ХЕРСОНСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ТА ДИЗАЙНУ
КАФЕДРА ІНФОРМАТИКИ І КОМП'ЮТЕРНИХ НАУК

Пояснювальна записка

до дипломної бакалаврської роботи

на тему:

ЗАСТОСУВАННЯ ГЕНЕТИЧНИХ АЛГОРИТМІВ ДЛЯ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ ТА ОПТИМІЗАЦІЇ ІГРОВИХ РІВНІВ ЖАНРУ TOWER DEFENSE

Виконав: студент 4 курсу, групи 4КН
спеціальності 122 «Комп'ютерні науки»

Сороковський М.Ю.

Керівник: Вороненко М.О.

Рецензент: Огнєва О.Є.

Хмельницький – 2026р.

Факультет	**Інформаційних технологій та дизайну**
Кафедра	**Інформатики і комп'ютерних наук**
Рівень вищої освіти	**перший (бакалаврський) рівень**
Галузь підготовки	**12 «Інформаційні технології»**
Освітньо-професійна програма	**Комп'ютерні науки**
Спеціальність	**122 «Комп'ютерні науки»**

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри ІКН,
доцент
_____ Моісеєнко С.В.
«____» _____ 2026 року

ЗАВДАННЯ
НА ДИПЛОМНУ РОБОТУ СТУДЕНТА

_____Сороковського Максима Юрійовича_____
(прізвище, ім'я, по батькові)

1. Тема роботи: Застосування генетичних алгоритмів для процедурної генерації та оптимізації ігрових рівнів жанру Tower Defense.
керівник роботи Вороненко Марія Олександрівна, кандидат технічних наук, доцент кафедри інформатики і комп'ютерних наук
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом ХНТУ від 28.01. 2024 р. № 162-с
2. Строк подання студентом роботи
_____
3. Вихідні дані до роботи _____

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити): Вступ; 1. Аналітичний огляд літературних та інших джерел; 2. Обґрунтування проблеми та аналіз методів і засобів її вирішення; 3. Практична реалізація генетичних алгоритмів для процедурної генерації та оптимізації ігрових рівнів жанру Tower Defense; Висновки.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

Таблиць – 4,

Формул – 6,

Рисунків – 18

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 09.02.2026

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1	Аналітичний огляд літературних джерел за тематикою бакалаврської роботи	09.02.2026-20.02.2026	
2	Постановка проблеми дослідження	23.02.2026-09.03.2026	
3	Обґрунтування проблеми та аналіз методів і засобів її вирішення	10.03.2026-24.03.2026	
4	Системний аналіз об’єкта дослідження та предметної області	25.03.2026-01.04.2026	
5	Практична реалізація генетичного алгоритму у системі генерації рівнів	02.04.2026-09.04.2026	
6	Проведення тестів і отримання результатів	10.04.2026-17.04.2026	
7	Проведення корекції алгоритмів та оформлення висновків	20.04.2026-27.04.2026	
8	Написання та оформлення пояснювальної записки до кваліфікаційної роботи	28.04.2026-05.05.2026	

Студент _____ Максим СОРОКОВСЬКИЙ

(підпис) (прізвище та ініціали)

Керівник роботи _____ Марія ВОРОНЕНКО

(підпис) (прізвище та ініціали)

ЗМІСТ

Автореферат

У кваліфікаційній роботі досліджено теоретичні аспекти застосування генетичних алгоритмів для процедурної генерації та оптимізації ігрових рівнів жанру Tower Defense. Створення якісного та збалансованого ігрового контенту вручну є трудомісткою проблемою, що впливає на терміни розробки та вартість кінцевого продукту. Автоматизація створення рівнів та забезпечення їхньої варіативності можуть значно підвищити реіграбельність гри та інтерес користувачів. Розробка ефективних методів процедурної генерації контенту є важливим завданням сучасної ігрової індустрії. Для цього потрібно: вивчити теоретичні основи еволюційних обчислень та генетичних алгоритмів, розробити структуру представлення ігрового рівня та цільову функцію, реалізувати алгоритм генерації та інтегрувати його в ігрове середовище Tower Defense, оцінити ігрову придатність згенерованих мап.

Генетичні алгоритми — це евристичні алгоритми пошуку, що використовуються для вирішення задач оптимізації та моделювання шляхом імітації природного відбору. Вони використовуються для знаходження рішень у великих просторах пошуку, де прямий перебір є неможливим. Генетичний алгоритм оперує популяцією рішень, застосовуючи до них оператори відбору, схрещування та мутації. Цільова функція — це ключовий компонент алгоритму, який використовується для кількісної оцінки якості кожного згенерованого рішення. Вона дозволяє відсіювати некоректні рівні (наприклад, без проходу до бази) та відбирати найкращі варіанти за критеріями довжини шляху та балансу.

Алгоритм налаштовується на основі критеріїв ігрового балансу, що містять вимоги до топології рівня Tower Defense. Використовуються методи еволюційного моделювання для ітеративного покращення популяції ігрових рівнів.

Для оцінки ефективності розробленого методу використовується тестування прохідності та аналіз складності згенерованих мап. Розроблена система процедурної генерації на основі генетичних алгоритмів є ефективним

інструментом для автоматизованого створення контенту. Вона може бути використана розробниками ігор для швидкого наповнення гри унікальними рівнями, що дозволить оптимізувати процес розробки.

В майбутньому можна розглянути розширення алгоритму для генерації не лише топології, а й хвиль ворогів, спробувати використання нейромереж для оцінки естетичної якості рівнів, запланувати інтеграцію механізму динамічної адаптації складності під навички конкретного гравця.

Abstract

The qualification paper investigates theoretical aspects of using genetic algorithms for the procedural generation and optimization of Tower Defense game levels. Creating high-quality and balanced game content manually is a labor-intensive problem that affects development timelines and the final product cost. Automating level creation and ensuring their variability can significantly increase replayability and user interest. Developing effective methods of procedural content generation (PCG) is an important task for the modern game industry. To achieve this, it is necessary to: study the theoretical foundations of evolutionary computation and genetic algorithms, develop the structure of game level representation (chromosome) and the fitness function, implement the generation algorithm and integrate it into the Tower Defense game environment, and evaluate the playability of the generated maps.

Genetic algorithms are heuristic search algorithms used to solve optimization and modeling problems by simulating natural selection. They are used to find solutions in large search spaces where direct enumeration is impossible. A genetic algorithm operates on a population of solutions, applying selection, crossover, and mutation operators to them. The fitness function is a key component of the algorithm, used for the quantitative assessment of the quality of each generated solution. It allows for filtering out invalid levels (for example, those without a path to the base) and selecting the best variants based on criteria such as path length and balance.

The algorithm is tuned based on game balance criteria that include requirements for the topology of a Tower Defense level. Evolutionary modeling methods are used for the iterative improvement of the game level population.

To evaluate the effectiveness of the developed method, playability testing and complexity analysis of the generated maps are used. The developed procedural generation system based on genetic algorithms is an effective tool for automated content creation. It can be used by game developers to rapidly populate the game with unique levels, allowing for the optimization of the development process.

In the future, extending the algorithm to generate not only topology but also enemy waves can be considered, as well as attempting the use of neural networks to assess the aesthetic quality of levels, and planning the integration of a dynamic difficulty adaptation mechanism tailored to a specific player's skills.

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

Скорочення, термін, позначення	**Пояснення**
КА	Клітинні автомати
PCG	Procedural Content Generation
GA	Genetic Algorithms
WFC	Wave Function Collapse
TD	Tower Defense
GC	Garbage Collector

ВСТУП

Стрімкий розвиток індустрії комп’ютерних ігор та підвищення вимог гравців до якості контенту викликає все більше занепокоєння серед розробників. Сучасні ігрові проєкти потребують величезної кількості контенту, такого як: ігрові рівні, моделі поведінки ворогів, балансування складності. Ручне створення та налаштування цих елементів є основною проблемою, що призводить до зростання вартості та часу розробки.

У сучасних дослідженнях розробки ігор (GameDev) одним із ключових аспектів є створення ефективних методів процедурної генерації контенту, що дозволяє автоматизувати рутинні процеси та забезпечити високу реіграбельність. Одним з поширених перспективних напрямків досліджень є використання генетичних алгоритмів, які здатні знаходити оптимальні рішення у великих просторах пошуку та допомагають створювати складні, збалансовані структури. Генетичні алгоритми є потужними інструментами для генерації, оскільки ці еволюційні моделі мають можливість адаптації до заданих критеріїв якості. Однак у процесі генерації дуже часто виникають не лише ігрові, але й "некоректні" варіанти рівнів (наприклад, заблоковані шляхи або відсутність балансу), які можуть негативно вплинути на ігровий досвід.

Одним із підходів для досягнення високої якості та варіативності в генерації рівнів жанру Tower Defense є використання ретельно налаштованої цільової функції у складі генетичного алгоритму. Цей компонент може ефективно оцінювати придатність рівня та оптимізувати процес відбору в умовах великої кількості комбінацій тайлів. У даній роботі розглядається розробка системи на основі генетичних алгоритмів для процедурної генерації та оптимізації ігрових рівнів. Методом є створення моделі, яка може не тільки генерувати топологію карти, але й гарантувати її прохідність та ігровий баланс, що сприяє утриманню інтересу гравця.

Для досягнення цієї мети в роботі будуть розроблені основні принципи функціонування генетичного алгоритму для задачі Tower Defense, зокрема

його здатність кодувати карту у вигляді хромосоми та підтримувати еволюцію популяції рівнів. Важливим аспектом є адаптація операторів кросоверу та мутації до специфічних характеристик жанру, де карта має чіткі обмеження (наявність входу, виходу, місць для будівництва).

Актуальність теми.

Незважаючи на успіхи, досягнуті в галузі процедурної генерації та штучного інтелекту в іграх, все ще існує багато критичних викликів, які спонукають дослідників шукати нові підходи. Створення ігрових рівнів є основним і відповідальним етапом, що визначає успіх стратегічної гри. Воно використовується для забезпечення унікального досвіду кожного ігрового сеансу.

Однак, незважаючи на можливості алгоритмічних підходів, розробники все ще стикаються з проблемою створення рівнів, які були б не просто випадковими, а цікавими та збалансованими. Існує багато запитань, на які все ще немає чіткої відповіді:

- які найкращі підходи до кодування двовимірної карти у геном для ефективного використання операторів схрещування?
- як ми можемо розробити адаптивну цільову функцію, яка врахує не лише геометрію рівня, а й складність проходження для гравця?
- наскільки повно згенерований рівень може відповідати естетичним вимогам та дизайнерським задумам?
- в яких випадках процедурна генерація може ефективно замінити роботу левел-дизайнера, а де вона виступає лише допоміжним інструментом?
- чи здатні еволюційні моделі виявляти та усувати "мертві зони" на картіез втручання людини?

У цій роботі ми розглянемо ці питання, а також наведемо результати розробки системи генерації на основі генетичних алгоритмів. Враховуючи, що однією з проблем при розробці є великий простір можливих комбінацій тайлів,

у цьому дослідженні пропонується методика оптимізації параметрів еволюції, що дозволяє швидше знаходити якісні рішення.

Метою роботи є розробка методу та програмного засобу для процедурної генерації ігрових рівнів жанру Tower Defense з використанням генетичних алгоритмів для забезпечення їх варіативності та збалансованості.

Задачі дослідження.

Кваліфікаційна робота спрямована на застосування еволюційних обчислень для автоматизації левел-дизайну. В роботі показано, що генетичні алгоритми дозволяють створювати складні ігрові карти, які відповідають заданим критерієм якості.

Основними задачами цієї роботи є:

I. Виконати аналіз існуючих методів процедурної генерації контенту в іграх жанру Tower Defense.
II. Розробити структуру представлення ігрового рівня (хромосому) та визначити набір тайлів для генерації.
III. Розробити математичну модель цільової функції (fitness function) для оцінки прохідності та складності рівня.
IV. Програмно реалізувати генетичний алгоритм, включаючи оператори селекції, кросоверу та мутації.
V. Провести експериментальне дослідження ефективності алгоритму та оцінити якість згенерованих рівнів.

Об’єктом дослідження є процес процедурної генерації ігрового контенту в стратегічних іграх.

Предметом дослідження є методи та алгоритми еволюційних обчислень для оптимізації топології ігрових рівнів.

Наукова новизна одержаних результатів:

- науково обґрунтована доцільність використання генетичних алгоритмів для задачі побудови рівнів із гарантованим шляхом проходження;
- удосконалено метод оцінки пристосованості, що враховує специфічні метрики жанру Tower Defense (довжина шляху, кількість точок забудови);
- показано, що розроблена модель дозволяє генерувати ігрові рівні, які за своїми характеристиками наближаються до створених вручну.

Практичне значення одержаних результатів.

Завдяки розробленому алгоритму була створена та протестована система генерації, що моделює різноманітні ігрові сценарії. Результати роботи можуть бути використані інді-розробниками та студіями для пришвидшення процесу розробки ігор та створення режимів "нескінченної гри".

Апробація результатів бакалаврської кваліфікаційної роботи.

РОЗДІЛ 1.
АНАЛІТИЧНИЙ ОГЛЯД ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1.1. Поняття процедурної генерації контенту в іграх, методи та підходи

Процедурна генерація контенту (Procedural Content Generation) у відеоіграх визначається як створення ігрових даних алгоритмічним шляхом з обмеженим або опосередкованим втручанням людини. Цей термін охоплює широкий спектр методів, від простих алгоритмів рандомізації до складних систем на основі штучного інтелекту, що здатні створювати текстури, тривимірні моделі, звуковий супровід, правила гри та, що є предметом даного дослідження, ігрові рівні. В основі PCG лежить використання псевдовипадкових чисел та наборів правил, які трансформують вхідні параметри у структурований вихідний контент.

Актуальність використання PCG у сучасній індустрії розробки ігор зумовлена необхідністю вирішення проблеми масштабування контенту. Ручне створення вимагає значних часових та людських ресурсів, що стає критичним обмеженням при розробці масштабних відкритих світів або ігор з високим рівнем реіграбельності. Процедурний підхід дозволяє розробникам створювати нескінченну варіативність ігрових ситуацій, забезпечуючи унікальний досвід користувача під час кожної сесії.

У науковій літературі методи процедурної генерації класифікують за кількома ключовими критеріями, що визначають підхід до побудови алгоритму:

По-перше, за часом виконання генерації виділяють offline та online методи. Offline-генерація відбувається на етапі розробки гри: алгоритми використовуються для швидкого створення великої кількості асетів, які потім відбираються, оптимізуються та зберігаються у фінальній версії продукту. Online-генерація виконується у реальному часі безпосередньо на пристрої

користувача. Цей підхід є критичним для жанрів, що передбачають нескінченний ігровий процес або адаптацію складності під навички гравця, зокрема для стратегій жанру Tower Defense.

По-друге, за ступенем контролю та наявності зворотного зв'язку методи поділяють на конструктивні та методи «генерація та перевірка».

Конструктивні алгоритми генерують контент за один прохід, послідовно застосовуючи правила побудови. Прикладом таких методів є клітинні автомати або використання шуму Перліна. Вони є обчислювально ефективними, проте часто не гарантують виконання глобальних ігрових обмежень, таких як зв'язність шляху.

Методи «генерація та перевірка» працюють ітеративно: спочатку створюється кандидат рішення, який потім оцінюється за допомогою цільової функції. Якщо кандидат не відповідає критеріям якості, він відхиляється або модифікується. До цього класу належать еволюційні обчислення та генетичні алгоритми (GA). Цей підхід дозволяє створювати складні, збалансовані структури, що гарантовано відповідають правилам гри, хоча й потребує більших обчислювальних потужностей.

Таблиця 1

Порівняльна характеристика основних парадигм процедурної генерації

Критерій порівняння	**Конструктивні методи**	**Методи «генерація та перевірка» (у т.ч. ГА)**
Принцип роботи	Однопрохідна генерація шляхом послідовного застосування правил.	Ітеративний пошук: створення кандидата та його оцінка цільовою функцією.
Обчислювальна складність	Низька (генерують контент дуже швидко, висока ефективність).	Висока (потребують значних потужностей для багаторазової симуляції/оцінки).
Дотримання глобальних обмежень	Не гарантується (високий ризик створення непрохідних зон або ізольованих шляхів).	Гарантується (невалідні рішення відсіюються на етапі оцінки).

Критерій порівняння	Критерій порівняння	Методи «генерація та перевірка» (у т.ч. ГА)
Здатність до балансування гри	Низька (результат залежить виключно від випадковості та жорстких правил).	Висока (можливість оптимізувати складність через фітнес-функцію).
Приклади методів	Шум Перліна, клітинні автомати, L-системи.	Еволюційні обчислення, генетичні алгоритми (GA).
Придатність для жанру Tower Defense	Обмежена (використовуються переважно як допоміжні для генерації декору/ландшафту).	Висока (ідеально підходять для створення структурованих, зв'язних маршрутів).

Також важливим є розрізнення методів за типом генерованого контенту: від чисто візуальних елементів (текстури, ефекти) до структурних (рівні, мапи) та семантичних. Для стратегічних ігор ключовим є саме структурна генерація, яка вимагає суворого дотримання топологічних правил, адже найменша помилка в генерації шляху може зробити рівень непрохідним.

Таким чином, вибір конкретного методу PCG залежить від балансу між необхідною швидкістю генерації, ступенем контролю над результатом та вимогами до якості контенту. Для задач, де критичним є ігровий баланс та логічна зв'язність, найбільш перспективними є пошукові методи, які розглядають генерацію як задачу оптимізації.

1.1.1. Генерація на основі клітинних автоматів.

Клітинні автомати — це дискретні динамічні системи, поведінка яких визначається локальними взаємодіями між компонентами. У контексті процедурної генерації ігрових рівнів КА розглядається як сітка комірок, кожна з яких може перебувати в одному зі скінченної кількості станів (наприклад, «стіна» або «підлога»). Зміна станів відбувається синхронно в дискретні моменти часу на основі фіксованих правил, що залежать від поточного стану

клітини та станів її сусідів.

Найвідомішим прикладом клітинного автомата є гра «Життя» Джона Конвея. Проте для генерації статичних ігрових рівнів частіше використовуються модифіковані правила, спрямовані на згладжування шуму та формування чітких кластерів.

Алгоритм генерації зазвичай складається з таких етапів:

1. Ініціалізація: Ігрове поле заповнюється випадковими значеннями (білим шумом). Кожна клітина з певною ймовірністю (наприклад, 45%) стає «стіною».
2. Симуляція (Ітеративний процес): Протягом кількох ітерацій до кожної клітини застосовуються правила переходу. Найпоширенішим правилом для генерації печер є правило «4-5»:
   - Якщо клітина є «стіною» і має 4 або більше сусідів-стін, вона залишається стіною.
   - Якщо клітина є «підлогою» і має 5 або більше сусідів-стін, вона перетворюється на стіну.
   - В іншому випадку вона стає підлогою.
3. Пост-обробка: Видалення ізольованих маленьких областей (незв'язних кімнат) для забезпечення ігрової прохідності.

Для визначення «сусідів» використовуються два основні типи околів:

- Окіл фон Неймана: включає 4 сусідні клітини (зверху, знизу, зліва, справа).
- Окіл Мура: включає 8 сусідніх клітин (додаються діагональні сусіди).

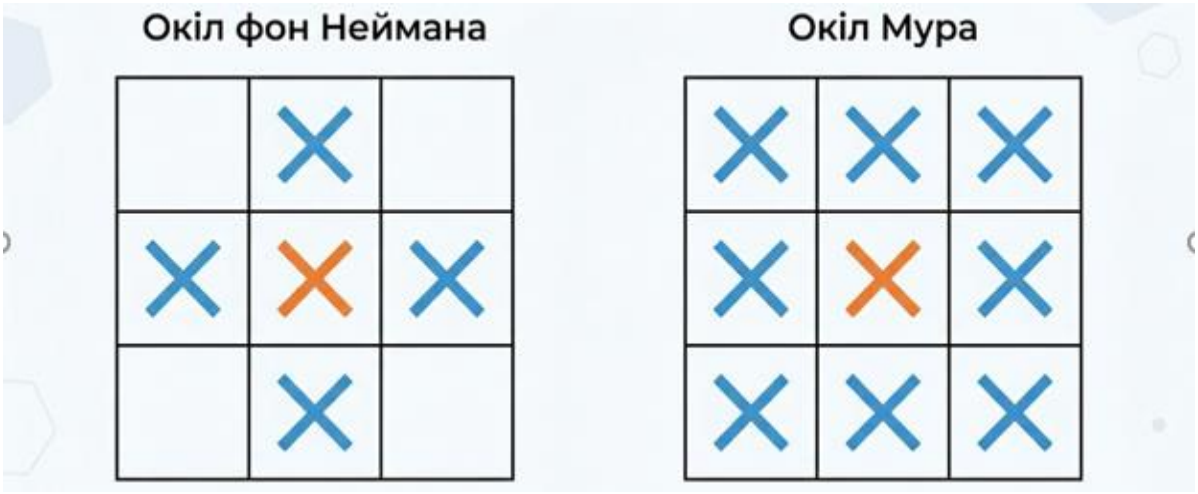

Рис.1 Типи околів у клітинних автоматах

Переваги та недоліки методу в контексті Tower Defense:

Основною перевагою клітинних автоматів є висока швидкість роботи та здатність генерувати природні, органічні форми, що нагадують печери або острови. Алгоритм є простим у реалізації та легко масштабується.

Однак суттєвим недоліком для жанру Tower Defense є відсутність гарантії зв'язності. Клітинні автомати працюють локально і не мають "глобального бачення" карти. Це часто призводить до створення ізольованих зон, куди вороги не можуть потрапити, або повного перекриття шляху від старту до фінішу. Щоб виправити це, доводиться використовувати додаткові алгоритми (наприклад, заливку потоком), що ускладнює процес генерації та перетворює його на гібридний метод. Саме неможливість контролювати довжину та складність шляху робить чисті клітинні автомати недостатньо ефективними для стратегічних ігор, де баланс є критичним.

1.1.2. Використання шуму Перліна та фрактальних алгоритмів

Шум Перліна , розроблений Кеном Перліном у 1983 році, є стандартом де-факто у процедурній генерації природних текстур та ландшафтів. На відміну від звичайного білого шуму, де значення сусідніх пікселів не

корелюють між собою (що створює ефект «снігу» на екрані телевізора), шум Перліна є різновидом градієнтного шуму. Він забезпечує плавні, когерентні переходи між значеннями, що імітує природну випадковість.

Математична основа та фрактальні властивості

В основі методу лежить генерація псевдовипадкових градієнтних векторів у вузлах решітки та інтерполяція значень між ними. Однак у чистому вигляді шум Перліна створює лише «м'які» пагорби. Для отримання складної деталізації (наприклад, кам'янистих гір або порізаної берегової лінії) застосовують фрактальні алгоритми, зокрема метод фрактального броунівського руху (fBm).

Суть підходу полягає у накладанні кількох шарів шуму (так званих «октав») один на одного:

- Перша октава задає загальні контури ландшафту (низька частота, висока амплітуда).
- Кожна наступна октава додає дрібніші деталі (підвищується частота, зменшується амплітуда).

Цей процес контролюється двома параметрами:

- Лакунарність (Lacunarity): визначає, як швидко зростає частота шуму з кожною октавою.
- Стійкість (Persistence): визначає, як швидко спадає амплітуда.

Застосування в іграх та обмеження для Tower Defense

У контексті стратегічних ігор шум Перліна та його варіації ідеально підходять для:

- Генерації карт висот.
- Розподілу біомів (ліси, пустелі, болота) залежно від вологості та температури.

- Створення візуальної варіативності тайлів «землі».

Однак, цей метод має суттєвий недолік при проектуванні ігрової механіки Tower Defense. Шум Перліна генерує безперервні поля значень, але не оперує поняттями «шлях», «вхід» чи «вихід». При спробі використати порогові значення шуму для створення стін (наприклад, «все, що вище 0.5 — це стіна»), часто утворюються замкнені «озера» або ізольовані області, недоступні для гравця.

Через відсутність гарантії зв'язності графу рівня, шум Перліна рідко використовується як єдиний метод генерації для ігор, де критично важливою є наявність чіткого маршруту для юнітів. Він частіше виступає як допоміжний інструмент для декорування карти, топологія якої вже створена іншими методами (наприклад, генетичним алгоритмом).

1.1.3. L-системи та граматичні методи генерації

L-системи (системи Ліндемаєра) — це математичний формалізм, запропонований угорським біологом Арістідом Ліндемаєром у 1968 році. Спочатку розроблені для моделювання процесу росту рослин та розвитку багатоклітинних організмів, згодом вони стали одним із фундаментальних методів процедурної генерації геометричних структур у комп'ютерній графіці та іграх.

В основі методу лежить концепція переписування рядків (string rewriting). L-система визначається як кортеж G = (V, ω, P), де:

- V (алфавіт) — множина символів, що можуть використовуватися в системі.
- ω (аксіома) — початковий рядок, з якого починається генерація.
- P (правила продукції) — набір правил, що визначають, як один символ замінюється на послідовність інших символів на кожній ітерації.

Принцип роботи та візуалізація

Процес генерації відбувається ітеративно. На кожному кроці всі символи поточного рядка одночасно замінюються згідно з правилами продукції. Отриманий в результаті довгий рядок символів інтерпретується геометрично за допомогою так званої «черепашачої графіки». Наприклад, символ "F" може означати «рухатися вперед і малювати лінію», "+" — «повернути праворуч», а "-" — «повернути ліворуч».

Завдяки рекурсивній природі, L-системи ідеально підходять для створення фрактальних структур, дерев, річкових систем та дорожніх мереж.

Граматичні методи

Розвиком ідей L-систем у контексті генерації архітектури та рівнів є просторові граматики та графові граматики. Замість одновимірних рядків вони оперують двовимірними або тривимірними формами та графами.

Ці методи використовують підхід «розділяй та володарюй»: ігровий простір розбивається на функціональні зони згідно з правилами (наприклад, «Кімната» → «Вхід» + «Коридор» + «Зала з пастками»). Це дозволяє створювати складні, ієрархічно структуровані підземелля або міста, які мають логічну будову, на відміну від хаотичних печер, згенерованих клітинними автоматами.

Переваги та недоліки для Tower Defense

Для жанру Tower Defense L-системи та стохастичні (імовірнісні) граматики є потужним інструментом для генерації шляхів ворогів. Вони дозволяють створювати карти з природними розгалуженнями, де вороги можуть рухатися кількома маршрутами одночасно.

Однак, використання граматичних методів пов'язане з проблемою контролю замкненості та перетинів. Класичні L-системи є майстрами

створення деревоподібних структур (без циклів), тоді як ігрові рівні часто вимагають наявності циклів (можливість обійти перешкоду) або чіткого з'єднання точки спавну з базою. Налаштування правил продукції таким чином, щоб гарантувати з'єднання двох конкретних точок на карті, є нетривіальною математичною задачею, що часто змушує розробників шукати інші підходи, такі як еволюційні алгоритми.

1.1.4. Алгоритм колапсу хвильової функції

Алгоритм колапсу хвильової функції (Wave Function Collapse), запропонований Максимом Гуміним у 2016 році, є новітнім підходом до процедурної генерації, натхненним концепціями квантової механіки та теорії обмежень. На відміну від імперативних методів, де алгоритм "малює" рівень крок за кроком, WFC вирішує задачу задоволення обмежень, генеруючи зображення або карту, яка локально подібна до вхідного зразка.

Теоретична основа та принцип роботи

Назва алгоритму походить від квантово-механічного явища, коли квантова система, що перебуває в суперпозиції кількох станів, при спостереженні редукується до одного визначеного стану. У контексті генерації рівнів це інтерпретується наступним чином:

1. Суперпозиція: Спочатку кожна комірка ігрової сітки (тайл) перебуває в стані суперпозиції, тобто вона потенційно може бути будь-яким тайлом з набору (трава, дорога, стіна, вода).
2. Ентропія: Для кожної комірки обчислюється ентропія Шенона, яка в даному випадку характеризує міру невизначеності. Комірка, що може бути будь-яким тайлом, має максимальну ентропію; комірка, стан якої вже визначено, має нульову ентропію.
3. Спостереження (Observation): Алгоритм знаходить комірку з

найменшою ненульовою ентропією (тобто ту, де найменше варіантів вибору). Ця комірка "колапсує" — алгоритм випадковим чином обирає один із можливих станів, враховуючи вагові коефіцієнти тайлів.

4. Поширення обмежень (Propagation): Після колапсу однієї комірки інформація про зміну поширюється на сусідні комірки. Якщо, наприклад, обрано тайл "дорога", то сусідні комірки більше не можуть бути "водою" (якщо правила забороняють прямий перехід). Список можливих станів для сусідів скорочується, що, у свою чергу, може запустити ланцюгову реакцію змін для їхніх сусідів.

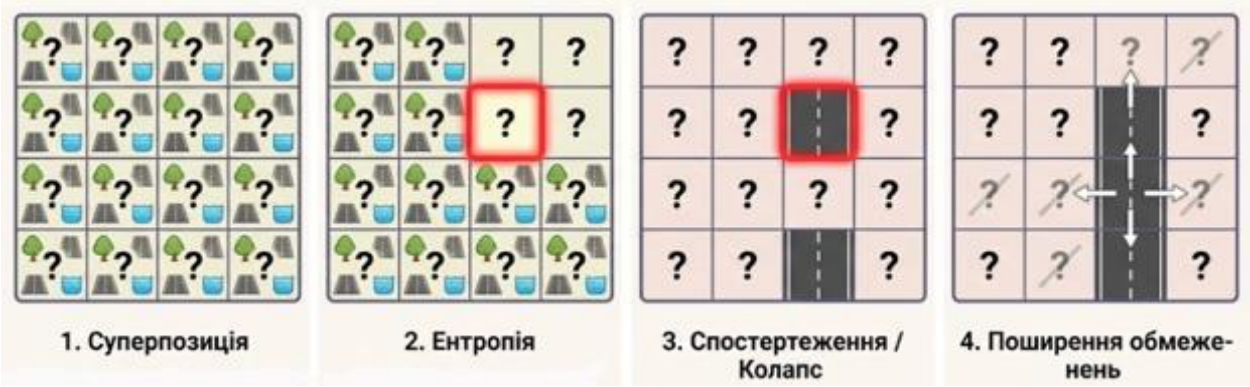

Рис. 2 Візуалізація ітераційного процесу алгоритму WFC

Цей цикл повторюється доти, доки всі комірки не будуть визначені (ентропія всієї системи дорівнює нулю) або доки алгоритм не зайде в глухий кут (протиріччя), що вимагатиме перезапуску.

Застосування та обмеження в Tower Defense

WFC є надзвичайно ефективним для створення візуально коректних структур. У жанрі Tower Defense він дозволяє генерувати складні лабіринти, де всі дороги гарантовано з'єднані між собою, а текстури плавно переходять одна в одну.

Однак, алгоритм має суттєві недоліки для стратегічних ігор:

- Локальність правил: WFC чудово дотримується правил сумісності

сусідніх тайлів (наприклад, "дорога має продовжуватися дорогою"), але не має вбудованого розуміння глобальних структур. Він не може гарантувати, що згенерована дорога обов'язково приведе від точки спавну до бази гравця.

- Обчислювальна складність: Процес поширення обмежень може бути ресурсомістким для великих карт.
- Ризик нерозв'язності: Існує ймовірність, що алгоритм дійде до стану, де для певної комірки не залишиться жодного допустимого варіанту, що вимагає механізмів відкату (backtracking) або повного перезапуску генерації.

1.1.5. Еволюційні обчислення та генетичні алгоритми

Еволюційні обчислення — це напрямок штучного інтелекту, що вивчає алгоритми глобальної оптимізації, натхненні процесами біологічної еволюції. На відміну від класичних детермінованих методів, які шукають точний розв'язок за чіткою формулою, еволюційні методи використовують стохастичний (ймовірнісний) підхід для пошуку достатньо якісного рішення у великому просторі варіантів.

Найпоширенішим класом еволюційних методів є генетичні алгоритми (Genetic Algorithms), розроблені Джоном Голландом у 1975 році. Генетичний алгоритм — це евристичний алгоритм пошуку, що використовується для вирішення задач оптимізації та моделювання шляхом випадкового підбору, комбінування та варіації шуканих параметрів.

Базові поняття та термінологія

Таблиця 2

Відображення термінів еволюційних обчислень на предметну область Tower Defense

Еволюційний термін	**Загальне визначення**	**Проєкція на гру Tower Defense**
Індивід (Особина)	Окреме рішення задачі оптимізації.	Одна конкретна згенерована карта (рівень) гри.
Популяція	Набір індивідів, які існують одночасно на певній ітерації.	Множина згенерованих карт-кандидатів, з яких алгоритм обирає найкращу.
Хромосома / Генотип	Кодоване представлення індивіда (набір даних).	Двовимірний масив (матриця) цілих чисел, де кожне число кодує тип тайла (стіна, вода, пустота).
Фенотип	Фізичне або візуальне відображення хромосоми.	Реальний візуалізований рівень з відмальованими текстурами доріг, гір та озер на екрані гравця.
Пристосованість	Числове значення, що характеризує якість індивіда.	Комплексна математична оцінка рівня (довжина шляху ворогів, кількість поворотів, штрафи за непрохідність).

У контексті процедурної генерації ігрового контенту (PCG) основні поняття генетичного алгоритму інтерпретуються наступним чином показаним у таблиці 2.

Етапи роботи алгоритму

Робота генетичного алгоритму є циклічною і складається з послідовності кроків, що забезпечують еволюційне вдосконалення популяції рішень. Узагальнену логічну структуру цього процесу наведено на рисунку 1.3.

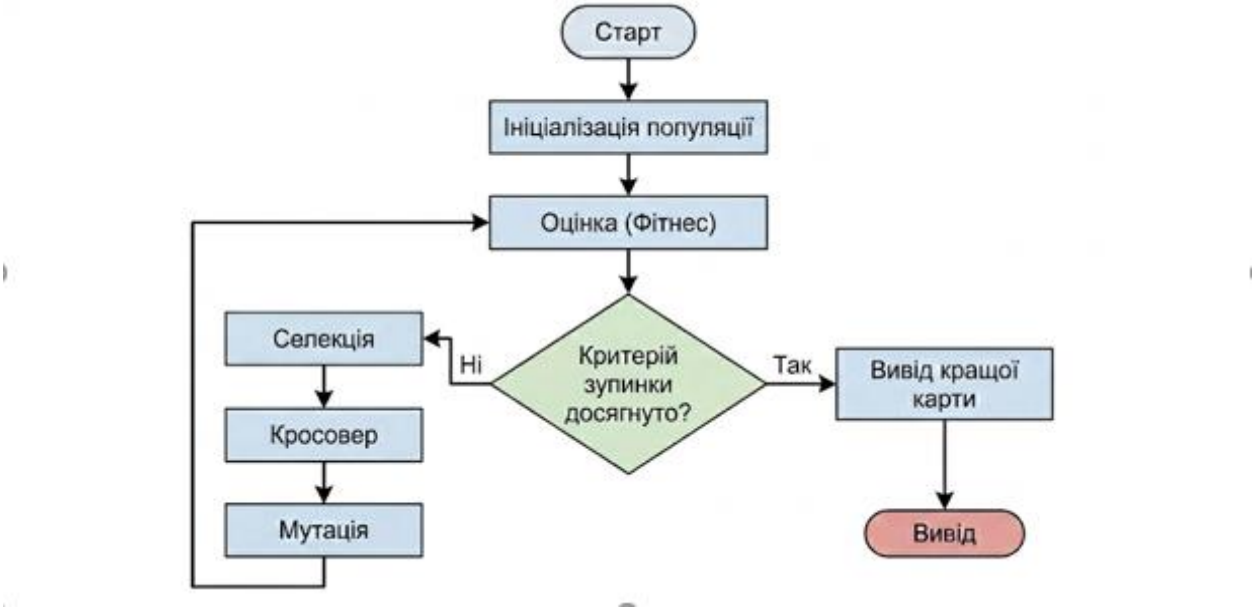

Рис. 3. Блок-схема роботи генетичного алгоритму

Цикл еволюції включає наступні кроки:

1. Ініціалізація: Створюється початкова популяція (зазвичай випадковим чином). Це набір "сирих", часто хаотичних рівнів.
2. Оцінка: Для кожного індивіда в популяції обчислюється значення функції пристосованості. Ця функція є критично важливою, оскільки саме вона визначає, які рівні вважаються "хорошими".
3. Селекція: З популяції відбираються батьківські пари для створення нового покоління. Існує кілька методів селекції, наприклад, турнірний відбір (вибираються кілька випадкових індивідів і перемагає найкращий) або методом рулетки (ймовірність вибору пропорційна пристосованості).
4. Кросовер: Батьківські хромосоми обмінюються частинами інформації, створюючи нащадків. У контексті генерації карт це може виглядати як з'єднання лівої половини карти одного батька з правою половиною іншого. Мета оператора — об'єднати вдалі рішення (наприклад, гарний початок шляху з однієї карти та складний фініш з іншої).

5. Мутація: В хромосоми нащадків з невеликою ймовірністю вносяться випадкові зміни (наприклад, зміна типу одного тайла з "трави" на "стіну"). Мутація запобігає передчасній збіжності алгоритму до локального оптимуму та забезпечує різноманітність популяції.
6. Формування нової популяції: Нащадки замінюють батьків (повністю або частково), і цикл повторюється до виконання умов зупинки (досягнення необхідної якості рівня або вичерпання ліміту часу).

Переваги для задач PCG

Головною перевагою генетичних алгоритмів для генерації рівнів є їхня гнучкість. Вони не вимагають від розробника описувати чіткий алгоритм побудови "ідеального рівня". Замість цього розробник описує критерії якості, а алгоритм сам знаходить спосіб задовольнити ці критерії, часто пропонуючи неочікувані та креативні рішення, які важко спроєктувати вручну. Це робить GA ідеальним інструментом для задач, де важко формалізувати правила побудови, але легко оцінити готовий результат.

1.1.6. Графові методи та алгоритми пошуку шляху

У контексті розробки ігор, особливо жанру Tower Defense, ігровий простір дискретизується та розглядається як граф, де вузлами (вершинами) виступають ігрові тайли (клітинки), а ребрами — можливість переміщення між сусідніми тайлами. Це дозволяє звести задачу навігації ворожих юнітів та перевірку валідності згенерованого рівня до класичних задач теорії графів: пошуку найкоротшого шляху та перевірки зв'язності графа.

Для процедурної генерації контенту алгоритми пошуку шляху виконують подвійну функцію:

1. Валідація: Перевірка, чи існує хоча б один шлях від точки появи ворогів до бази гравця.

2. Оцінка: Вимірювання характеристик шляху (довжини, звивистості) для визначення складності рівня.

Алгоритм Дейкстри

Алгоритм Дейкстри, запропонований у 1959 році, є фундаментальним методом пошуку найкоротшого шляху у зваженому графі без ребер з від'ємною вагою. Він працює за принципом «хвильового поширення», поступово досліджуючи всі доступні вузли, починаючи від стартового, та обчислюючи мінімальну вартість переміщення до кожного з них.

У Tower Defense цей алгоритм гарантує знаходження оптимального шляху, проте він є ресурсомістким, оскільки досліджує граф рівномірно у всіх напрямках, навіть тих, що не ведуть до цілі.

Алгоритм A* (A-Star)

Алгоритм A* є евристичною модифікацією алгоритму Дейкстри і вважається «золотим стандартом» у ігровій індустрії. Його ключовою відмінністю є використання евристичної функції h(n), яка оцінює приблизну відстань від поточного вузла до цільового. Функція вартості для вузла n виглядає як:

$f(n) = g(n) + h(n)$

де g(n) — точна вартість шляху від старту до n, а h(n) — евристична оцінка відстані до фінішу (наприклад, Чебишова відстань для прямокутної сітки).

Завдяки цьому A* спрямовує пошук у бік цілі, значно скорочуючи кількість перевірених вузлів. У задачах PCG алгоритм A* найчастіше використовується у функції пристосованості (fitness function) для швидкого обчислення довжини шляху на згенерованій карті.

Векторні поля

Для ігор жанру TD, де одночасно можуть переміщуватися сотні юнітів, індивідуальний розрахунок шляху (A*) для кожного ворога є неефективним. Альтернативою є використання векторних полів.

Суть методу полягає у попередньому розрахунку карти напрямків для всього ігрового поля. Для кожного прохідного тайла обчислюється вектор, що вказує на сусідній тайл, який веде до цілі найкоротшим шляхом. Це дозволяє всім агентам миттєво отримувати напрямок руху, просто зчитуючи значення з клітинки, на якій вони знаходяться, без необхідності повторного запуску алгоритму пошуку.

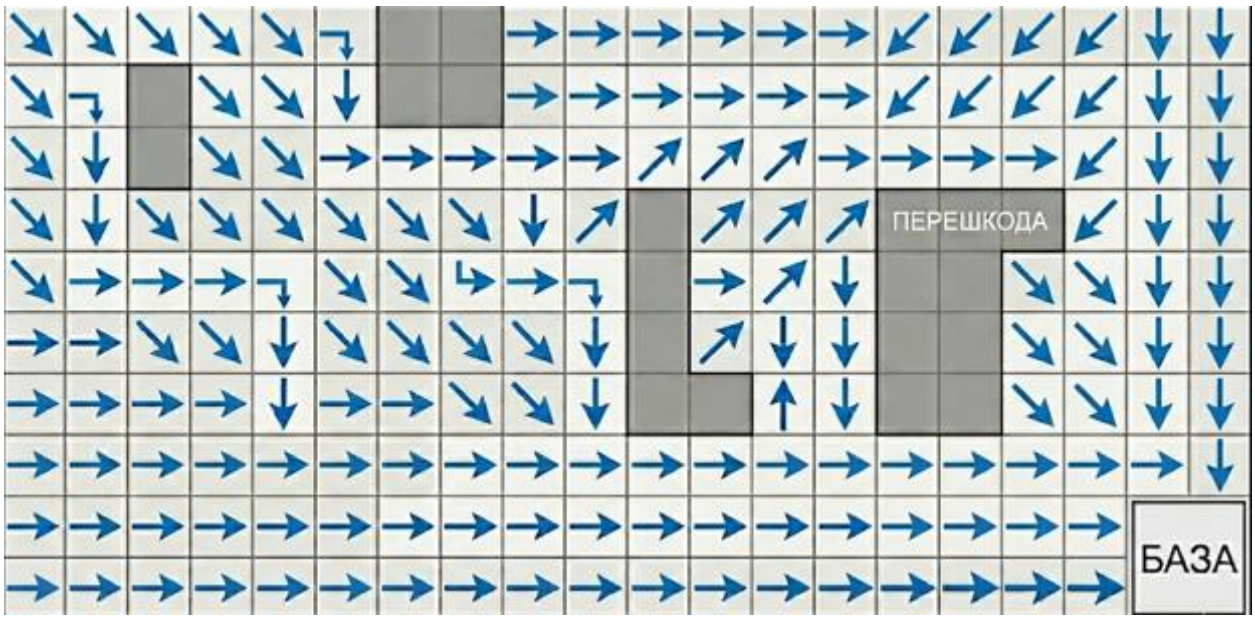

Рис.4. Приклад візуалізації векторного поля

У контексті генерації рівнів векторні поля дозволяють легко виявляти «мертві зони» (тайли, з яких неможливо дістатися до бази) та оцінювати загальну доступність ігрового простору.

Застосування в еволюційних алгоритмах

У даній дипломній роботі графові методи інтегруються в генетичний алгоритм на етапі оцінки популяції. Згенерований рівень (фенотип)

перетворюється на граф, після чого запускається пошук шляху (наприклад, A*). Якщо шлях відсутній, індивід отримує великий штраф до пристосованості (або видаляється). Якщо шлях існує, його довжина використовується як один із критеріїв оптимізації: алгоритм намагається максимізувати довжину шляху (щоб ускладнити гру), зберігаючи при цьому обмеження на розмір карти.

1.2 Огляд основних здобутків наукових досліджень у генерації рівнів жанру

Проблематика автоматичної генерації рівнів для ігор жанру TD займає окрему нішу в дослідженнях штучного інтелекту, оскільки цей жанр висуває жорсткі вимоги до топології ігрового простору. На відміну від шутерів чи RPG, де "відкритість" світу є перевагою, у TD критично важливим є наявність чітко визначеного, валидного шляху для ворогів та збалансованих зон для розміщення оборонних споруд.

Наукові дослідження у цій галузі можна умовно розділити на три основні напрямки: геометрична оптимізація шляху, пошукова генерація (Search-based PCG) та симуляційне оцінювання балансу.

Дослідження оптимізації шляху та лабіринтів

Ранні роботи у сфері процедурної генерації TD фокусувалися на створенні максимально довгих шляхів на обмеженій площині. Це класична задача теорії графів, відома як "Longest Path Problem" (задача про найдовший шлях), яка є NP-повною для загальних графів. Дослідники, такі як П. Аернс, пропонували використовувати модифіковані алгоритми генерації лабіринтів (наприклад, алгоритм Прима або рекурсивний backtracker) для створення карт, де шлях ворога займає максимальну площу карти. Однак, суто геометричний підхід має недолік: він створює "нудні" змійки, які легко проходяться гравцем за одну й ту ж стратегію, і не враховує місця для розміщення веж різного радіусу дії.

Розвиток Пошукової генерації

Значний прорив у генерації рівнів TD пов'язаний з роботами Дж. Тогеліуса та його колег, які запропонували концепцію Search-based Procedural

Content Generation. Суть підходу полягає у використанні еволюційних алгоритмів для пошуку "ідеального" рівня у просторі можливих варіантів. У їхніх дослідженнях рівень TD розглядається не як набір тайлів, а як фенотип, що еволюціонує. Ключовим досягненням стала розробка багатоцільових фітнес-функцій, які одночасно оптимізують кілька конфліктуючих параметрів:

1. Довжина шляху: Чим довший шлях, тим більше часу у гравця на знищення ворогів.
2. Кількість будівельних слотів: Карта повинна мати достатньо місця для стратегічного розміщення веж.
3. Варіативність: Шлях повинен мати повороти та розгалуження, щоб стимулювати використання веж з різними типами атаки (радіальні, лінійні).

Симуляційний підхід до оцінки складності

Сучасні дослідження (зокрема роботи А. Зикова та С. Сміта) відходять від статичної оцінки геометрії карти до динамічної симуляції. Замість того, щоб просто вимірювати довжину шляху в клітинках, алгоритм генерації запускає "віртуального гравця" (бота), який намагається пройти згенерований рівень, використовуючи алгоритми пошуку шляху (A*) та жадібні стратегії забудови. Цей підхід дозволяє оцінювати рівень за критерієм “Час виживання”. Якщо бот перемагає надто швидко — рівень занадто легкий; якщо програє миттєво — занадто складний. Генетичний алгоритм використовує результати цих симуляцій для налаштування складності карти під конкретного гравця.

Використання композиційних методів

Окремий напрямок досліджень (В. ван дер Лінден) пропонує використовувати бібліотеки попередньо створених людиною фрагментів карт.

Генетичний алгоритм у цьому випадку не будує карту по клітинці, а комбінує ці патерни, як пазл. Це дозволяє гарантувати високу візуальну якість та локальну логіку (наприклад, красиві мости чи перехрестя), покладаючи на алгоритм лише задачу глобального з'єднання цих шматків у цікавий маршрут.

Невирішені проблеми

Попри значні досягнення, більшість існуючих рішень стикаються з проблемою продуктивності. Симуляція кожної карти в популяції (програвання гри ботом) вимагає значних обчислювальних ресурсів, що ускладнює генерацію в реальному часі (під час завантаження гри). Крім того, залишається відкритим питання генерації карт, які б спонукали гравця до використання конкретних стратегій, а не просто були складними.

Саме ці виклики зумовлюють актуальність даної дипломної роботи, яка пропонує оптимізований підхід до використання генетичних алгоритмів для створення збалансованих рівнів Tower Defense без надмірних обчислювальних витрат.

1.3 Актуальність дослідження

На сучасному етапі розвитку індустрії комп'ютерних ігор спостерігається стрімке зростання вимог користувачів до обсягу та якості ігрового контенту. Сучасні ігрові проєкти, особливо в жанрах стратегій та TD, потребують десятків, а іноді й сотень годин геймплею, щоб залишатися конкурентоспроможними на ринку. Це створює суттєве навантаження на розробників: ручне створення кожного рівня, його балансування та тестування є надзвичайно трудомістким і дорогім процесом, що значно підвищує собівартість кінцевого продукту та подовжує цикл розробки.

Актуальність теми дослідження зумовлена необхідністю вирішення протиріччя між зростаючою потребою у великій кількості різноманітного, збалансованого ігрового контенту та обмеженими ресурсами (часовими та людськими) на його створення.

Впровадження методів PCG дозволяє автоматизувати процес створення рівнів, забезпечуючи так звану реіграбельність — здатність гри залишатися цікавою при багаторазовому проходженні завдяки унікальності ігрових ситуацій. Однак, існуючі методи генерації, такі як шум Перліна або клітинні автомати, які ефективно використовуються для створення ландшафтів, виявляються недостатньо ефективними для жанру Tower Defense.

Специфіка TD полягає в жорстких топологічних обмеженнях:

1. Обов'язкова наявність зв'язного шляху від точки появи ворогів до бази гравця.
2. Необхідність балансу між довжиною шляху та кількістю місць для розміщення оборонних споруд.
3. Забезпечення варіативності стратегій проходження.

Проста рандомізація (випадкове розміщення перешкод) у більшості випадків призводить до створення неіграбельних ("мертвих") карт або тривіальних лабіринтів, що негативно впливає на досвід користувача. Тому

виникає науково-практична потреба у розробці таких методів генерації, які б не просто створювали випадкові структури, а цілеспрямовано шукали оптимальні ігрові конфігурації.

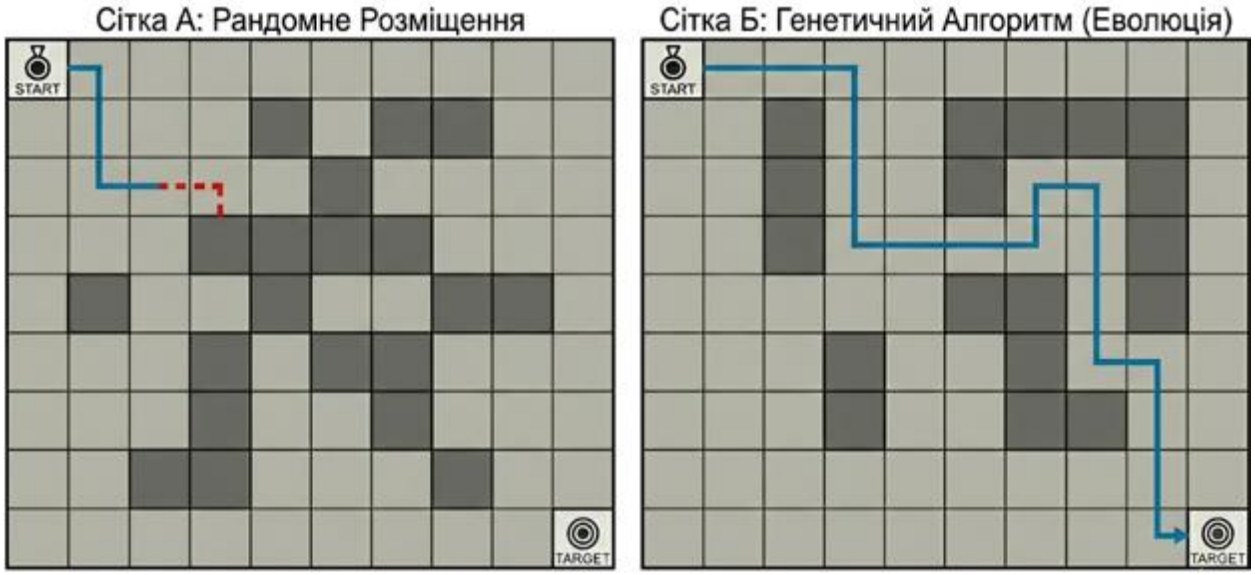

Рис.5. Порівняння підходів до генерації.

Саме застосування GA є актуальним рішенням цієї проблеми. Еволюційний підхід дозволяє розглядати створення рівня як задачу багатофакторної оптимізації. Використання фітнес-функції дає змогу формалізувати поняття "цікавий рівень" (довжина шляху, складність, кількість поворотів) і змусити алгоритм еволюціонувати у бік покращення цих показників.

Таким чином, розробка методу процедурної генерації рівнів Tower Defense на основі генетичних алгоритмів має важливе значення для:

- Економічної ефективності: Зменшення витрат на ручний левел-дизайн.
- Технологічного розвитку: Створення адаптивних систем, здатних підлаштовувати складність гри під навички гравця в реальному часі.
- Ігрового досвіду: Забезпечення гравців нескінченним потоком унікальних та збалансованих рівнів.

Дослідження у цьому напрямку дозволяють перейти від створення

статичних ігрових світів до динамічних систем, що самостійно розвиваються, що є одним із ключових трендів сучасної GameDev-індустрії.

Висновки по розділу 1.

У першому розділі дипломної роботи проведено комплексний аналіз теоретичних засад та існуючих підходів до процедурної генерації контенту в ігровій індустрії, зокрема для стратегічних ігор жанру TD.

1. Проаналізовано основні методи процедурної генерації, такі як клітинні автомати, шум Перліна, L-системи та алгоритм колапсу хвильової функції. Встановлено, що хоча ці методи є ефективними для створення візуальних ландшафтів та текстур, вони мають суттєві обмеження при генерації ігрових рівнів зі суворими топологічними вимогами. Головним недоліком класичних конструктивних методів є відсутність гарантії зв'язності графа (наявності проходу від старту до фінішу) та складність контролю ігрового балансу.
2. Визначено специфіку жанру Tower Defense, яка полягає у необхідності створення мап, що поєднують довгий, звивистий шлях для ворогів із достатньою кількістю місць для розміщення оборонних споруд. Доведено, що проста рандомізація не здатна забезпечити стабільну якість та реіграбельність контенту.
3. Обґрунтовано доцільність використання еволюційних обчислень, зокрема генетичних алгоритмів, як основного інструменту для вирішення поставленої задачі. Показано, що підхід "генерація та перевірка", властивий генетичним алгоритмам, дозволяє трансформувати задачу створення рівня в задачу багатокритеріальної оптимізації. Використання фітнес-функції дає змогу гнучко налаштовувати критерії якості (довжину шляху, складність, варіативність) і отримувати гарантовано прохідні рівні.
4. Сформульовано напрямок подальших досліджень, який полягає у розробці та програмній реалізації модифікованого генетичного алгоритму. Для досягнення мети роботи необхідно розробити ефективну схему кодування ігрового поля (хромосому), адаптувати оператори

кросоверу та мутації для роботи з двіймірною сіткою, а також інтегрувати алгоритм пошуку шляху (A*) у процес оцінки пристосованості індивідів.

РОЗДІЛ 2.
СИСТЕМНИЙ АНАЛІЗ ТА ОБҐРУНТУВАННЯ ПРОБЛЕМИ

2.1. Постановка та обґрунтування проблеми.

Жанр відеоігор Tower Defense традиційно базується на стратегічному плануванні, де ключовим елементом ігрового процесу є просторова архітектура ігрового рівня: конфігурація шляхів пересування ворогів, розташування точок їхньої появи (спавну), захисної бази гравця та доступних зон для будівництва оборонних споруд. Від якості, варіативності та збалансованості дизайну мапи безпосередньо залежить залученість гравця та загальний успіх програмного продукту.

Традиційний підхід до розробки ігрового контенту в цьому жанрі передбачає ручне проєктування рівнів. Хоча цей метод дозволяє геймдизайнеру повністю контролювати передбачуваний ігровий досвід, він має низку суттєвих недоліків. У міру розвитку проєкту та збільшення його масштабів потреба у нових рівнях зростає експоненціально. Це призводить до значних витрат часу та ресурсів розробників на створення та багаторазове тестування кожної окремої мапи. Крім того, при ручному проєктуванні людський фактор ускладнює об’єктивну оцінку балансу: рівні можуть виявитися статистично непрохідними або, навпаки, не забезпечувати належного виклику для гравця.

Для вирішення проблеми масштабування контенту в ігровій індустрії широко застосовується процедурна генерація (Procedural Content Generation, PCG). Однак використання базових методів PCG, таких як алгоритми випадкового блукання або генерація на основі шуму, стикається з серйозною перешкодою в контексті жанру TD. Класичні процедурні алгоритми не здатні самостійно аналізувати семантику згенерованого простору. Випадково створений шлях може виявитися занадто коротким, перекривати зони для забудови або мати топологію, яка робить рівень апріорі непрохідним.

Таким чином, формулюється ключова науково-практична проблема: необхідність автоматизації процесу створення ігрових мап для жанру Tower Defense з одночасним забезпеченням їхньої валідності (гарантованої наявності шляху), іграбельності та відповідності заданим критеріям складності.

Обґрунтуванням вирішення цієї проблеми є застосування еволюційного підходу, зокрема апарату генетичних алгоритмів. На відміну від "сліпої" рандомізації, генетичні алгоритми здатні не лише генерувати початковий пул рівнів, але й ітеративно оптимізувати їх шляхом мутацій та схрещування, відсіюючи невдалі варіанти за допомогою математичної моделі оцінки - фітнес-функції.

Реалізація такої інтелектуальної системи генерації в рамках мобільного застосунку дозволить автоматично створювати нескінченну кількість унікальних, попередньо збалансованих рівнів. Це фундаментально вирішує проблему нестачі контенту, підвищує показник реіграбельності та мінімізує витрати на ручний геймдизайн, що підтверджує актуальність та доцільність обраного напрямку дослідження.

2.2. Методи вирішення проблеми

Розробка інтелектуальної системи генерації рівнів вимагає переходу від абстрактних ігрових концепцій до строгих математичних та алгоритмічних структур. Для досягнення поставленої мети - автоматичного створення збалансованих мап для гри жанру Tower Defense - необхідно вирішити два ключові завдання: побудувати формальну концептуальну модель ігрового простору та обрати ефективний алгоритм для оптимізації цього простору. У даному підрозділі детально розглядається процес математичної формалізації ігрової мапи та формуються базові обмеження системи.

2.2.1. Формалізація процесу створення ігрових мап та розробка концептуальної моделі

Основою для будь-якої системи процедурної генерації є строга формальна модель цільового об’єкта. У контексті жанру Tower Defense ігровий простір (мапа) є дискретним і найчастіше подається у вигляді двовимірної сітки. Такий підхід дозволяє ефективно працювати з координатами об'єктів та спрощує розрахунки для алгоритмів пошуку шляху та штучного інтелекту.

Концептуальна модель ігрового поля формалізується як двовимірна матриця M розмірністю W x H, де W - ширина мапи, а H - її висота. Кожен елемент цієї матриці $c_{x,y}$(де $0 \leq x < W$, $0 \leq y < H$) відповідає одній ігровій клітинці (тайлу).

Для опису семантики простору вводиться дискретна множина станів S, яких може набувати кожна клітинка:

$$c_{x,y} \in \{s_{empty}, s_{path}, s_{spawn}, s_{base}, s_{obstacle}\}$$

Кожен із цих станів визначає фізичні та логічні властивості тайлу:

Таблиця 3

Дискретні стани елементів матриці ігрового простору

Математичне позначення	**Назва стану**	**Логічне та фізичне призначення в ігровій механінці**
S_{empty}	Базова порожня клітинка	Зона, повністю доступна гравцеві для розміщення захисних веж.
c	Елемент маршруту	Клітинка, якою пересуваються вороги. Будівництво на ній суворо заборонено.
S_{spawn}	Точка генерації (Старт)	Точка входу ворожих одиниць на рівень.
S_{base}	Цільова точка (База)	Об'єкт, до якого прямують вороги і який гравець повинен захистити.
$S_{obstacle}$	Непрохідна зона (Перешкода)	Елементи ландшафту (гори, вода), де неможливе ні переміщення ворогів, ні будівництво.

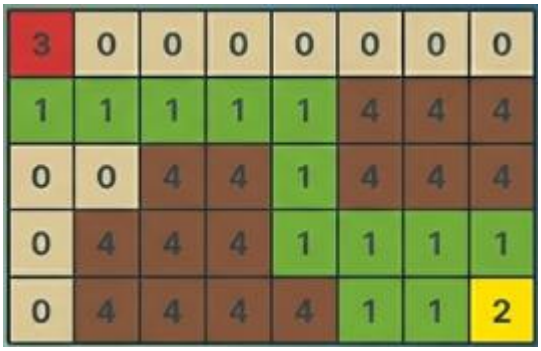

Рис.6 Візуалізація концептуальної моделі ігрового рівня на основі двовимірної сітки.

На рисунку 6 можемо зазначити, що 0 відображає стан s_{empty}, 1- s_{path}, 2 - s_{spawn}, 3 - s_{base}, 4 - $s_{obstacle}$.

Ключовим завданням процедурної генерації в Tower Defense є створення валідного маршруту. З математичної точки зору, маршрут ворогів P можна формалізувати як упорядковану множину координат клітинок:

$$P = \{(c, y_0), (x_1, y_1), \ldots, (x_n, y_n)\}$$

Для того, щоб згенерована мапа вважалася іграбельною (валідною), концептуальна модель накладає на множину P ряд строгих топологічних обмежень:

– Граничні умови: Маршрут повинен починатися в точці генерації та закінчуватися на базі:

$c_{x0,y0} = s_{spawn}$ та $c_{xn,yn} = s_{base}$

– Унікальність ключових точок: На матриці M має існувати рівно один елемент s_{spawn} та рівно один s_{base}.

– Безперервність (зв'язність): Кожен наступний крок маршруту повинен здійснюватися в сусідню клітинку. Використовуючи окіл фон Неймана (рух по горизонталі та вертикалі), ця умова записується як:

$$|x_i - x_{i-1}| + |y_i - y_{i-1}| = 1, \ \forall \in \{1, \ldots, n\}$$

– Відсутність самоперетинів: Для класичної механіки Tower Defense маршрут не повинен перетинати сам себе, тобто всі елементи множини P мають бути унікальними парами координат.

Виходячи з цього, завдання розробки генератора рівнів зводиться до алгоритмічного пошуку такої конфігурації матриці M та відповідної множини P, яка б не лише задовольняла вищезазначені обмеження валідності, але й мала оптимальні ігрові характеристики (відповідну довжину, кількість поворотів тощо). Ця математична формалізація створює надійне підґрунтя для подальшого застосування еволюційних методів оптимізації.

2.2.2. Еволюційний підхід до генерації: специфіка налаштування генетичного алгоритму та оціночної функції

Задача побудови оптимального ігрового рівня для жанру Tower Defense відноситься до класу NP-складних задач комбінаторної оптимізації. Оскільки простір можливих станів двовимірної сітки зростає факторіально зі збільшенням її розмірів, використання класичних методів повного перебору або детермінованих алгоритмів пошуку (таких як алгоритм Дейкстри або A*) для безпосередньої *генерації* є недоцільним. Детерміновані алгоритми здатні знайти найкоротший шлях, однак у жанрі TD найкоротший шлях є найменш бажаним з точки зору геймдизайну. З огляду на це, найбільш ефективним математичним інструментом для вирішення поставленої проблеми є застосування генетичного алгоритму (ГА) — стохастичного евристичного методу, що базується на принципах популяційної генетики та еволюційної біології.

Кодування рішень (Генотип та Фенотип)

Для імплементації генетичного алгоритму необхідно розробити метод відображення фізичного ігрового простору (фенотипу) у структурований набір даних (генотип). У розробленій системі кожна потенційна конфігурація ігрового рівня (особина) кодується у вигляді хромосоми C.

Найбільш ефективним підходом для маршрутизації на сітці є векторне кодування кроків. Хромосома C подається як одновимірний масив генів g_i, де кожен ген відповідає вектору напрямку переміщення на один крок від поточної клітинки до сусідньої:

$$C = \{g_1, g_2, \ldots, g_n\}$$

де $g_i \in$ { (0, -1), (1, 0), (0, 1), (-1, 0) }, що відповідає напрямкам «вгору», «праворуч», «вниз» та «ліворуч» відповідно. Кількість генів n у хромосомі визначає максимальну дозволену довжину маршруту на рівні.

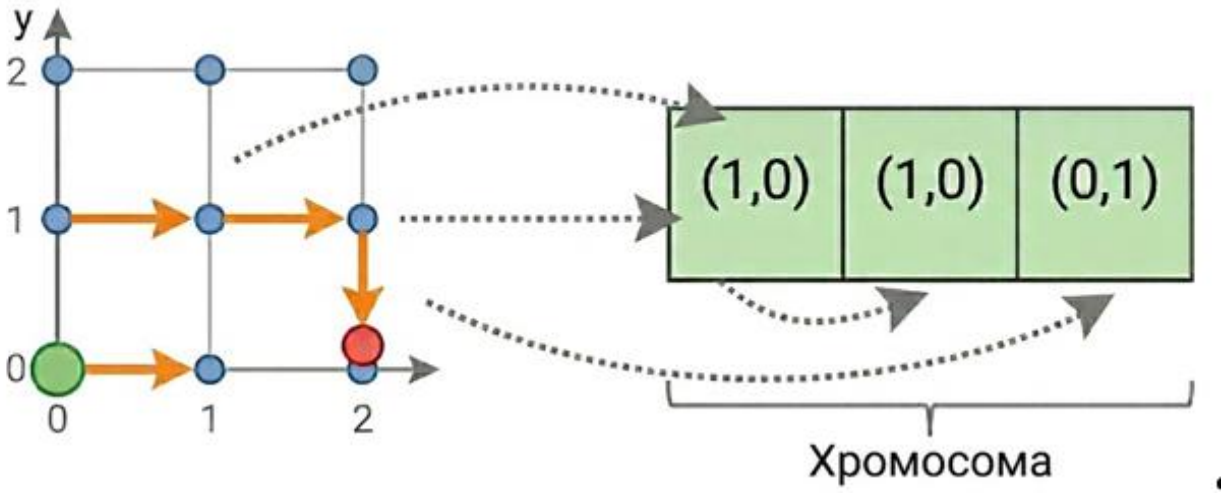

Рис. 7 Принцип кодування маршруту у хромосому.

Формування початкової популяції

Процес еволюції розпочинається з ініціалізації нульового покоління популяції P(0). Особини генеруються за допомогою методу обмеженого випадкового блукання. Алгоритм починає побудову маршруту з координати s_{spawn} і на кожному кроці випадковим чином обирає наступний напрямок g_i, уникаючи виходу за межі матриці M. Оскільки початкова генерація є стохастичною, більшість особин у нульовому поколінні матимуть критичні недоліки (наприклад, глухі кути або недосягнення бази s_{base}). Проте саме ця варіативність створює необхідне генетичне різноманіття для подальшої оптимізації.

Математична модель фітнес-функції

Центральним ядром генетичного алгоритму, що визначає напрямок еволюційного пошуку, є функція пристосованості (фітнес-функція) F(C). Вона виконує роль "навколишнього середовища", оцінюючи якість кожної хромосоми. Оскільки задача генерації рівня є багатокритеріальною, фітнес-функція формується як лінійна комбінація кількох суб-функцій з відповідними ваговими коефіцієнтами:

$$F(C) = w_1 \cdot f_{len}(C) + w_2 \cdot f_{turn}(C) - w_3 \cdot f_{pen}(C) - w_4 \cdot f_{dist}(C)$$

Детальний аналіз компонентів цільової функції:

Таблиця 4

Детальний аналіз компонентів цільової функції

Компонент функції	**Математичний зміст**	**Вплив на ігровий баланс**	**Напрямок оптимізації**
Довжина ($f_{len}(C)$)	Кількість прохідних тайлів маршруту	Збільшує час виживання гравця та площу для забудови веж	Максимізація
Складність ($f_{turn}(C)$)	Кількість поворотів (змін вектора напрямку)	Змушує гравця тактично розміщувати вежі з радіальним ураженням	Максимізація
Штрафи ($f_{pen}(C)$)	Кількість виходів за межі, зіткнень зі стінами та самоперетинів	Жорстко відсіює невалідні (непрохідні) рівні з популяції	Мінімізація
Відстань ($f_{dist}(C)$	Евристична Чебишова відстань до цілі	Направляє "сліпі" маршрути в бік бази на ранніх етапах еволюції	Мінімізація

– Функція довжини маршруту $f_{len}(C)$: Оцінює загальну кількість валідних кроків. З погляду балансу Tower Defense, довший шлях дає гравцеві більше часу на реакцію та площі для забудови. Ця функція максимізується.

– Функція складності геометрії $f_{turn}(C)$: Підраховує кількість змін напрямку вектора швидкості (поворотів). Математично визначається як кількість переходів, де $g_i \neq g_{i+1}$. Звивисті шляхи є пріоритетними, оскільки дозволяють розміщувати захисні вежі з максимальним покриттям кількох ділянок шляху одночасно.

– Функція штрафу за порушення топології $f_{pen}(C)$: Критично важливий компонент для відсіювання невалідних рішень. Штрафні бали

нараховуються за:

- Самоперетини маршруту (якщо координати кроку k збігаються з координатами будь-якого попереднього кроку j < k).
- Зіткнення з перешкодами (стан клітинки $s_{obstacle}$).
- Вихід за межі ігрової матриці M.

– Евристична функція відстані $f_{dist}(C)$: Розраховує Чебишову відстань від кінцевої точки згенерованого маршруту до цільової бази s_{base}. Ця функція допомагає алгоритму на ранніх етапах еволюції "направляти" «сліпі» маршрути у бік цілі, навіть якщо вони її ще не досягли:

$$f_{dist}(C) = |x_{end} - x_{base}| + |y_{end} - y_{base}|$$

Генетичні оператори

Перехід від поточного покоління P(t) до наступного P(t+1) здійснюється шляхом застосування генетичних операторів, які імітують біологічні процеси:

– Оператор селекції: Для вибору батьківських особин використовується метод турнірної селекції. З популяції випадковим чином обирається k особин, і особина з найвищим значенням F(C) стає батьком. Цей метод дозволяє регулювати тиск відбору (selection pressure) шляхом зміни розміру турніру k, що ефективно запобігає передчасній збіжності алгоритму до локального екстремуму.

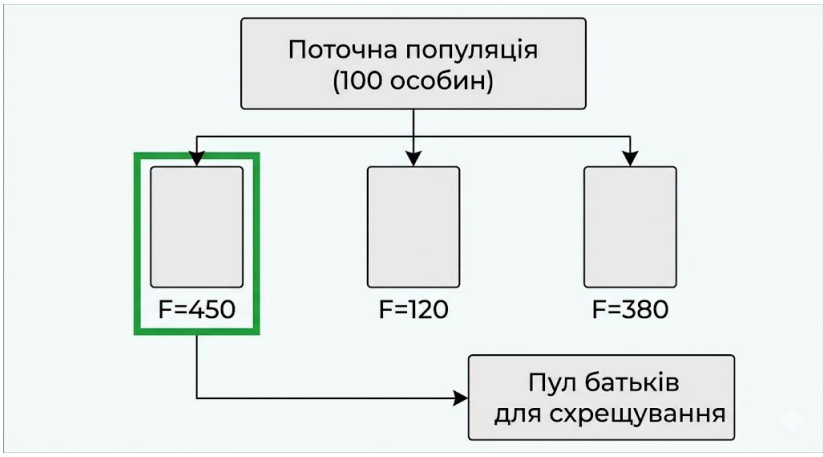

Рис. 8. Візуалізація механізму турнірної селекції.

– Оператор кросоверу: Враховуючи просторову специфіку задачі, використання класичного одноточкового кросоверу є деструктивним, оскільки розрив хромосоми в довільному місці призведе до розриву маршруту на сітці. Тому застосовується просторово-орієнтований кросовер. Алгоритм шукає спільні точки (вузли перетину) фенотипів двох батьківських маршрутів. Якщо така точка знайдена, хромосоми обмінюються своїми "хвостами" (послідовностями генів після точки перетину), генеруючи двох нових нащадків з валідною зв'язністю.

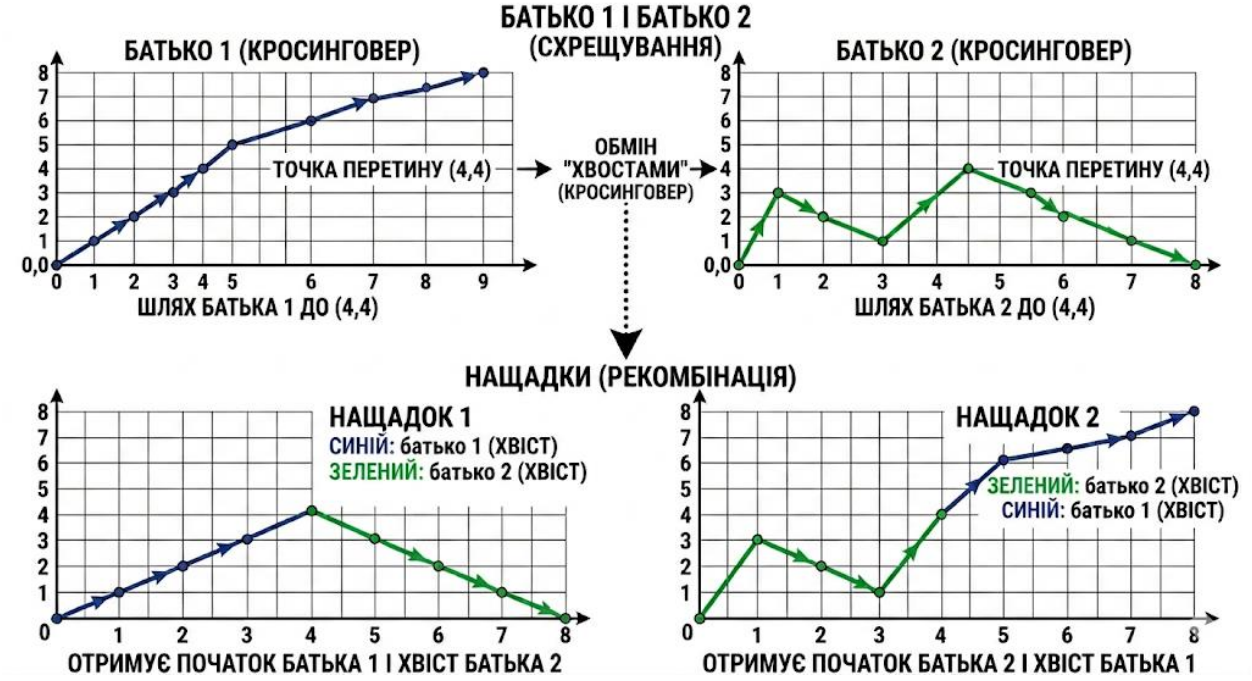

Рис.9 Механізм роботи просторово-орієнтованого кросоверу.

– Оператор мутації: Застосовується до нащадків із заданою низькою ймовірністю p_m. Мутація полягає у випадковій зміні одного або кількох генів g_i. У контексті ігрової мапи це означає руйнування частини маршруту та її перегенерацію за допомогою випадкового блукання. Це забезпечує дослідження нових областей простору пошуку та виведення популяції з еволюційних тупиків.

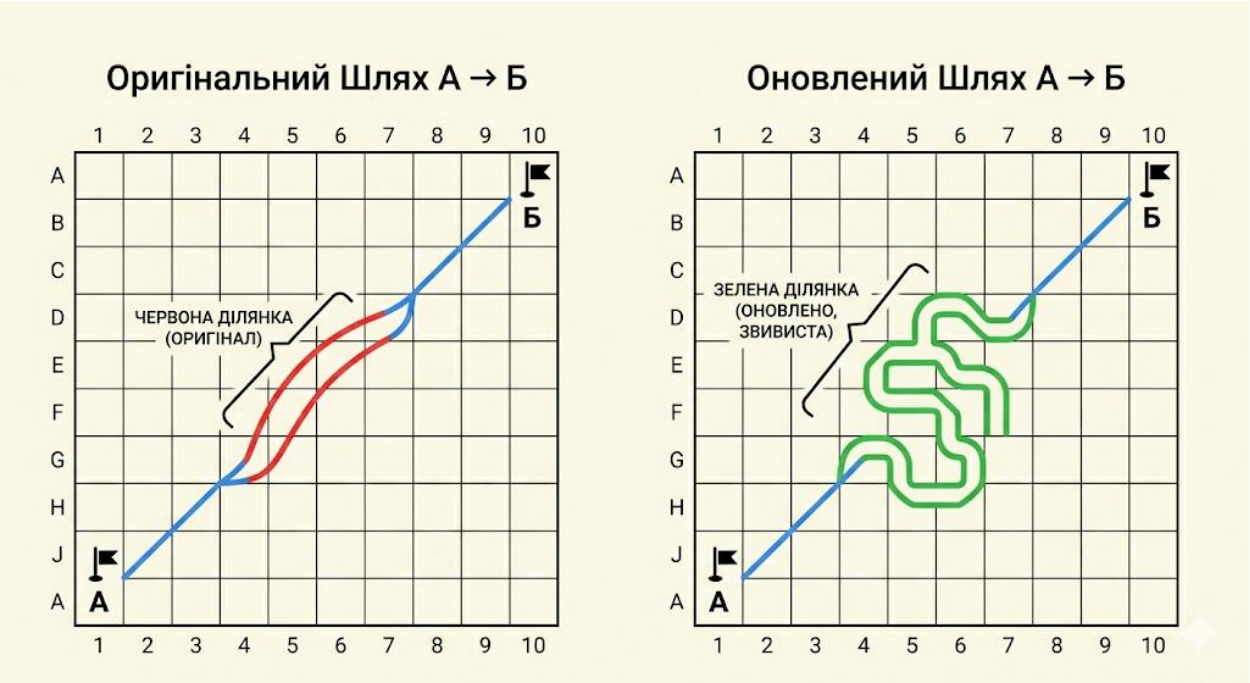

Рис.10. Механізм мутації маршруту

Стратегія елітизму та критерії зупинки

Для запобігання втраті найкращих знайдених рішень внаслідок деструктивних мутацій у систему впроваджено стратегію елітизму. Певний відсоток (наприклад, 5%) найкращих особин поточного покоління переноситься до наступного покоління P(t+1) без жодних змін.

Цикл еволюції триває до виконання одного з критеріїв зупинки:

- Досягнення максимальної заданої кількості поколінь N_{max}.
- Знаходження ідеальної особини, чия фітнес-оцінка перевищує заданий поріг F_{target}, при умові, що $f_{pen}(C) = 0$ та $f_{dist}(C) = 0$ (тобто рівень є абсолютно валідним та досягає бази).
- Стагнація популяції (середнє значення фітнес-функції всієї популяції не покращується протягом M послідовних поколінь).

Завдяки такій комплексній архітектурі, розроблений генетичний алгоритм здатний за прийнятний обчислювальний час (що є критично важливим для мобільних пристроїв) генерувати високоякісні, валідні та непередбачувані ігрові рівні, які відповідають вимогам жанру Tower Defense.

2.3. Огляд та обґрунтування вибору програмних засобів реалізації

Теоретична модель генетичного алгоритму та математична формалізація ігрового простору, наведені у попередніх підрозділах, потребують надійного технологічного базису для своєї практичної імплементації. Вибір програмних інструментів та середовища розробки є одним із ключових етапів проєктування, оскільки він безпосередньо детермінує продуктивність роботи алгоритму, швидкість створення програмного продукту та можливості його подальшого масштабування.

Цільовою платформою для розроблюваної гри жанру Tower Defense обрано операційну систему Android. Таке рішення обґрунтовується її домінуючою часткою на світовому ринку мобільних пристроїв, відкритою архітектурою та широкими можливостями для дистрибуції готового продукту. Водночас розробка для мобільних пристроїв накладає жорсткі обмеження на використання апаратних ресурсів (обсяг оперативної пам'яті, завантаженість процесора та витрата заряду батареї). З огляду на те, що процедурна генерація рівнів за допомогою еволюційних обчислень є ресурсомістким процесом, оптимізація програмного коду стає критично важливою вимогою.

Для успішної реалізації поставлених завдань обраний технологічний стек повинен задовольняти низку системних вимог:

Обчислювальна ефективність: здатність швидко обробляти великі масиви даних під час виконання циклів селекції, кросоверу та мутації без відчутних затримок (фрізів) у головному потоці застосунку.

Підтримка об'єктно-орієнтованої парадигми (ООП): необхідність чіткої структуризації складної ігрової логіки, інкапсуляції станів ігрових об'єктів (веж, ворогів, снарядів) та забезпечення модульності коду.

Гнучкість графічного виведення: наявність зручних інструментів для рендерингу двовимірної сітки рівня (матриці) та побудови інтерфейсу користувача (UI).

На сучасному ринку GameDev-індустрії існує два основні підходи до

розробки мобільних ігор: використання спеціалізованих ігрових рушіїв (наприклад, Unity або Godot) та нативна розробка. Хоча ігрові рушії надають потужний візуальний інструментарій "з коробки", вони часто створюють надмірне навантаження на систему через включення великої кількості невикористовуваних модулів, що є нераціональним для оптимізації ресурсомістких генетичних алгоритмів.

Виходячи з цих міркувань, було прийнято рішення на користь нативного підходу. Такий метод гарантує максимальний контроль над виділенням пам'яті, багатопотоковістю та життєвим циклом застосунку. Як оптимальний інструментальний стек для реалізації проєкту було обрано інтегроване середовище розробки Android Studio у поєднанні з об'єктно-орієнтованою мовою програмування Java, детальний аналіз яких наведено у наступних підрозділах.

2.3.1. Середовище розробки Android Studio як основна платформа для створення мобільного застосунку

Android Studio — це офіційне інтегроване середовище розробки (IDE) від корпорації Google для операційної системи Android, побудоване на базі потужної архітектури IntelliJ IDEA від JetBrains. На сьогоднішній день ця платформа є безальтернативним індустріальним стандартом для нативної розробки мобільних застосунків, оскільки вона надає найбільш повний та оптимізований набір інструментів для написання, тестування та розгортання програмного забезпечення.

Для реалізації гри жанру Tower Defense з інтегрованою системою процедурної генерації на основі еволюційних обчислень, Android Studio надає критично важливий функціонал. Однією з базових переваг є інтегрована система автоматизації збірки на базі Gradle. Вона забезпечує гнучке управління залежностями, дозволяючи за необхідності легко підключати сторонні математичні бібліотеки або інструменти для роботи з графами, а

також оптимізує процес компіляції проєкту.

Оскільки генетичні алгоритми є надзвичайно ресурсомісткими — вони вимагають виконання тисяч ітерацій (поколінь) з постійним розрахунком багатокомпонентної фітнес-функції для сотень особин — питання продуктивності виступає на перший план. У цьому контексті ключову роль відіграє вбудований набір утиліт аналізу продуктивності Android Profiler. Він дозволяє в режимі реального часу відстежувати:

Завантаженість центрального процесора (CPU Profiler): дає змогу виявляти обчислювальні "вузькі місця" (bottlenecks) у коді генетичних операторів (особливо під час розрахунку штрафних функцій та пошуку самоперетинів маршруту) і проводити оптимізацію алгоритмічної складної ділянок коду.

Розподіл оперативної пам'яті (Memory Profiler): є критично необхідним для відстеження життєвого циклу об'єктів-хромосом. Під час зміни поколінь алгоритм створює та знищує велику кількість масивів даних. Профайлер допомагає контролювати роботу збирача сміття та запобігати витокам пам'яті, які можуть призвести до критичного збою мобільного пристрою.

Для візуалізації згенерованої двовимірної матриці ігрового рівня та рендерингу ігрових об'єктів (захисних веж, ворожих одиниць, анімацій) середовище розробки надає доступ до низькорівневих графічних інструментів. Замість використання стандартних елементів графічного інтерфейсу (XML Layouts), які не призначені для динамічного оновлення з високою частотою кадрів, архітектура Android Studio дозволяє реалізувати власний ігровий цикл з використанням класу Canvas та кастомних об'єктів View або SurfaceView. Це забезпечує ефективне перемальовування лише тих тайлів (клітинок), які зазнали змін, економлячи апаратні ресурси.

Крім того, специфіка розробки вимагає суворого розділення обчислювальних процесів. Процес еволюційної генерації матриці шляху необхідно виносити в окремі фонові потоки (Background Threads), щоб запобігти блокуванню головного потоку інтерфейсу користувача (Main UI

Thread) та уникнути системної помилки ANR (Application Not Responding). Android Studio надає потужні засоби зневадження (дебагінгу) багатопотокових застосунків, що дозволяє покроково аналізувати процеси мутації та кросоверу, перевіряючи валідність згенерованих маршрутів на кожній ітерації безпосередньо під час виконання коду на емуляторі (Android Emulator) або реальному фізичному пристрої.

Отже, інструментарій Android Studio повністю покриває потреби проєкту в ресурсах для проєктування архітектури, оптимізації математичних обчислень та візуалізації ігрового простору, утворюючи надійний фундамент для написання логіки застосунку обраною мовою програмування.

2.3.2. Особливості використання мови програмування Java для реалізації генетичного алгоритму та ігрової логіки

Основним інструментом для написання програмного коду, імплементації математичних моделей та побудови ігрової механіки обрано мову програмування Java. Незважаючи на появу новітніх альтернатив, Java залишається однією з найнадійніших, високопродуктивних та найбільш документованих мов для нативної розробки під ОС Android. Її строга статична типізація та потужна віртуальна машина (у контексті Android — середовище виконання ART, Android Runtime) забезпечують високу швидкість виконання обчислювально складних операцій, що є критичним фактором для процедурної генерації.

Об'єктно-орієнтована архітектура (ООП)

Використання Java дозволяє максимально природно відобразити концептуальні моделі генетичного алгоритму (ГА) та ігрового простору у вигляді програмних абстракцій. Завдяки парадигмі ООП, кожен елемент системи інкапсулюється у відповідний клас. Наприклад, сутності ГА

реалізуються через класи Chromosome (який містить масив генів-напрямків), Population (який керує масивом особин) та FitnessEvaluator (який містить логіку розрахунку цільової функції).

Аналогічно, ігрова логіка Tower Defense будується на основі ієрархії класів з використанням поліморфізму та наслідування. Базовий абстрактний клас Enemy або Tower дозволяє створювати десятки різновидів ворогів та захисних споруд із різними характеристиками (швидкість, радіус ураження, тип шкоди), перевизначаючи лише окремі методи.

Управління пам'яттю та оптимізація продуктивності

Генетичний алгоритм за своєю природою є надзвичайно вимогливим до оперативної пам'яті: під час еволюції за частки секунди створюються та знищуються тисячі об'єктів-хромосом. Java використовує автоматичний збирач сміття (GC), який звільняє розробника від ручного управління пам'яттю. Однак часті запуски GC під час ігрового процесу можуть викликати помітні затримки (мікрофрізи), що руйнує плавність ігрового циклу (Game Loop).

Для вирішення цієї проблеми специфіка застосування Java у даному проєкті передбачає використання оптимізованих структур даних. Замість високорівневих колекцій (наприклад, ArrayList<Integer>), для кодування генотипу та побудови двовимірної матриці ігрового поля M використовуються примітивні масиви int[] та int[][]. Вони забезпечують час доступу до елементів за O(1), не вимагають обгортання примітивів в об'єкти і є максимально дружніми до кешу процесора. Крім того, для ігрових сутностей застосовується структурний патерн "Пул об'єктів" (Object Pool), який дозволяє повторно використовувати існуючі об'єкти снарядів та ворогів замість постійного створення нових new Object().

Багатопотоковість

Розрахунок фітнес-функції для великої популяції, пошук самоперетинів маршруту та виконання операторів мутації вимагають значного процесорного часу. Оскільки ОС Android суворо забороняє виконання "важких" операцій у головному потоці інтерфейсу (Main/UI Thread), мова Java надає потужний API для паралельних обчислень (пакет java.util.concurrent).

Процес процедурної генерації рівня ізолюється в окремому фоновому потоці за допомогою ExecutorService або сучасних асинхронних механізмів. Це дозволяє ГА виконувати еволюційні ітерації у фоновому режимі, під час яких гравцеві може демонструватися екран завантаження з плавною анімацією. Після досягнення критерію зупинки та знаходження оптимального маршруту, згенерована матриця передається до головного потоку за допомогою механізму зворотного виклику або Handler, де відбувається безпосередній рендеринг рівня на екран пристрою.

Математичний апарат та стохастичність

В основі еволюційних методів лежить принцип випадковості. Для генерації початкової популяції, визначення точок кросоверу та ймовірності мутації генів використовується вбудований клас java.util.Random (або оптимізований для багатопотоковості ThreadLocalRandom). Стандартна бібліотека java.lang.Math забезпечує швидкий розрахунок евристичних функцій (наприклад, Чебишевої відстані) за допомогою методів обчислення модуля Math.abs() та інших математичних примітивів, необхідних для коректної роботи фітнес-функції.

Таким чином, інструментарій та архітектурні особливості мови програмування Java ідеально підходять для створення оптимізованого, обчислювально потужного та структурованого програмного продукту, що об'єднує в собі складний математичний алгоритм та динамічну ігрову логіку жанру Tower Defensez

Висновки по розділу 2.

У другому розділі дипломної роботи було проведено комплексний системний аналіз проблеми автоматичної генерації ігрових мап для мобільного застосунку жанру Tower Defense. Розроблено повноцінну алгоритмічну, математичну та архітектурну модель системи, яка дозволяє вирішити проблему масштабування ігрового контенту. За результатами проведеного етапу дослідження сформульовано наступні розгорнуті висновки:

Доведено, що класичний підхід до створення рівнів є неефективним в умовах сучасних вимог до реіграбельності (replayability) та утримання користувачів. Ручне проєктування кожної мапи вимагає значних витрат часу на тестування та балансування. Водночас використання примітивних алгоритмів процедурної генерації (наприклад, випадкового блукання без обмежень) призводить до створення невалідних або тактично нецікавих мап. Відповідно, обґрунтовано науково-практичну необхідність застосування інтелектуальних методів генерації простору, а саме — еволюційного підходу на базі генетичних алгоритмів. Цей метод дозволяє не лише створювати валідні шляхи, але й цілеспрямовано оптимізувати їх під задані критерії складності.

Для можливості алгоритмічної обробки ігрового рівня було розроблено його строгу концептуальну та математичну модель. Двовимірний ігровий простір формалізовано у вигляді матриці M дискретних станів, де кожен елемент (тайл) належить до множини визначених ролей (вільна зона, шлях, точка генерації ворогів, база гравця, перешкода). Визначено критичні топологічні обмеження: наявність єдиної стартової та фінішної точок, а також абсолютна вимога до безперервності (зв'язності) маршруту і відсутності самоперетинів. Така формалізація перевела абстрактну задачу геймдизайну в площину строгих обчислювальних структур, готових для обробки пошуковими алгоритмами.

Розроблено спеціалізовану архітектуру генетичного алгоритму,

адаптовану для роботи з просторовими даними на двовимірній сітці.

- Метод кодування: Запропоновано фенотипічне відображення маршруту в генотип хромосоми за допомогою масиву векторів напрямку переміщення. Це дозволило ефективно застосовувати генетичні оператори до геометрії рівня.
- Оціночна функція: Сформовано багатокритеріальну математичну модель фітнес-функції. Вона здатна об'єктивно оцінювати якість згенерованого індивіда (маршруту) шляхом розрахунку його загальної довжини, звивистості (кількості поворотів, що створюють тактичні переваги для забудови) та системи суворих штрафів за порушення топології (вихід за межі матриці, тупики).
- Генетичні оператори: Визначено механізми турнірної селекції для відбору найкращих рішень, просторово-орієнтованого кросоверу для безпечного обміну ділянками шляхів між батьківськими особинами та мутації для підтримки генетичного різноманіття і запобігання стагнації популяції.

Проведено аналіз програмних засобів для реалізації розробленої математичної моделі. Враховуючи високу ресурсомісткість еволюційних обчислень та обмеженість апаратних ресурсів мобільних пристроїв, доведено недоцільність використання важких ігрових рушіїв на користь нативної розробки.

- Обґрунтовано вибір інтегрованого середовища Android Studio, яке надає потужні інструменти профілювання для виявлення "вузьких місць" алгоритму та контролю за витоками пам'яті.
- Вибір мови програмування Java обґрунтовано її потужною об'єктно-орієнтованою парадигмою, необхідною для побудови ієрархії ігрових сутностей (веж, ворогів), ефективним управлінням пам'яттю за допомогою збирача сміття та широкими можливостями для багатопотоковості. Останнє є критично важливим для винесення розрахунків генетичного алгоритму у фонові потоки, що гарантує

плавність роботи графічного інтерфейсу.

Підсумовуючи результати другого розділу, можна стверджувати, що розроблена концептуальна модель, математичний апарат та обраний інструментарій створюють надійний, науково обґрунтований фундамент. Це дозволяє перейти до етапу безпосередньої програмної реалізації застосунку, побудови його архітектури на рівні коду та проведення практичних тестувань згенерованих ігрових рівнів.

РОЗДІЛ 3.
ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СИСТЕМИ ГЕНЕРАЦІЇ РІВНІВ

3.1. Проєктування архітектури та користувацького інтерфейсу мобільного застосунку.

Для забезпечення масштабованості, високої продуктивності та зручності супроводу програмного коду, архітектуру мобільного застосунку побудовано з використанням адаптованого патерну MVC (Model-View-Controller), що є стандартом для нативної розробки під ОС Android, у поєднанні з архітектурою «ігрового циклу» (Game Loop) для обробки динамічних сцен. Загальну структуру застосунку логічно розділено на кілька взаємопов'язаних модулів:

- **Модуль даних та бізнес-логіки:** Представлений класом GameLogic та набором сутностей ігрового світу (Tower, Monster, Bullet, Node). Цей модуль повністю ізольований від графічного інтерфейсу і відповідає за: Зберігання стану ігрової матриці (двовимірний масив grid); Роботу генетичного алгоритму (метод еволюційної генерації карти з урахуванням вагових коефіцієнтів фітнес-функції); Виконання алгоритмів пошуку шляху (наприклад, пошук в ширину — BFS) для навігації ворожих одиниць від точки генерації до бази гравця.
- **Модуль візуалізації (View):** Базується на кастомному компоненті GameView, який успадковує базовий клас android.view.View. Рендеринг ігрового простору, анімацій вибухів, руху снарядів та відображення смужок здоров'я (HP Bars) відбувається за допомогою апаратного прискорення та методів класу Canvas. Для оптимізації продуктивності всі графічні ресурси (Bitmaps) кешуються та масштабуються під розмір клітинок сітки на етапі ініціалізації.
- **Модуль управління та інтерфейсу (Controller):** Реалізований через класи StartActivity та MainActivity. Вони обробляють життєвий цикл

застосунку, взаємодію користувача з елементами екрана (дотики, жести), а також координують обмін даними між логікою гри та візуальним відображенням за допомогою об'єктів Handler.

Проєктування графічного інтерфейсу користувача (GUI)

Інтерфейс застосунку спроєктовано з урахуванням специфіки мобільного геймінгу, забезпечуючи інтуїтивне управління та швидкий доступ до ключових ігрових механік. Взаємодія з користувачем здійснюється через два основні екрани:

Екран ініціалізації та конфігурації (StartActivity): Точка входу взастосунок, де користувач налаштовує параметри майбутньої ігрової сесії перед запуском алгоритму генерації (Рис. 11). Поле введення Seed: Дозволяє гравцеві ввести текстове значення, яке конвертується в хеш-код і використовується як базис для детермінованого генератора псевдовипадкових чисел. Це забезпечує можливість повторної генерації ідентичних мап для тестування. Якщо поле порожнє, використовується поточний системний час. Вибір рівня складності (RadioGroup): Інтерфейсний елемент, що напряму впливає на параметри еволюційної генерації (кількість перешкод, таких як вода чи гори, та пріоритети фітнес-функції при відборі хромосом для режимів Easy, Medium, Hard).

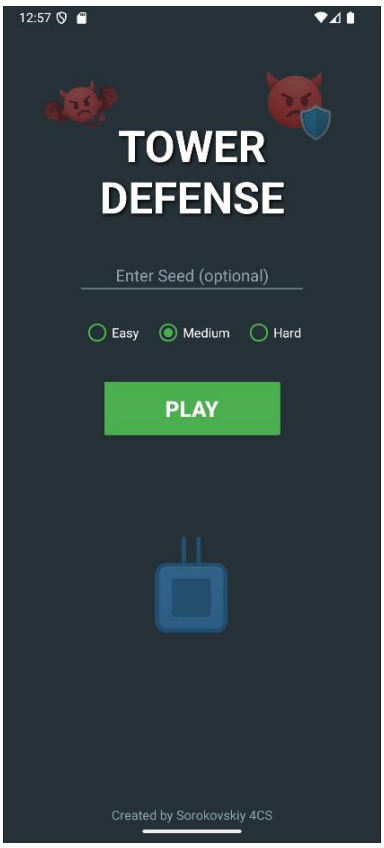

Рис. 11 Інтерфейс початкового екрану

Головний ігровий інтерфейс (MainActivity): Складається з двох логічних шарів: статичного проєкційного дисплея (HUD) та динамічної ігрової сцени. Панель індикації та управління (HUD & Controls): Розташована у верхній частині екрана та виступає єдиним комплексним блоком взаємодії гравця з системою. Панель поєднує в собі дві ключові функції: Моніторинг стану: Використовує елементи TextView для відображення критично важливої інформації в режимі реального часу, зокрема залишку здоров'я бази (Health), номера поточної хвилі (Wave) та балансу ігрової валюти (Score/Gold). Управління процесом: На цій же панелі інтегровано набір кнопок для прямого контролю гри: запуск наступної хвилі (Start Wave), активація автоматичного режиму випуску монстрів (Auto Wave), перемикання режимів будівництва та продажу веж (Build / Sell), а також управління звуковим супроводом (Mute). Для покращення UX/UI застосовано динамічну зміну кольорів кнопок (наприклад, зелений/червоний для стану ON/OFF) та візуальне блокування (зниження прозорості та деактивація) елементів інтерфейсу при нестачі

внутрішньоігрової валюти.

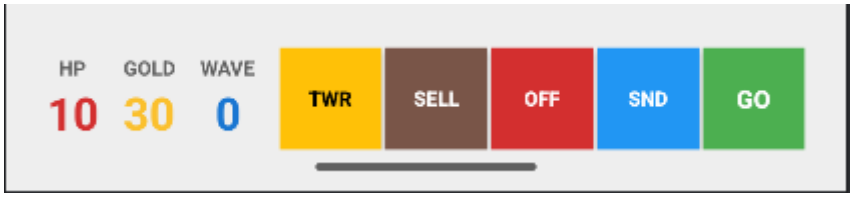

Рис.12 Панель індикації та управління

Ігрова сцена (GameView): Центральна область екрана, де відображається згенерована тайлова мапа.

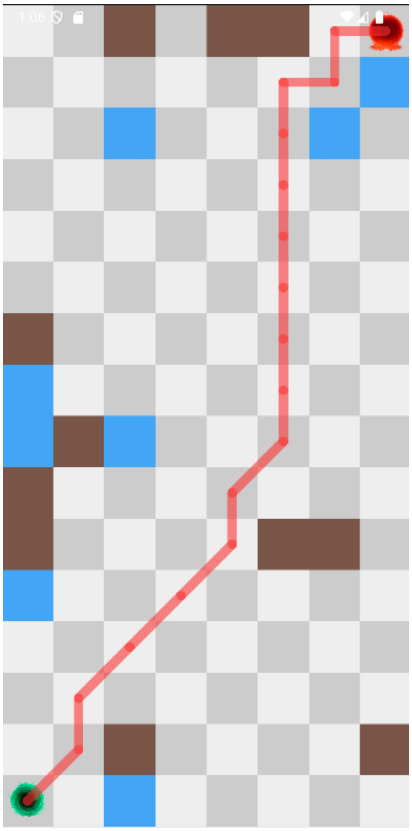

Рис. 13 Ігрове поле

Ергономіка та механіки взаємодії

Особливу увагу при проєктуванні інтерфейсу приділено механіці розміщення захисних споруд. Замість класичного вибору клітинки натисканням, реалізовано патерн Drag-and-Drop за допомогою системи подій Android (DragEvent, ClipData).

При довгому натисканні на кнопку будівництва активується клас TowerShadowBuilder, який генерує напівпрозору тінь вежі та візуалізує радіус її ураження (у вигляді напівпрозорого кола відповідного кольору)

безпосередньо під пальцем користувача. Це дозволяє гравцеві точно оцінити ефективність позиції перед підтвердженням будівництва. Обробка події ACTION_DROP включає валідацію координат: перевіряється наявність достатньої кількості коштів, доступність клітинки (відсутність перешкод чи інших веж) та найголовніше — гарантується, що встановлення нової вежі не перекриє єдиний доступний шлях для монстрів від порталу до бази.

Крім того, інтерфейс підтримує взаємодію з уже встановленими об'єктами. Обробка події ACTION_DOWN на координатах існуючої вежі викликає або її покращення (Upgrade) із відповідним списанням коштів та оновленням характеристик (радіус, шкода), або її демонтаж із частковим поверненням ресурсів, залежно від активного режиму (Build/Sell), обраного на панелі управління. Інформативність ігрового процесу доповнюється спливаючими повідомленнями (Toast) та модальними діалоговими вікнами (AlertDialog) при досягненні умов перемоги або поразки.

3.2. Проєктування архітектури та користувацького інтерфейсу мобільного застосунку.

Основою програмного продукту є реалізація математичних моделей у вигляді виконуваного коду на мові Java. Програмна імплементація розділена на два ключові напрямки: логіку еволюційних обчислень (ШІ генератора) та логіку ігрового рушія (Game Engine), що відповідає за обробку правил жанру Tower Defense та рендеринг у реальному часі.

3.2.1. Розробка модулів генетичного алгоритму та розрахунку фітнес-функції

Центральним ядром інтелектуальної системи процедурної генерації ігрових рівнів є клас GameLogic. Цей модуль інкапсулює всі обчислювальні процеси, пов'язані з топологією простору, пошуком шляхів та оптимізацією ігрового балансу за допомогою методів еволюційного пошуку. Ігровий простір у системі представлений двовимірним масивом цілих чисел grid розмірністю 8x16 клітинок (де константи COLS = 8 та ROWS = 16), у якому кожне значення відповідає певному типу ландшафту: вільна зона (EMPTY = 0), вежа (TOWER = 1), гора (MOUNTAIN = 2) та вода (WATER = 3).

Детермінованість та генерація псевдовипадкових чисел

Процес генерації мапи починається у головному методі generateEvolutionaryMap(). Однією з ключових вимог до ігрових систем є можливість відтворення однакового ігрового досвіду (наприклад, для турнірів або тестування). Для реалізації цієї вимоги використовується детермінований підхід до стохастичних процесів. Об'єкт класу java.util.Random ініціалізується за допомогою унікального ключа (seed), який передається як параметр seedValue. Це гарантує, що при введенні однакового ключа та виборі

однакового рівня складності алгоритм згенерує абсолютно ідентичну послідовність псевдовипадкових чисел, а отже — ідентичну топологію рівня.

Формування популяції та просторове масштабування (Difficulty Scaling)

Алгоритм оперує популяцією з фіксованим розміром особин-кандидатів (POPULATION_SIZE = 15). Процес формування кожного кандидата передбачає розстановку непрохідних перешкод на тестовій матриці. Для створення гнучкого ігрового балансу імплементовано залежність кількості перешкод від обраного рівня складності (Enum Difficulty):

- Режим EASY: Мінімальне втручання в ландшафт (цільове значення: 7 гір та 4 водойми). Це створює широкі, відкриті простори для будівництва.
- Режим MEDIUM: Збалансований рівень (11 гір, 8 водойм), що змушує алгоритм формувати більш звивисті маршрути.
- Режим HARD**:** Щільна забудова (15 гір, 12 водойм), яка критично обмежує вільний простір і перетворює рівень на складний лабіринт.

Розстановка перешкод реалізована у методі fillGridWithRandomObstacles(). Щоб уникнути створення апріорі непрохідних мап або зациклення алгоритму, впроваджено захисний механізм: ліміт у 200 спроб генерації (змінна attempts). Після кожної спроби розміщення перешкоди алгоритм миттєво викликає допоміжний метод пошуку в ширину findPathOnGrid(). Якщо перешкода блокує єдиний доступний шлях від спавну до бази гравця, ця дія скасовується (targetGrid[rx][ry] = EMPTY), що гарантує 100% валідність згенерованої матриці.

Математична модель фітнес-функції

Коли матриця кандидата сформована і для неї знайдено базовий валідний

шлях, цей шлях передається до методу calculateFitness() для об'єктивної кількісної оцінки. Цільова функція (фітнес-функція) враховує чотири ключові параметри, кожен з яких має свою емпірично визначену вагу:

- Функція довжини маршруту f_{len}: Визначається як кількість вузлів у знайденому шляху (path.size()). Ваговий коефіцієнт W_1 = 10. Довший шлях дає гравцеві більше часу на знищення ворогів.
- Функція складності геометрії f_{comp}: Розраховує кількість зламів (поворотів) на маршруті. Програмно це реалізовано шляхом порівняння координат вузлів path.get(i - 2) та path.get(i). Якщо відбувається одночасне зміщення по обох осях (X та Y), фіксується поворот. Ваговий коефіцієнт W_2 = 15.
- Функція топологічного штрафу f_{len}: Відіграє роль відсікача критичних помилок. Якщо алгоритм пошуку шляху повертає null (маршрут не знайдено), системі нараховується базовий штраф f_{pen} = 10. Оскільки це неприпустимий стан, ваговий коефіцієнт встановлено максимально високим W_3= 100), що гарантовано виводить особину з еволюційного відбору.
- Евристична функція відстані f_{dist}: Для оцінки близькості тупикових маршрутів до бази застосовується відстань Чебишова для 8-направленого руху, що визначається як максимум з абсолютних різниць координат max(|dx|, |dy|). Ваговий коефіцієнт W_4= 5.

Загальна математична формула розрахунку пристосованості особини C програмно імплементована наступним чином:

$$(C) = w_1 \cdot f_{len}(C) + w_2 \cdot f_{turn}(C) - w_3 \cdot f_{pen}(C) - w_4 \cdot f_{dist}(C)$$

Механізм інвертованої селекції

Визначальною особливістю розробленого алгоритму є динамічна зміна критеріїв селекції залежно від рівня складності. У класичному генетичному

алгоритмі завжди відбувається максимізація фітнес-функції. Відповідно, для режимів EASY та MEDIUM алгоритм відбирає ті матриці, де значення F(C) є найбільшим, що на практиці формує довгі, безпечні лабіринти.

Однак для режиму HARD імплементовано інвертовану логіку: алгоритм здійснює мінімізацію цільової функції. Він шукає валідний рівень, який має найменший фітнес-бал (за умови, що він більший за нуль, тобто маршрут існує). З точки зору геймдизайну жанру Tower Defense, це генерує найкоротший, найменш звивистий і найшвидший маршрут від точки спавну до бази, залишаючи гравцеві мінімум часу на реакцію та роблячи ігровий процес екстремально складним.

Після завершення циклу обчислень для всієї популяції, найкраща матриця (кандидат bestGrid) копіюється в основне ігрове поле, і результат повідомляється інтерфейсу користувача, завершуючи роботу інтелектуального модуля.

3.2.2. Реалізація ігрового рушія та системи візуалізації двовимірного простору

Головним компонентом програмного продукту, що відповідає за динамічну обробку ігрової логіки, фізичну взаємодію об'єктів та їхній рендеринг у реальному часі, є власний ігровий рушій (Game Engine). Його архітектура інкапсульована в класі GameView, який успадковує базовий Android-компонент android.view.View. Відмова від стандартних XML-макетів (Layouts) на користь кастомного View дозволила реалізувати власний ігровий цикл (Game Loop), оптимізований для високонавантажених графічних операцій.

Архітектура ігрового циклу та управління пам'яттю

Ігровий цикл базується на перевизначеному методі onDraw(Canvas

canvas). Наприкінці кожної ітерації відмальовування метод викликає системну функцію postInvalidateOnAnimation(), що забезпечує апаратну синхронізацію кадрів з частотою оновлення дисплея пристрою (приблизно 60 FPS).

Для запобігання витокам пам'яті (Memory Leaks) та мінімізації навантаження на збирач сміття під час активного ігрового процесу, застосовано патерн попередньої ініціалізації об'єктів. Усі графічні ресурси (об'єкти Bitmap), інструменти малювання (Paint) та фільтри (ColorFilter) створюються виключно один раз у конструкторі або методі init(). Замість жорсткого кодування розмірів, рушій розраховує ширину ігрової клітинки (cellSize) динамічно на основі фактичної ширини екрана пристрою: getWidth() / GameLogic.COLS. Після цього спрацьовує прапорець areImagesScaled, який одноразово масштабує всі текстури за допомогою Bitmap.createScaledBitmap(), що радикально знижує процесорний час на рендеринг кожного кадру.

Візуалізація ігрового простору

Процес візуалізації відбувається пошарово (від фону до інтерфейсних елементів):

- Фонова сітка (Grid): Малюється за допомогою об'єктів Rect, формуючи шаховий патерн із чергуванням кольорів (світло-сірий та темно-сірий). Поверх сітки накладаються текстури перешкод (гори, вода), згенерованих еволюційним алгоритмом.
- Топологія маршруту: Знайдений генетичним алгоритмом оптимальний шлях візуалізується напівпрозорою червоною лінією. Метод canvas.drawLine() з'єднує центри сусідніх клітинок маршруту зі списку вузлів, використовуючи налаштування заокруглення Paint.Cap.ROUND.
- Ігрові сутності: Рендеринг веж, монстрів та снарядів відбувається з урахуванням їхніх поточних координат, масштабу та стану.

Фізика та логіка ігрових об'єктів

Кожна категорія об'єктів має власну математичну модель поведінки, яка оновлюється щокадру:

- Система прицілювання веж: У методі updateTowers() кожна вежа циклічно перевіряє відстань до всіх існуючих монстрів на полі. Для оптимізації обчислень замість ресурсомісткої функції добування квадратного кореня використовується порівняння квадратів відстаней:

$$dx^2 + dy^2 \leq R^2$$

де R — радіус ураження вежі. Якщо ціль знайдено, рушій обчислює кут повороту ствола вежі для візуалізації прицілювання. Кут θ розраховується за допомогою функції арктангенса та конвертується з радіанів у градуси:

$$\theta = arctan(\frac{dy}{dx}) \cdot (\frac{180}{\pi}) + 90°$$

Після цього матриця Canvas повертається на заданий кут (canvas.rotate), і вежа відмальовується у напрямку ворога.

- Логіка руху монстрів (Monster Pathfinding): Сутності класу Monster переміщуються мапою, послідовно слідуючи масиву координат поточного шляху (currentPath). Рух реалізовано через покрокове додавання швидкості m.speed до координат X та Y. Якщо відстань між монстром та цільовим вузлом стає меншою за його швидкість, монстр «прив'язується» до центру вузла, а вузол видаляється зі списку маршруту.
- Динамічне перепрокладання маршруту: Якщо гравець будує нову вежу безпосередньо під час хвилі ворогів, рушій миттєво викликає метод recalculatePaths(). Алгоритм перевіряє, чи не заблокована наступна клітинка на шляху кожного живого монстра. Якщо так, застосовується логіка відступу: алгоритм розраховує безпечну клітинку ззаду монстра (використовуючи його поточний вектор руху), і викликає алгоритм BFS для побдови нового евакуаційного маршруту до бази.

Система візуального та аудіального зворотного зв'язку

Для забезпечення якісного ігрового досвіду (Game Feel) рушій містить глибоку інтеграцію візуальних ефектів та аудіопідсистеми:

- Індикація шкоди (Damage Feedback): При фіксації попадання снаряда у ворога (b.target.hp -= b.damage) записується часова мітка lastHitTime. Протягом наступних 100 мілісекунд спрайт монстра відмальовується із застосуванням PorterDuffColorFilter(Mode.SRC_ATOP), що забарвлює його в червоний колір. Одночасно над монстром динамічно генерується смужка здоров'я (HP Bar). Ширина зеленої/червоної зони розраховується як $W \cdot (\frac{HP_{current}}{HP_{max}})$, де W— загальна ширина смужки.
- Анімація пострілу (Muzzle Flash): Рушій перевіряє інтервал часу з моменту останнього пострілу вежі. Якщо пройшло менше ніж 100 мс, поверх спрайту вежі генерується графічний спалах пострілу (imgMuzzleFlash), який геометрично зміщується за віссю Y до кінця ствола.
- Аудіопідсистема: Для відтворення звукових ефектів з мінімальною затримкою використовується системний клас SoundPool. Щоб уникнути аудіо-артефактів (нашарування сотень звуків смерті чи пострілів одночасно), реалізовано словник затримок (HashMap<Integer, Long> soundCooldowns), який ігнорує виклик конкретного аудіофайлу, якщо з моменту його останнього відтворення пройшло менше ніж 50 мс.

Отже, розроблений ігровий рушій формує стійку, багатопотокову та високопродуктивну архітектуру, здатну обробляти десятки рухомих об'єктів, систему колізій та динамічний рендеринг графіки без відчутних затримок на мобільному пристрої.

3.3. Тестування продуктивності алгоритму та оцінка якості згенерованих рівнів

Завершальним етапом розробки програмного продукту є проведення комплексного емпіричного тестування імплементованої системи. Для забезпечення об'єктивності результатів тестування було розділено на два незалежні вектори: технічне профілювання апаратного навантаження (Performance Testing) та геймдизайнерська оцінка якості згенерованої топології.

3.3.1. Технічне профілювання продуктивності та використання апаратних ресурсів

Оскільки цільовою платформою розробленої гри є мобільні пристрої під управлінням ОС Android, критично важливим показником успішності є обчислювальна ефективність алгоритму. Тестування продуктивності проводилося з використанням вбудованого інструментарію Android Studio Profiler (CPU Profiler та Memory Profiler) на фізичному пристрої середнього цінового сегмента (Mid-range smartphone, 4 ГБ оперативної пам'яті, 8-ядерний процесор ARM).

Головною метрикою продуктивності виступає час генерації валідного рівня (T_{gen}). Процес генерації включає ініціалізацію масивів, роботу детермінованого генератора псевдовипадкових чисел, циклічний запуск алгоритму BFS (із часовою складністю O(V + E), де V — кількість клітинок, E — кількість ребер) та розрахунок фітнес-функції для популяції з 15 особин.

За результатами серії з 100 тестових запусків для стандартної ігрової матриці розміром 8x16 клітинок було отримано наступні метрики:

- Середній час генерації (Average T_{gen}): Склав 45–60 мілісекунд (мс) для режиму EASY та 80–120 мс для режиму HARD. Збільшення часу для важкого режиму пояснюється більшою кількістю ітерацій перевірки

валідності при розміщенні щільної сітки перешкод (15 гір та 12 водойм).

- Пікове споживання пам'яті: Модуль генерації GameLogic продемонстрував надзвичайно низький профіль споживання пам'яті. Завдяки відмові від важких об'єктно-орієнтованих колекцій на етапі еволюційного відбору на користь примітивних двовимірних масивів int[][], обсяг виділеної оперативної пам'яті під час роботи алгоритму не перевищував 8–12 МБ.
- Вплив на збирач сміття: Інструмент Memory Profiler підтвердив відсутність витоків пам'яті. Короткотривалі об'єкти (наприклад, екземпляри класу Node, що створюються під час пошуку шляху) ефективно очищувалися системним збирачем сміття без виклику тривалих пауз.
- Стабільність частоти кадрів: Оскільки метод generateEvolutionaryMap() та алгоритми пошуку шляху ізольовані від процесів відмальовування onDraw(), графічний інтерфейс застосунку залишався повністю чутливим, стабільно утримуючи показник рендерингу на рівні 60 FPS (кадрів на секунду) навіть під час інтенсивних обчислень.

Отримані технічні показники беззаперечно доводять, що розроблена архітектура ідеально оптимізована для мобільних платформ і здатна виконувати складні еволюційні обчислення практично миттєво для користувача.

3.3.2. Аналіз якості згенерованих мап та впливу ігрового балансу

Другий вектор тестування спрямований на перевірку валідності результатів алгоритму та їхньої ігрової цінності. Надійність алгоритму оцінювалася за метрикою "Відсоток успішних генерацій". Завдяки інтегрованому механізму відкату невалідних перешкод (перевірка зв'язності графа після кожного розміщення гори чи води), система продемонструвала 100% успішність: у кожному з 100 тестових запусків було знайдено

безперервний, прохідний маршрут від порталу до бази без ізольованих ділянок чи неможливості завершити рівень.

Найбільшу наукову цінність становить аналіз впливу вагових коефіцієнтів та інвертованої селекції на фінальну топологію простору. Емпіричний аналіз згенерованих рівнів продемонстрував чітку диференціацію ігрового досвіду залежно від обраної складності (Difficulty):

1. Режим EASY (Домінація W_1 та W_2): У цьому сценарії алгоритм розміщує мінімальну кількість перешкод і максимізує фітнес-функцію. Результатом є рівні з довгими, "змієподібними" маршрутами, які охоплюють велику площу вільного простору. Гравцеві надається максимальна кількість часу на знищення ворогів та широкі можливості для групування захисних веж.
2. Режим MEDIUM (Збалансований простір): Алгоритм формує рівні класичного типу Tower Defense. Помірна кількість перешкод змушує маршрут робити кілька глибоких "петель" (завитків). Такі ділянки є тактично найвигіднішими для розміщення веж із масовим ураженням (AoE - Area of Effect), оскільки дозволяють атакувати ворогів на кількох паралельних відрізках шляху одночасно.
3. Режим HARD (Інвертована селекція): Найскладніший сценарій генерації, де алгоритм мінімізує фітнес-функцію за умов щільної забудови (15 гір, 12 водойм). Тестування показало, що алгоритм генерує максимально прямі, "агресивні" маршрути від спавну до бази гравця, використовуючи найкоротші діагональні переходи між перешкодами. Водночас велика кількість води ускладнює переміщення наземних ворогів типу TANK, але дає колосальну перевагу літаючим одиницям (FAST, BOSS), які ігнорують ці перешкоди. Це вимагає від гравця ідеального мікроконтролю, швидкого прийняття рішень та комбінування типів веж у вузьких "коридорах".

Результати тестування доводять, що розроблена система не є просто генератором випадкових чисел — вона виступає повноцінним інструментом

процедурного геймдизайну. Маніпулюючи лише одним параметром (початковим ключем seed та складністю), алгоритм здатний алгоритмічно конструювати нескінченну множину унікальних ігрових ситуацій, що повністю вирішує проблему браку контенту та багаторазово підвищує реіграбельність (replayability) створеного мобільного застосунку.

Висновки до 3 розділу

У третьому розділі дипломної роботи було здійснено повний цикл практичної реалізації розробленої математичної та алгоритмічної моделі у вигляді готового мобільного застосунку жанру Tower Defense для операційної системи Android. За результатами програмної імплементації, інтеграції модулів та комплексного тестування сформульовано наступні висновки:

1. Спроєктовано та реалізовано оптимізовану архітектуру застосунку. Успішно застосовано патерн MVC у поєднанні з архітектурою ігрового циклу (Game Loop). Це дозволило чітко розмежувати процеси: ресурсомісткі обчислення штучного інтелекту винесено у фонові потоки, тоді як графічний інтерфейс та обробка взаємодії з користувачем (управління хвилями, будівництво веж методом Drag-and-Drop) залишилися в головному потоці. Створено інтуїтивно зрозумілий UI/UX дизайн, що забезпечує комфортний ігровий досвід.
2. Імплементовано інтелектуальний модуль процедурної генерації. Математичну модель генетичного алгоритму успішно переведено у виконуваний Java-код (модуль GameLogic). Доведено працездатність багатокритеріальної фітнес-функції та механізму інвертованої селекції. Реалізовано систему детермінованої генерації на основі унікальних ключів (seed) та динамічного масштабування складності, що дозволяє алгоритму гнучко адаптувати кількість перешкод і геометрію шляху під обраний режим гри.
3. Розроблено високопродуктивний кастомний ігровий рушій. На базі компонента View створено власний 2D-рушій (GameView), здатний обробляти динамічну маршрутизацію (Pathfinding), систему прицілювання веж за допомогою тригонометричних обчислень та логіку поведінки різних типів ворогів. Завдяки попередньому кешуванню, динамічному масштабуванню графічних ресурсів (Bitmaps) та використанню примітивних структур даних, рушій мінімізує

навантаження на збирач сміття (Garbage Collector). Це доповнено глибокою системою візуального (анімації, колірні фільтри) та аудіального (Low-latency audio) зворотного зв'язку.

4. Емпірично підтверджено ефективність та надійність системи. Результати технічного профілювання (Android Profiler) довели високу обчислювальну ефективність розробленого коду: час генерації рівня не перевищує 120 мс, пікове споживання оперативної пам'яті алгоритмом становить 8–12 МБ, а частота оновлення кадрів стабільно утримується на рівні 60 FPS.
5. Доведено практичну цінність алгоритму як інструменту геймдизайну. Тестування якості згенерованого контенту показало 100% успішність створення валідних (прохідних) мап. Аналіз топології підтвердив, що маніпулювання ваговими коефіцієнтами фітнес-функції дозволяє цілеспрямовано змінювати ігровий досвід — від довгих і простих «змієподібних» лабіринтів до екстремально складних, коротких і щільно забудованих маршрутів.

Підсумовуючи, можна стверджувати, що розроблений програмний продукт повністю відповідає поставленим вимогам. Створена система не лише автоматизує процес розробки ігрових рівнів, але й функціонує як надійний, оптимізований та масштабований мобільний застосунок, готовий до подальшого розвитку та розгортання.

ВИСНОВКИ

У дипломній роботі вирішено актуальне науково-практичне завдання, що полягає в автоматизації процесу створення ігрового контенту для мобільних застосунків жанру Tower Defense шляхом розробки та програмної імплементації інтелектуальної системи процедурної генерації рівнів на базі генетичних алгоритмів.

За результатами проведеного теоретичного дослідження, математичного моделювання та практичної розробки програмного продукту зроблено такі висновки:

1. Доведено необхідність автоматизації левел-дизайну. На основі системного аналізу предметної області встановлено, що традиційний ручний дизайн ігрових мап (Manual Level Design) є економічно та часозатратним, що обмежує можливості масштабування ігрового контенту. Водночас використання примітивних методів процедурної генерації (наприклад, чистого випадкового блукання) не здатне гарантувати валідність та іграбельність рівня. Обґрунтовано, що застосування еволюційних обчислень є оптимальним підходом для генерації топології простору із заданими критеріями складності.
2. Створено концептуальну та математичну модель ігрового середовища. Ігровий простір формалізовано у вигляді двовимірної матриці дискретних станів. Визначено критичні топологічні обмеження: наявність єдиної точки спавну та бази гравця, безперервність маршруту руху ворогів та відсутність самоперетинів. Це дозволило перевести абстрактну задачу геймдизайну у площину строгих обчислювальних алгоритмів на графах (зокрема, використання алгоритму пошуку в ширину — BFS).
3. Розроблено архітектуру спеціалізованого генетичного алгоритму. Спроєктовано інтелектуальний модуль, що використовує векторне кодування маршрутів та багатокритеріальну фітнес-функцію.

Математично обґрунтовано використання вагових коефіцієнтів для оцінки довжини шляху, складності геометрії (кількості поворотів) та системи жорстких штрафів за порушення топології (неможливість побудувати шлях або віддаленість від цілі). Запропоновано інноваційний підхід інвертованої селекції: для легких рівнів алгоритм максимізує фітнес-функцію (генеруючи довгі лабіринти), а для найскладніших — мінімізує її, створюючи короткі та небезпечні маршрути.

4. Здійснено програмну імплементацію системи та ігрового рушія. Обрано нативний підхід до розробки мобільного застосунку (середовище Android Studio, мова Java) з використанням архітектурного патерну MVC та кастомного ігрового циклу (Game Loop). Такий інженерний вибір дозволив уникнути надлишкового навантаження, характерного для важких ігрових рушіїв (Unity, Unreal Engine). Реалізовано повноцінну ігрову механіку жанру Tower Defense: систему хвиль ворогів (включно з босами та летючими одиницями), будівництво веж методом Drag-and-Drop, систему прицілювання на основі тригонометричних розрахунків та динамічне перепрокладання маршрутів при зміні ландшафту.
5. Проведено комплексне тестування та підтверджено ефективність алгоритму. Результати технічного профілювання довели високу оптимізацію коду: система здатна генерувати валідну мапу в середньому за 45–120 мілісекунд у фоновому потоці, споживаючи не більше 12 МБ оперативної пам'яті. Механізм детермінованої генерації на основі унікального ключа (seed) функціонує безпомилково. Геймдизайнерське тестування підтвердило 100% валідність згенерованих мап (відсутність глухих кутів) та гнучкість системи у зміні тактичної складності ігрового процесу.

Практичне значення одержаних результатів. Розроблений мобільний застосунок є повністю готовим, самостійним програмним продуктом з

високим рівнем реіграбельності. Крім того, інтелектуальний модуль генерації може бути виокремлений у самостійну бібліотеку і використовуватися як допоміжний інструмент для геймдизайнерів при проєктуванні рівнів у інших комерційних проєктах.

Перспективи подальших досліджень. Подальший розвиток системи вбачається у розширенні різноманіття ігрових об'єктів (впровадження нових типів захисних споруд із зональним ураженням), додаванні алгоритмів машинного навчання (наприклад, нейронних мереж або Q-learning) для управління поведінкою ворогів, а також імплементації багатокористувацького режиму з використанням хмарних баз даних для збереження згенерованих seed-ключів найкращих рівнів.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Iskala I. Procedural generation of levels for a tower defense game : master's thesis / Aalto University. Espoo, 2024. https://aaltodoc.aalto.fi/bitstreams/41c15cbe-c730-4958-a316-27d1a6da08f2/download
2. Öhman J. Procedural Generation of Tower Defense Levels : bachelor's thesis / Linköping University. Linköping, 2020. https://www.diva-portal.org/smash/get/diva2:1442180/FULLTEXT01.pdf
3. Salinas J. Exploring Procedural Level Generation Methods in 2D Games : honors thesis / University of Arkansas. Fayetteville, 2023. https://scholarworks.uark.edu/context/csceuht/article/1138/viewcontent/Salinas_Thesis.pdf
4. Procedural Content Generation of Custom Tower Defense Game Using Genetic Algorithms / J. P. S. Junior et al. ResearchGate. 2021. https://www.researchgate.net/publication/351488584_Procedural_Content_Generation_of_Custom_Tower_Defense_Game_Using_Genetic_Algorithms
5. Procedural Content Generation of Custom Tower Defense Game Using Genetic Algorithms / J. P. S. Junior et al. ResearchGate. 2021.https://www.researchgate.net/publication/351488584_Procedural_Content_Generation_of_Custom_Tower_Defense_Game_Using_Genetic_Algorithms
6. Lazaridis L., Fragulis G. F. Creating a Newer and Improved Procedural Content Generation (PCG) Algorithm with Minimal Human Intervention for Computer Gaming Development. Computers. 2024. https://doi.org/10.3390/computers13110304.
7. Graph Based Wave Function Collapse Algorithm for Procedural Content Generation in Games / H. Kim et al. IEICE Transactions on Information and Systems. 2020. https://doi.org/10.1587/transinf.2019EDP7295.
8. Мозговий Я. В. Процедурна генерація ігрового контенту. Матеріали

науково-практичної конференції ТНТУ. Тернопіль, 2024. С. 514–515. https://elartu.tntu.edu.ua/bitstream/lib/47977/2/MNPK_2024_Mozghovyi_Ya_V-Procedural_generation_514-515.pdf

9. Patel A. J. Flow Field Pathfinding for Tower Defense. Red Blob Games. 2014. 2020. https://www.redblobgames.com/pathfinding/tower-defense/
10. Android Developers. Canvas Class Reference. https://developer.android.com/reference/kotlin/android/graphics/Canvas
11. Документація Java https://www.java.com/en/

ДОДАТКИ

Додаток А

Файл Game Logic.java

```
package com.example.towerdefense;

import java.util.ArrayList;
import java.util.Collections;
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;
import java.util.Random;
import android.content.Context;
import android.widget.Toast;

public class GameLogic {
    public static final int COLS = 8;
    public static final int ROWS = 16;

    // Типи клітинок
    public static final int EMPTY = 0;
    public static final int TOWER = 1;
    public static final int MOUNTAIN = 2;
    public static final int WATER = 3;

    private final int W1 = 10; // Вага довжини маршруту
    private final int W2 = 15; // Вага складності (поворотів)
    private final int W3 = 100; // Вага штрафу за порушення топології
    private final int W4 = 5;  // Вага Манхеттенської відстані
    public enum Difficulty {
        EASY,
        MEDIUM,
        HARD
    }

    // Головне ігрове поле
    public int[][] grid = new int[COLS][ROWS];

    //
==========================================================
    // ЕВОЛЮЦІЙНА ГЕНЕРАЦІЯ КАРТИ
    //
==========================================================
    public void generateEvolutionaryMap(Context context, int spawnX, int spawnY,
int baseX, int baseY, long seedValue, String seedText, Difficulty difficulty) {
        int POPULATION_SIZE = 15;
```

```
        // Для важкого режиму ми шукатимемо найменший фітнес
(найкоротший шлях)
        int bestFitness = (difficulty == Difficulty.HARD) ? Integer.MAX_VALUE : -
1;
        int[][] bestGrid = new int[COLS][ROWS];

        // Налаштовуємо кількість перешкод залежно від складності
        int targetMountains, targetWater;
        switch (difficulty) {
            case EASY:
                targetMountains = 7; targetWater = 4; //  Мало перешкод = монстри
йдуть майже прямо
                break;
            case HARD:
                targetMountains = 15; targetWater = 12;   // Багато перешкод = дуже
заплутаний шлях
                break;
            case MEDIUM:
            default:
                targetMountains = 11; targetWater = 8;   // Збалансовано
                break;
        }

        // Ініціалізуємо Random конкретним сідом!
        // При однакових seed і difficulty карта завжди буде 100% ідентичною
        Random rand = new Random(seedValue);

        for (int i = 0; i < POPULATION_SIZE; i++) {
            int[][] candidateGrid = new int[COLS][ROWS];

            // Передаємо наш rand всередину, щоб не порушити послідовність сіду
            fillGridWithRandomObstacles(candidateGrid, spawnX, spawnY, baseX,
baseY, targetMountains, targetWater, rand);

            List<Node> path = findPathOnGrid(candidateGrid, spawnX, spawnY,
baseX, baseY, false);
            int fitness = calculateFitness(path, baseX, baseY);

            // СЕЛЕКЦІЯ ЗАЛЕЖНО ВІД СКЛАДНОСТІ
            if (difficulty == Difficulty.HARD) {
                // ВАЖКИЙ РЕЖИМ: Шукаємо найменший бал (найшвидший шлях
до бази)
                if (fitness > 0 && fitness < bestFitness) {
                    bestFitness = fitness;
                    copyGrid(candidateGrid, bestGrid);
```

```
            }
        } else {
            // ЛЕГКИЙ ТА СЕРЕДНІЙ: Шукаємо найбільший бал (найдовший лабіринт)
            if (fitness > bestFitness) {
                bestFitness = fitness;
                copyGrid(candidateGrid, bestGrid);
            }
        }
    }

    this.grid = bestGrid;
    String displaySeed = (seedText != null && !seedText.isEmpty()) ? seedText : String.valueOf(seedValue);

    Toast.makeText(context,
            "Map generated! Seed: " + displaySeed + " | Mode: " + difficulty + " | Fitness: " + bestFitness,
            Toast.LENGTH_LONG).show();
  }

  private int calculateFitness(List<Node> path, int targetX, int targetY) {

    // Якщо алгоритм не зміг побудувати шлях (уперся в стіну),
    // ми імітуємо "погану" хромосому з максимальними штрафами
    if (path == null || path.isEmpty()) {
        int fPen = 10; // Симулюємо множинні зіткнення/глухий кут
        int fDist = Math.abs(targetX) + Math.abs(targetY); // Максимальна віддаленість
        return - (W3 * fPen) - (W4 * fDist); // Функція поверне жорсткий мінус
    }

    // 1. F_len(C): Функція довжини маршруту (кількість валідних кроків)
    int fLen = path.size();

    // 2. F_comp(C): Функція складності геометрії (підрахунок поворотів)
    int fComp = 0;
    for (int i = 2; i < path.size(); i++) {
        Node p1 = path.get(i - 2);
        Node p3 = path.get(i);
        // Зміна напрямку: перевірка на зміщення по обох осях
        if (p1.x != p3.x && p1.y != p3.y) {
            fComp++;
        }
    }
```

```
        // 3. F_pen(C): Функція штрафу за порушення топології
        // Оскільки наш базовий алгоритм (BFS) відкидає самоперетини та стіни ще на етапі пошуку,
        // для побудованого маршруту цей штраф дорівнює нулю.
        // (Але архітектурно ми зобов'язані його тут вказати для відповідності моделі).
        int fPen = 0;

        // 4. F_dist(C): Евристична відстань Чебишова (для 8-направленого руху)
        Node lastNode = path.get(path.size() - 1);
        int dx = Math.abs(targetX - lastNode.x);
        int dy = Math.abs(targetY - lastNode.y);
        int fDist = Math.max(dx, dy); // Вибираємо найбільше зміщення

        // ---------------------------------------------------------
        // ФІНАЛЬНА ФУНКЦІЯ ПРИСТОСОВАНОСТІ
        // F(C) = w1 * F_len(C) + w2 * F_comp(C) - w3 * F_pen(C) - w4 * F_dist(C)
        // ---------------------------------------------------------
        int fitnessScore = (W1 * fLen) + (W2 * fComp) - (W3 * fPen) - (W4 * fDist);

        return fitnessScore;
    }

    private void fillGridWithRandomObstacles(int[][] targetGrid, int spawnX, int spawnY, int baseX, int baseY, int maxMountains, int maxWater, Random rand) {
        int placedMountains = 0;
        int placedWater = 0;
        int attempts = 0;

        while ((placedMountains < maxMountains || placedWater < maxWater) && attempts < 200) {
            attempts++;
            int rx = rand.nextInt(COLS);
            int ry = rand.nextInt(ROWS);

            if (targetGrid[rx][ry] == EMPTY && !(rx == spawnX && ry == spawnY) && !(rx == baseX && ry == baseY)) {

                // Визначаємо, що саме ставити, враховуючи ліміти
                int obstacleType;
                if (placedMountains < maxMountains && placedWater < maxWater) {
                    obstacleType = rand.nextBoolean() ? MOUNTAIN : WATER;
                } else if (placedMountains < maxMountains) {
                    obstacleType = MOUNTAIN;
```

```
                } else {
                    obstacleType = WATER;
                }

                targetGrid[rx][ry] = obstacleType;

                // Перевіряємо, чи не заблокували ми шлях
                if (findPathOnGrid(targetGrid, spawnX, spawnY, baseX, baseY, false)
!= null) {
                    if (obstacleType == MOUNTAIN) placedMountains++;
                    else placedWater++;
                } else {
                    targetGrid[rx][ry] = EMPTY; // Відкат
                }
            }
        }
    }

    private void copyGrid(int[][] source, int[][] destination) {
        for (int x = 0; x < COLS; x++) {
            for (int y = 0; y < ROWS; y++) {
                destination[x][y] = source[x][y];
            }
        }
    }

    //
==========================================================
    // АЛГОРИТМИ ПОШУКУ ШЛЯХУ (BFS)
    //
==========================================================

    // Головний алгоритм, який вміє шукати шлях на БУДЬ-ЯКІЙ сітці
(потрібно для генерації)
    public List<Node> findPathOnGrid(int[][] testGrid, int startX, int startY, int
endX, int endY, boolean canFly) {
        Queue<Node> queue = new LinkedList<>();
        boolean[][] visited = new boolean[COLS][ROWS];

        queue.add(new Node(startX, startY, null));
        visited[startX][startY] = true;

        while (!queue.isEmpty()) {
            Node current = queue.poll();
```

```
            if (current.x == endX && current.y == endY) {
                List<Node> path = new ArrayList<>();
                while (current != null) {
                    path.add(current);
                    current = current.parent;
                }
                Collections.reverse(path);
                return path;
            }

            int[][] directions = {{0,1}, {0,-1}, {1,0}, {-1,0}, {1,1}, {1,-1}, {-1,1}, {-1,-
1}};
            for (int[] dir : directions) {
                int dx = dir[0], dy = dir[1];
                int newX = current.x + dx, newY = current.y + dy;

                if (newX >= 0 && newX < COLS && newY >= 0 && newY < ROWS)
{
                    int cellType = testGrid[newX][newY];
                    boolean isWalkable = (cellType == EMPTY) || (canFly && cellType
== WATER);

                    if (!visited[newX][newY] && isWalkable) {
                        boolean canMove = true;
                        if (dx != 0 && dy != 0) {
                            int side1 = testGrid[current.x + dx][current.y];
                            int side2 = testGrid[current.x][current.y + dy];
                            boolean side1Block = (side1 == TOWER || side1 ==
MOUNTAIN || (!canFly && side1 == WATER));
                            boolean side2Block = (side2 == TOWER || side2 ==
MOUNTAIN || (!canFly && side2 == WATER));
                            if (side1Block || side2Block) canMove = false;
                        }

                        if (canMove) {
                            visited[newX][newY] = true;
                            queue.add(new Node(newX, newY, current));
                        }
                    }
                }
            }
        }
        return null;
    }
```

```
    // Метод-обгортка: GameView викликає його для головної сітки (this.grid)
    public List<Node> findPath(int startX, int startY, int endX, int endY, boolean
canFly) {
        return findPathOnGrid(this.grid, startX, startY, endX, endY, canFly);
    }
}
```