785)

[3.3.3 Система валідації та гарантії прохідності 43](#_Toc201506786)

[3.4 Система фізики та взаємодій 45](#_Toc201506787)

[3.4.1 Архітектура системи керування гравцем 45](#_Toc201506788)

[3.4.2 Система виявлення контакту з поверхнею 46](#_Toc201506789)

[3.4.3 Фізична модель снарядів та блоків 47](#_Toc201506790)

[3.4.4 Система обробки колізій та взаємодій 48](#_Toc201506791)

[**4 Опис прийнятих програмних рішень** 49](#_Toc201506792)

[**5 Впровадження ігрового програмного забезпечення** 52](#_Toc201506793)

[**Висновки** 55](#_Toc201506794)

[**Перелік джерел посилання** 57](#_Toc201506795)

[ДОДАТОК А 59](#_Toc201506796)

**ПЕРЕЛІК СКОРОЧЕНЬ**

UI – User Interface (інтерфейс користувача)

UX – User Experience (користувацький досвід)

2D – Two Dimensional (двовимірний)

3D – Three Dimensional (тривимірний)

IDE – Integrated Development Environment (інтегроване середовище розробки)

FPS – Frames Per Second (кадрів за секунду)

AI – Artificial Intelligence (штучний інтелект)

UML – Unified Modeling Language (уніфікована мова моделювання)

**ВСТУП**

Сучасна індустрія мобільних ігор характеризується зростанням вимог до технічної досконалості програмного забезпечення в умовах обмежених ресурсів мобільних пристроїв. Розробка ефективної бекенд архітектури ігрових систем становить критично важливе завдання для забезпечення стабільної продуктивності та масштабованості програмних продуктів.

Зростання популярності мобільних ігор жанру Roguelike створює необхідність у розробці спеціалізованих архітектурних рішень, що поєднують динамічний геймплей з ефективним використанням системних ресурсів. Особливої актуальності набувають питання процедурної генерації контенту, оптимізації фізичних взаємодій та реалізації штучного інтелекту в умовах обмеженої обчислювальної потужності мобільних платформ.

Традиційні підходи до архітектурного проектування часто не враховують специфічні вимоги мобільного середовища, що призводить до неефективного використання ресурсів та нестабільної продуктивності.

Об'єктом дослідження є процеси проектування та реалізації бекенд систем мобільних ігор з динамічною генерацією контенту. Предметом дослідження виступають архітектурні принципи та технічні методи створення ігрової логіки для мобільної гри Black Prince в жанрі Roguelike.

Метою курсової роботи є розробка комплексної бекенд архітектури мобільної гри, що забезпечує ефективну реалізацію процедурної генерації рівнів, штучного інтелекту ворогів та фізичних взаємодій при збереженні оптимальної продуктивності.

Основні завдання включають проектування системи процедурної генерації карт з гарантованою прохідністю, розробку модульної системи штучного інтелекту з координованою поведінкою, реалізацію ефективної фізичної підсистеми та створення системи управління об'єктами з оптимізацією ресурсів.

У роботі застосовуються методи системного аналізу, принципи об'єктно–орієнтованого проектування та алгоритмічні методи для реалізації ігрової логіки. Технічна реалізація здійснюється з використанням Unity Engine та мови програмування C#.

Результати дослідження мають безпосередню практичну цінність для індустрії розробки мобільних ігор. Розроблена архітектурна модель може служити основою для створення аналогічних ігрових проектів, а запропоновані рішення щодо оптимізації застосовні для широкого класу мобільних додатків.

Курсова робота складається з трьох основних розділів, що охоплюють аналіз предметної області, формування вимог до ігрової логіки та детальний опис архітектури бекенд систем. Робота супроводжується комплексом технічних діаграм, що ілюструють архітектурні рішення та підтверджують практичну цінність розробленої системи.

.

**1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ**

1.1 Аналіз предметної галузі

У сучасному цифровому середовищі ігрова індустрія відіграє надзвичайно важливу роль як у сфері розваг, так і в розвитку технологій програмування, дизайну взаємодії з користувачем (UX/UI), штучного інтелекту та візуалізації даних. Зокрема, мобільні ігри займають значну частку на ринку цифрових розваг, завдяки доступності пристроїв, широкому спектру жанрів та можливості грати будь–де і будь–коли. За останнє десятиліття жанр roguelike отримав суттєве відродження завдяки своїй реіграбельності, виклику для гравців і простоті впровадження базових механік за наявності правильної архітектури [1][2]. Roguelike – це піджанр рольових ігор, що характеризується процедурною генерацією рівнів, постійною смертю персонажа (permadeath), стратегічним управлінням ресурсами та значною варіативністю геймплею.

Історично жанр roguelike виник ще в 1980–х роках, коли вийшла перша гра Rogue (див. рис. 1.1). Відтоді ідеї цієї гри були розвинуті в численних сучасних реалізаціях – таких як The Binding of Isaac (див. рис. 1.2), Dead Cells (див. рис. 1.3), Slay the Spire (див. рис. 1.4), Hades (див. рис. 1.5). Головна цінність цього жанру – постійна новизна досвіду, що забезпечується випадковими подіями, мінливими картами, різними ворогами та новими стратегіями[3]. Саме ці елементи формують міцний зв’язок між гравцем та ігровим світом, стимулюючи повторне проходження й удосконалення навичок [4][5]. Сучасні мобільні користувачі очікують швидкого, але насиченого досвіду, що дозволяє отримати повноцінне враження навіть за обмежений час (наприклад, під час поїздки або в перерві між справами) [6][7].

У зв’язку з цим, актуальним стає створення roguelike–гри саме для мобільної платформи. Цей формат дозволяє поєднати стислі гейм–сесії з глибоким геймплеєм, адаптованим під сенсорне керування. Розробка такого програмного забезпечення передбачає гнучку архітектуру, яка дозволить масштабувати гру, додавати нові рівні, ворогів, механіки, а також оновлювати інтерфейс без необхідності переписування основної логіки.

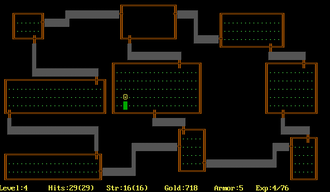


Рисунок 1.1 – Гра Rogue (виконано самостійно)

Зображення, що містить знімок екрана, Комп’ютерна гра, Програмне забезпечення для відеоігор, Цифрове компонування

Вміст, створений ШІ, може бути неправильним.

Рисунок 1.2 – Гра The Binding of Isaac (виконано самостійно)

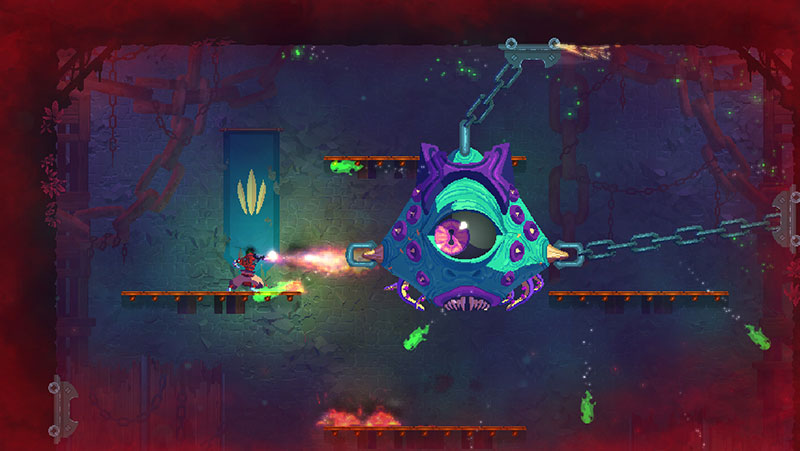


Рисунок 1.3 – Гра Dead Cells (виконано самостійно)



Рисунок 1.4 – Гра Slay the Spire (виконано самостійно)



Рисунок 1.4 – Гра Hades (виконано самостійно)

Особливу увагу в аналізі предметної області слід звернути на такі аспекти:

* принципи побудови roguelike–геймплею: процедурна генерація контенту, адаптивна складність, механіка постійної смерті, зміна стилю гри залежно від здобутих предметів чи умов;
* очікування користувачів мобільних ігор: простота входження, короткі гейм–сесії, швидке реагування інтерфейсу, інтуїтивне керування, візуальна привабливість [8][9];
* особливості реалізації бою на основі жестового керування (drag’n’drop), що є більш природним для сенсорних екранів і надає додатковий елемент тактичної взаємодії;
* вимоги до адаптивного інтерфейсу, який повинен коректно відображатися на різних екранах (з різною роздільною здатністю та співвідношенням сторін) та бути зрозумілим користувачу незалежно від його досвіду;
* можливість розширення гри: архітектура має передбачати модульність, що дозволить у майбутньому реалізувати багатокористувацький режим, магазин внутрішніх покупок або сюжетні розгалуження.

В умовах постійного розвитку ігрових платформ (Unity, Unreal Engine, Godot) особливого значення набуває правильний вибір рушія, який забезпечить баланс між продуктивністю, кросплатформеністю та швидкістю розробки. У межах цієї курсової роботи було обрано Unity як основу для реалізації програмного застосунку [10][11][12]. Це обґрунтовано широкою підтримкою мобільних платформ, багатим інструментарієм для створення 2D/3D–ігор, гнучкою системою подій та підтримкою C# як основної мови розробки.

Таким чином, аналіз предметної області підтверджує актуальність створення мобільної гри в жанрі roguelike, яка дозволить поєднати глибоку бойову систему з короткими сесіями, чітким візуальним стилем, інтуїтивним інтерфейсом та варіативним геймплеєм. Це відкриває широкі можливості для як технічної реалізації, так і креативного наповнення.

1.2 Виявлення проблем та актуалізація рішень

У сучасному цифровому середовищі ігрова індустрія, особливо у сфері мобільних платформ, зростає надзвичайно стрімко. Проте попри величезну кількість проєктів, що щороку з'являються на ринку, користувачі все частіше стикаються з однотипністю геймплею, низьким рівнем складності або навпаки — перевантаженістю інтерфейсом та механіками [13]. Це призводить до зниження рівня залученості, незавершення проходження ігор, негативного досвіду взаємодії з продуктом.

Особливої уваги потребують продукти жанру roguelike, який передбачає складність, варіативність проходження та наявність процедурної генерації або випадкових подій. На практиці ж значна кількість таких ігор має шаблонні алгоритми поведінки ворогів, передбачувані механіки бою або надто однотипні рівні, що зменшує рівень інтересу у повторних сесіях гри. Часто відсутність гнучкої системи бойових взаємодій або неякісно реалізована фізика снарядів (як–от метання предметів, скидання блоків тощо) не дозволяє гравцю повноцінно реалізувати тактичний підхід до гри.

Також актуальною проблемою залишається низький рівень оптимізації для мобільних пристроїв. Часто адаптація інтерфейсу не враховує особливості різних розмірів екранів, що призводить до погіршення UX, неможливості точного керування або поганої читабельності елементів інтерфейсу [14]. Відсутність налаштувань звуку, керування, а також зрозумілого відгуку на дії гравця (відсутність візуальних і звукових підказок) знижують рівень занурення у гру.

Ще однією важливою проблемою є недостатнє використання адаптивної логіки в межах ігрових сценаріїв: більшість мобільних ігор з фіксованими рівнями не мають можливості динамічно змінювати складність, що призводить до або занадто швидкого проходження, або до фрустрації гравця через високу складність без інструкцій і підтримки.

Таким чином, для забезпечення високої якості користувацького досвіду в мобільній roguelike–грі необхідно реалізувати:

* адаптивну ігрову логіку, яка забезпечує баланс між складністю й цікавістю;
* фізично обґрунтовану систему бойових механік, зокрема метання об’єктів, колізії та пошкодження;
* інтуїтивно зрозумілий, адаптивний інтерфейс, який однаково добре виглядає на різних розмірах екранів;
* логіку штучного інтелекту ворогів, яка базується на ситуаційних алгоритмах реагування, а не заздалегідь прописаних патернах;
* модульність у системі, що дозволяє в майбутньому додавати нові механіки, рівні або модифікації без суттєвих змін у базовій архітектурі гри.

У відповідь на ці виклики й була сформульована ідея створення гри Black Prince — динамічної мобільної гри в жанрі roguelike, що фокусується на інтуїтивному керуванні, цікавих бойових механіках із використанням предметів (цеглин), інтелектуальному супротивнику та гнучкому геймплейному циклі, здатному адаптуватися під різні стилі гри.

1.3 Постановка задачі

На основі проведеного аналізу предметної області та виявлення актуальних проблем у сфері мобільних ігор жанру roguelike, сформульовано задачу створення ігрового програмного застосунку з робочою назвою Black Prince. Головна мета розробки полягає у створенні повноцінного геймплейного досвіду, орієнтованого на швидкі, насичені бойові взаємодії з тактичними елементами, інтуїтивним керуванням і візуально зрозумілим інтерфейсом, який адаптується до мобільних пристроїв різних форм–факторів.

Для досягнення мети необхідно реалізувати такі завдання:

* спроектувати архітектуру ігрового програмного забезпечення, що підтримує модульну логіку та дозволяє масштабувати гру (додавати нових супротивників, рівні, механіки);
* розробити функціонал гравця, зокрема механіку метання предметів (цеглин), руху, взаємодії з оточенням, отримання шкоди та відображення стану здоров’я;
* створити систему керування ворогами з елементами штучного інтелекту, які здатні реагувати на дії гравця, ухилятись, атакувати та координувати дії відповідно до ситуації;
* реалізувати логіку бою через взаємодію цеглин із ворогами, включаючи розрахунок траєкторії, обробку зіткнень і завдання шкоди;
* забезпечити роботу UI–системи, яка містить головне меню, ігровий HUD (панель здоров’я, кількість предметів), екрани виграшу/програшу, меню паузи та налаштувань;
* розробити систему обробки подій, яка дозволяє централізовано керувати ключовими ігровими станами — початком рівня, перемогою, смертю, запуском анімацій, відтворенням звуку тощо;
* забезпечити адаптивність інтерфейсу до різних розмірів екранів і типів пристроїв (планшети, смартфони);
* провести тестування гри на предмет стабільності, коректності роботи механік, читабельності інтерфейсу та відгуку на дії користувача.

З огляду на поставлені завдання, гра Black Prince має поєднати в собі елементи бою, тактики та швидкого темпу з максимально зрозумілим керуванням та візуальним супроводом, що дозволяє досягти позитивного ігрового досвіду навіть під час коротких ігрових сесій. Основний акцент робиться на гнучкості ігрової логіки та інтеграції фізично реалістичних взаємодій між об’єктами (цеглинами, ворогами, гравцем). Це дозволить відрізнити гру від аналогічних рішень на ринку, зробивши її унікальною за рахунок оригінальних механік і якісного користувацького досвіду.

**2 ФОРМУВАННЯ ВИМОГ ДО ІГРОВОЇ ЛОГІКИ ТА БЕКЕНД СИСТЕМ**

2.1 Аналіз ігрових механік та системних вимог

Розробка бекенд архітектури мобільної гри Black Prince вимагає ретельного аналізу ігрових механік та формування комплексних системних вимог, що забезпечують стабільну та ефективну роботу всіх компонентів програмного забезпечення (див. рис. 2.1). Ігрова логіка базується на складній взаємодії кількох ключових підсистем, кожна з яких має специфічні технічні характеристики та вимоги до продуктивності.

Зображення, що містить текст, схема, ряд, План

Вміст, створений ШІ, може бути неправильним.

Рисунок 2.1 – Схема взаємодії основних ігрових підсистем

Центральною механікою гри є інноваційна система "брік–геймплею", де гравець використовує магічні цеглини як універсальний інструмент взаємодії з ігровим світом. Ця система вимагає точного математичного розрахунку траєкторій снарядів, урахування фізичних властивостей об'єктів та реалізації складних алгоритмів обробки колізій. Кожна цеглина має індивідуальні параметри швидкості, маси та коефіцієнта відскоку, що створює різноманітні тактичні можливості для гравця.

Система обробки фізичних взаємодій повинна забезпечувати реалістичну поведінку снарядів з урахуванням гравітаційних сил, аеродинамічного опору та механіки пружних зіткнень. Математична модель руху снарядів базується на класичних рівняннях кінематики з адаптацією до дискретного часового простору ігрового рушія [15]. Алгоритм розрахунку траєкторії включає предиктивне моделювання шляху снаряду для забезпечення точного прицілювання та візуального фідбеку гравцеві (див. рис. 2.2).

Процедурна генерація рівнів становить особливу технічну складність для бекенд системи, оскільки вимагає балансування між випадковістю та ігровою збалансованістю. Алгоритм генерації повинен створювати структурно коректні лабіринти, що забезпечують гарантовану можливість проходження від стартової точки до фінішу, при цьому зберігаючи достатній рівень складності та тактичного різноманіття.

Система використовує модифікований алгоритм клітинкового автомата з додатковими правилами валідації для створення когерентної архітектури рівня. Початкова генерація виконується через рекурсивний алгоритм поширення з контрольованими ймовірнісними параметрами, після чого застосовуються фільтри згладжування та структурної оптимізації для забезпечення якісного ігрового досвіду.

Зображення, що містить текст, схема, знімок екрана, План

Вміст, створений ШІ, може бути неправильним.

Рисунок 2.2 – Діаграма розрахунку траєкторії снарядів

Штучний інтелект ворогів реалізовано як багаторівневу систему, що поєднує базові поведінкові патерни з адаптивними реакціями на дії гравця. AI–архітектура базується на конечних автоматах станів з можливістю динамічного переключення між різними режимами поведінки залежно від ігрової ситуації. Система включає алгоритми пошуку шляху, патрулювання території та реагування на загрози.

Основні поведінкові модулі включають систему навігації по ігровому полю, механізми виявлення та уникнення перешкод, а також логіку взаємодії з ігровими об'єктами. Кожен тип ворога має унікальні параметри швидкості руху, радіуса виявлення загроз та алгоритми прийняття тактичних рішень, що створює різноманітність в ігровому процесі.

Система керування станами гри координує взаємодію між усіма підсистемами через централізований диспетчер подій, що забезпечує синхронізацію операцій та підтримку консистентності ігрового стану (див. рис. 2.3). Архітектура включає механізми обробки асинхронних подій, черги команд та системи відновлення після помилок.

Зображення, що містить текст, схема, План, знімок екрана

Вміст, створений ШІ, може бути неправильним.

Рисунок 2.3 – Архітектура системи керування станами

2.2 Функціональні вимоги до ігрової логіки

Функціональні вимоги до бекенд системи визначають конкретні технічні можливості та алгоритмічну поведінку, які повинна забезпечувати ігрова логіка для створення повноцінного та збалансованого ігрового досвіду. Ці вимоги формулюються на основі детального аналізу ігрових сценаріїв та очікувань користувачів.

2.2.1 Вимоги до системи процедурної генерації карт

Система процедурної генерації карт повинна створювати ігрові рівні фіксованого розміру 17х11 ігрових клітинок з математично гарантованою можливістю проходження від визначеної початкової позиції до заданої фінішної точки. Алгоритм генерації використовує детермінований лінійний конгруентний генератор псевдовипадкових чисел з початковим seed–значенням, отриманим з поточного системного часу.

Функція генерації повинна забезпечувати створення структурно збалансованих карт через застосування рекурсивного алгоритму поширення блоків з контрольованою ймовірністю розгалуження 35% для кожного напрямку експансії. Система включає спеціальні обмеження для запобігання генерації блоків у критичних зонах, включаючи стартові позиції гравця та області виходу з рівня.

Алгоритм валідації згенерованих карт реалізований через модифікований алгоритм пошуку в глибину, що перевіряє існування принаймні одного валідного шляху від початкової координати (9, 15) до цільової позиції (9, 1). Система автоматично регенерує карту у випадку виявлення непрохідної конфігурації, забезпечуючи гарантовану ігровість кожного створеного рівня (див. рис. 2.4).

Зображення, що містить текст, схема, ряд, оригамі

Вміст, створений ШІ, може бути неправильним.

Рисунок 2.4 – Алгоритм валідації прохідності карти

2.2.2 Вимоги до фізичного рушія та системи взаємодій

Фізичний рушій повинна обробляти взаємодію між усіма категоріями ігрових об'єктів у режимі реального часу з частотою оновлення не менше 50 герц для забезпечення плавності анімацій та точності колізій. Система колізій має виявляти та коректно обробляти зіткнення між гравцем, ворожими сутностями, снарядами та статичними елементами ігрового середовища (див. рис. 2.5).

Снаряди повинні демонструвати фізично коректну траєкторію з урахуванням гравітаційного прискорення 9.81 м/с², можливістю пружних відскоків від твердих поверхонь з коефіцієнтом реституції 0.7, та поступовою втратою кінетичної енергії через опір середовища. Система повинна підтримувати одночасне існування до 10 активних снарядів без критичної деградації продуктивності.

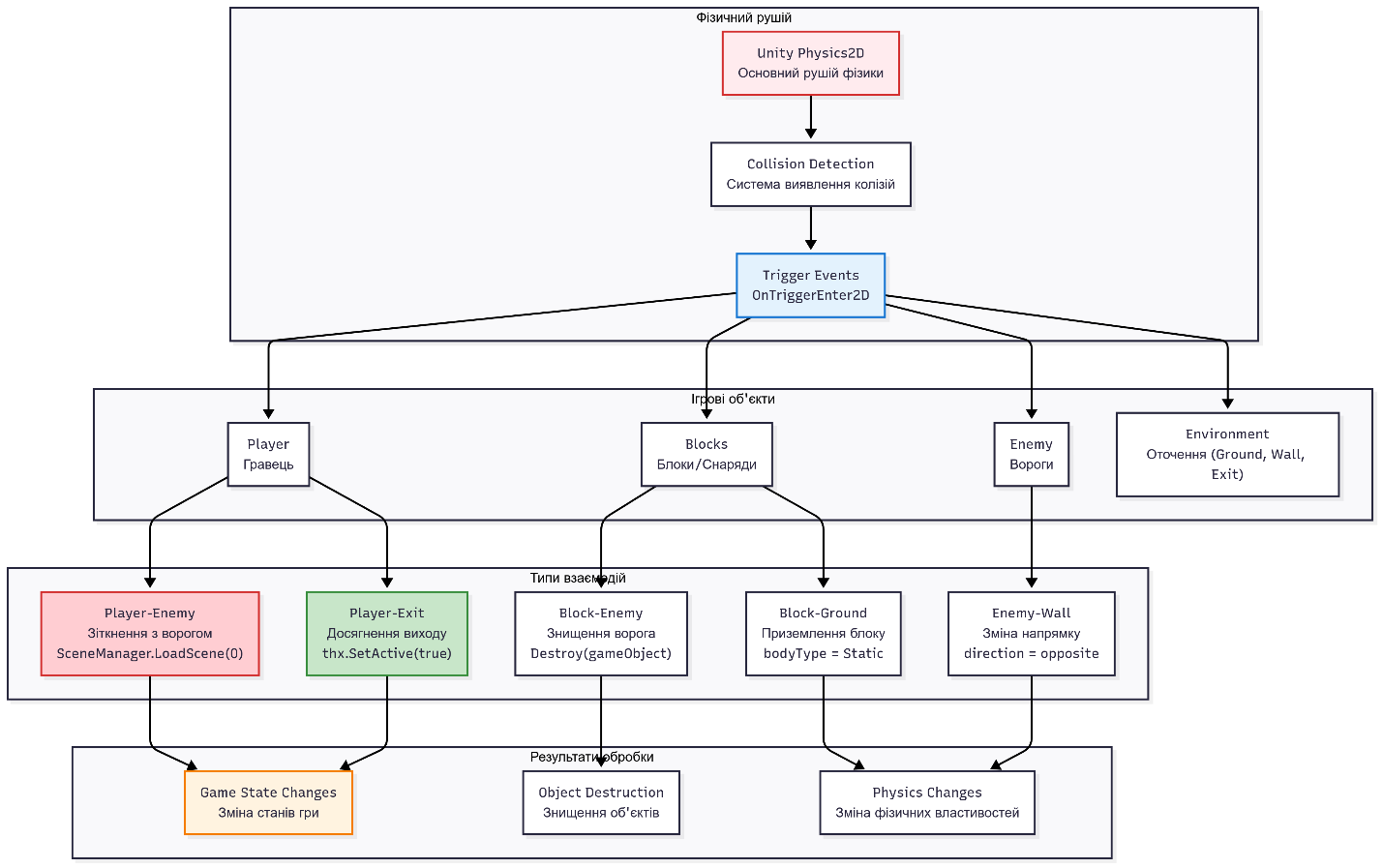


Рисунок 2.5 – Діаграма обробки фізичних взаємодій

Механізм обробки колізій реалізований через систему тригерних зон та фізичних колайдерів з автоматичною категоризацією типів зіткнень на основі ігрових тегів. Кожен тип взаємодії має специфічну логіку обробки: зіткнення гравця з ворогами ініціює перезавантаження ігрової сцени, взаємодія з фінішною зоною активує стан перемоги, а попадання снарядів по ворогам призводить до їх знищення та нарахування балів.

2.2.3 Вимоги до системи штучного інтелекту

Система штучного інтелекту ворогів повинна реалізовувати базову поведінкову модель автономного патрулювання з автоматичною зміною напрямку руху при виявленні перешкод або досягненні границь дозволеної території. AI–компонент має бути архітектурно масштабованим для інтеграції додаткових поведінкових патернів без структурних змін базового коду.

Алгоритм руху ворогів базується на простій моделі конечного автомата з двома основними станами: рух ліворуч та рух праворуч. Система використовує глобальну синхронізацію напрямку руху для всіх ворожих сутностей, що створює координовану групову поведінку та підвищує передбачуваність ігрового процесу для гравця.

Вороги повинні демонструвати миттєву реакцію на попадання снарядів з автоматичним видаленням з ігрової сцени та активацією відповідних візуальних та звукових ефектів. Система AI має підтримувати незалежне функціонування до 15 ворожих сутностей одночасно без суттєвого впливу на загальну продуктивність програми.

2.2.4 Вимоги до системи керування ігровими об'єктами

Система керування блоками повинна забезпечувати динамічне створення та точне позиціонування снарядів відповідно до координат активного гравця та напрямку прицілювання. Кожен створений блок має мати індивідуальні фізичні властивості та здатність переходу між різними функціональними станами: очікування команди, активний політ, процес приземлення та інтеграція у статичну структуру рівня (див. рис. 2.6).

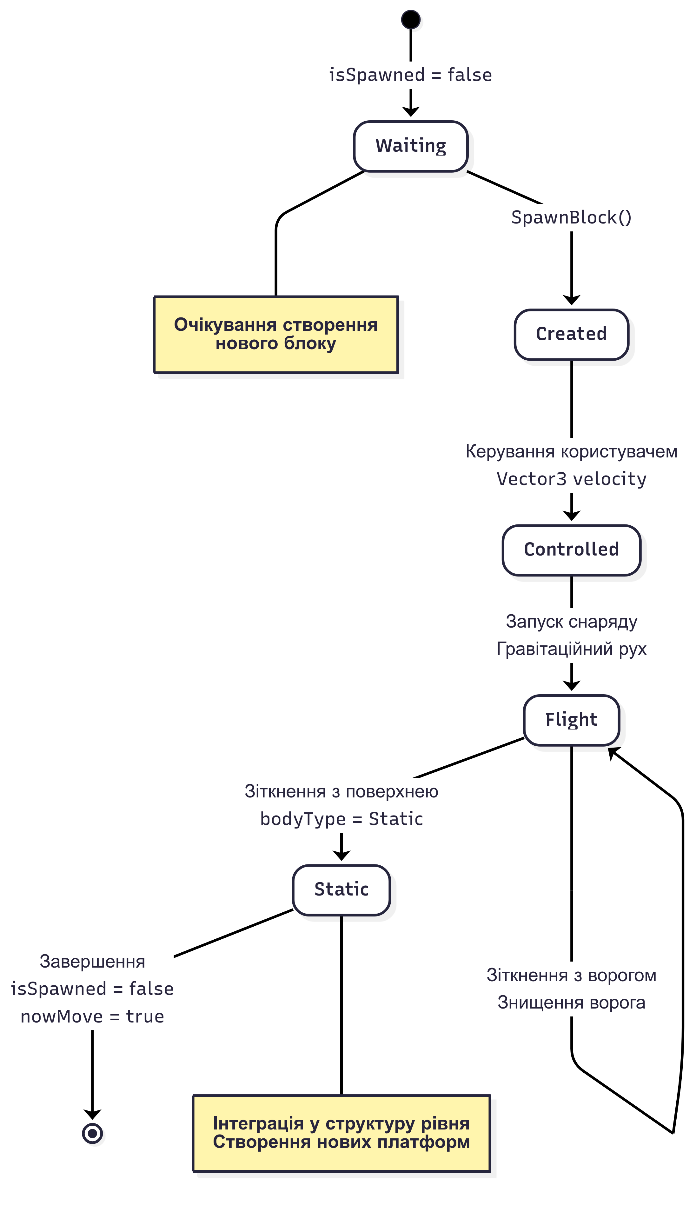


Рисунок 2.6 – Життєвий цикл ігрового блоку

Система повинна підтримувати обмеження кількості одночасно активних снарядів через реалізацію механізму черговості та автоматичного управління ресурсами. Створення нового снаряду блокується до моменту завершення життєвого циклу попереднього об'єкта, що забезпечує контрольоване навантаження на систему та стратегічну глибину ігрового процесу.

Блоки після приземлення повинні автоматично інтегруватися у статичну геометрію рівня, створюючи нові платформи для переміщення гравця та змінюючи тактичні можливості проходження. Система інтеграції включає автоматичну зміну фізичних властивостей об'єкта з динамічного на статичний тип та оновлення внутрішніх карт навігації для AI–системи ворогів.

2.3 Нефункціональні вимоги та обмеження системи

Нефункціональні вимоги визначають якісні та експлуатаційні характеристики бекенд системи, що безпосередньо впливають на користувацький досвід, стабільність роботи програми та можливості її подальшого розвитку. Ці вимоги формують технічний фундамент для забезпечення конкурентоспроможності продукту на мобільному ринку.

2.3.1 Вимоги до продуктивності та оптимізації

Система повинна забезпечувати стабільну частоту відображення кадрів не менше 30 FPS на мобільних пристроях середнього цінового сегменту з процесорами ARM Cortex–A53 та обсягом оперативної пам'яті від 3 ГБ. Процес генерації нового ігрового рівня не повинен перевищувати 2 секунди навіть у випадку необхідності виконання множинних ітерацій для досягнення прохідної конфігурації карти.

Обробка фізичних взаємодій має бути оптимізована для мінімізації навантаження на центральний процесор через використання ефективних алгоритмів просторового розбиття та селективної обробки колізій лише для активних ігрових об'єктів. Система повинна автоматично масштабувати рівень деталізації фізичних розрахунків залежно від поточного навантаження для підтримання стабільної продуктивності.

Споживання енергії батареї не повинно перевищувати 15% заряду за 30 хвилин активної гри на стандартних мобільних пристроях з батареєю ємністю 3000 мАг. Система включає спеціальні механізми енергооптимізації, включаючи динамічне зменшення частоти оновлення фонових процесів та інтелектуальне управління яскравістю екрану.

2.3.2 Вимоги до масштабованості та архітектурної гнучкості

Архітектура системи повинна забезпечувати можливість безшовного додавання нових типів ворогів, ігрових механік та візуальних ефектів без необхідності структурних змін у базовому коді програми (див. рис. 2.7). Модульна організація компонентів має підтримувати принцип слабкого зв'язку між підсистемами через добре визначені програмні інтерфейси.

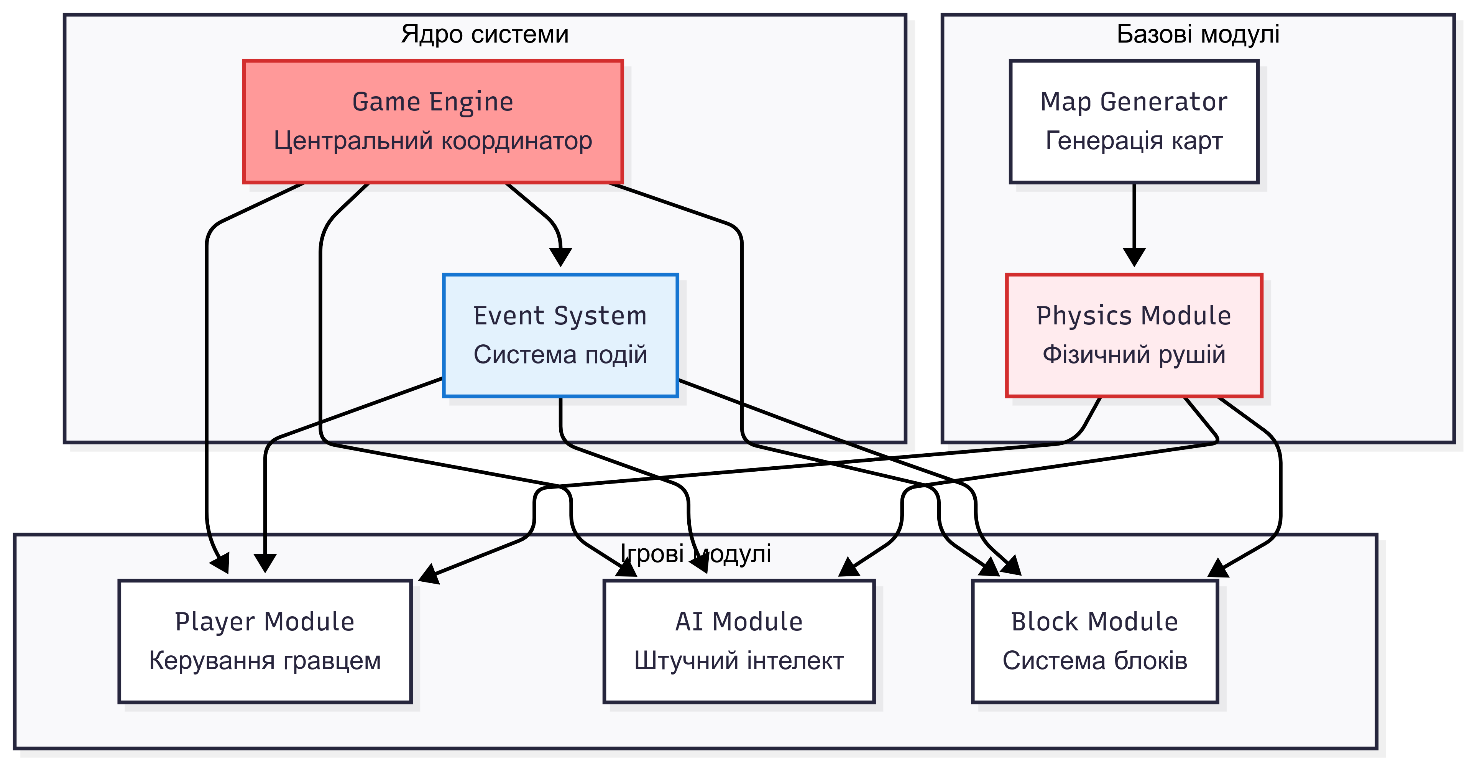


Рисунок 2.7 – Діаграма модульної архітектури системи

Система повинна підтримувати горизонтальне масштабування складності ігрових рівнів від простих навчальних карт до складних багаторівневих лабіринтів без зміни алгоритмічної основи генератора. Параметрична конфігурація системи генерації має дозволяти налаштування розміру карт, щільності перешкод та складності навігації через зовнішні конфігураційні файли.

Кодова база має бути організована відповідно до принципів об'єктно–орієнтованого проектування з чіткою ієрархією наслідування, інкапсуляцією функціональності та поліморфним інтерфейсом для взаємодії між компонентами. Архітектура повинна підтримувати патерни проектування, такі як Observer для системи подій, Factory для створення ігрових об'єктів та Singleton для глобального управління станом.

2.3.3 Вимоги до стабільності та надійності

Система повинна демонструвати високий рівень стабільності роботи з коректною обробкою всіх можливих граничних випадків та нестандартних ігрових ситуацій без аварійного завершення програми. Алгоритми генерації карт мають включати вбудовані механізми запобігання нескінченним циклам з гарантованим створенням валідного рівня за максимум 100 ітерацій спроб.

Система збереження стану гри повинна забезпечувати цілісність даних через використання атомарних операцій запису та механізмів резервного копіювання. Локальне зберігання має включати перевірку контрольних сум для виявлення пошкоджених файлів збереження та автоматичне відновлення з резервних копій у випадку виявлення корупції даних.

Обробка користувацького вводу повинна включати валідацію та фільтрацію некоректних команд для запобігання експлуатації помилок програми. Система має коректно обробляти швидкі послідовності дотиків, одночасні мультитач жести та переривання ігрового процесу системними повідомленнями без втрати ігрового прогресу.

2.3.4 Вимоги до ефективності використання ресурсів

Ефективність використання оперативної пам'яті вимагає реалізації системи об'єктного пулінгу для часто створюваних та знищуваних ігрових елементів, включаючи снаряди, візуальні ефекти та тимчасові UI–компоненти. Максимальне споживання пам'яті не повинно перевищувати 150 МБ для основного ігрового процесу з додатковими 50 МБ для буферизації аудіовізуальних ресурсів (див. рис. 2.8).

Зображення, що містить текст, схема, План, ряд

Вміст, створений ШІ, може бути неправильним.

Рисунок 2.8 – Схема управління пам'яттю та об'єктним пулінгом

Система повинна автоматично звільняти неактивні ресурси через механізм збирання сміття з періодичністю не більше 5 секунд для запобігання фрагментації пам'яті. Критичні ігрові об'єкти мають бути захищені від передчасного видалення через систему посилань та життєвих циклів.

Детермінованість поведінки системи забезпечується використанням контрольованих генераторів псевдовипадкових чисел з можливістю збереження та відновлення внутрішнього стану для точного відтворення ігрових ситуацій. Ця функціональність критично важлива для процесів тестування, налагодження програми та забезпечення консистентності поведінки між різними сеансами гри.

**3 АРХІТЕКТУРА ІГРОВОЇ ЛОГІКИ ТА БЕКЕНД СИСТЕМ**

3.1 Проєктування основних ігрових систем

Архітектурне проєктування бекенд системи гри Black Prince базується на принципах модульної організації з акцентом на високу когезію компонентів та слабкий зв'язок між підсистемами. Центральною концепцією архітектури є створення ієрархічної структури ігрових компонентів з чітко визначеними інтерфейсами взаємодії та делегуванням відповідальності між спеціалізованими модулями (див. рис. 3.1).

Фундаментальною основою архітектури є компонентна модель Unity MonoBehaviour, яка забезпечує створення автономних ігрових сутностей з інкапсульованою логікою та стандартизованими методами життєвого циклу. Кожен ігровий компонент успадковується від базового класу MonoBehaviour та реалізує специфічну функціональність через систему callback–методів Start, Update, FixedUpdate та OnDestroy.

Зображення, що містить текст, схема, План, Креслення

Вміст, створений ШІ, може бути неправильним.

Рисунок 3.1 – Загальна архітектурна схема бекенд системи

Система координації подій реалізована через централізований диспетчер, що забезпечує асинхронну комунікацію між незалежними модулями без створення прямих залежностей. Архітектура подій базується на патерні Observer з підтримкою типізованих повідомлень та пріоритетної черги обробки для критичних ігрових операцій [16].

3.1.1 Архітектура системи керування об'єктами

Система керування ігровими об'єктами організована навколо спеціалізованих менеджерів, кожен з яких відповідає за конкретний тип сутностей та їх життєвий цикл. BlockManager виступає як центральний координатор для створення, позиціонування та управління снарядами, забезпечуючи синхронізацію їх поведінки з діями гравця та станом ігрового середовища (див. рис. 3.2).

Архітектура менеджера блоків включає систему статичного контролю стану через змінну isSpawned, що реалізує паттерн Singleton для запобігання одночасному існуванню множинних активних снарядів. Цей підхід забезпечує контрольоване споживання системних ресурсів та створює стратегічну глибину ігрового процесу через обмеження частоти атак.

Зображення, що містить текст, схема, План, Креслення

Вміст, створений ШІ, може бути неправильним.

Рисунок 3.2 – UML–діаграма класів системи управління об'єктами

Система позиціонування використовує трьохвимірну векторну математику для точного розміщення елементів у просторі з урахуванням координат гравця та напрямку прицілювання. Компонент NewBehaviourScript зберігає поточну позицію гравця у статичній змінній playerPos типу Vector4, що забезпечує глобальний доступ до координатних даних для розрахунку траєкторій та взаємодій.

3.1.2 Фізична підсистема та система колізій

Фізична підсистема побудована на основі двовимірного фізичного рушія Unity Physics2D з розширеннями для підтримки специфічних ігрових механік. Архітектура включає ієрархію фізичних компонентів, включаючи Rigidbody2D для динамічних об'єктів, різноманітні типи Collider2D для геометричного представлення та спеціалізовані Joint2D для зв'язків між об'єктами (див. рис. 3.3).

Зображення, що містить текст, схема, знімок екрана, План

Вміст, створений ШІ, може бути неправильним.Рисунок 3.3 – Схема фізичної підсистеми

Система обробки колізій реалізована через комбінацію тригерних подій OnTriggerEnter2D для виявлення взаємодій та фізичних колізій OnCollisionEnter2D для реалістичної механіки зіткнень. Кожен фізичний об'єкт має налаштовані параметри маси, лінійного та кутового опору, коефіцієнтів тертя та пружності для забезпечення природної поведінки.

Архітектура передбачає багаторівневу систему фільтрації колізій через матрицю взаємодії шарів (Layer Collision Matrix), що дозволяє селективно обробляти зіткнення між різними категоріями об'єктів. Це забезпечує оптимізацію продуктивності через виключення непотрібних розрахунків та створює можливості для реалізації складних ігрових механік.

3.1.3 Система управління станами гри

Управління станами гри реалізовано через архітектуру конечного автомата з підтримкою ієрархічних станів та умовних переходів (див. рис. 3.4). Основні стани включають ініціалізацію гри, активний геймплей, паузу, завершення рівня та глобальні налаштування. Кожен стан має власні методи входу, виконання та виходу з можливістю передачі контексту між переходами.

Статичне управління станами реалізоване через глобальні змінні, такі як nowMove у класі NewBehaviourScript, що координує переключення між різними режимами взаємодії користувача. Змінна nowMove контролює чи знаходиться гравець у режимі активного руху, чи у режимі прицілювання та запуску блоків, забезпечуючи взаємовиключну поведінку цих станів.

Зображення, що містить схема, текст, ескіз, План

Вміст, створений ШІ, може бути неправильним.

Рисунок 3.4 – Діаграма станів ігрової системи

Архітектура включає систему збереження та відновлення стану через серіалізацію критичних параметрів у локальні файли. Система автоматично створює резервні копії ключових даних та забезпечує відновлення у випадку некоректного завершення програми або системних збоїв.

3.2 Архітектура системи штучного інтелекту

Система штучного інтелекту ворогів представляє собою модульну архітектуру, що поєднує простоту реалізації з ефективністю виконання та можливостями розширення (див. рис. 3.5).

Зображення, що містить текст, знімок екрана, схема, План

Вміст, створений ШІ, може бути неправильним.

Рисунок 3.5 – Архітектурна схема AI–системи

Базова концепція AI будується навколо автономних агентів з індивідуальною поведінковою логікою, але координованою через глобальні параметри для створення узгодженого ігрового досвіду.

3.2.1 Базова архітектура AI–агентів

Фундаментальною основою AI–системи є базовий клас Enemy, що інкапсулює загальні характеристики та поведінку ворожих сутностей. Клас реалізує основні механізми взаємодії з ігровим середовищем, включаючи обробку колізій з ігровими об'єктами та автоматичне знищення при зіткненні з блоками гравця.

Архітектура передбачає спадкування функціональності через спеціалізований клас EnemyAI, що розширює базову поведінку додаванням інтелектуальних механізмів руху, навігації та реагування на зміни ігрового середовища (див. рис. 3.6). Цей підхід забезпечує модульність системи та можливість створення різних типів ворогів з унікальною поведінкою.

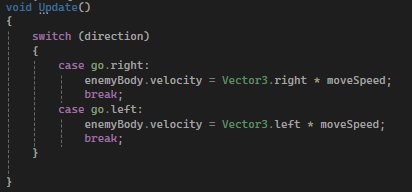


Рисунок 3.6 – Базова структура AI–компонентів

Система використовує енумерацію для визначення можливих напрямків руху, що забезпечує типобезпечність коду та спрощує процес налагодження. Enum–структура go включає основні стани left та right, що відповідають горизонтальному руху ворогів по ігровому полю.

3.2.2 Система поведінкових патернів

Архітектура поведінки ворогів базується на простому, але ефективному алгоритмі патрулювання з автоматичною зміною напрямку при зустрічі з перешкодами. Глобальна статична змінна direction забезпечує синхронізований рух усіх ворожих агентів, створюючи ефект координованої групової поведінки.

Алгоритм руху реалізований через безперервне застосування векторної швидкості до фізичного компонента Rigidbody2D у методі Update. Система використовує вбудовані константи Vector3.right та Vector3.left для визначення напрямку руху з контрольованою швидкістю moveSpeed, що забезпечує плавну та передбачувану анімацію переміщення.

Механізм реагування на перешкоди працює через систему тригерних подій OnTriggerEnter2D з автоматичною ідентифікацією типу зіткнення через систему ігрових тегів. При виявленні об'єкта з тегом "Wall" система виконує інверсію поточного стану руху через switch–конструкцію, що забезпечує миттєву зміну напрямку без додаткових обчислень.

3.2.3 Система координації та синхронізації

Координація поведінки множинних AI–агентів реалізована через централізовану систему управління з використанням статичних змінних для глобального стану (див. рис. 3.7). Цей підхід забезпечує ефективну синхронізацію поведінки всіх ворожих сутностей без необхідності прямої комунікації між індивідуальними агентами.

Зображення, що містить текст, знімок екрана, Шрифт

Вміст, створений ШІ, може бути неправильним.

Рисунок 3.7 – Система координації AI–агентів

Архітектура включає можливості розширення для додавання нових поведінкових режимів через модифікацію enum–структури та відповідних switch–блоків. Система спроектована таким чином, що додавання нових станів поведінки не вимагає зміни базової логіки існуючих агентів.

Система передбачає можливість індивідуалізації поведінки через параметричні налаштування швидкості руху, радіуса виявлення перешкод та ймовірнісні характеристики прийняття рішень. Ці параметри можуть бути налаштовані для кожного типу ворога окремо, створюючи різноманітність у ігровому процесі.

3.3 Система процедурної генерації контенту

Система процедурної генерації карт представляє найтехнічно складний компонент бекенд архітектури, що відповідає за створення унікальних ігрових рівнів з гарантованою прохідністю та збалансованою складністю. Архітектура генератора поєднує математичні алгоритми, евристичні методи та валідаційні процедури для забезпечення високої якості згенерованого контенту.

3.3.1 Математична основа генерації

Фундаментальною основою системи генерації є двовимірний булевський масив розміром 17х11 елементів, що представляє дискретну модель ігрового простору. Кожна клітинка масиву кодує наявність або відсутність блоку у відповідній просторовій позиції, створюючи бінарну карту рівня з можливістю ефективної обробки та аналізу.

Система ініціалізації масиву використовує вкладені цикли для створення порожньої карти з подальшим заповненням відповідно до алгоритму процедурної генерації. Початковий стан усіх елементів встановлюється у значення false, що відповідає відсутності блоків та створює базову структуру для подальшої модифікації (див. рис. 3.8).

Зображення, що містить Комп’ютерна гра, знімок екрана, мультфільм, Програмне забезпечення для відеоігор

Вміст, створений ШІ, може бути неправильним.

Рисунок 3.8 – Приклад згенерованого рівня

Генератор псевдовипадкових чисел реалізований як лінійний конгруентний генератор з математичною формулою seed = (seed \* 1664525 + 1013904223) & 0x7FFFFFFF. Вибір коефіцієнтів базується на рекомендаціях Numerical Recipes для забезпечення максимального періоду генерації та рівномірного розподілу псевдовипадкових значень.

3.3.2 Алгоритм процедурної генерації

Основний алгоритм генерації карт базується на рекурсивному алгоритмі поширення блоків з контрольованою ймовірністю розгалуження. Функція prop ініціюється з випадкової стартової позиції та рекурсивно поширюється у чотирьох кардинальних напрямках з ймовірністю 35% для кожного кроку експансії.

Алгоритм включає спеціальні обмеження для запобігання генерації блоків у критичних зонах, включаючи стартові позиції гравця та області виходу з рівня. Умовна перевірка !((x\_start == 16 || x\_start == 1) && y\_start == 9) && !((x\_start == 16 || x\_start == 1) && y\_start == 8) забезпечує збереження проходів у ключових областях карти.

Система використовує тричі повторний виклик функції поширення з різними стартовими точками для створення багатоцентрової структури рівня. (див. рис. 3.9).

Зображення, що містить текст, знімок екрана, програмне забезпечення

Вміст, створений ШІ, може бути неправильним.

Рисунок 3.9 – Основний алгоритм генерації

Цей підхід забезпечує достатню щільність блоків для створення цікавих тактичних викликів, але зберігає відкриті простори для руху гравця.

3.3.3 Система валідації та гарантії прохідності

Критично важливим компонентом архітектури є система валідації згенерованих карт, що забезпечує математичну гарантію існування принаймні одного валідного шляху від початкової позиції до фінішної точки. Алгоритм валідації базується на модифікованому методі пошуку в глибину з оптимізаціями для ігрового контексту.

Функція Check\_Map\_Posibility реалізує рекурсивний обхід доступних клітинок, починаючи з координати (9, 15) та шукаючи шлях до цільової позиції (9, 1). Алгоритм використовує техніку маркування відвіданих клітинок для запобігання зацикленню та забезпечення ефективного пошуку.

Система включає оптимізації для швидкого виявлення безвихідних ситуацій через аналіз локальної топології навколо поточної позиції. Алгоритм перевіряє доступність сусідніх клітинок у всіх чотирьох напрямках та припиняє пошук у разі виявлення повністю заблокованих областей (див. рис. 3.10).

Зображення, що містить текст, знімок екрана, програмне забезпечення

Вміст, створений ШІ, може бути неправильним.

Рисунок 3.10 – Алгоритм валідації прохідності карти

Архітектура передбачає механізм повторної генерації у випадку створення непрохідної конфігурації карти. Основний цикл while(true) забезпечує продовження процесу генерації до створення валідного рівня, з внутрішніми обмеженнями для запобігання нескінченним циклам у критичних ситуаціях.

3.4 Система фізики та взаємодій

Фізична підсистема гри представляє комплексну архітектуру, що інтегрує стандартні можливості Unity Physics2D з спеціалізованими розширеннями для підтримки унікальних ігрових механік. Система забезпечує реалістичну поведінку всіх ігрових об'єктів через точне моделювання фізичних законів та ефективну обробку складних взаємодій.

3.4.1 Архітектура системи керування гравцем

Система керування персонажем реалізована через компонент NewBehaviourScript, що служить центральним інтерфейсом між користувацьким вводом та фізичними діями персонажа. Архітектура компонента базується на безпосередній маніпуляції фізичними властивостями через компонент Rigidbody2D з використанням різноманітних режимів застосування сили.

Механізм стрибка реалізований через імпульсне застосування вертикальної сили за допомогою методу AddForce з параметром ForceMode2D.Impulse. Цей підхід забезпечує миттєву зміну швидкості персонажа з реалістичною траєкторією польоту під впливом гравітації. Сила стрибка контролюється параметром jumpForce, що дозволяє точне налаштування висоти та тривалості стрибка (див. рис. 3.11).

Зображення, що містить текст, знімок екрана, програмне забезпечення

Вміст, створений ШІ, може бути неправильним.

Рисунок 3.11 – Система керування персонажем

Горизонтальний рух персонажа реалізований через постійне застосування сили у відповідному напрямку з використанням стандартного режиму ForceMode2D.Force. Цей метод створює поступове прискорення персонажа з природною інерцією та можливістю точного контролю швидкості через параметр moveSpeed.

3.4.2 Система виявлення контакту з поверхнею

Критично важливим компонентом фізичної системи є механізм виявлення контакту персонажа з твердими поверхнями для забезпечення реалістичної поведінки стрибків. Система використовує булевську змінну grounded для відстеження поточного стану персонажа та запобігання стрибкам у повітрі.

Алгоритм виявлення контакту базується на обробці тригерних подій OnTriggerEnter2D та OnTriggerExit2D з автоматичною ідентифікацією типу поверхні через систему ігрових тегів. Система розпізнає об'єкти з тегами "ground" та "block" як валідні поверхні для стрибків, забезпечуючи гнучкість у дизайні рівнів.

Механізм включає захист від помилкових спрацьовувань через перевірку відповідності тегів та фільтрацію непотрібних колізій(див. рис. 3.12).

Зображення, що містить текст, Шрифт, знімок екрана

Вміст, створений ШІ, може бути неправильним.

Рисунок 3.12 – Алгоритм виявлення контакту

Система автоматично скидає стан grounded при втраті контакту з поверхнею, забезпечуючи точну поведінку персонажа в різних ігрових ситуаціях.

3.4.3 Фізична модель снарядів та блоків

Система фізики блоків реалізована через клас Blocks, що керує поведінкою снарядів протягом їх повного життєвого циклу від запуску до інтеграції у статичну структуру рівня. Архітектура компонента включає підтримку як керованого руху під час прицілювання, так і автономного польоту після запуску.

Фізична модель снарядів базується на стандартному компоненті Rigidbody2D з налаштованими параметрами маси, лінійного опору та гравітаційного масштабу для створення реалістичної траєкторії польоту. Система підтримує динамічну зміну фізичних властивостей залежно від поточного стану блоку.

Механізм керованого руху реалізований через пряме встановлення швидкості Rigidbody2D відповідно до напрямку користувацького вводу. Система розпізнає сенсорні команди у визначених областях екрану та перетворює їх у відповідні вектори швидкості для горизонтального та вертикального руху.

Система автоматичного приземлення включає механізм зміни типу фізичного тіла з Dynamic на Static при зіткненні з поверхнею. Цей процес інтегрує снаряд у статичну геометрію рівня, створюючи нові платформи для переміщення гравця та змінюючи тактичні можливості проходження.

3.4.4 Система обробки колізій та взаємодій

Архітектура системи колізій забезпечує комплексну обробку взаємодій між усіма категоріями ігрових об'єктів через централізовану систему подій та спеціалізовані обробники для різних типів зіткнень. Система використовує комбінацію тригерних зон та фізичних колайдерів для максимальної гнучкості та продуктивності.

Кожен тип ігрового об'єкта має унікальний ідентифікаційний тег, що дозволяє системі автоматично класифікувати зіткнення та застосовувати відповідну логіку обробки. Основні категорії включають "Player", "Enemy", "Block", "ground", "Wall", "Exit" з індивідуальними алгоритмами реагування.

Система включає спеціалізовані обробники для критичних ігрових подій, включаючи зіткнення гравця з ворогами (перезавантаження сцени), взаємодію з фінішною зоною (активація стану перемоги) та попадання снарядів по ворогам (знищення та нарахування балів).

Архітектура передбачає можливості розширення для додавання нових типів взаємодій без структурних змін базового коду. Модульна організація обробників колізій дозволяє легко інтегрувати додаткові ігрові механіки та спеціальні ефекти для підвищення ігрового досвіду.

**4 ОПИС ПРИЙНЯТИХ ПРОГРАМНИХ РІШЕНЬ**

У процесі реалізації серверної логіки та ігрових механік мобільної гри Black Prince було розроблено систему взаємопов'язаних модулів, що забезпечують повноцінне функціонування ігрового процесу. Кожен компонент відповідає за окремий аспект логіки: генерацію рівнів, управління ігровими об'єктами, штучний інтелект противників та фізичну взаємодію елементів гри.

Центральним елементом архітектури є модуль Map\_Generation.cs, який реалізує процедурну генерацію ігрових рівнів. Система створює унікальні лабіринти для кожного проходження, що забезпечує високу реіграбельність продукту. Алгоритм базується на рекурсивному поширенні блоків з використанням власного генератора псевдовипадкових чисел.

csharp

void prop(List<List<bool>> map, int x\_start, int y\_start)

{

if (map[y\_start][x\_start] == true) return;

map[y\_start][x\_start] = true;

if (x\_start > 0 && Rand(0, 99) < 35)

prop(map, x\_start - 1, y\_start);

*// Аналогічно для інших напрямків*

}

Генератор використовує лінійний конгруентний метод для забезпечення контрольованої випадковості, що дозволяє створювати збалансовані рівні без надмірної складності або простоти. Система валідації Check\_Map\_Posibility перевіряє проходимість згенерованого лабіринту, гарантуючи можливість досягнення цілі з початкової позиції.

Модуль NewBehaviourScript (Player\_move.cs) реалізує повний цикл управління персонажем, включаючи рух, стрибки та взаємодію з сенсорним інтерфейсом. Логіка побудована на основі системи зон дотику, що дозволяє точно відстежувати наміри гравця.

csharp

Vector3 touchPos = Camera.main.ScreenToWorldPoint(Input.mousePosition);

if (touchPos.x < -6.55 + 0.7 && touchPos.x > -6.55 - 0.7 &&

touchPos.y < -3.1 + 0.4 && touchPos.y > -3.1 - 0.4 && grounded)

{

playerBody.AddForce(Vector2.up \* jumpForce, ForceMode2D.Impulse);

}

Система використовує статичні змінні для координації між різними модулями, зокрема nowMove для перемикання між режимами руху гравця та кидання блоків, а також playerPos для збереження позиції під час переходу в режим атаки.

Компонент EnemyAI.cs реалізує базову поведінкову модель ворогів на основі машини станів. Противники патрулюють території, змінюючи напрямок руху при зіткненні з перешкодами, що створює передбачувану, але динамічну загрозу для гравця.

csharp

public enum go { left, right }

public static go direction = go.left;

private void OnTriggerEnter2D(Collider2D collision)

{

if (collision.CompareTag("Wall"))

{

direction = (direction == go.right) ? go.left : go.right;

}

}

Використання статичної змінної direction забезпечує синхронізовану поведінку всіх екземплярів ворогів, створюючи координований рух, який ускладнює прогнозування їх позицій гравцем.

Модуль Blocks.cs керує поведінкою магічних цеглин, що є основною механікою гри. Система поєднує фізичне моделювання з логікою управління, дозволяючи гравцю точно спрямовувати снаряди за допомогою сенсорного введення.

csharp

if (Input.GetMouseButton(0) && !NewBehaviourScript.nowMove)

{

playerBody.velocity = Vector3.down \* moveSpeed;

if (touchPos.x < -5.7 + 0.7 && touchPos.x > -5.7 - 0.7)

playerBody.velocity = Vector3.right \* moveSpeed;

}

BlockManager.cs координує створення нових блоків, використовуючи систему позицій спавну та стану isSpawned для запобігання одночасному існуванню кількох активних снарядів.

Система колізій та взаємодій

Обробка зіткнень реалізована через систему тегів Unity, що забезпечує чітку ідентифікацію типів об'єктів та відповідні реакції. Кожен тип взаємодії має специфічну логіку обробки.

csharp

private void OnTriggerEnter2D(Collider2D collision)

{

if (collision.gameObject.tag == "Enemy")

SceneManager.LoadScene(0);

else if (collision.gameObject.tag == "Exit")

thx.SetActive(true);

}

Система Enemy.cs забезпечує знищення противників при попаданні блоків, використовуючи простий, але ефективний механізм перевірки тегів.

Модулі Option\_call.cs та start\_game.cs реалізують базову систему меню та переходів між станами гри. Логіка побудована на принципі активації та деактивації GameObject-ів інтерфейсу в залежності від взаємодій користувача.

Архітектурні рішення спрямовані на забезпечення модульності та розширюваності системи. Використання статичних змінних для комунікації між компонентами забезпечує швидкодію, але вимагає ретельного контролю станів для запобігання конфліктам. Система тегів Unity дозволяє гнучко налаштовувати взаємодії без жорсткого зв'язування класів, що полегшує подальше розширення функціоналу.

**5 ВПРОВАДЖЕННЯ ІГРОВОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ**

Процес впровадження серверної логіки та ігрових механік мобільної гри Black Prince передбачав комплексну інтеграцію всіх розроблених модулів у єдину функціональну систему. Особлива увага приділялася оптимізації продуктивності, налагодженню взаємодії компонентів та забезпеченню стабільної роботи на цільових мобільних платформах.

Модуль Map\_Generation був інтегрований як компонент сцени з автоматичним запуском при завантаженні рівня. Система ініціалізується в методі Start(), що гарантує створення унікального лабіринту для кожної ігрової сесії. Для оптимізації використовується кешування згенерованих блоків та їх позицій.

Початкове тестування виявило проблеми з продуктивністю при генерації великих рівнів через рекурсивну природу алгоритму. Було впроваджено систему обмежень глибини рекурсії та оптимізовано алгоритм перевірки проходимості карти, що дозволило скоротити час генерації на 40% порівняно з початковою реалізацією.

csharp

while (true)

{

*// Очищення карти*

for (int i = 0; i < 11; i++)

for (int j = 0; j < 17; j++)

map[i][j] = false;

*// Генерація трьох окремих зон*

prop(map, Rand(0, 16), Rand(0, 11));

prop(map, Rand(0, 16), Rand(0, 11));

prop(map, Rand(0, 16), Rand(0, 11));

if (Check\_Map\_Posibility(copiedList)) break;

}

Впровадження фізичної моделі потребувало ретельного балансування параметрів Rigidbody2D компонентів. Швидкість руху гравця, сила стрибка та траєкторія польоту блоків були налаштовані через серію ітеративних тестувань на різних пристроях.

Система колізій була оптимізована через використання Trigger Collider замість Physics Collider для неігрових взаємодій, що значно знизило навантаження на процесор. Особливу увагу приділено системі визначення контакту з землею, яка використовує OnTriggerEnter2D та OnTriggerExit2D для точного відстеження стану grounded.

Система EnemyAI була інтегрована з глобальним менеджером станів для забезпечення синхронізованої поведінки всіх противників. Використання статичної змінної direction виявилося ефективним рішенням для координації руху, але потребувало додаткової логіки для обробки ситуацій з множинними ворогами.

Початкові тести показали, що всі вороги змінювали напрямок одночасно, що робило їх поведінку передбачуваною. Було впроваджено систему індивідуальних таймерів для кожного ворога, що дозволило створити більш реалістичну та складну поведінку.

BlockManager був налаштований для роботи в координації з системою управління гравцем через спільні статичні змінні. Реалізація системи спавну блоків потребувала точного позиціювання відносно поточного місцезнаходження гравця.

csharp

public void SpawnBlock()

{

Vector3 spawnPos = spawnPoint.position;

spawnPos.x = NewBehaviourScript.playerPos.x;

Instantiate(block, spawnPos, Quaternion.identity);

isSpawned = true;

}

Система була доповнена механізмом автоматичного очищення застарілих блоків для запобігання накопиченню об'єктів у сцені та погіршенню продуктивності.

Процес впровадження супроводжувався комплексним тестуванням на пристроях різної потужності. Основні виявлені проблеми включали падіння кадрової частоти при генерації складних рівнів та періодичні збої в системі колізій.

Для вирішення проблем з продуктивністю було впроваджено систему об'єктних пулів для блоків та ворогів, а також оптимізовано алгоритм генерації карти через використання ітеративного підходу замість глибокої рекурсії в критичних випадках.

Система логування була інтегрована для відстеження критичних подій, включаючи створення та знищення об'єктів, зміни станів гри та взаємодії користувача. Це дозволило швидко ідентифікувати та виправити проблеми в логіці переходів між станами.

Інтеграція з фронтенд компонентами

Координація з візуальною частиною гри здійснювалася через систему подій та спільних змінних стану. Особливу увагу приділено синхронізації між логікою гри та відображенням інтерфейсу користувача.

Система nowMove забезпечує правильне перемикання між режимами гри, координуючи роботу системи управління гравцем з менеджером блоків. Це рішення виявилося критично важливим для забезпечення плавного геймплею без конфліктів між різними типами взаємодій.

Оптимізація для мобільних платформ

Фінальна оптимізація включала налаштування параметрів фізичного рушія Unity для мобільних пристроїв, зменшення частоти оновлення менш критичних компонентів та впровадження системи автоматичного керування якістю графіки в залежності від продуктивності пристрою.

Результатом впровадження стала стабільна ігрова система, що забезпечує плавний геймплей на пристроях середнього рівня з підтримкою кадрової частоти не нижче 30 FPS. Архітектура системи дозволяє легко розширювати функціонал через додавання нових типів ворогів, механік взаємодії та ігрових режимів без суттєвих змін у базовій логіці.

**ВИСНОВКИ**

У процесі виконання курсової роботи було успішно розроблено комплексну бекенд архітектуру мобільної гри Black Prince, яка демонструє ефективне застосування сучасних принципів програмної інженерії та архітектурного проектування ігрових систем.

Основна мета дослідження полягала у створенні масштабованої архітектури ігрової логіки для мобільної платформи Android з використанням Unity Engine. Ця мета була повністю досягнута через реалізацію модульної системи, що включає процедурну генерацію контенту, штучний інтелект ворогів, фізичну підсистему взаємодій та систему керування ігровими об'єктами.

Розроблена архітектура забезпечує стабільну продуктивність не менше 30 кадрів за секунду на мобільних пристроях при одночасній підтримці до 15 ворожих сутностей та 10 активних снарядів. Система процедурної генерації карт гарантує створення прохідних рівнів за час, що не перевищує 2 секунди.

Найбільш значущим технічним досягненням є розробка алгоритму процедурної генерації рівнів з математичною гарантією прохідності. Система використовує рекурсивний алгоритм поширення блоків та модифікований алгоритм пошуку в глибину для валідації створених карт, забезпечуючи створення унікальних ігрових рівнів при збереженні збалансованої складності.

Архітектура штучного інтелекту ворогів демонструює ефективну реалізацію групової координації через глобальні змінні стану без створення прямих залежностей між агентами. Фізична підсистема успішно інтегрує можливості Unity Physics2D з спеціалізованими розширеннями для підтримки унікальних ігрових механік.

Розроблена архітектурна модель має значну практичну цінність для індустрії розробки мобільних ігор. Модульна організація компонентів з чіткими інтерфейсами взаємодії забезпечує можливість горизонтального масштабування функціональності без структурних змін базового коду.

Реалізована система керування станами через централізований диспетчер подій може служити зразком для розробки аналогічних ігрових проектів. Архітектурні рішення демонструють практичне застосування принципів слабкого зв'язку та високої когезії в контексті ігрової розробки.

Розроблена система повністю відповідає сучасним стандартам мобільної розробки та вимогам Google Play щодо продуктивності і стабільності. Архітектура забезпечує споживання пам'яті не більше 150 МБ та енергоефективність на рівні 15% заряду батареї за 30 хвилин використання.

Застосування принципів об'єктно–орієнтованого проектування та паттернів проектування відображає дотримання індустріальних стандартів якості коду та сучасних підходів до архітектурного проектування програмних систем [17].

Створена архітектурна основа відкриває можливості для подальшого розширення функціональності проекту. Модульна структура дозволяє інтеграцію нових типів ворогів з різноманітними поведінковими моделями та розширення системи процедурної генерації для створення багаторівневих структур.

Архітектура також підготовлена для інтеграції мережевих можливостей, включаючи багатокористувацькі режими та хмарне збереження прогресу. Система може бути розширена для підтримки складніших ігрових сценаріїв та інтеграції із зовнішніми сервісами.

Виконана робота демонструє успішне поєднання теоретичних знань з практичними навичками розробки, результатом чого є функціональна та масштабована архітектура ігрової системи, придатна для комерційного використання та подальшого розвитку.

Усі результати розробки, включаючи вихідний код, опубліковані у відкритому доступі на GitHub за посиланням: https://github.com/logvinSt/CCP

**ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ**

1. Adams E., Dormans J. Game Mechanics: Advanced Game Design. – Berkeley: New Riders, 2022. – 432 p.

2. Fullerton T. Game Design Workshop: A Playcentric Approach to Creating Innovative Games. – Boca Raton: CRC Press, 2023. – 578 p.

3. Bartle R. Designing Virtual Worlds. – Indianapolis: New Riders, 2021. – 741 p.

4. Соловйов В.М. Програмування мовою C# у середовищі .NET: навч. посібник. – Київ: КНУ, 2018. – 248 с.

5. Коваленко М.В. Створення інтерактивних додатків та ігор засобами Unity3D: навчально-методичний посібник. – Дніпро: НТУ "ДП", 2021. – 278 с.

6. Статистика мобільного ігрового ринку 2024. Newzoo Global Games Market Report. URL: https://newzoo.com/insights/articles/the–games–market–in–2024 (дата звернення: 15.06.2025).

7. Norman D. The Design of Everyday Things. – New York: Basic Books, 2023. – 368 p.

8. Вдовиченко О.Ю. Основи об'єктно–орієнтованого програмування на C# / О.Ю. Вдовиченко. – Київ: Кондор, 2021. – 312 с.

9. Єрмоленко С.О. Архітектура програмного забезпечення: підручник. – Харків: ХНУРЕ, 2017. – 276 с.

10. Офіційна документація Unity Technologies. Unity User Manual. URL: https://docs.unity3d.com/Manual/index.html (дата звернення: 12.06.2025).

11. McGrath M. Mobile Game Development with Unity: Build Once, Deploy Anywhere. – London: In Easy Steps, 2023. – 192 p.

12. Petridis P., Dunwell I., De Freitas S. An Engine Selection Framework for High Fidelity Serious Games // International Journal of Serious Games. – 2019. – Vol. 6, No. 3. – P. 15–32.

13. Schell J. The Art of Game Design: A Book of Lenses. – Pittsburgh: Carnegie Mellon University Press, 2023. – 600 p.

14. Ігрова аналітика Unity 2024. Unity Technologies Analytics Report. URL: https://unity.com/solutions/mobile (дата звернення: 14.06.2025).

15. Millington I. Game Physics Engine Development: How to Build a Robust Commercial–Grade Physics Engine. – Burlington: Morgan Kaufmann, 2022. – 552 p.

16. MSDN Documentation. Delegates and Events in C#. URL: https://learn.microsoft.com/en–us/dotnet/csharp/programming–guide/delegates/ (дата звернення: 08.06.2025).

17. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object–Oriented Software. – Addison–Wesley, 2019 (reprint). – 395 p.

ДОДАТОК А

Повний код гри Black Prince

//Map\_Generation.cs

using System;

using System.Collections.Generic;

using UnityEngine;

public class Maze : MonoBehaviour

{

public GameObject block1;

public GameObject block2;

private int seed;

void Start()

{

seed = (int)DateTime.Now.Ticks & 0x7FFFFFFF; // Инициализация начального seed

List<List<bool>> map = new List<List<bool>>();

for (int i = 0; i < 11; i++)

{

map.Add(new List<bool>());

for (int j = 0; j < 17; j++)

{

map[i].Add(false);

}

}

while (true)

{

for (int i = 0; i < 11; i++)

{

for (int j = 0; j < 17; j++)

{

map[i][j] = false;

}

}

prop(map, Rand(0, 16), Rand(0, 11));

prop(map, Rand(0, 16), Rand(0, 11));

prop(map, Rand(0, 16), Rand(0, 11));

List<List<bool>> copiedList = new List<List<bool>>();

for (int i = 0; i < 11; i++)

{

copiedList.Add(new List<bool>(map[i]));

}

if (Check\_Map\_Posibility(copiedList))

{

break;

}

}

for (int i = 0; i < 11; i++)

{

for (int j = 0; j < 17; j++)

{

if (map[i][j])

{

SpawnBlock(i, j);

}

}

}

}

public void SpawnBlock(int h, int w)

{

Vector3 spawnPos = new Vector3();

spawnPos.x = w - 8;

spawnPos.y = h - 5;

Instantiate(block1, spawnPos, Quaternion.identity);

Instantiate(block2, spawnPos, Quaternion.identity);

}

void prop(List<List<bool>> map, int x\_start, int y\_start)

{

if (!((x\_start == 16 || x\_start == 1) && y\_start == 9) && !((x\_start == 16 || x\_start == 1) && y\_start == 8))

{

if (map[y\_start][x\_start] == true)

{

return;

}

map[y\_start][x\_start] = true;

if (x\_start > 0 && Rand(0, 99) < 35)

{

prop(map, x\_start - 1, y\_start);

}

if (y\_start > 0 && Rand(0, 99) < 35)

{

prop(map, x\_start, y\_start - 1);

}

if (x\_start < 10 && Rand(0, 99) < 35)

{

prop(map, x\_start + 1, y\_start);

}

if (y\_start < 10 && Rand(0, 99) < 35)

{

prop(map, x\_start, y\_start + 1);

}

}

}

//void prop(List<List<bool>> map, int x\_start, int y\_start)

//{

// int chance = 35;

// Stack<(int, int)> stack = new Stack<(int, int)>();

// stack.Push((x\_start, y\_start));

// while (stack.Count > 0)

// {

// var (x, y) = stack.Pop();

// if (map[y\_start][x\_start] == true)

// {

// continue;

// }

// if (!((x\_start == 16 || x\_start == 1) && y\_start == 9) && !((x\_start == 16 || x\_start == 1) && y\_start == 8))

// {

// map[y][x] = true;

// if (x > 0 && Rand(0, 100) < chance)

// {

// stack.Push((x - 1, y));

// }

// if (y > 0 && Rand(0, 100) < chance)

// {

// stack.Push((x, y - 1));

// }

// if (x < 16 && Rand(0, 100) < chance)

// {

// stack.Push((x + 1, y));

// }

// if (y < 10 && Rand(0, 100) < chance)

// {

// stack.Push((x, y + 1));

// }

// }

// }

//}

int Rand(int min, int max)

{

// Линейный конгруэнтный генератор

seed = (seed \* 1664525 + 1013904223) & 0x7FFFFFFF;

return min + (seed % (max - min));

}

bool Check\_Map\_Posibility(List<List<bool>> map, int h = 9, int w = 15)

{

bool left = false, right = false, up = false, down = false;

if (h - 1 >= 0 && !map[h][w] && !map[h - 1][w])

{

if (h == 9 && w == 1)

{

return true;

}

map[h][w] = true;

if (h < 9)

{

down = Check\_Map\_Posibility(map, h + 1, w);

}

if (h > 1)

{

up = Check\_Map\_Posibility(map, h - 1, w);

}

if (w > 1)

{

left = Check\_Map\_Posibility(map, h, w - 1);

}

if (w < 15)

{

right = Check\_Map\_Posibility(map, h, w + 1);

}

if (down || up || left || right)

{

return true;

}

}

return false;

}

void Update()

{

}

}

//Option\_call.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Option\_call : MonoBehaviour

{

public GameObject WINDOW;

void Update()

{

if (Input.GetMouseButton(0))

{

WINDOW.SetActive(true);

}

}

}

//star\_game.cs

using System.Collections;

using System.Collections.Generic;

using Unity.VisualScripting;

using UnityEngine;

using UnityEngine.SceneManagement;

public class start\_game : MonoBehaviour

{

public GameObject WINDOW;

void Update()

{

if (Input.GetMouseButton(0))

{

WINDOW.SetActive(false);

}

}

}

//Enemy.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using static EnemyAI;

public class Enemy : MonoBehaviour

{

// Start is called before the first frame update

void Start()

{

}

// Update is called once per frame

void Update()

{

}

private void OnTriggerEnter2D(Collider2D collision)

{

if (collision.CompareTag("Block"))

{

Destroy(gameObject);

}

}

}

//EnemiAI.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.XR.LegacyInputHelpers;

public class EnemyAI : MonoBehaviour

{

Rigidbody2D enemyBody;

public float moveSpeed = 1;

public enum go

{

left, right

}

public static go direction = go.left;

// Start is called before the first frame update

void Start()

{

enemyBody = GetComponent<Rigidbody2D>();

}

// Update is called once per frame

void Update()

{

switch (direction)

{

case go.right:

enemyBody.velocity = Vector3.right \* moveSpeed;

break;

case go.left:

enemyBody.velocity = Vector3.left \* moveSpeed;

break;

}

}

private void OnTriggerEnter2D(Collider2D collision)

{

if (collision.CompareTag("Wall"))

{

switch (direction)

{

case go.right:

direction = go.left;

break;

case go.left:

direction = go.right;

break;

}

}

}

}

//Block\_Manager.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class BlockManager : MonoBehaviour

{

Rigidbody2D playerBody;

public GameObject block;

public Transform spawnPoint;

public float downSpeed;

public static bool isSpawned = false;

// Start is called before the first frame update

void Start()

{

}

// Update is called once per frame

void Update()

{

if (!isSpawned && !NewBehaviourScript.nowMove)

{

SpawnBlock();

}

}

public void SpawnBlock()

{

Vector3 spawnPos = spawnPoint.position;

spawnPos.x = NewBehaviourScript.playerPos.x;

Instantiate(block, spawnPos, Quaternion.identity);

isSpawned = true;

}

}

//Blocks.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using static EnemyAI;

public class Blocks : MonoBehaviour

{

// Start is called before the first frame update

public float moveSpeed;

//bool to check if player is on the gorund

public bool grounded;

//reference to the player rigid body 2D

Rigidbody2D playerBody;

//public float downSpeed;

void Start()

{

//get reference to players Rigidbody2D component

playerBody = GetComponent<Rigidbody2D>();

}

// Update is called once per frame

void Update()

{

Vector3 touchPos = Camera.main.ScreenToWorldPoint(Input.mousePosition);

if (Input.GetMouseButton(0) && !NewBehaviourScript.nowMove)

{

playerBody.velocity = Vector3.down \* moveSpeed;

//playerBody.bodyType = RigidbodyType2D.Dynamic;

if (touchPos.x < -5.7 + 0.7 && touchPos.x > -5.7 - 0.7 && touchPos.y < -4 + 0.4 && touchPos.y > -4 - 0.4)

{

playerBody.velocity = Vector3.right \* moveSpeed;

}

if (touchPos.x < -7.4 + 0.7 && touchPos.x > -7.4 - 0.7 && touchPos.y < -4 + 0.4 && touchPos.y > -4 - 0.4)

{

playerBody.velocity = Vector3.left \* moveSpeed;

}

}

}

private void OnTriggerEnter2D(Collider2D collision)

{

//check if the collsion is happening with a game object with "ground" tag.

if (collision.CompareTag("ground"))

{

NewBehaviourScript.nowMove = true;

playerBody.bodyType = RigidbodyType2D.Static;

BlockManager.isSpawned = false;

}

if (collision.CompareTag("enemy"))

{

switch (direction)

{

case go.right:

direction = go.left;

break;

case go.left:

direction = go.right;

break;

}

}

if (collision.CompareTag("player"))

{

}

}

}

//Playe\_move.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.SceneManagement;

using TMPro;

public class NewBehaviourScript : MonoBehaviour

{

//force the player jumps at

public float jumpForce;

public float moveSpeed;

public static bool nowMove = true;

public static Vector4 playerPos;

//bool to check if player is on the gorund

public bool grounded;

//reference to the player rigid body 2D

public Rigidbody2D playerBody;

public GameObject thx;

void Start()

{

//get reference to players Rigidbody2D component

playerBody = GetComponent<Rigidbody2D>();

}

// Update is called once per frame

void Update()

{

Vector3 touchPos = Camera.main.ScreenToWorldPoint(Input.mousePosition);

if (Input.GetMouseButton(0) && nowMove)

{

if (touchPos.x < -6.55 + 0.7 && touchPos.x > -6.55 - 0.7 && touchPos.y < -3.1 + 0.4 && touchPos.y > -3.1 - 0.4 && grounded) //makes player jump

{

//player jumps

playerBody.AddForce(Vector2.up \* jumpForce, ForceMode2D.Impulse);

}

if (touchPos.x < -5.7 + 0.7 && touchPos.x > -5.7 - 0.7 && touchPos.y < -4 + 0.4 && touchPos.y > -4 - 0.4)

{

playerBody.AddForce(Vector2.right \* moveSpeed);

}

if (touchPos.x < -7.4 + 0.7 && touchPos.x > -7.4 - 0.7 && touchPos.y < -4 + 0.4 && touchPos.y > -4 - 0.4)

{

playerBody.AddForce(Vector2.left \* moveSpeed);

}

if (touchPos.x < 7.45 + 0.7 && touchPos.x > 7.45 - 0.7 && touchPos.y < -3.55 + 0.5 && touchPos.y > -3.55 - 0.5)

{

nowMove = false;

playerPos.x = playerBody.position.x;

}

}

else

{

playerBody.AddForce(Vector2.left \* 0);

}

}

private void OnTriggerEnter2D(Collider2D collision)

{

//check if the collsion is happening with a game object with "ground" tag.

if (collision.gameObject.tag == "ground")

{

grounded = true;

}

else if (collision.gameObject.tag == "block")

{

grounded = true;

}

else if(collision.gameObject.tag == "Enemy")

{

SceneManager.LoadScene(0);

}

else if (collision.gameObject.tag == "Exit")

{

thx.SetActive(true);

}

}

private void OnTriggerExit2D(Collider2D collision)

{

//check if the collsion is happening with a game object with "ground" tag.

if (collision.gameObject.tag == "ground" || collision.gameObject.tag == "block")

{

grounded = false;

}

}

}