КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
БУДІВНИЦТВА І АРХІТЕКТУРИ

Факультет автоматизації інформаційних технологій

Кафедра інформаційних технологій

(назва випускової кафедри)

ПОЯСНЮВАЛЬНА ЗАПИСКА

ДО КВАЛІФІКАЦІЙНОЇ ВИПУСКНОЇ РОБОТИ

НА ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ БАКАЛАВР

на тему:

ПРОЦЕДУРНА ГЕНЕРАЦІЯ КОНТЕНТУ В ВІДЕОІГРАХ

Калюжко Матвей Миколайович

Київ 2025 р.

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
БУДІВНИЦТВА І АРХІТЕКТУРИ

Факультет автоматизації інформаційних технологій

Кафедра інформаційних технологій

ЗАТВЕРДЖУЮ
Завідувач кафедри ІТ
Тетяна Гончаренко
_____
„___”_____2025 року

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО КВАЛІФІКАЦІЙНОЇ ВИПУСКНОЇ РОБОТИ НА ЗДОБУТТЯ
ОСВІТНЬОГО СТУПЕНЯ БАКАЛАВР

ПРОЦЕДУРНА ГЕНЕРАЦІЯ КОНТЕНТУ В ВІДЕОІГРАХ

Я як здобувач вищої освіти КНУБА розумію і підтримую політику закладу з академічної доброчесності. Я не надавав(-ла) і не одержував(-ла) недозволену допомогу під час підготовки цієї роботи. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

Здобувач
Калюжко Матвей Миколайович
122 «Комп’ютерні науки»
(спеціальність)
Інформаційні управляючі системи і технології
(освітня програма)
Групи КН-21-1
Керівник Рябчун Ю.В.
(прізвище та ініціали)
Доктор філософії
(вчене звання, науковий ступінь)
Рецензент к.т.н., доц. Доля О.В.
(Прізвище та ініціали)

Ідентичність підтверджую

Київ 2025 р.

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БУДІВНИЦТВА І АРХІТЕКТУРИ

Факультет:	Автоматизації інформаційних технологій
Випускова кафедра:	Інформаційних технологій
Освітній ступінь:	Бакалавр
Спеціальність:	Комп'ютерні науки
Освітня програма:	Інформаційні управляючі системи та технології

ЗАТВЕРДЖУЮ

Завідувач кафедри ІТ

Тетяна ГОНЧАРЕНКО

_____

„___"_____2025 року

З А В Д А Н Н Я

ДО ВИКОНАННЯ КВАЛІФІКАЦІЙНОЇ ВИПУСКНОЇ РОБОТИ НА ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ БАКАЛАВР

	Калюжко Матвею Миколайовичу

1. Тема роботи ПРОЦЕДУРНА ГЕНЕРАЦІЯ КОНТЕНТУ В ВІДЕОІГРАХ

затверджена наказом ректора КНУБА № 235/23/25 від «14» лютого 2025 року

2. Керівник роботи	Рябчун Юлія Володимирівна, PhD

3. Строк подання Здобувачем роботи до захисту_____

4. Зміст пояснювальної записки за розділами:

Р.1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

Р.2 МЕТОДИ ТА МОДЕЛІ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ ОПОВІДНОГО КОНТЕНТУ

Р.3 РОЗРОБКА СИСТЕМИ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ ОПОВІДНОГО КОНТЕНТУ

Р.4 ПРАКТИЧНА РЕАЛІЗАЦІЯ СИСТЕМИ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ СЮЖЕТНОГО КОНТЕНТУ

5. Графічний матеріал за розділами:

Р.1_____

Р.2_____

Р.3_____

Р.4_____

6. Календарний план виконання роботи:

Види робіт та їх зміст	Дата виконання
Розділ 1	31.01.2025
Розділ 2	23.02.2025
Розділ 3	09.03.2025
Розділ 4	14.05.2025
Остаточне оформлення роботи	20.05.2025
Направлення роботи для перевірки на плагіат	22.05.2025
Попередній захист роботи на випусковій кафедрі	22.05.2025
Направлення роботи на рецензування	23.05.2025

7. Консультанти розділів атестаційної випускної роботи

Розділ	Прізвище, ініціали та посада консультанта	Перевірив	
		дата	підпис
Розділ 1	Рябчун Ю.В., доц.каф.ІТ	31.01.2025	
Розділ 2	Рябчун Ю.В., доц.каф.ІТ	23.02.2025	
Розділ 3	Рябчун Ю.В., доц.каф.ІТ	09.03.2025	
Розділ 4	Рябчун Ю.В., доц.каф.ІТ	14.05.2025	

8. Дата видачі завдання _____

Зав. кафедри			Гончаренко Т.А.
	(підпис)		(прізвище та ініціали)
Керівники			Рябчун Ю.В.
	(підпис)		(прізвище та ініціали)
Здобувач			Калюжко М.М.
	(підпис)		(прізвище та ініціали)

АНОТАЦІЯ

Калюжко М.М. Процедурна генерація контенту в відеоіграх.

Атестаційна випускна робота бакалавра за спеціальністю 122 «Комп’ютерні науки», освітня програма «Інформаційні управляючі системи та технології». – Київський національний університет будівництва та архітектури. – Київ, 2025.

Дана дипломна робота присвячена дослідженню та розробці системи процедурної генерації квестів у відеоіграх. Основна мета роботи – створення алгоритму, який дозволяє динамічно генерувати сюжетні завдання з урахуванням контексту гри, вибору користувача та внутрішніх механік, забезпечуючи при цьому логічну зв’язність оповідання.

Робота включає аналіз існуючих методів процедурної генерації, визначення ключових проблем при створенні адаптивних квестів та розробку моделі генерації, яка балансує між випадковістю та зв’язністю оповідання. Практична частина містить реалізацію прототипу, що демонструє роботу запропонованої системи та її можливості у формуванні динамічного оповідного контенту.

Експериментальна частина роботи оцінює якість згенерованих квестів, їх різноманітність та відповідність сюжетному контексту. На основі отриманих результатів сформовано висновки та рекомендації щодо подальшого вдосконалення підходів до процедурної генерації оповідного контенту у відеоіграх.

Робота викладена на 90 аркушах, містить 2 додатки, 19 рисунків, список використаних джерел із 35 найменувань.

Ключові слова: PCG, ПЗ, Процедурна Генерація Оповідання, Java.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

PCG (Procedural Content Generation) – Процедурна генерація контенту;

ПЗ – Програмне Забезпечення;

Процедурна Генерація Оповідання –

Java

ВСТУП

У сучасному світі, відеоігри займають важливе місце не лише в індустрії розваг, а й у розвитку технологій. Якщо раніше люди розглядали їх як суто черговий вид розваг та дозвілля, у реаліях сьогодення, індустрія розробки ігор вже є перспективним напрямком у сфері ІТ. Із кожним днем індустрія розробки ігор стрімко розвивається, є рушійною силою розвитку нових технологій, та посідає значуще місце у сфері цифрового мистецтва.

З постійним зростанням обсягів ігрового контенту, розробники стикаються із викликами, пов’язаними з необхідністю створення великих та різноманітних ігрових світів при збереженні високої якості й унікальності контенту. У відповідь на ці виклики все ширше застосовується процедурна генерація контенту (Procedural Content Generation, PCG) – метод використання алгоритмів для створення ігрового контенту автоматично.

PCG використовується у різних аспектах розробки відеоігор: створення рівнів, персонажів, квестів, діалогів, карт світу тощо. Відомі ігри, такі як Minecraft, No Man’s Sky, The Elder Scrolls II: Daggerfall, та The Binding of Isaac активно використовують PCG для забезпечення унікального ігрового досвіду. Сучасні алгоритми процедурної генерації базуються на різних методах, включаючи випадкові алгоритми, генетичні алгоритми, нейронні мережі та інші підходи.

Попри значні переваги PCG, такі як економія ресурсів розробки та підвищення реіграбельності, існує низка проблем, зокрема недостатня контрольованість створеного контенту, ризик отримання неякісних або нелогічних результатів, а також складнощі у створенні процедурно згенерованих сюжетів, які були б цілісними та захопливими.

Сучасні дослідження спрямовані на вдосконалення PCG у сфері інтерактивного оповідання та глибшої інтеграції з системами штучного інтелекту. Це відкриває нові можливості для розробки ігор з адаптивними історіями, де сюжет динамічно змінюється відповідно до дій користувача.

Дана дипломна робота присвячена аналізу процедурної генерації контенту в відеоіграх, зокрема у сфері оповідання, та дослідженню можливих методів покращення цієї технології для створення більш якісних інтерактивних історій.

Метою даної роботи є аналіз сучасних підходів та процедурної генерації контенту у відеоіграх та дослідження можливостей використання PCG для створення інтерактивних сюжетів.

Для досягнення цієї мети було поставлено такі завдання:

- Провести аналіз сучасного стану PCG у відеоіграх.
- Дослідити алгоритми та підходи для створення оповідного контенту.
- Визначити основні проблеми та виклики, пов’язані з використанням PCG для дизайну оповідного контенту.
- Розробити та протестувати прототип системи, що використовує PCG у оповідному контексті.
- Оцінити вплив запропонованого прототипу та методики на ігровий досвід.

Завдання дослідження:

Об’єкт дослідження – процес автоматичної генерації ігрового контенту за допомогою алгоритмічних методів, а також підходи до реалізації процедурної генерації у сучасних відеоіграх.

Предмет дослідження – алгоритми та методи процедурної генерації контенту.

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Поняття процедурної та випадкової генерацій

У сучасній індустрії розробки відеоігор процедурна генерація контенту відіграє важливу роль у процесі створення унікальних наповнених контентом ігрових світів, персонажів, та інших елементів. Але що ж таке ця процедурна генерація і чому вона називається саме так?

Вся справа в тому, що процедурна генерація (PCG) – автоматизований метод створення даних за певним рядом правил (алгоритмів), поєднаних із елементом випадковості (часто контрольованої) [1]. Вона використовується для автоматичного створення великої кількості контенту у відеоіграх, моделей та текстур у сфері комп'ютерної графіки, для створення композицій у різних жанрах музики, та навіть для синтезу мовлення. Зазвичай PCG протиставляють ручному способу створення контенту аніматорами, дизайнерами, художниками тощо.

Хоча розробники ПЗ користуються засобами PCG вже роками, кінцевий результат частіше включає вже напрацьовані та «відфільтровані» варіації різних одне від одного елементів. Це пов'язано із тим фактом, що при використанні PCG варіація та кількість згенерованих елементів та унікальних варіацій, з теоретичної точки зору, насправді, може сягати нескінченності. Однак без додаткової обробки результати PCG можуть здаватися повторюваними. Це явище отримало назву «процедурна вівсянка», яку ввела письменниця Кейт Комптон. Вона зазначає, що хоча математично можливо сформувати тисячі унікальних варіацій, вони все одно здаватимуться однаковими, якщо їм бракує глибини та виразності. [2]

Чому ж тоді PCG використовується так часто, якщо увесь контент потенційно може стати просто «процедурною вівсянкою»? Якщо усі результати генерацій треба все одно переглядати вручну, та обирати підходящі? Відповідь, насправді, доволі проста – вся справа у перевагах цього методу над ручною генерацією. А саме:

- Швидкість – комп'ютерні алгоритми дозволяють генерувати великий обсяг контенту значно швидше, ніж це можливо вручну.

- Контрольованість та налаштованість – оскільки PCG створюється алгоритмами, можливо налаштувати алгоритми під спеціальні вимоги, аби отримати необхідний контент.
- Економія часу – комп’ютер створює унікальний контент швидше аніж людина.
- Економія ресурсів – оскільки контент генерується комп’ютером, на задачу використовується менше ресурсів, даючи можливість розподілити їх на інші аспекти розробки.
- Багаторазове використання – генератори що генерують контент можуть бути використані у інших проектах, чи додатках.
- Практичність – методи PCG передбачають «конструювання» контенту з наданих параметрів, нібито з конструктору. Це дає свободу до генерації тисяч можливих варіацій контенту, що займатиме врази менше місця у вигляді шматочків що використовуються багато разів, аніж великих конструкцій вже згенерованого контенту.
- Унікальність досвіду – методи PCG мають потенціал надати унікальний досвід кожному користувачу, таким чином роблячи продукт більш привабливим у очах користувачів. Підвищуючи потенціал до реіграбельності.
- «Жива» симуляція – PCG має змогу генерувати контент, що не здаватиметься репетативним, та непередбачуваним, що є позитивною ознакою.

До всього до цього, хоча саме поняття PCG частіше сприймається як повністю випадкова генерація контенту, вона, насправді, не завжди повністю випадкова. Більшість сучасних алгоритмів використовують контрольовану випадковість, коли результат визначається певними правилами та параметрами. Але усього існує 3 основних підходи до використання PCG:

- Повна випадковість [3] – коли кожен елемент створюється незалежно та без обмежень (наприклад, випадкові карти у The Binding of Isaac) .

- Псевдовипадковість [4] – коли при створенні контенту використовуються генератори псевдовипадкових чисел, що дають передбачувані результати при однакових початкових умовах (як seed-и у Terraria та Minecraft).
- Керована випадковість [5] – комбінація випадкових та детермінованих правил, що гарантують узгодженість та геймплейний баланс (наприклад, у Hades).

Процедурна генерація відіграє ключову роль у створенні різноманітного та динамічного ігрового досвіду. Подальші підрозділи розглянуть основні алгоритми PCG та їхні переваги та недоліки.

1.2. Основні алгоритми процедурної генерації

Процедурна генерація базується на використанні алгоритмів, що дозволяють автоматично створювати контент за певним рядом правил. На сьогодні типів алгоритмів PCG існує дійсно безліч. Розглянемо декілька з найбільш поширених.

1.2.1. Алгоритми на основі шуму

Одним із основних методів у PCG є метод генерації шуму. Цей метод дозволяє створювати реалістичні текстури, висотні карти та природні форми. До найбільш поширених алгоритмів PCG за допомогою генерації шуму належать:

- **Шум Перліна** – розроблений Кеном Перліном у 1983 році як результат його розчарування у машинному вигляді комп’ютерної графіки тих часів. Є примітивом процедурних текстур, що належить до градієнтних шумів. Художники використовують його для того аби зробити комп’ютерну графіку більш реалістичною (див. рис. 1.1). Результат алгоритму є псевдовипадковим, але всі візуальні деталі мають однаковий розмір. Часто використовується під час генерації текстур у комп’ютерній графіці. [6]

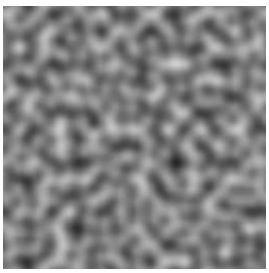


Рисунок 1.1 Результат роботи генератора Шуму Перліна
Джерело: [26]

- **Симплекс-шум** – покращена версія Шуму Перліна. Являє собою метод побудови n-вимірної функції шуму, але з меншою кількістю артефактів (див. рис. 1.2). Розроблений Кеном Перліном задля усунення обмеження класичної функції шуму, особливо помітно за вищої кількості вимірів. [7]

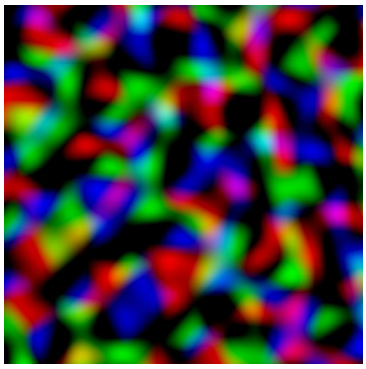


Рисунок 1.2 Результат роботи генератора Симплекс-шуму

Джерело: [27]

- **Шум Уорлі-Вороной** – метод генерації PCG, основою якого є використання діаграми Вороной (Voronoi diagram) [8]. Розроблено у 1996 році Стівеном Уорлі (Steven Worley). Може бути диференційовано для створення нормальної карти шуму. (див. рис. 1.3 та 1.4). [9]

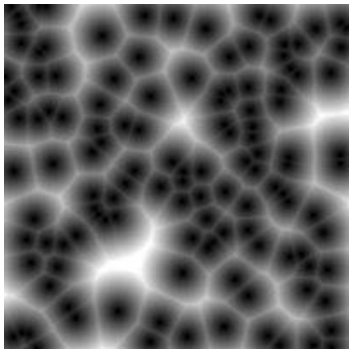


Рисунок 1.3 Результат роботи алгоритму Уорлі

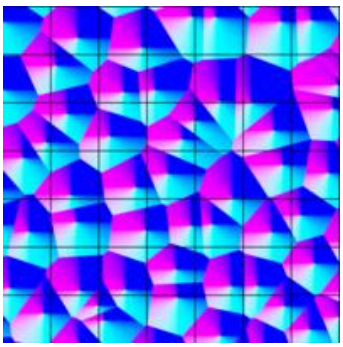


Рисунок 1.4 Результат роботи алгоритму Уорлі (диференційований)

1.2.2. Фрактальні алгоритми

Для створення більш складних та детальних текстур за допомогою PCG використовують метод генерації фракталів. Фрактал [10] – у поширеному розумінні становить собою структуру, що складається з частин, які в певному сенсі подібні до цілого. Тобто, точно, або наближено, збігається із частиною себе. Не всі самоподібні множини є фрактальними і не всі фрактальні множини є самоподібними. Загалом у PCG фрактальні алгоритми використовують для генерації ландшафтів, рослини, та текстур.

- **L-системи (Lindenmayer Systems)** – системи паралельного запису, що були розроблені угорським біологом Арістідом Лінденмаєром (Aristid Lindenmayer) н 1968 році із ціллю моделювання поведінки клітин рослин. Система складається з алфавіту символів, та працює завдяки правилам розширення початкових символів у більші рядки символів. Початковий рядок, з якого починається побудова та розширення в такій системі називають «аксіомою». Після ряду перетворень та розширень початкової «аксіоми» у згенеровані рядки, відповідно до системи правил, згадані рядки надалі переводяться у геометричні структури (див. рис. 1.5). [11] L-системи часто використовують для генерації дерев та рослин у комп’ютерній графіці та іграх.

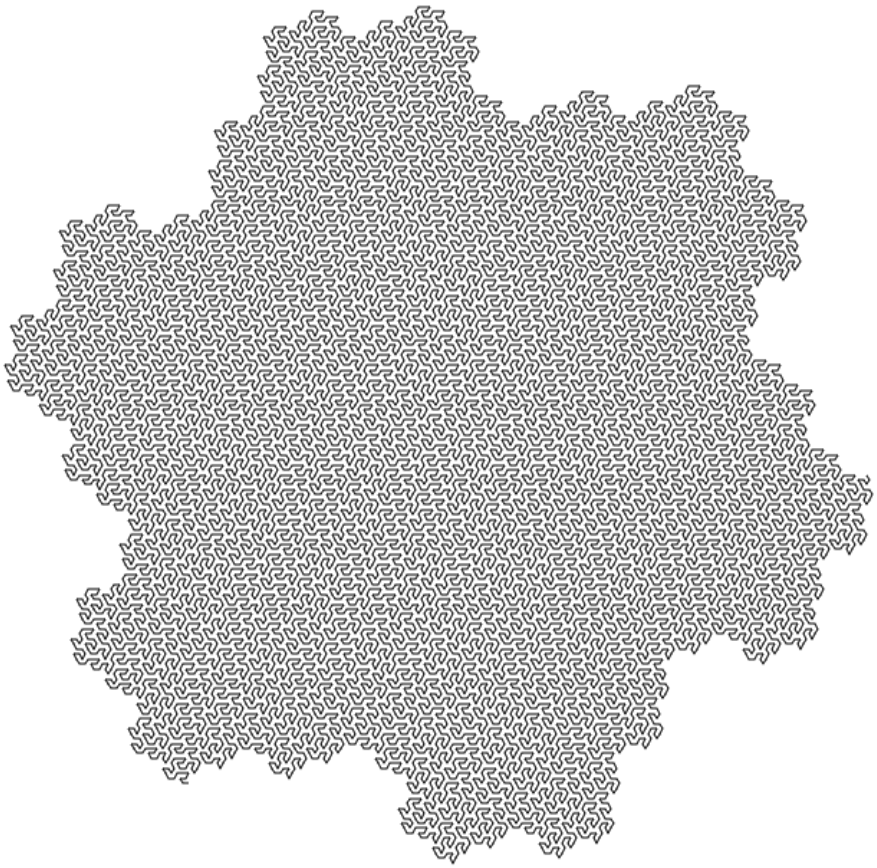


Рисунок 1.5 Результат роботи генератора L-системи

Джерело: [28]

- **Фрактальний ландшафт (Diamond-Square)** – метод створення карт висот для комп'ютерної графіки. Цей алгоритм є кращим за три-вимірну реалізацію алгоритму зміщення середньої точки фракталів, що створює у результаті двовимірні пейзажі (див. рис. 1.6). Цей метод також відомий як «метод генерації плазми», «метод генерації фракталів хмар», та «метод випадкового змішення середньої точки фракталів». Уперше ідею даного методу представили у 1982 році Фурньє (Fournier), Фюссел (Fussell), та Карпентер (Carpenter) у журналі SIGGRAPH. Основна ідею алгоритму в тому, що на двовимірній сітці випадковим чином генерується висота місцевості з чотирьох початкових значень, розташованих у сітці точок, щоб уся площина була покрита квадратами. [12]

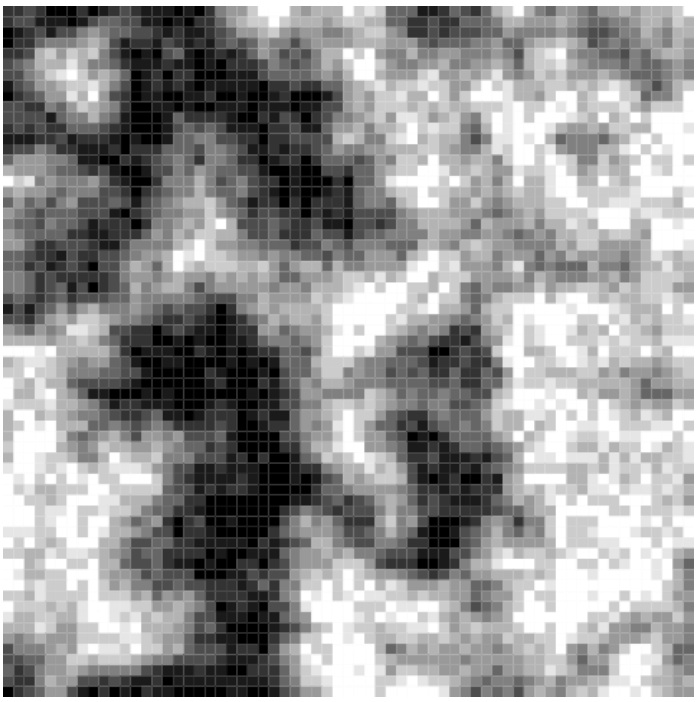


Рисунок 1.6 Результат роботи генератора за алгоритмом Diamond-Square

Джерело: [29]

1.2.3. Графові алгоритми та генерація карт

Для створення рівнів, доріг, та комплексних біомів.

- **Клітинні автомати (Cellular Automata)** – дискретна математична модель системи об’єктів «клітин», що живуть на сітці. Клітини перебувають у одному з двох станів «активна», чи «неактивна», що в бінарній системі представлено у вигляді 0 (нема сигналу, неактивна) та 1 (є сигнал, активна). Для кожної клітинки набір клітинок-сусідів називається околицею, та частіше представлено у вигляді списку. Після початкової ідентифікації станів для кожної комірки на сітці, задається ряд правил, що зазвичай є однаковим для кожної клітини. Початкові дані зберігаються із початковим часом (t=0), та далі відбувається прогресія відповідно до заданих правил (частіше визначених у вигляді математичної функції), результатом є створення наступного покоління (t=1). Основний принцип – зміна стану клітин спираючись на стан клітин минулого покоління, та стан клітин-сусідів у околиці. Правило для оновлення клітинок, зазвичай, є фіксованим, та не змінюється із часом, при цьому застосовуючись на всю сітку

одночасно. Створення методу приписується Станіславу Уламу (Stanislaw Ulam) та Джону фон Нейману (John von Neumann) у 1940 році у Національній Лабораторії Лос-Аламос в Нью-Мексико. [13]

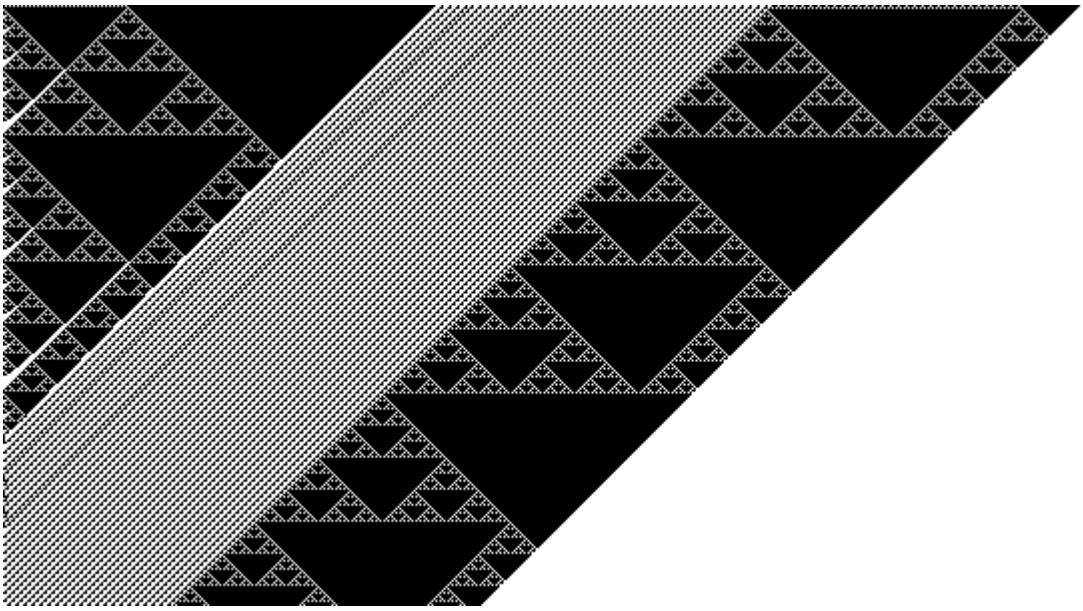


Рисунок 1.7 Результат роботи генератора Елементарного Клітинного Автомата

Джерело: [30]

- **Алгоритм тріангуляції Делоне (Delaunay Triangulation)** – розроблений у 1934 році Борисом Делоне (Boris Delaunay), цей метод, будучи частиною обчислювальної геометрії, являє собою таку тріангуляцію множини точок, що жодна точка множини не знаходиться всередині описаних довкола трикутників кіл у наданій множині. Умова Делоне стверджує, що мережа трикутників є тріангуляцією Делоне, якщо всі описані кола трикутників пусті. Коло описане навколо трикутника вважається пустим, якщо воно не містить вершин трикутника інших ніж ті три, що становлять його. Загалом, цей метод використовується у двовимірних площинах, але він також можу бути адоптований для тривимірних площин, якщо замість кіл використовувати сфери. [14]
- **Метод Бінарного Розподілення Простору (BSP – Binary Space Partitioning)** – метод розбиття простору, що рекурсивно розбиває евклідів простір на два опуклі множини, використовуючи гіперплощини як розбиття.

Розроблений у 1969 році в контексті тривимірної комп'ютерної графіки, структура цього метода є корисною для візуалізації об'єктів на сцені відносно користувача, оскільки вона ефективно надає просторову інформацію про об'єкти в сцені. Через свої особливості використовується для генерації кімнат та коридорів. [15]

1.2.4. Евристичні та еволюційні алгоритми

Деякі підходи використовують генетичні алгоритми та штучний інтелект

- **Генетичні алгоритми (Genetic Algorithms)** [16] – еволюційний алгоритм пошуку, що використовується для генерації високоякісних рішень питань оптимізації та пошуку, методами натхненними такими біологічними еволюційними процесами, як селекція (вибір), схрещення або кросовер (рекомбінація), та мутація. Генетичні алгоритми часто напряму пов'язані із системами пошуку та штучним інтелектом. Фактично, являючись симуляцією процесу еволюції, цей метод складається з 3 основних стадій: ініціалізації – задається «популяція», що складається з «генетичного коду» (рядків бітів); селекція – із кожною ітерацією алгоритму (із кожним новим поколінням) із існуючої популяції стохастично відбирається, як правило, найбільш підходяща та пристосована частина «популяції» що є найближчою до заданого конструктивного рішення проблеми, напряму спираючись на функцію пристосованості [17] (тип цільової функції що використовується для порівняння існуючих значень із бажаним результатом вирішення проблеми), що буде використана під час наступної ітерації; мутації та рекомбінація – до часток популяції обраної селекцією застосовуються «генетичні оператори» - оператор схрещення та оператор мутації, результати яких зберігаються для наступної ітерації популяції. Стадія селекції і стадія мутації та рекомбінації повторюються із наступними поколіннями до тих пір поки не отримано бажаного результату та алгоритм не буде зупинено, або поки процес циклування не відбувся задану кількість разів. Якщо неможливо вичислити функцію пристосованості, то прибігають

до використання симуляцій та вичислення фенотипів (виразів генетичного коду у реальному середовищі). В таких випадках може потребуватися людська оцінка для отримання найбільш наближеного до бажаного результату. Приклад прогресії генетичного алгоритму зображено нижче (див. рис. 1.8 – 1.10). Джерело: [31]

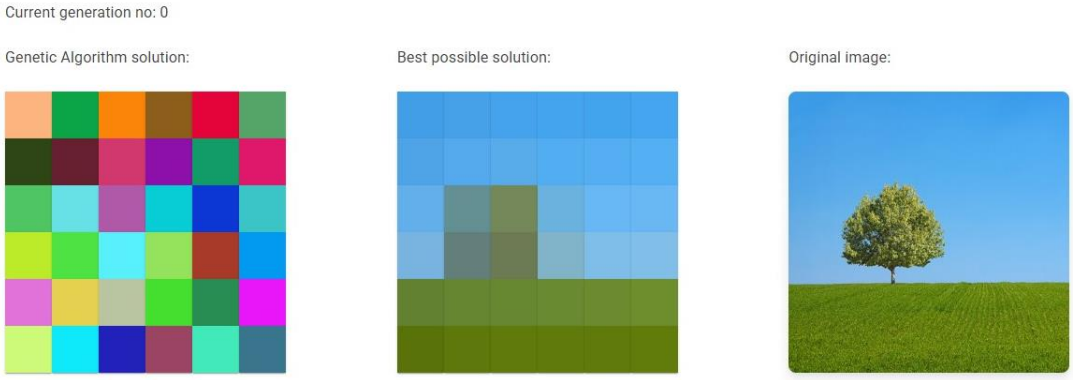


Рисунок 1.8 Початкові дані у генераторі Генетичного Алгоритму.(ліве зображення – початкова популяція, праве зображення – бажаний результат очами людини, центральне зображення – ідеальний можливий результат)

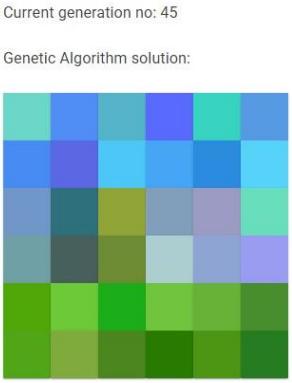


Рисунок 1.9 Отримана популяція після 45-ої ітерації у генераторі Генетичного Алгоритму

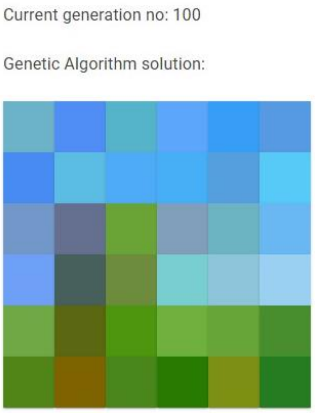


Рисунок 1.10 Фінальний результат нашої популяції після 100-ої ітерації у генераторі Генетичного Алгоритму

- **Ланцюг Маркова (Markov Chains)** – у теорії ймовірностей ланцюг Маркова є стохастичним процесом, що описує послідовність подій, у яких ймовірність кожної наступної події залежить напряму від стану, досягнутого в попередній події. Іншими словами, прогнози базуються виключно на основі поточного стану. Ланцюги Маркова мають багато застосувань як статистичні модулі даних та процесів реального світу. Вони забезпечують основу для загальних методів стохастичного моделювання, відомих як ланцюг Маркова Монте-Карло. [18] Ланцюги Маркова також є потужним інструментом для генерації тексту, особливо коли необхідно створити послідовність, яка статистично нагадує вхідний матеріал. Цей метод базується на аналізі ймовірностей переходу між елементами тексту, такими як символи, слова або фрази, залежно від обраного порядку моделі. Це може бути корисним для створення поезії, імітації стилю певного автора або навіть для генерації випадкових технічних текстів. Наприклад, у роботі "Порівняння ланцюгів Маркова та нейромереж для вирішення задачі генерації" описано алгоритми генерації віршів за допомогою ланцюгів Маркова та нейромереж, а також охарактеризовано переваги та недоліки кожного з підходів. [19]

1.3. Порівняльний аналіз алгоритмів PCG

1.3.1. Алгоритми на основі шуму

Алгоритм	Основна ідея	Застосування	Переваги	Недоліки
Шум Перліна	Генерує згладжений шум із плавними градієнтами.	Карти висот, текстури, хмари, вода.	Легко контролювати, створює природний вигляд.	Вразливий до повторень у великих масштабах.
Симплекс-шум	Покращена версія шуму Перліна, яка використовує симплекси замість ґраток для зменшення артефактів.	Генерація карт висот, текстур, процедурних світів, ефектів у 2D/3D/4D просторах.	Менше артефактів, швидший у високих вимірах, плавніші переходи..	Складніший у реалізації.
Шум Уорлі-Вороной	Створює мозаїкоподібні структури, розбиті на осередки.	Біоми, розподіл територій, картографія.	Генерує природні патерни, підходить для генерації карт.	Не забезпечує гладкого переходу між зонами.

1.3.2. Фрактальні алгоритми

Алгоритм	Основна ідея	Застосування	Переваги	Недоліки
L-системи	Рекурсивні правила для генерації рослинних структур.	Дерева, рослини, архітектурні елементи.	Проста структура, висока деталізація.	Не завжди ефективний у великомасштабних світах.

Фрактальний ландшафт (Diamond-square)	Рекурсивний поділ площини для створення гірських масивів.	Ландшафти, карти висот.	Генерує реалістичні природні структури.	Потребує постпроцесингу для усунення артефактів.

1.3.3. Графові алгоритми та генерація карт

Алгоритм	Основна ідея	Застосування	Переваги	Недоліки
Клітинні автомати	Локальні правила визначають розвиток клітин у сітці.	Генерація печер, лабіринтів, біомів.	Простота реалізації, хороші результати.	Важко контролювати кінцевий вигляд карти.
Тріангуляція Делоне	Створює зв’язки між точками для отримання графа.	Рівні з відкритими просторами, дороги, біоми.	Висока контрольованість форми.	Не підходить для вузьких коридорів.
Метод Бінарного Розподілення Простору	Ієрархічний поділ простору на секції.	Підземелля, будівлі, рівні з кімнатами.	Дозволяє контроль над розміром кімнат.	Може створювати надто регулярні структури.

1.3.4. Еврестичні та еволюційні алгоритми

Алгоритм	Основна ідея	Застосування	Переваги	Недоліки
Генетичні алгоритми	Відбір, схрещення та мутація для пошуку	Оптимізація рівнів, налаштування AI.	Може знаходити унікальні рішення.	Високі обчислювальні витрати.

	найкращих рішень.			
Ланцюг Маркова	Імовірнісний підхід до прогнозування наступного стану.	Генерація тексту, діалогів, музики.	Добре підходить для послідовних структур.	Обмежена контрольованість, потребує великого обсягу даних.

Кожен алгоритм має свої сильні та слабкі сторони. Для створення складного ігрового світу зазвичай використовується комбінація кількох методів, щоб збалансувати випадковість, контроль і продуктивність.

1.4. Основні проблеми процедурної генерації

Як ми вже розглянули, методи процедурної генерації мають значні переваги перед ручним створенням контенту у відеоіграх. Вони дозволяють значно знизити витрати часу та ресурсів, а також створювати динамічні світи. Здається, що достатньо лише розробити правильний алгоритм і можна повністю автоматизувати процес: гра створюватиметься сама собою, а розробники лише коригуватимуть результати.

На жаль, на практиці все не так просто. Незважаючи на ефективність PCG, повна автоматизація контенту залишається складним завданням. Процедурна генерація стикається з низкою проблем, які обмежують її використання як єдиного інструменту для розробки ігор. Нижче ми розглянемо основні виклики пов’язані із використанням PCG.

1.4.1. Контрольованість та передбачуваність

Процедурна генерація може створювати величезну кількість варіацій, але це також означає, що результат може бути важко передбачити. Розробники змушені ретельно налаштовувати алгоритми, щоб уникнути генерації нелогічного контенту. Наприклад, у генерації рівнів важливо, щоб користувач міг їх пройти, а в генерації сюжетів – щоб історія не втрачала сенсу.

1.4.2. Якість контенту та його сприйняття користувачем

Ручна робота дозволяє розробникам втілювати унікальні та детально опрацьовані ідеї, тоді як PCG часто дає передбачувані або «знеособлені» результати. Наприклад, багато ігор з процедурно згенерованими рівнями можуть страждати від одноманітності або відсутності унікального дизайну, що робить їх менш цікавими для гравців.

1.4.3. Оптимізація продуктивності

Процедурні алгоритми можуть бути ресурсомісткими, особливо якщо вони працюють у реальному часі. Чим складніший алгоритм, тим більше обчислювальної потужності він потребує.

1.4.4. Тестування та налагодження

Процедурно згенерований контент складно тестувати, оскільки кожний запуск алгоритму може давати нові результати. Це, у свою чергу, означає, що розробники не можуть протестувати всі можливі варіанти вручну, що ускладнює пошук і виправлення помилок та можливих багів.

1.4.5. Баланс та відповідність ігровому дизайну

Важливо не лише створити контент, а й зробити його цікавим для користувача. Процедурна генерація може створювати занадто складні або, навпаки, надто прості рівні, а також незбалансовані механіки.

1.4.6. Складність інтеграції у сюжетні ігри

У сюжетних іграх важливо, щоб історія мала логічний розвиток. Процедурна генерація діалогів або квестів може призводити до нелогічних або повторюваних ситуацій, що знижує якість занурення у гру. Це одна з причин, чому PCG частіше використовується у геймплейних елементах (генерації рівнів, ландшафтів), а не у сюжеті.

Та попри всі ці виклики процедурна генерація залишається високо затребуваною у розробці відеоігор. Її використання дозволяє створювати унікальний

і динамічний ігровий досвід, але для досягнення найкращих результатів розробники часто комбінують PCG із ручною розробкою та контролем. Баланс між автоматизацією та контролем є ключем до створення якісного PCG.

1.5. Аналіз існуючих рішень

Сучасні технології не стоять на місці. Більшість проблем згаданих у минулому підрозділі вже мають методи, або техніки, що допомагають частково або повністю їх вирішити… Розглянемо їх нижче.

1.5.1. Гібридні методи генерації

Гібридні методи в процедурній генерації контенту поєднують як автоматичні алгоритми, так і елементи ручного дизайну. Вони дозволяють поєднати кращі риси різних підходів, зберігаючи високу ефективність і контроль за результатом. Гібридні методи активно використовуються в індустрії, оскільки дозволяють отримати широкий спектр результатів, одночасно підтримуючи певний рівень креативного контролю.

- **Ручна розмітка + алгоритмічна генерація** [20] – Один із найпоширеніших підходів гібридних методів генерації – це комбінування ручної розмітки та алгоритмічної генерації. В цьому випадку дизайнери задають основну структуру рівня або середовища, а алгоритм генерує деталі. Це дозволяє зберегти контроль за глобальною організацією контенту, при цьому забезпечуючи автоматичну генерацію нескінченних варіантів. **Приклад:** У Shadows of Doubt поєднано алгоритмічну генерацію контенту з частковою ручною розміткою, надаючи унікальний досвід кожного разу. Генерація карти міста, наповнюваності апартаментів, імен людей, та обставин злочинів для кожного розслідування є випадковою, що забезпечує високу варіативність у геймплеї…
- **Шаблонні структури та елементи випадковості** [21] – Цей підхід включає використання заздалегідь створених структур або блоків, які потім випадково комбінуються для створення різноманітних результатів. Генерація рівнів,

середовищ або контенту відбувається шляхом поєднання цих структур з урахуванням випадковості та певних правил.
Приклад: У Minecraft для генерації структур використовуються заздалегідь створені шаблони компонентів. Якщо розглядати поселення, то шаблонами будуть будівлі, що розміщуються випадковим чином у межах певної території. Алгоритм не генерує поселення з нуля, але визначає де і як ці шаблони будуть з'являтися на карті, забезпечуючи величезну варіативність без втрати узгодженості.

- **Розширені графові алгоритми** [22] – Графові алгоритми використовують структури, що складаються з вузлів і з'єднаних між собою елементів (ребер). Цей підхід зазвичай застосовується для генерації карт або рівнів, що мають логічну взаємозв'язку між різними частинами. Розширені графові алгоритми комбінують принципи процедурної генерації і ручного контролю, щоб досягти ефективної та організованої генерації.
Приклад: No Man's Sky використовує складну систему алгоритмів для створення процедурних планет та їхніх екосистем. Алгоритми генерують планети на основі заздалегідь розроблених шаблонів і даних (наприклад, тип атмосфери, ландшафт), а також забезпечують велику варіативність, змішуючи ці шаблони

Гібридні методи генерації є надзвичайно перспективними для ігрової індустрії, адже дозволяють зберігати баланс між контролем і варіативністю, а також оптимізувати процес створення контенту. Однак, незважаючи на їхні переваги, вони все ж мають певні обмеження в плані гарантованої якості та складності налаштування. Вибір методу завжди залежить від конкретних цілей і вимог проекту, тому важливо ретельно оцінювати, які техніки комбінувати для досягнення найкращого результату

1.5.2. Використання ШІ та нейромереж

- **Генерація тексту та діалогів** [23] - Однією з найбільш революційних змін у світі процедурної генерації є використання великих мовних моделей,

таких як GPT (Generative Pretrained Transformer), для створення унікальних і динамічних текстів. Ці моделі можуть генерувати природні діалоги, описи подій і навіть цілі історії, що можуть бути інтегровані в ігровий процес. Особливо потужними є можливості генерації текстів для персонажів у відкритих світах, де кожна взаємодія може бути унікальною та залежати від вибору користувача.

Приклад: У AI Dungeon використовується модель GPT для генерації тексту, кожен вибір користувача може призвести до нової частини історії, яку модель генерує на основі попередніх дій. Це дозволяє створювати безліч варіантів сюжетів, що адаптуються до стилю гри кожного користувача.

- **Допомога в балансуванні рівнів** [24] – Штучний інтелект активно застосовується для балансування рівнів і складності в іграх. Збір даних про проходження користувачами та їхні вибори дозволяє ШІ оптимізувати рівні для різних стилів гри, що покращує досвід користувачів і робить його більш інклюзивним та адаптивним.

  Приклад: У Left 4 Dead використовується так званий «мозковий модуль» (AI Director), який динамічно змінює рівень складності на основі дій користувача. Якщо користувач або група починають успішно долати рівень, ШІ може збільшити складність, додавши більше ворогів або змінюючи ландшафт **рівня.**

- **Генерація нових механік і рівнів** [25] – Іншим важливим напрямком є використання нейромереж для створення нових механік і рівнів. Завдяки здатності нейромереж навчатися на існуючих даних, вони можуть генерувати нові варіанти рівнів або ігрових механік, зберігаючи при цьому баланс та інтеграцію з іншими елементами гри. Такі системи можуть навчатися на попередніх рівнях, а потім пропонувати нові варіанти на основі їхніх характеристик.

  Приклад: У No Man’s Sky використовується нейромережевий підхід для генерування нових планет, на яких враховуються екосистеми, ресурси, типи

місцевості і атмосферні умови. Це дозволяє створювати тисячі унікальних планет, які все одно виглядають органічно, навіть якщо їхні характеристики визначаються лише алгоритмами.

Штучний інтелект та нейромережі мають великий потенціал для революції в процедурній генерації контенту. Вони дозволяють створювати адаптивні, гнучкі системи генерації, які можуть ефективно реагувати на вибори гравців, генерувати складні тексти та історії, а також створювати нові механіки і рівні. Однак ці технології також несуть певні виклики, зокрема потребу в великих обчислювальних ресурсах та можливі проблеми з якістю контенту. Тому важливо правильно налаштовувати ці системи, щоб досягти балансу між автоматизацією і контролем

Процедурна генерація контенту має безліч переваг у розробці ігор, надаючи можливість створювати масштабні, варіативні та адаптивні ігрові світи та механіки. Використання таких підходів дозволяє значно зменшити обсяг ручної праці і забезпечити унікальний досвід для кожного користувача. Однак, як показано в аналізі, існують і деякі обмеження, пов’язані з контролем якості контенту, технічними складнощами реалізації і ризиком повторюваності елементів. Враховуючи ці проблеми, необхідно впроваджувати додаткові методи, які дозволять оптимізувати та покращити використання процедурної генерації в іграх.

Зараз ми маємо чітке розуміння як можливостей, так і проблем, що виникають у процесі використання PCG. Однак для повного усвідомлення потенціалу цієї технології важливо зрозуміти, які конкретні задачі можуть бути вирішені в рамках сучасних ігор за допомогою процедурної генерації, а також на яких етапах розробки вона може бути максимально ефективною.

1.6. Постановка задачі дослідження

Враховуючи раніше обговорені проблеми та переваги процедурної генерації контенту, наступним етапом є визначення конкретних напрямків, де ці методи можуть бути найбільш ефективними. У рамках цієї роботи ми зосередимося на

вивченні застосування PCG у створенні оповідного контенту – однієї з найбільш складних і творчих складових ігор.

Процедурна генерація оповідань відкриває нові можливості для створення унікальних і захоплюючих сюжетів, які адаптуються до вибору користувача та його взаємодії з ігровим світом. Вона дозволяє не лише знизити витрати на створення великої кількості тексту, але й створювати динамічні історії, де кожен вибір має значний вплив на розвиток подій. Однак реалізація цього підходу пов’язана з певними труднощами, зокрема, у забезпеченні зв’язності контенту, збереженні логіки сюжету та інтеграції з іншими ігровими механіками.

Задачею дослідження є вивчення ефективних методів застосування PCG для створення оповідного контенту, зокрема фокусуючись на алгоритмах, які здатні збалансувати випадковість і зв’язність історії. Увага буде приділена розробці системи генерації квестів, яка дозволяє гнучко інтегрувати адаптивні місії у сюжет, зберігаючи логічну послідовність та взаємодію з іншими елементами гри. У процесі роботи також буде проаналізовано обмеження цих методів і запропоновано можливі способи їх подолання.

1.7. Дерево цілей

2. МЕТОДИ ТА МОДЕЛІ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ ОПОВІДНОГО КОНТЕНТУ

2.1. Аналіз підходів до процедурної генерації сюжетного контенту

Процедурна генерація сюжетного або оповідного контенту є окремим напрямом у галузі процедурної генерації, що спрямована на створення динамічних сюжетів, завдань або діалогів з урахуванням контексту гри та взаємодії користувача. Основна мета таких систем – створення різноманітного, логічно пов’язаного ігрового досвіду, що адаптується до дій користувача.

Існує кілька основних підходів до реалізації процедурної генерації сюжетного контенту, з яких виділяють:

- **Шаблонний (template-based)** – найпростіший підхід, що базується на заздалегідь підготовлених структурах із заповненням змінних частин (наприклад, «[Персонаж] просить допомогти з [проблема] у [місце]»). Перевага полягає у простоті реалізації та передбачуваності результату, однак суттєвим недоліком є обмежена варіативність та швидке повторення патернів.
- **Правиловий (rule-based)** – підхід що передбачає використання набору логічних умов або дерев прийняття рішень для формування сюжету. Цей підхід забезпечує більшу адаптивність, дозволяючи враховувати контекст гри, поведінку користувача та попередні події. Недоліком є складність масштабування та підтримки великої кількості правил.
- **Графовий (graph-based/story graphs)** – підхід де сюжет представлений у вигляді графа з вузлами (події, ситуації) та зв’язками (умовами переходу). Користувач подорожує цим графом залежно від власних дій, а система може генерувати нові вузли або комбінувати існуючі. Перевага – гнучкість у побудові складних структур оповіді; недолік – складність у підтриманні цілісності сюжету.

- **Планувальний (planning-based)** – підхід що застосовується у випадках, коли система має досягти певної кінцевої цілі, генеруючи послідовність дій (сюжетних подій), які до неї ведуть. Подібні підходи використовуються у ШІ-агентах, що мають досягти цілі через логічну послідовність рішень. Це дає змогу досягати сюжетної логіки, однак створює високі обчислювальні навантаження.
- **Стохастичний (probabilistic)** – підхід побудований на ймовірнісному виборі подій, що дозволяє досягти високої варіативності. Зазвичай використовується у поєднанні з іншими підходами, як-от системою ваг або модифікаторів, які керують імовірністю вибору певного елементу. Цей метод добре підходить для створення унікальних вражень, але потребує додаткових механізмів для забезпечення логіки.
- **Гібридні моделі** – сучасні системи часто поєднують декілька підходів для досягнення балансу між варіативністю, зв’язністю та продуктивністю. Наприклад, графова структура із стохастичним вибором вузлів, доповнена моральною системою та системою cooldown.

Успішна реалізація процедурної генерації сюжетного контенту потребує врахування як технічних аспектів (продуктивність, тестованість), так і оповідьних – зв’язність історії, емоційна логіка, інтеграція з ігровим світом та механіками. Саме тому сучасні дослідження все частіше орієнтуються на модульні підходи та адаптивні системи, які здатні підлаштовуватись до унікального досвіду кожного користувача. Модульна структура системи генерації дозволяє гнучко поєднувати шаблонні, стохастичні та графові підходи залежно від контексту та потреб.

2.2. Структура квесту як об’єкта генерації

У контексті процедурної генерації сюжетного контенту квест виступає основною структурною одиницею, яка поєднує в собі елементи оповідання, геймплейні завдання та мотивацію користувача. Щоб забезпечити ефективну генерацію, необхідно чітко визначити, з яких компонентів складається типовий

квест, як ці компоненти можуть змінюватися, та які зв’язки між ними мають бути збережені.

Основними складовими квесту є:

- **Ціль (Goal):** основна задача, яку має виконати користувач (напр., знищити ворога, знайти предмет, доставити повідомлення).
- **Ініціатор (Quest Giver):** персонаж або подія, що запускає квест. Цей елемент може бути варіативним і залежати від локації, фракції, чи репутації користувача.
- **Контекст (Context):** короткий опис або обґрунтування, чому квест виникає у світі гри (наприклад, напад монстрів на поселення, або зникнення NPC).
- **Етапи (Stages):** квест може складатись з одного або кількох кроків, що логічно пов’язані між собою (наприклад: отримати завдання > знайти локацію > перемогти ворога > повернутись за нагородою).
- **Місце дії (Location):** точка або регіон на карті, де відбувається виконання квесту. Це може бути поселення, печера, шлях або інша генеративна зона.
- **Умови завершення (Completion Conditions):** критерії, за якими квест вважається виконаним.
- **Нагорода (Reward):** матеріальний або сюжетний результат, що мотивує користувача до виконання.

Окрім основних складових, до структури квесту можуть входити додаткові елементи:

- **Часові обмеження або тригери (Timers/Triggers):** створюють напругу або контролюють появу квестів.
- **Розгалуження (Branches):** наявність альтернативних шляхів проходження, що залежать від дій користувача.
- **Вплив на світ гри (Consequences):** квести можуть змінювати стан ігрового світу (наприклад, зруйноване поселення після провалу місії).

Формалізація квесту як об’єкта генерації дає змогу представити його у вигляді шаблону або структури даних, яка підтримує параметризацію. Чітке визначення структури квесту дозволяє системі процедурної генерації варіативно комбінувати

окремі елементи, змінювати їхню кількість, складність або порядок, зберігаючи при цьому логічну зв’язність та ігрову цінність.

2.3. Категоризація та параметризація квестів

Щоб забезпечити ефективну генерацію різноманітних, логічно пов’язаних та ігрово цінних квестів, важливо не лише визначити їхню структуру, але й здійснити категоризацію та параметризацію. Це дозволяє системі генерації оперувати не окремими випадками, а узагальненими шаблонами, які можна адаптувати до конкретного контексту гри.

Квести можуть бути класифіковані за кількома критеріями:

- **За типом завдання:**
  - Знищення цілі (Elimination)
  - Пошук об’єкта (Fetch)
  - Супровід (Escort)
  - Розслідування (Investigation)
  - Оборона території (Defense)
  - Дипломатична взаємодія (Diplomacy)
  - Комбіновані (наприклад, знищення + пошук)
- **За джерелом ініціації:**
  - Пряме звернення NPC
  - Випадкова подія у світі
  - Вплив користувача (наслідок попередніх дій)
  - Місцева тригерна зона
- **За важливістю:**
  - Основні (Main story quests)
  - Побічні (Side quests)
  - Динамічні або ситуативні (Emergent quests)

Параметри квесту визначають варіативність і адаптивність кожного завдання. Кожен квест може мати такі змінні:

- **Складність (Difficulty):** кількість етапів, сила супротивників, віддаленість локацій.
- **Тривалість (Duration):** орієнтовний час на виконання; може впливати на час появи або зникнення квесту.
- **Ургентність (Urgency):** впливає на таймери або швидкість змін у світі при невиконанні.
- **Нагорода (Reward Type & Value):** визначає мотивацію користувача – матеріальну, сюжетну, репутаційну.
- **Місце дії (Location Type):** місто, ліс, печера, відкритий світ – впливає на склад квесту та можливі події.
- **Дійові особи (Characters):** NPC, вороги, фракції, до яких належить завдання. Це дозволяє персоналізувати сюжет.

Система процедурної генерації поєднує категорії та параметри для створення узгодженого шаблону завдання. Наприклад:

Тип: Захист поселення

Локація: Мала ферма поблизу міста

Складність: Середня

Ініціатор: NPC-фермер

Нагорода: Їжа + репутація в регіоні

Часовий ліміт: 1 ігрова доба

Це дозволяє створити велику кількість унікальних завдань на основі невеликої кількості загальних категорій та параметрів. Якщо підсумувати, то параметризація надає такі переваги:

- Зменшення кількості жорстко закодованих сценаріїв.
- Гнучкість при розширенні системи.
- Можливість адаптації до рівня користувача або ігрової ситуації.
- Підтримка зв’язності між квестами за допомогою спільних змінних (фракція, регіон, попередні дії).

Таким чином, категоризація та параметризація квестів є основою для динамічного, модульного підходу до процедурної генерації, що дозволяє системі

адаптувати завдання до конкретного контексту гри, зберігаючи при цьому оповідьну логіку та ігрову глибину.

2.4. Алгоритмічні підходи до генерації сюжетів

У системах процедурної генерації сюжетного контенту важливим завданням є не лише створення окремих квестів, а й забезпечення їхньої різноманітності, унікальності та логічної послідовності протягом всієї гри. Для цього використовуються спеціалізовані алгоритмічні підходи, що дозволяють автоматично керувати повторюваністю елементів, впливом користувача на оповідь та адаптацією до контексту.

Одними з ключових технічних механізмів, що застосовуються в таких системах, є:

- система ваг параметрів – для керування ймовірністю появи певних елементів у нових квестах;
- система cooldown – для зниження шансів повторного використання щойно згенерованих сюжетних тем;
- модифікатори ваг – для динамічної адаптації сценаріїв на основі історії попередніх дій користувача.

У поєднанні ці механізми дозволяють створювати більш живу, реактивну та варіативну оповідну структуру.

2.4.1. Система cooldown та ваг параметрів

Механізм ваг полягає у присвоєнні кожному можливому елементу (наприклад, типу квесту, мотиву, місця дії або типу персонажа) певного числового значення, що визначає його базову ймовірність потрапити до наступного згенерованого квесту. Вага може бути змінною, і змінюється залежно від загального контексту, статистики вибору, а також взаємодії з іншими елементами.

Система cooldown служить для тимчасового зменшення ваги елемента після його використання, щоб уникнути надмірної повторюваності. Наприклад, якщо

квест, пов’язаний із захистом поселення, був використаний у попередній генерації, то вага подібного мотиву знижується до певного мінімуму (наприклад, з 100 до 5) і лише поступово повертається до свого початкового значення з часом або після певної кількості інших квестів.

Типова логіка cooldown реалізується у вигляді:

- миттєвого зниження ваги після використання;
- експоненційного або лінійного відновлення ваги з кожною новою ітерацією;
- можливості відслідковування декількох останніх використаних елементів у «черзі відновлення».

Такий підхід дозволяє підтримувати баланс між варіативністю та повторюваністю, не забороняючи жодного сценарію повністю, але надаючи перевагу тим, які довше не використовувалися.

2.4.2. Модифікатори ваг на основі минулих подій

Окрім базових ваг та cooldown, для додаткової гнучкості можуть застосовуватись модифікатори, які змінюють вагу елементів на основі історії попередніх дій користувача або загального оповідного стану гри.
Типові модифікатори включають:

- контекстні модифікатори: знижують вагу тем, що вже були нещодавно використані, навіть якщо сам елемент не повторюється (наприклад, якщо був квест про хворобу в селі – наступний квест із темою хвороб отримає понижену вагу);
- категорійні модифікатори: впливають на всі елементи певної категорії (наприклад, зниження ймовірності квестів, пов’язаних з монстрами, якщо вони домінували останнім часом);
- адаптивні модифікатори: враховують уподобання користувача (наприклад, якщо користувач часто ігнорує дипломатичні місії – їхня вага поступово знижується);
- послідовні модифікатори: підсилюють або послаблюють певні теми в залежності від нещодавньої послідовності подій (наприклад, тема зради може

мати більшу ймовірність, якщо в попередніх квестах були натяки на нестабільність фракцій).

Модифікатори можуть мати різну силу (наприклад, коефіцієнти 0.9, 0.6 або 0.3) та обчислюватись на основі фіксованої кількості останніх згенерованих квестів. Така система дозволяє досягти балансу між логічною послідовністю, варіативністю та унікальністю кожної ігрової сесії.

2.5. Врахування поведінки користувача при генерації

Один із ключових принципів сучасної процедурної генерації сюжетного контенту полягає в адаптації до дій користувача. Система повинна не лише генерувати випадкові чи логічно пов’язані квести, а й реагувати на стиль гри, вибір у минулих завданнях, частоту взаємодії з певними персонажами чи локаціями. Такий підхід підвищує занурення у гру, створює відчуття персоналізованого наративу та дозволяє формувати нелінійний, унікальний досвід для кожного користувача.

Аналіз поведінки користувача може включати:

- вибірковість (які типи квестів користувач приймає/ігнорує),
- ефективність (як швидко виконує завдання, які часто провалює),
- стиль гри (агресивний, дипломатичний, дослідницький тощо),
- повторювані шаблони дій (наприклад, завжди допомагає слабким або завжди зраджує союзників).

Залежно від цього, система може змінювати:

- ймовірності появи квестів певного типу (наприклад, більше бойових місій для агресивного користувача),
- реакцію NPC (персонажі можуть боятись, довіряти або ставитись з підозрою),
- лінії діалогів і можливі вибори (відкриваються або блокуються опції залежно від минулих рішень),
- сюжетні події або наслідки (наприклад, місто, яке користувач часто ігнорував, потрапляє в біду).

Для реалізації цього принципу в системі може зберігатись набір внутрішніх статистик, які автоматично оновлюються після кожної взаємодії та впливають на подальшу генерацію. Такі статистики не завжди відображаються гравцю напряму, але мають вирішальний вплив на побудову оповідання.

Таким чином, врахування поведінки користувача дозволяє:

- створити реактивну систему квестів, де сюжет «відповідає» на дії користувача;
- покращити занурення та емоційну залученість;
- підтримувати варіативність без потреби жорсткого сценарного контролю.

2.5.1. Внутрішня статистика та система моральності

Для забезпечення адаптивної генерації сюжетного контенту необхідно зберігати і аналізувати певні показники, які відображають дії користувача у грі. Ці показники формують внутрішню статистику, що використовується системою як база для прийняття рішень при генерації подальших квестів, діалогів або подій.

Внутрішня статистика – це набір змінних, які автоматично оновлюються залежно від поведінки користувача. Вони не завжди є видимими для самого користувача, але активно впливають на оповідну логіку. Прикладами таких статистик є:

- Частота прийняття/відмови від квестів певного типу (наприклад, бойових чи дипломатичних);
- Кількість завершених або в провалених місій;
- Взаємодія з фракціями (довіра, ворожість, нейтралітет);
- Стиль виконання завдань (використання сили, хитрості, обходу конфліктів);
- Рівень впливу на світ гри (наприклад, скільки поселень було врятовано або зруйновано).

На основі цієї статистики може формуватися система моральності – прихована шкала, яка оцінює дії користувача з точки зору етичних чи світоглядних категорій. Така система не обов’язково повинна бути бінарною («добро – зло»), її можна реалізувати як багатовимірну модель з різними осями, наприклад:

- Альтруїзм – Егоїзм
- Порядок – Хаос
- Жорстокість – Милосердя
- Лояльність – Зрада

Ці моральні статистики змінюються відповідно до рішень користувача, і надалі впливають на:

- генерацію квестів з моральною дилемою;
- поведінку NPC, які можуть підтримувати або засуджувати дії користувача;
- доступність окремих сюжетних гілок, що відкриваються лише при певному рівні моральної схильності;
- світогляд персонажа, який відображається в діалогах і репутації у світі гри.

Такий підхід дозволяє не лише зберігати оповідну логіку, а й стимулює користувача задумуватись над наслідками своїх рішень. Крім того, це створює довготривалу зв’язність у сюжеті: навіть ранні вибори можуть мати вплив на подальший перебіг гри, підвищуючи її глибину та емоційну значущість.

2.5.2. Адаптація ймовірностей на основі дій користувача

Однією з ключових особливостей процедурної генерації сюжетного контенту є реактивність – здатність системи підлаштовуватись під стиль гри конкретного користувача. Для досягнення цього використовується механізм адаптації ймовірностей, за якого шанси появи певних типів контенту змінюються на основі попередніх дій користувача.

Основна ідея полягає у тому, що кожен сюжетний шаблон, квест або мотив має ваговий коефіцієнт, який визначає ймовірність його обрання системою при генерації. Ці ваги динамічно змінюються відповідно до поведінки користувача:

- Якщо користувач часто приймає бойові завдання, вага таких квестів поступово збільшується, підвищуючи ймовірність їх подальшої генерації.
- Якщо користувач ігнорує або відмовляється від певного типу контенту (наприклад, супровід NPC або дослідження локацій), вага подібних завдань знижується.

- У разі провалу місій, пов’язаних з певною темою, вага відповідних шаблонів тимчасово зменшується, щоб не перевантажувати користувача.
- Успішне завершення рідкісного або складного завдання може підвищити шанси на появу сюжетів того ж типу в майбутньому.

Цей підхід дозволяє створити відчуття "логічного продовження" ігрового досвіду, у якому дії користувача прямо впливають на зміст наступних подій. Крім того, адаптація ваг дозволяє досягти кращого балансу між повторюваністю та новизною: система не буде надмірно генерувати одні й ті самі квести, навіть якщо вони є улюбленими, завдяки внутрішнім обмеженням (наприклад, cooldown або множникам зв’язності).

З технічної точки зору, адаптація може реалізовуватись через:

- Прості модифікатори ваг (збільшення/зменшення ваги на фіксовану величину);
- Експоненційні функції згасання (наприклад, зменшення впливу старіших дій);
- Використання пам’яті останніх N квестів, що задає контекст короткострокових уподобань;
- Систему рекомендацій, яка комбінує минулу активність з внутрішніми обмеженнями (наприклад, темами, які нещодавно не використовувались).

Таким чином, механізм адаптації ймовірностей виступає важливим інструментом у забезпеченні персоналізованого, різноманітного та логічно зв’язаного ігрового досвіду, орієнтованого на унікальні вибори користувача.

2.6. Узагальнена модель роботи системи

На основі попередньо описаних компонентів, у цьому підпункті представлено узагальнену модель роботи системи процедурної генерації квестів, яка поєднує алгоритмічні підходи, параметризацію контенту та адаптацію до дій користувача. Ця модель демонструє, як відбувається процес генерації сюжетного контенту у динамічному, контекстно-залежному середовищі.

1) Ініціація запиту на генерацію

Генерація нового квесту відбувається у відповідь на:

- подію у світі гри (наприклад, гравець прибув до поселення, знищив монстра, завершив попереднє завдання);
- внутрішні тригери (досягнення певного рівня, статусу, репутації тощо);
- часові умови (наприклад, оновлення квестів щодня або після ігрового тижня).

На цьому етапі система фіксує:

- місце розташування гравця;
- стан світу;
- внутрішню статистику гравця (поведінкові параметри, моральність, стиль гри);
- активні події та незавершені квести.

2) Відбір релевантних шаблонів

Із бази шаблонів квестів (структурованих шаблонів із параметрами) система фільтрує лише ті, що:

- відповідають поточному регіону;
- узгоджуються з темпом гри та складністю;
- не мають активного cooldown (нещодавно використані шаблони тимчасово відсікаються);
- не суперечать поточному стану ігрового світу чи вже прийнятим рішенням гравця.

3) Розрахунок ваг і ймовірностей

До кожного з потенційних шаблонів застосовуються числові ваги на основі кількох факторів:

- поведінкова модель гравця (наприклад, переважання агресивних чи дипломатичних рішень);
- моральна система (впливає на типи сюжетів, які можуть бути запропоновані);
- зв’язки з попередніми квестами (категоріальні модифікатори, уникнення повторень);
- ефект cooldown (зменшення ваг нещодавно використаних тем або мотивів);
- контекст середовища (фракції, NPC, події в регіоні).

Результатом є список шаблонів із адаптивними ймовірностями, що відображають ігрову динаміку.

4) Стохастичний вибір і генерація квесту

Використовуючи стохастичний вибір на основі розрахованих ваг, система обирає найбільш релевантний шаблон. Далі:

- підставляються конкретні параметри: місце, персонажі, тип ворога, предмети тощо;
- у випадку складних квестів – формується послідовність етапів, кожен з яких пов’язаний із подіями, діями чи умовами;
- створюються діалоги, внутрішні описи та текстові варіанти на основі шаблонів і локального контексту.

5) Презентація завдання гравцю

Система визначає спосіб подачі: через NPC, внутрішній монолог, випадкову подію або об’єкт у світі. Якщо шаблон передбачає варіанти вибору – одразу визначаються умови, за яких ті чи інші гілки можуть стати доступними (наприклад, високий рівень моралі відкриває «мирну» розв’язку).

6) Відстеження результатів і оновлення даних

Після завершення, провалу чи відмови:

- оновлюється внутрішня статистика гравця;
- система зберігає наслідки квесту (зміни в стані світу, фракціях, NPC);
- коригуються ваги та ймовірності шаблонів;
- активується cooldown на використання схожих тем, щоб уникнути повторень;
- враховуються моральні наслідки, що можуть вплинути на наступні події.

Ця модель забезпечує цикл самонавчання системи, в якій кожна дія гравця впливає на подальший контент. У результаті, ігровий досвід стає унікальним і персоналізованим, а процедура генерації – не лише випадковою, а контекстно-обґрунтованою та інтегрованою у загальну логіку гри.

3. РОЗРОБКА СИСТЕМИ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ ОПОВІДНОГО КОНТЕНТУ

3.1. Загальна архітектура системи

Система процедурної генерації сюжетного контенту розроблена з використанням мови програмування Java , яка є зручною для об’єктно-орієнтованого проєктування та модульної побудови програмних рішень. Основною метою розробки є створення гнучкого механізму для автоматичної генерації ігрових завдань (квестів), які відрізняються за типами, складністю, описами, умовами виконання та нагородами.

Архітектура системи побудована у вигляді набора взаємодіючих модулів, кожен з яких відповідає за окремий аспект створення квесту:

- Класи (QuestTemplate, ObjectiveModifier, RewardModifier) що включають основні структури шаблонів та методи для отримання та вказання значень у відповідних структурах.
- Генератор шаблонів (QuestTemplateGenerator) – створює об'єкти квестів з урахуванням випадкових параметрів, зокрема типу завдання, складності, винагород тощо.
- Клас управління квестами (QuestManager) – містить активні квести, реалізує додавання, перегляд, оновлення та автоматичне видалення “застарілих” завдань.
- Система шаблонів (DataQ, ConfigQ, ConfigQLoader, DataI, ConfigI, ConfigILoader) – забезпечує завантаження текстових шаблонів та іменованих змінних із зовнішніх JSON-файлів.
- Генератор змінних (ObjectiveGenerator) – відповідає за заповнення сюжетних шаблонів змінними (наприклад, імена NPC, локації, предмети).
- Генератор нагород (RewardGenerator) – створює набір винагород, що відповідає складності завдання.

Усі модулі взаємодіють між собою через чітко визначені інтерфейси. Кожен згенерований квест створюється незалежно й може бути унікальним завдяки поєднанню шаблонного підходу та випадкових змінних.

Користувач взаємодіє із системою через консольне меню (реалізоване у класі Main), де може:

- згенерувати новий квест,
- переглянути список поточних завдань,
- спостерігати за їхнім старінням та автоматичним зникненням.

Розроблена система має модульну структуру, що дозволяє з легкістю додавати нові типи квестів, змінні, шаблони або логіку їх формування без порушення цілісності вже реалізованого функціоналу.

3.2. Механізм генерації завдань

Процес створення квесту у системі процедурної генерації сюжетного контенту базується на поєднанні заздалегідь підготовлених шаблонів, наборів змінних та елементів випадковості. Такий підхід дозволяє забезпечити різноманітність завдань без потреби вручну створювати кожен із них.

Генерація одного квесту відбувається за таким алгоритмом:

1. Вибір типу квесту
    - Випадково обирається один із доступних типів квестів, таких як квест на супровід, доставку, захист тощо. Тип квесту визначає подальшу логіку – зокрема, які змінні будуть використані, які шаблони застосовуються, і які параметри квесту мають найбільше значення.
2. Генерація сюжетних змінних
    - Відповідно до обраного типу квесту, система звертається до зовнішніх JSON-файлів, що містять шаблонні змінні (наприклад, імена персонажів, назви локацій, типи предметів). Генератор змінних (ObjectiveGenerator) обирає відповідні значення та готує їх для вставки у текстовий шаблон.
3. Формування назви та опису квесту

- Із заздалегідь підготовлених шаблонів, що містять заповнювачі (наприклад: "Супроводити {target} до {location}"), система підставляє обрані змінні, формуючи унікальну назву та опис для кожного завдання.
- Створюється окремий «дублюючий масив» де зберігається кількість використання кожного типу кожного елементу, що напряму впливає на шанс обирання елементів із меншими результатами випадіння.

4. Розрахунок складності та мінімального рівня
    - Залежно від типу завдання, система випадково генерує рівень складності у вигляді числового значення. Це дозволяє масштабувати квести відповідно до прогресу гравця.
5. Генерація винагород
    - На основі складності квесту за допомогою модуля RewardGenerator визначається набір винагород: ігрова валюта, досвід, цінні предмети тощо. Винагороди генеруються в випадковому порядку та зі зверненням до зовнішнього JSON-файлу задля підбору нагороди із списку можливих нагород.
6. Генерація унікального ідентифікатора квесту
    - Кожному новому квесту присвоюється унікальний ID, який має вигляд Q#### (наприклад, Q3046). У разі, якщо такий ID уже існує серед активних квестів, система повторно генерує новий до досягнення унікальності.
7. Формування повного об’єкта квесту
    - Всі згенеровані елементи збираються у єдиний об’єкт типу QuestTemplate, який включає:
        1. ID,
        2. Назву,
        3. Опис,
        4. Тип квесту,
        5. Складність,

6. Мінімальний рівень (що у плані цієї роботи виконано у вигляді генерації випадкового числа),
7. Дані з ціллю та локацією цілі,
8. Нагороди,
9. Час до старіння.

Такий об’єкт додається до списку активних квестів у менеджері (QuestManager) і зберігається доти, доки не буде виконаний, відхилений або не втратить актуальність із часом. У рамках цієї роботи реалізовано лише можливість користувача відхилити квест, чи дати йому застаріти.

3.3. Управління активними квестами

Управління активними квестами реалізовано за допомогою спеціального модуля QuestManager, який відповідає за зберігання, оновлення та видалення згенерованих завдань. Цей компонент забезпечує централізовану роботу з усіма актуальними квестами та підтримує стабільність системи, унеможливлюючи дублювання ідентифікаторів та накопичення «застарілих» записів.

Основні функції QuestManager включають:

1. Додавання нового квесту до активного списку
   - Після створення нового квесту за допомогою QuestTemplateGenerator, він додається до внутрішнього контейнера activeQuests, реалізованого у вигляді Map<String, QuestTemplate>, де ключем виступає унікальний ідентифікатор.
   - Перед додаванням проводиться перевірка унікальності ID. У випадку колізії запускається повторна генерація, доки не буде отримано унікальний результат.
2. Перегляд поточних завдань
   - Користувач може у будь-який момент переглянути список усіх активних квестів через консольне меню. Система виводить повну інформацію про кожен квест: ID, назву, опис, тип, складність, час до старіння, винагороди тощо.

- Для зручності сприйняття кожен квест виводиться у форматованому вигляді.

3. Старіння та автоматичне видалення завдань
   - Кожен квест має параметр «час до старіння» (Alive Time), який зменшується з кожним оновленням системи.
   - При досягненні нульового значення відповідне завдання вважається неактуальним і автоматично видаляється зі списку activeQuests.
   - Цей механізм імітує втрату актуальності квестів, роблячи систему більш динамічною.
4. Відхилення завдання користувачем
   - Користувач має змогу вручну відхилити будь-яке активне завдання через відповідний пункт меню.
   - У такому випадку квест також видаляється зі списку активних, і не чинить подальшого впливу на ігровий процес.
5. Унікальність ідентифікаторів
   - З метою уникнення дублювання ідентифікаторів для нових квестів, система реалізує механізм перевірки зайнятості ID перед додаванням.
   - Якщо ID уже існує, система генерує новий до тих пір, поки не буде отриманий унікальний. Це забезпечує стабільність доступу до квестів та уникнення конфліктів при зверненні до них.

Таким чином, система управління квестами виконує роль «ядра», що координує життєвий цикл завдань – від моменту генерації до зникнення. Вона дозволяє зберігати порядок, уникати повторів та створює враження живого ігрового світу, що постійно оновлюється.

3.4. Робота з шаблонами та конфігураційними файлами

Одним з ключових принципів побудови системи процедурної генерації сюжетного контенту є відокремлення структури квестів від їхнього текстового

наповнення. Для цього використовуються зовнішні конфігураційні файли у форматі JSON, які містять:

- шаблони описів квестів,
- списки змінних (імена, локації, предмети тощо),
- шаблони нагород,
- параметри, що визначають логіку генерації.

Цей підхід дозволяє легко оновлювати або розширювати наративну складову без необхідності вносити зміни до коду програми.

Основні компоненти роботи з файлами:

1. Шаблони текстів квестів
    - Зберігаються у файлах типу ConfigQ.json, які містять набір типових фраз із плейсхолдерами (наприклад: {target}, {location}).
    - При генерації квесту система підставляє реальні значення у відповідні поля.
    - Шаблони мають умовну класифікацію за типами квестів, що дозволяє кожному типу мати свої унікальні формулювання.
2. Шаблони змінних
    - У файлах ConfigI.json та ConfigQ.json зберігаються списки потенційних змінних (типи предметів, імена NPC, назви локацій).
    - Кожна змінна має свій тип і контекст використання.
    - Випадковий вибір змінної відбувається при кожній генерації, що дозволяє створювати варіативні завдання.
3. Шаблони нагород
    - У файлах конфігурації зберігаються списки можливих винагород, які використовуються RewardGenerator-ом.
    - Нагороди обираються відповідно до рівня складності квесту та доступності для поточного рівня гравця (у поточній реалізації – випадково, без адаптації під гравця).
4. Завантаження конфігурацій

- Завантаження здійснюється спеціальними класами ConfigQLoader, ConfigILoader, які парсять JSON-файли й перетворюють їх у внутрішні структури даних (DataQ, DataI).
- Для зручності роботи ці класи реалізовані з кешуванням, що дозволяє уникнути зайвого доступу до файлової системи.

Переваги шаблонного підходу:

- Гнучкість – нові шаблони можна додавати без змін у коді.
- Масштабованість – просте розширення бази даних сюжетних елементів.
- Локалізація – підтримка кількох мов при зміні лише текстових файлів.
- Повторне використання – одні й ті самі шаблони можна використовувати у різних контекстах з іншими змінними.

Таким чином, розділення коду логіки та текстового наповнення дозволяє створити систему, придатну для повторного використання, розширення та адаптації під нові вимоги без ризику порушити роботу основного функціоналу.

3.5. Інтерфейс користувача та взаємодія з системою

Для забезпечення зручної взаємодії користувача з системою процедурної генерації сюжетного контенту було реалізовано консольний інтерфейс, який виконує роль простого засобу керування функціоналом системи.

Основні функції інтерфейсу включають:

- Генерація нового квесту – користувач може ініціювати створення нового квесту за допомогою відповідної команди меню.
- Перегляд активних квестів – відображення списку поточних завдань з їхніми ключовими параметрами, такими як назва, опис, час до застаріння.
- Оновлення стану системи – автоматичне зменшення часу життя квестів, видалення застарілих завдань.
- Відхилення квестів – можливість вручну видалити небажані завдання.

Консольний інтерфейс реалізований у класі Main, де користувач вводить відповідні команди, що викликають методи менеджера квестів (QuestManager).

Такий підхід дозволяє зосередитися на тестуванні ключових механізмів генерації та управління квестами без необхідності розробки складного графічного інтерфейсу.

Незважаючи на мінімалістичний вигляд, інтерфейс надає повний доступ до основних функцій системи та слугує зручним інструментом для демонстрації працездатності розробленої архітектури.

3.6. Тестування роботи системи

Для забезпечення коректної роботи системи процедурної генерації сюжетного контенту було проведено комплексне тестування, що включало як автоматичні, так і ручні перевірки.

Тестування охоплювало такі аспекти:

- Перевірка генерації квестів різних типів. Система успішно створювала квести із різними параметрами – типами завдань, складністю, змінними та нагородами. Кожен згенерований квест відповідав заданим обмеженням та умовам.
- Валідація унікальності ідентифікаторів квестів. Перевірено, що для кожного нового квесту генерується унікальний ID, що запобігає дублюванню та конфліктам у списку активних завдань.
- Тестування механізму старіння квестів. Було підтверджено, що квести коректно зберігають інформацію про залишковий час життя та автоматично видаляються з активного списку після завершення цього часу.
- Перевірка взаємодії з користувачем через консольний інтерфейс. Тестування показало, що базові функції системи – генерація, перегляд, відхилення завдань – працюють стабільно, забезпечуючи користувачу необхідний контроль над активними квестами.

Результати тестування підтверджують працездатність системи та її відповідність вимогам, визначеним у технічному завданні. Виявлені дрібні недоліки були усунуті під час розробки.

3.7. Можливість подальшого розвитку та покращення

Розроблена система процедурної генерації сюжетного контенту має модульну структуру, що дозволяє її подальше розширення та удосконалення. Основні напрямки розвитку включають:

- Розширення типів квестів. Можливе додавання нових різновидів завдань із унікальними умовами, логікою та нагородами, що підвищить різноманітність ігрового процесу.
- Ускладнення механізмів генерації. Впровадження більш складних алгоритмів вибору параметрів, залежних від прогресу гравця, його рішень та минулих квестів, що зробить сюжет більш адаптивним та цікавим.
- Інтеграція графічного інтерфейсу користувача. Реалізація візуального інтерфейсу замість або разом із консольним для підвищення зручності взаємодії.
- Впровадження системи взаємозв’язків між квестами. Залежності між завданнями, які впливають одне на одного, створять більш складний і багатошаровий сюжет.
- Адаптація під різні платформи. Розробка версій системи для мобільних пристроїв, чи версії для веб додатків.

Запропоновані напрямки розвитку є основою для подальших наукових досліджень та практичних впроваджень у сфері процедурної генерації сюжетного контенту.

4. ПРАКТИЧНА РЕАЛІЗАЦІЯ СИСТЕМИ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ СЮЖЕТНОГО КОНТЕНТУ

4.1. Ключові класи та їх реалізація

4.1.1. Клас QuestTemplate

Клас QuestTemplate є базовою структурою даних для представлення шаблону квесту. Він об'єднує всі ключові параметри, що описують конкретний квест: унікальний ідентифікатор, назву, опис, тип, складність, мінімальний рівень гравця, час до автоматичного зникнення, список цілей (змінних) та список нагород.

Цей клас слугує контейнером, який наповнюється даними під час генерації квесту і використовується в подальшій логіці проєкту для збереження, відображення чи виконання завдань.

Код класу:

```
import java.util.List;

public class QuestTemplate {
    private final String id;
    private final String name;
    private final String description;
    private final String type;
    private final double difficulty;
    private final int minLevel;
    private double aliveTime;
    private final List<ObjectiveModifier> variables;
    private final List<RewardModifier> rewards;

    public QuestTemplate(String id, String name, String description, String type,
                         double difficulty, int minLevel, double aliveTime,
                         List<ObjectiveModifier> variables, List<RewardModifier>
rewards) {
        this.id = id;
        this.name = name;
        this.description = description;
        this.type = type;
        this.difficulty = difficulty;
        this.minLevel = minLevel;
        this.aliveTime = aliveTime;
        this.variables = variables;
        this.rewards = rewards;
    }

    …

    @Override
    public String toString() {
        return "\nQuestTemplate {" +
                "\n  Id: '" + id + '\'' +
                ",\n  Name: '" + name + '\'' +
```

```
                ",\n  Description: '" + description + '\'' +
                ",\n  Type: '" + type + '\'' +
                ",\n  Difficulty:  " + difficulty +
                ",\n  MinLevel: " + minLevel  +
                ",\n  Time until decay: " + aliveTime +
                ",\n  Variables: " + variables +
                ",\n  Rewards: " + rewards +
                "\n}";
    }
}
```

4.1.2. Генератор квестів QuestTemplateGenerator

Клас QuestTemplateGenerator відповідає за створення повноцінного об'єкта квесту із використанням процедурної генерації. Під час створення квесту випадково обирається його тип, визначаються базові параметри, такі як складність, мінімальний рівень, час життя, ім’я та опис. Також генеруються змінні/цілі та відповідні нагороди.

Основні етапи генерації:

1. Випадковий вибір типу квесту з enum QuestType;
2. Генерація мети (тип задачі, ціль, локація) через ObjectiveGenerator;
3. Формування назви та опису квесту на основі шаблонів;
4. Розрахунок складності та тривалості квесту;
5. Генерація нагород відповідно до складності.

Код класу:

```
import java.util.List;
import java.util.Random;

public class QuestTemplateGenerator {
    private static final Random random = new Random();

    public static QuestTemplate generate(String id) {
        QuestType questType =
QuestType.values()[random.nextInt(QuestType.values().length)];
        ObjectiveModifier objective =
ObjectiveGenerator.generateObjective(questType);
        String name = ObjectiveGenerator.generateQuestName(objective);
        String description = ObjectiveGenerator.generateQuestDescription(objective);
        double difficulty = generateDifficulty(questType);
        int minLevel = generateminLevel();
        int aliveTime = (int) Math.round(generateAliveTime(questType));


        List<ObjectiveModifier> vars = ObjectiveGenerator.generateOV(questType);
        List<RewardModifier> rewards = RewardGenerator.generateRewards(difficulty);

        return new QuestTemplate(
```

```
                id,
                name,
                description,
                questType.name(),
                difficulty,
                minLevel,
                aliveTime,
                vars,
                rewards
        );
    }

    private static double generateDifficulty(QuestType type) {
        return switch (type) {
            case MURDER -> 3.5 + random.nextDouble(13.25) * 2;
            case DELIVERY -> 1.5 + random.nextDouble(19) * 1.5;
            case GATHERING -> 3.0 + random.nextDouble(18) * 1.5;
            case ESCORT -> 3.0 + random.nextDouble(13.5) * 2;
            case DEFENSE -> 4.0 + random.nextDouble(13) * 2;
        };
    }

    private static int generateminLevel() {
        return random.nextInt(5) + 1;
    }

    private static double generateAliveTime(QuestType type) {
        return switch (type) {
            case MURDER -> 45 + random.nextDouble() * 30;
            case DELIVERY -> 20 + random.nextDouble() * 20;
            case GATHERING -> 30 + random.nextDouble() * 30;
            case ESCORT -> 40 + random.nextDouble() * 30;
            case DEFENSE -> 50 + random.nextDouble() * 40;
        };
    }
}
```

4.1.3. Клас ObjectiveModifier

Клас ObjectiveModifier відповідає за структуру змінних, які наповнюють основну мету квесту (ціль, локація тощо). Він реалізує механізм автоматичного заповнення даних у разі їх відсутності та закладає основу для системи ваг, що дозволяє урізноманітнити генерацію та зменшити повторюваність контенту.

Окрім цього, в класі реалізована логіка врахування спеціальних умов, наприклад, для квестів типу DEFEND ціль і локація ототожнюються.

Основні функції:

- Зберігання типу задачі (objectiveType), цілі (target) та локації (location);
- Автоматичне заповнення змінних із використанням вагових коефіцієнтів;

- Допоміжна функція для вибору значення з урахуванням ваг (weightedRandomIndex).

Код класу:

```
import java.util.List;
import java.util.Random;

public class ObjectiveModifier {
    private final ObjectiveType objectiveType;
    private String target;
    private String location;

    public ObjectiveModifier(ObjectiveType objectiveType, String target, String
location, int quantity) {
        this.objectiveType = objectiveType;
        this.target = null;
        this.location = null;
        // quantity зарезервовано для майбутнього розширення функціоналу
    }

    public void ensureFilled() {
        if (target == null || location == null) {
            DataQ data = ConfigQ.get(objectiveType.name());
            if (data != null) {
                if (target == null && data.targets != null &&
!data.targets.isEmpty()) {
                    List<Integer> weights =
ObjectiveGenerator.usageToWeights(data.targetsUsage);
                    int index = weightedRandomIndex(weights);
                    target = data.targets.get(index);
                    data.targetsUsage.set(index, data.targetsUsage.get(index) + 1);
                }

                if (location == null && data.locations != null &&
!data.locations.isEmpty()) {
                    List<Integer> weights =
ObjectiveGenerator.usageToWeights(data.locationsUsage);
                    int index = weightedRandomIndex(weights);
                    location = data.locations.get(index);
                    data.locationsUsage.set(index, data.locationsUsage.get(index) +
1);
                }
            }
        }

        if (objectiveType == ObjectiveType.DEFEND) {
            target = location;
        }
    }

    public ObjectiveType getObjectiveType() { return objectiveType; }
    public String getTarget() { return target; }
    public String getLocation() { return location; }

    public static int weightedRandomIndex(List<Integer> weights) {
        int total = weights.stream().mapToInt(Integer::intValue).sum();
        int rand = new Random().nextInt(total);
        int cumulative = 0;
        for (int i = 0; i < weights.size(); i++) {
            cumulative += weights.get(i);
```

```
            if (rand < cumulative) return i;
        }
        return weights.size() - 1;
    }

    @Override
    public String toString() {
        if (!target.equals(location)) {
            return "target = " + target + ", location = " + location;
        }
        return "location = " + location;
    }
}
```

4.1.4. Генератор ObjectiveGenerator

Клас ObjectiveGenerator відповідає за генерацію об'єктів типу ObjectiveModifier, які визначають конкретні параметри завдання – зокрема ціль, локацію та опис. Він реалізує логіку обрання унікального наповнення квесту, використовуючи вагові коефіцієнти для зменшення повторюваності.

Основні функції:

- Генерація мети квесту залежно від типу (QuestType);
- Підстановка значень у шаблони назв та описів квестів;
- Перетворення лічильників використання на ваги для випадкового вибору.

Код класу:

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class ObjectiveGenerator {
    private static final Random random = new Random();

    public static ObjectiveModifier generateObjective(QuestType questType) {
        return switch (questType) {
            case MURDER -> new ObjectiveModifier(ObjectiveType.KILL, "", "", 1);
            case DELIVERY -> new ObjectiveModifier(ObjectiveType.DELIVER, "", "", 1);
            case GATHERING -> new ObjectiveModifier(ObjectiveType.GATHER, "", "",
random.nextInt(3) + 2);
            case ESCORT -> new ObjectiveModifier(ObjectiveType.ESCORT, "", "", 1);
            case DEFENSE -> new ObjectiveModifier(ObjectiveType.DEFEND, "", "", 1);
        };
    }

    public static List<ObjectiveModifier> generateOV(QuestType questType) {
        List<ObjectiveModifier> vars = new ArrayList<>();
        ObjectiveModifier objective = generateObjective(questType);
```

```
        objective.ensureFilled();
        vars.add(objective);
        return vars;
    }

    public static String generateQuestName(ObjectiveModifier objective) {
        ObjectiveType questType = objective.getObjectiveType();
        ConfigQ.loadDataFromJson("data/data.json");
        DataQ data = ConfigQ.get(questType.name());
        objective.ensureFilled();

        if (data != null && data.names != null && !data.names.isEmpty()) {
            List<Integer> weights = usageToWeights(data.namesUsage);
            int index = ObjectiveModifier.weightedRandomIndex(weights);
            String questName = data.names.get(index);
            data.namesUsage.set(index, data.namesUsage.get(index) + 1);

            return questName
                    .replace("{target}", objective.getTarget())
                    .replace("{location}", objective.getLocation());
        }
        return "Unknown Quest Name";
    }

    public static String generateQuestDescription(ObjectiveModifier objective) {
        ObjectiveType questType = objective.getObjectiveType();
        ConfigQ.loadDataFromJson("data/data.json");
        DataQ data = ConfigQ.get(questType.name());
        objective.ensureFilled();

        if (data != null && data.descriptions != null &&
!data.descriptions.isEmpty()) {
            List<Integer> weights = usageToWeights(data.descriptionsUsage);
            int index = ObjectiveModifier.weightedRandomIndex(weights);
            String questDescription = data.descriptions.get(index);
            data.descriptionsUsage.set(index, data.descriptionsUsage.get(index) + 1);

            return questDescription
                    .replace("{target}", objective.getTarget())
                    .replace("{location}", objective.getLocation());
        }
        return "Unknown Quest Description";
    }

    public static List<Integer> usageToWeights(List<Integer> usageCounters) {
        int max = usageCounters.stream().mapToInt(i -> i).max().orElse(0);
        List<Integer> weights = new ArrayList<>();
        for (int usage : usageCounters) {
            int weight = (max - usage) + 1;
            weights.add(weight);
        }
        return weights;
    }
}
```

4.1.5. Клас RewardModifier

Клас RewardModifier представляє окрему одиницю нагороди, яку гравець може отримати після виконання завдання. Він зберігає тип нагороди (згідно з перерахуванням RewardType), кількість (або умовну вагу), а також – за потреби – назву конкретного предмета, якщо йдеться про матеріальні винагороди (зброю, броню, матеріали тощо).

Клас надає метод toString() для зручного виводу вмісту, а також відповідні гетери для доступу до даних.

Основні функції:

- Зберігання типу, кількості та імені нагороди;
- Універсальне представлення у вигляді рядка;
- Простий і гнучкий у використанні разом із генератором нагород.

Код класу:

```
public class RewardModifier {
    private final RewardType rewardType;
    private final int amount;
    private final String itemName;

    public RewardModifier(RewardType rewardType, int amount, String itemName) {
        this.rewardType = rewardType;
        this.amount = amount;
        this.itemName = itemName;
    }

    public String toString() {
        return "[" +
                "rewardType = '" + rewardType + '\'' +
                (amount != 1 ? ", amount = " + amount : "") +
                (itemName != null ? ", itemName = '" + itemName + '\'' : "") +
                " ]";
    }
    public RewardType getRewardType() {
        return rewardType;
    }

    public int getAmount() {
        return amount;
    }

    public String getItemName() {
        return itemName;
    }
}
```

4.1.6. Генератор RewardGenerator

Клас RewardGenerator відповідає за створення динамічного списку нагород для квесту, враховуючи його рівень складності. Генерація базується на імовірностях, масштабуванні значень нагород залежно від складності, а також випадковому виборі предметів із зовнішнього джерела (JSON-файлу rewards.json).

Основні функції:

- Розрахунок кількості нагород залежно від складності (difficulty);
- Генерація золота (GOLD) та досвіду (EXPERIENCE) з масштабуванням;
- Додавання випадкових матеріальних винагород (предмети, зброя, броня, цінності);
- Об'єднання однакових нагород у загальний запис через метод mergeDuplicateRewards.

Код класу:

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class RewardGenerator {
    private static final Random random = new Random();

    public static List<RewardModifier> generateRewards(double difficulty) {
        List<RewardModifier> rewards = new ArrayList<>();

        int minRewards;
        int maxRewards;
        int nXP = getBaseRewardValue(RewardType.EXPERIENCE);
        int nGold = getBaseRewardValue(RewardType.GOLD);
        int diff = (int) Math.round(difficulty);

        if (diff <= 10) {
            minRewards = 1;
            maxRewards = 3;
        }
        else if (diff <= 20) {
            minRewards = 2;
            maxRewards = 4;
        }
        else {
            minRewards = 3;
            maxRewards = 5;
        }

        int nRewards = minRewards + random.nextInt(maxRewards - minRewards);

        RewardType[] rewardTypes = RewardType.values();

        int qDiff = (int) Math.round(difficulty) / 5;
```

```
        for (int i = 0; i < nRewards; i++) {
            RewardType chosenType = rewardTypes[random.nextInt(rewardTypes.length)];
            switch (chosenType) {
                case RewardType.GOLD:
                    nGold += Math.round((random.nextInt(100) + 50) * qDiff);
                    break;
                case RewardType.EXPERIENCE:
                    nXP += Math.round((random.nextInt(200) + 100) * qDiff);
                    break;
                case RewardType.ITEM:
                    rewards.add(new RewardModifier(RewardType.ITEM, 1,
getRandomReward(RewardType.ITEM)));
                    break;
                case RewardType.ARMOUR:
                    rewards.add(new RewardModifier(RewardType.ARMOUR, 1,
getRandomReward(RewardType.ARMOUR)));
                    break;
                case RewardType.WEAPON:
                    rewards.add(new RewardModifier(RewardType.WEAPON, 1,
getRandomReward(RewardType.WEAPON)));
                    break;
                case RewardType.MATERIAL:
                    rewards.add(new RewardModifier(RewardType.MATERIAL, 1,
getRandomReward(RewardType.MATERIAL)));
                    break;
                case RewardType.VALUABLE:
                    rewards.add(new RewardModifier(RewardType.VALUABLE, 1,
getRandomReward(RewardType.VALUABLE)));
                    break;
            }
        }
        rewards.add(new RewardModifier(RewardType.EXPERIENCE, nXP, null));
        rewards.add(new RewardModifier(RewardType.GOLD, nGold, null));

        return mergeDuplicateRewards(rewards);
    }

    private static List<RewardModifier> mergeDuplicateRewards(List<RewardModifier>
rewards) {
        for (int i = 0; i < rewards.size(); i++) {
            RewardModifier base = rewards.get(i);
            for (int j = i + 1; j < rewards.size(); j++) {
                RewardModifier compare = rewards.get(j);

                boolean sameType =
base.getRewardType().equals(compare.getRewardType());
                boolean sameItem =
                        (base.getItemName() == null && compare.getItemName() == null)
||
                                (base.getItemName() != null &&
base.getItemName().equals(compare.getItemName()));

                if (sameType && sameItem) {
                    int combinedAmount = base.getAmount() + compare.getAmount();
                    rewards.set(i, new RewardModifier(base.getRewardType(),
combinedAmount, base.getItemName()));
                    rewards.remove(j);
                    j--;
                }
            }
        }
        return rewards;
```

```
    }


    private static String getRandomReward(RewardType rt) {
        ConfigI.loadDataFromJson("data/rewards.json");
        DataI data = ConfigI.get(rt.name());
        if (data != null && data.names != null && !data.names.isEmpty())
            return data.names.get(random.nextInt(data.names.size()));
        return "Unknown Reward";
    }
    private static int getBaseRewardValue(RewardType type) {
        ConfigI.loadDataFromJson("data/rewards.json");
        DataI data = ConfigI.get(type.name());
        return data.baseValue;
    }
}
```

4.2. Запуск системи та консольний інтерфейс

Розроблена система процедурної генерації сюжетного контенту реалізована у вигляді Java-додатку з консольним інтерфейсом користувача. Її запуск виконується через основний клас Main, який ініціалізує необхідні компоненти, завантажує конфігураційні дані та запускає генерацію квестів на основі випадково вибраних параметрів.

Для запуску проєкту використовується середовище розробки IntelliJ IDEA Community Edition 2024.

Послідовність запуску:

1. Ініціалізується генератор квестів;
2. Завантажуються JSON-файли із даними;
3. Виводиться міні-меню із опціями для подальшого тестування та компіляції коду;
4. Відповідно до виборів користувача, можлива генерація квестів, емуляція пропуску часу, перегляд усіх активних доступних квестів у вигляді короткого списку, подробний перегляд одного з квестів на вибір, видалення одного з активних квестів на вибір, завершення роботи програми;

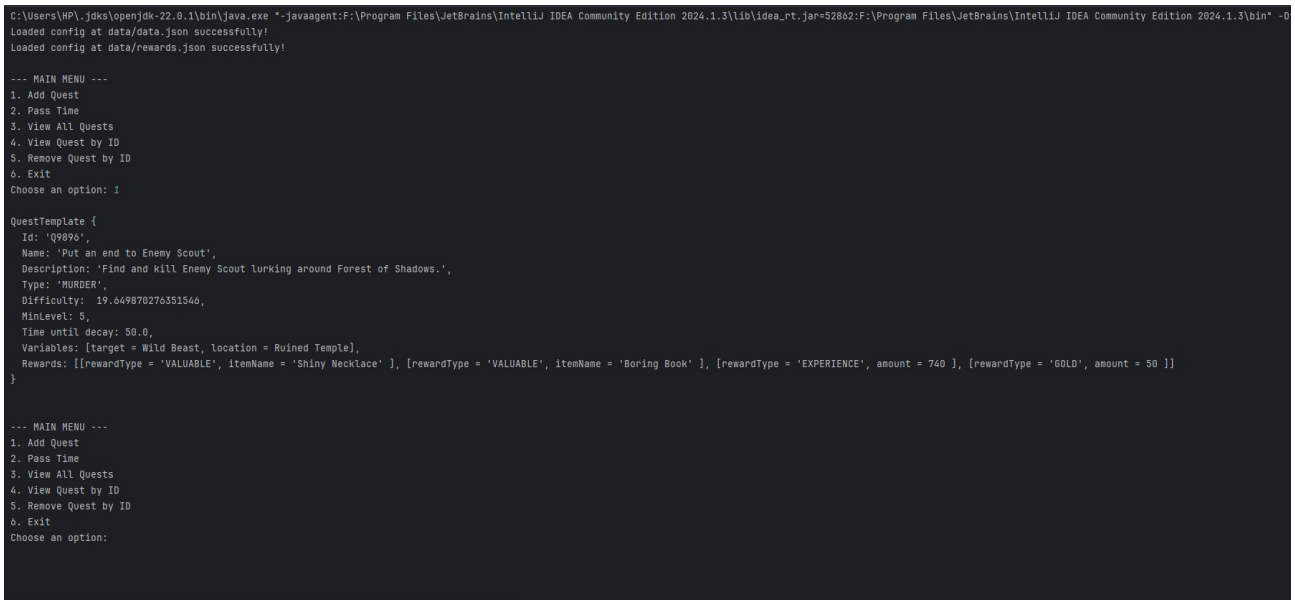
Консольний інтерфейс створений з метою спрощеного тестування та демонстрації результатів генерації. Він дозволяє розробнику відслідковувати хід генерації, переглядати параметри згенерованих квестів, а також перевіряти правильність логіки нагород.

Основні особливості:

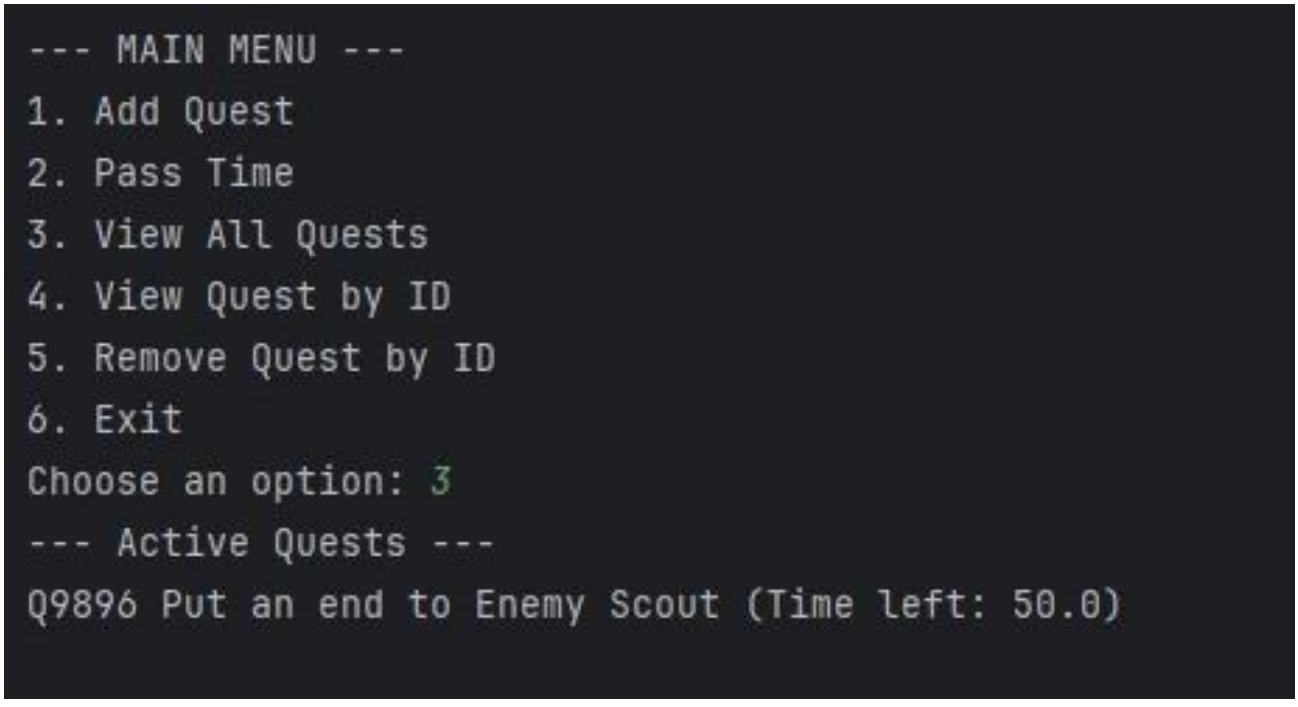
- Вивід згенерованих квестів у читабельному форматі;
- Повідомлення про можливі помилки конфігурації (наприклад, відсутній JSON-файл);
- Можливість багаторазової генерації шляхом повторного запуску програми;
- Легке розширення – у майбутньому можна реалізувати введення параметрів через консоль.

4.3. Приклади роботи системи

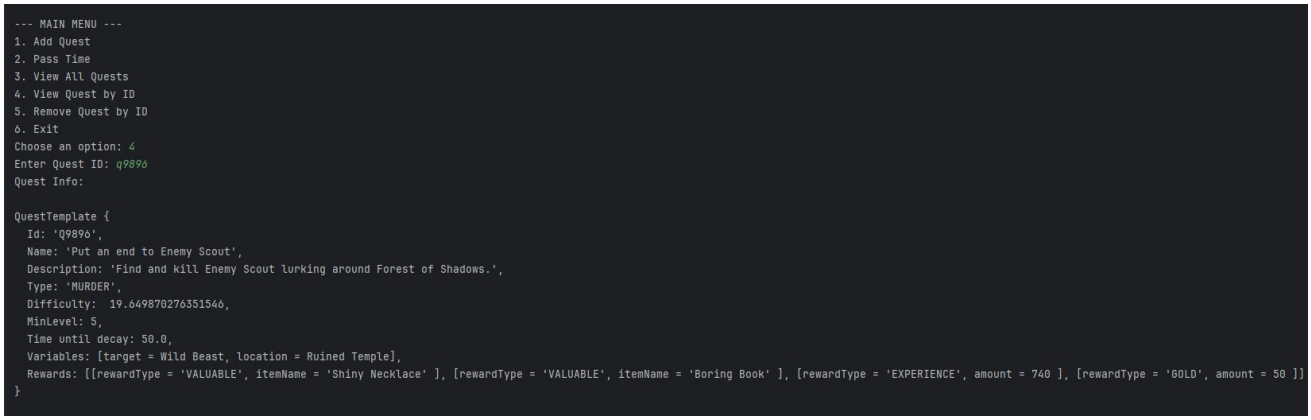
Перший запуск застосунку та генерація першого квесту:



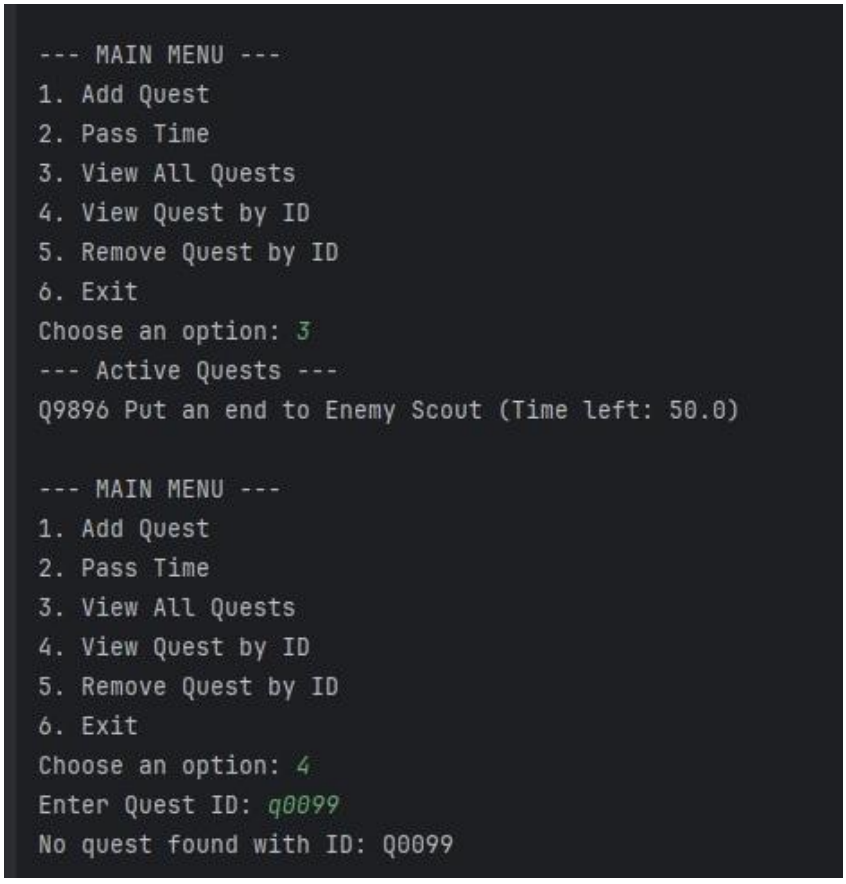
Короткий перегляд усіх активних квестів із лише одним згенерованим квестом:



Подробний перегляд активного квесту:



Неправильно введений ідентифікатор квесту при спробі показати подробну інформацію квесту:



Видалення квесту зі списку активних квестів:

```
--- MAIN MENU ---
1. Add Quest
2. Pass Time
3. View All Quests
4. View Quest by ID
5. Remove Quest by ID
6. Exit
Choose an option: 5
Enter Quest ID to remove: q9896
Quest Q9896 removed.

--- MAIN MENU ---
1. Add Quest
2. Pass Time
3. View All Quests
4. View Quest by ID
5. Remove Quest by ID
6. Exit
Choose an option: 3
--- Active Quests ---
```

Емуляція пропуску часу із декількома згенерованими квестами:

```
--- Active Quests ---
Q6345 Eliminate Wild Beast (Time left: 69.0)
Q7587 Protect the Riverside from incoming threat (Time left: 60.0)
Q2891 Escort the Merchant (Time left: 62.0)
Q8158 Collect some Medical Herbs (Time left: 36.0)
Q2341 Get your hands on some Monster Teeth (Time left: 32.0)
Q1954 Protect Scholar on the road (Time left: 45.0)
Q4827 Escort the Scholar (Time left: 43.0)
Q8426 Protect the Riverside from incoming threat (Time left: 81.0)

--- MAIN MENU ---
1. Add Quest
2. Pass Time
3. View All Quests
4. View Quest by ID
5. Remove Quest by ID
6. Exit
Choose an option: 2

How much time do you want to progress?
Please input your value: 45
Quest Q8158 has decayed.
Quest Q2341 has decayed.
Quest Q1954 has decayed.
Quest Q4827 has decayed.
Time ticked. Expired quests removed.
```

```
--- MAIN MENU ---
1. Add Quest
2. Pass Time
3. View All Quests
4. View Quest by ID
5. Remove Quest by ID
6. Exit
Choose an option: 3
--- Active Quests ---
Q6345 Eliminate Wild Beast (Time left: 24.0)
Q7587 Protect the Riverside from incoming threat (Time left: 15.0)
Q2891 Escort the Merchant (Time left: 17.0)
Q8426 Protect the Riverside from incoming threat (Time left: 36.0)
```

Завершення роботи коду через пункт закриття застосунку:

```
--- MAIN MENU ---
1. Add Quest
2. Pass Time
3. View All Quests
4. View Quest by ID
5. Remove Quest by ID
6. Exit
Choose an option: 6
Exiting...

Process finished with exit code 0
```

4.4. Висновки щодо практичної реалізації

У ході практичної реалізації системи процедурної генерації сюжетного контенту було створено базову архітектуру генерації квестів, яка передбачає використання шаблонів завдань, генераторів цілей та нагород, а також системи модифікаторів для налаштування змісту.

Основними результатами реалізації є:

- Побудова повнофункціонального прототипу генератора квестів із використанням об'єктно-орієнтованого підходу;
- Впровадження механізмів випадкової генерації на основі шаблонів;
- Розробка консольного інтерфейсу, що дозволяє запускати генерацію, переглядати результати та тестувати сценарії;
- Формування структури, що дозволяє легко додавати нові шаблони, типи нагород або параметри генерації.

Реалізована система підтвердила працездатність закладених ідей: усі компоненти генерації працюють узгоджено, забезпечуючи різноманітність створених квестів при збереженні структурної цілісності.

Система вже на цьому етапі може використовуватися як базова платформа для розширення – наприклад, для інтеграції із графічним інтерфейсом, додавання механізмів збереження історії або впровадження більш складних логік поведінки персонажів та наслідків рішень гравця.

ВИСНОВКИ

У ході виконання дипломної роботи було здійснено комплексне дослідження методів та підходів до процедурної генерації контенту у відеоіграх, зокрема в контексті створення сюжетного наповнення та квестів.

На етапі теоретичного аналізу були розглянуті базові принципи процедурної генерації та її застосування у сфері інтерактивного оповідання. Було проаналізовано існуючі методи генерації тексту й сюжетів, а також виділено ключові переваги й виклики, пов’язані із впровадженням PCG у наративну структуру гри. Окрему увагу приділено особливостям генерації квестів, де були ідентифіковані проблеми підтримання логічної зв’язності, гнучкості, балансу та відповідності до контексту гри.

У результаті аналізу були сформульовані вимоги до системи процедурної генерації квестів, після чого було розроблено концептуальну модель, що включає категоризацію квестів за типами, тематикою, масштабом, складністю, учасниками, а також механізмами прийняття чи відхилення завдань. Крім того, була запропонована система обробки моральних рішень гравця, яка впливає на перебіг подій у грі та варіативність сюжетних гілок.

На практичному етапі був реалізований програмний прототип системи генерації квестів із використанням мови програмування Java. Реалізована система дозволяє динамічно створювати квести з урахуванням категорій, контексту гравця, частоти використання тем, а також балансу між різними типами завдань. Було протестовано функціональність генератора, виявлено його сильні сторони (гнучкість, масштабованість, модульність) і обмеження (потреба в більш глибокій інтеграції з системою подій у грі).

За результатами дослідження можна зробити такі висновки:

1. Процедурна генерація має значний потенціал у контексті створення сюжетного наповнення відеоігор і дозволяє автоматизувати рутинні процеси розробки контенту.

2. Генерація квестів є складним завданням, що вимагає врахування логіки, варіативності та ігрового балансу. Використання категорій, обмежень і механізмів контролю повторень забезпечує більшу якість згенерованого контенту.
3. Інтеграція моральних систем у процес генерації дозволяє зробити сюжет більш адаптивним до дій гравця та підвищити загальну глибину оповідного контенту.
4. Розроблений прототип продемонстрував практичну доцільність та життєздатність запропонованої моделі генерації, забезпечивши базову функціональність та основу для подальшого розвитку.

Перспективами подальших досліджень є:

- поглиблення адаптивності системи до особистісного стилю гравця;
- інтеграція з графічним рушієм гри;
- застосування ШІ для покращення релевантності квестів;
- розширення бази категорій і сценаріїв взаємодії.

Таким чином, мета дипломної роботи була досягнута, а результати дослідження підтверджують ефективність використання процедурної генерації для створення інтерактивного сюжетного контенту у відеоіграх.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Процедурна генерація. URL: https://uk.wikipedia.org/wiki/Процедурна_генерація (дата звернення 07.01.2025).

2. Кейт Комптон URL: https://www.academia.edu/69211262/Generative_Methods

3. Повна випадковість URL: https://www.wired.com/2003/08/random/ (дата звернення 07.01.2025).

4. Псевдовипадковість URL: https://en.wikipedia.org/wiki/Pseudorandom_generator (дата звернення 07.01.2025).

5. Анураг Саркар та Сет Купер «Procedural Content Generation using Behavior Trees» URL: https://arxiv.org/abs/2107.06638

6. Шум Перліна. URL: https://uk.wikipedia.org/wiki/Шум_Перлина (дата звернення 08.01.2025).

7. Симплекс-шум. URL: https://uk.wikipedia.org/wiki/Симплекс-шум (дата звернення 08.01.2025).

8. Voronoi Diagram URL: https://en.wikipedia.org/wiki/Voronoi_diagram (лата звернення 08.01.2025)

9. Worley Noise. URL: https://en.wikipedia.org/wiki/Worley_noise (дата звернення 08.01.2025).

10. Фрактали. URL: https://en.wikipedia.org/wiki/Fractal (дата звернення 09.01.2025).

11. L-system. URL: https://en.wikipedia.org/wiki/L-system (дата звернення 09.01.2025).

12. Diamond-Square Algorithm. URL: https://en.wikipedia.org/wiki/Diamond-square_algorithm (дата звернення 09.01.2025).

13. Cellular Automaton. URL: https://en.wikipedia.org/wiki/Cellular_automaton (дата звернення 10.01.2025).

14. Delaunay Triangulation. URL: https://en.wikipedia.org/wiki/Delaunay_triangulation (дата звернення 11.01.2025).

15. Binary Space Partitioning. URL:

https://en.wikipedia.org/wiki/Binary_space_partitioning (дата звернення 11.01.2025).

16. Genetic Algorithm. URL: https://en.wikipedia.org/wiki/Genetic_algorithm (дата звернення 11.01.2025).

17. Функція пристосованості. URL: https://en.wikipedia.org/wiki/Fitness_function (дата звернення 11.01.2025).

18. Markov Chain. URL: https://en.wikipedia.org/wiki/Markov_chain (дата звернення 11.01.2025).

19. Митник Д.О. «ПОРІВНЯННЯ ЛАНЦЮГІВ МАРКОВА ТА НЕЙРОМЕРЕЖ ДЛЯ ВИРІШЕННЯ ЗАДАЧІ ГЕНЕРАЦІЇ ВІРШІВ» URL: https://ela.kpi.ua/server/api/core/bitstreams/16beb0c6-d563-442d-9d27-6dfd457aaf4e/content

20. Александр Гелел та Пенні Світсер «A Hybrid Approach to Procedural Generation of Roguelike Video Game Levels» URL: openresearch-repository.anu.edu.au/bitstream/1885/205015/5/FDG20_PCG-submitted.pdf

21. Шаблонні структури та елементи випадковості URL: http://www.reddit.com/r/proceduralgeneration/comments/uaxik8/procedural_actionadventure_map_including_skill/ (дата звернення 11.01.2025)

22. Гібридна генерація контенту із розширеними графовими алгоритмами URL: https://www.reddit.com/r/NoMansSkyTheGame/comments/mq3egk/a_practical_look_at_procedural_generation_20_or/?utm_source=chatgpt.com (дата звернення 11.01.2025)

23. Генерація текстів та діалогів з використанням ШІ URL: doomlaser.com/openai-api-generated-video-game-dialog-with-real-time-text-to-speech (дата звернення 11.01.2025)

24. Використання ШІ у балансуванні рівнів URL: https://www.getgud.io/blog/leveraging-ai-for-procedural-content-generation-in-game-development/

25. Сіню Мао «Procedural Content Generation via Generative Artificial Intelligence» URL: https://arxiv.org/html/2407.09013v1

26. Генератор Шуму Перліна. URL: http://kitfox.com/projects/perlinNoiseMaker/

27. Генератор Сиплекс-шуму. URL: https://codesandbox.io/p/sandbox/simplex-noise-

texture-generator-ufjxs

28. Генератор L-системи. URL: https://nolandc.com/sandbox/fractals/

29. Генератор методу Diamond-Square. URL: https://diamond-square.netlify.app

30. Генератор Елементарного Клітинного Автомата. URL: https://devinacker.github.io/celldemo/

31. Генератор Генетичного Алгоритму. URL: https://www.geneticalgorithms.online/imagegeneration

32. Буданов А.О. «Розробка алгоритму процедурної генерації у мультимедійній сфері». URL: https://krs.chmnu.edu.ua/jspui/bitstream/123456789/2340/1/401%20Буданов%20Андрій%20Олегович.pdf

33. Балінт Д.Т., Бідарра Р. «Procedural Generation of Narrative Worlds» URL: https://pure.tudelft.nl/ws/portalfiles/portal/155297989/Procedural_Generation_of_Narrative_Worlds.pdf

34. Арамбепола Н., Ранасінгха Ч.П. «Large Language Models for dynamic game content: procedural side-quest generation» URL: https://www.researchgate.net/publication/385292792_Large_Language_Models_for_dynamic_game_content_procedural_side-quest_generation

35. Йосуп О., Й. Велден «Procedural Content Generation for Games: A Survey» URL: https://www.researchgate.net/publication/262327212_Procedural_Content_Generation_for_Games_A_Survey

ДОДАТКИ

Додаток А. Повний код програмного застосунку

Main.java:

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        QuestManager questManager = new QuestManager();
        Scanner scanner = new Scanner(System.in);
        boolean running = true;

        ConfigQ.loadDataFromJson("data/data.json");
        ConfigI.loadDataFromJson("data/rewards.json");

        while (running) {
            System.out.println("\n--- MAIN MENU ---");
            System.out.println("1. Add Quest");
            System.out.println("2. Pass Time");
            System.out.println("3. View All Quests");
            System.out.println("4. View Quest by ID");
            System.out.println("5. Remove Quest by ID");
            System.out.println("6. Exit");
            System.out.print("Choose an option: ");

            String choice = scanner.nextLine();

            switch (choice) {
                case "1" -> {
                    questManager.addQuest();
                    System.out.println();
                }
                case "2" -> {
                    questManager.update();
                    System.out.println("Time ticked. Expired quests removed.");
                }
                case "3" -> {
                    System.out.println("--- Active Quests ---");
                    for (QuestTemplate q : questManager.getActiveQuests()) {
                        System.out.println(q.getId() + " " + q.getName() + " (Time
left: " + q.getAliveTime() + ")");
                    }
                }
                case "4" -> {
                    System.out.print("Enter Quest ID: ");
                    String id = scanner.nextLine().toUpperCase();
                    QuestTemplate quest = questManager.getQuestById(id);
                    if (quest != null) {
                        System.out.println("Quest Info:\n" + quest);
                    } else {
                        System.out.println("No quest found with ID: " + id);
                    }
                }
                case "5" -> {
                    System.out.print("Enter Quest ID to remove: ");
                    String id = scanner.nextLine().toUpperCase();
                    if (questManager.containsQuest(id)) {
                        questManager.removeQuestById(id);
```

```
                        System.out.println("Quest " + id + " removed.");
                    } else {
                        System.out.println("No quest found with that ID.");
                    }
                }
                case "6" -> {
                    running = false;
                    System.out.println("Exiting...");
                }
                default -> System.out.println("Invalid option. Try again.");
            }
        }

        scanner.close();
    }
}
```

QuestTemplate.java:

```
import java.util.List;

public class QuestTemplate {
    private final String id;
    private final String name;
    private final String description;
    private final String type;
    private final double difficulty;
    private final int minLevel;
    private double aliveTime;
    private final List<ObjectiveModifier> variables;
    private final List<RewardModifier> rewards;

    public QuestTemplate(String id, String name, String description, String type,
                         double difficulty, int minLevel, double aliveTime,
                         List<ObjectiveModifier> variables, List<RewardModifier>
rewards) {
        this.id = id;
        this.name = name;
        this.description = description;
        this.type = type;
        this.difficulty = difficulty;
        this.minLevel = minLevel;
        this.aliveTime = aliveTime;
        this.variables = variables;
        this.rewards = rewards;
    }

    public String getName() {
        return name;
    }

    public double getAliveTime() {
        return aliveTime;
    }

    public String getId() {
        return id;
    }

    public void setAliveTime(double aliveTime) {
        this.aliveTime = aliveTime;
    }
```

```
    @Override
    public String toString() {
        return "\nQuestTemplate {" +
                "\n  Id: '" + id + '\'' +
                ",\n  Name: '" + name + '\'' +
                ",\n  Description: '" + description + '\'' +
                ",\n  Type: '" + type + '\'' +
                ",\n  Difficulty:  " + difficulty +
                ",\n  MinLevel: " + minLevel  +
                ",\n  Time until decay: " + aliveTime +
                ",\n  Variables: " + variables +
                ",\n  Rewards: " + rewards +
                "\n}";
    }
}
```

QuestTemplateGenerator.java:

```
import java.util.List;
import java.util.Random;

public class QuestTemplateGenerator {
    private static final Random random = new Random();

    public static QuestTemplate generate(String id) {
        QuestType questType =
QuestType.values()[random.nextInt(QuestType.values().length)];
        ObjectiveModifier objective =
ObjectiveGenerator.generateObjective(questType);
        String name = ObjectiveGenerator.generateQuestName(objective);
        String description = ObjectiveGenerator.generateQuestDescription(objective);
        double difficulty = generateDifficulty(questType);
        int minLevel = generateminLevel();
        int aliveTime = (int) Math.round(generateAliveTime(questType));


        List<ObjectiveModifier> vars = ObjectiveGenerator.generateOV(questType);
        List<RewardModifier> rewards = RewardGenerator.generateRewards(difficulty);

        return new QuestTemplate(
                id,
                name,
                description,
                questType.name(),
                difficulty,
                minLevel,
                aliveTime,
                vars,
                rewards
        );
    }

    private static double generateDifficulty(QuestType type) {
        return switch (type) {
            case MURDER -> 3.5 + random.nextDouble(13.25) * 2;
            case DELIVERY -> 1.5 + random.nextDouble(19) * 1.5;
            case GATHERING -> 3.0 + random.nextDouble(18) * 1.5;
            case ESCORT -> 3.0 + random.nextDouble(13.5) * 2;
            case DEFENSE -> 4.0 + random.nextDouble(13) * 2;
        };
    }
```

```
    private static int generateminLevel() {
        return random.nextInt(5) + 1;
    }

    private static double generateAliveTime(QuestType type) {
        return switch (type) {
            case MURDER -> 45 + random.nextDouble() * 30;
            case DELIVERY -> 20 + random.nextDouble() * 20;
            case GATHERING -> 30 + random.nextDouble() * 30;
            case ESCORT -> 40 + random.nextDouble() * 30;
            case DEFENSE -> 50 + random.nextDouble() * 40;
        };
    }
}
```

QuestManager.java:

```
import java.util.*;

public class QuestManager {
    private final Map<String, QuestTemplate> activeQuests = new HashMap<>();

    private static final Random random = new Random();

    Scanner scanner = new Scanner(System.in);

    public void addQuest() {
        String id = generateUniqueQuestId();
        QuestTemplate template = QuestTemplateGenerator.generate(id);
        activeQuests.put(id, template);
        System.out.println(template);
    }

    public void update() {
        List<String> toRemove = new ArrayList<>();
        System.out.print("\nHow much time do you want to progress?\nPlease input your
value: ");
        double chosenTime = scanner.nextInt();
        for (QuestTemplate quest : activeQuests.values()) {
            double time = quest.getAliveTime();
            time = time - chosenTime;
            quest.setAliveTime(time);

            if (time <= 0) {
                toRemove.add(quest.getId());
            }
        }
        for (String id : toRemove) {
            activeQuests.remove(id);
            System.out.println("Quest " + id + " has decayed.");
        }
    }

    private String generateUniqueQuestId() {
        String id;
        do {
            id = "Q" + (1000 + random.nextInt(9000));
        } while (activeQuests.containsKey(id));
        return id;
    }
```

```
    public QuestTemplate getQuestById(String id) {
        return activeQuests.get(id);
    }

    public Collection<QuestTemplate> getActiveQuests() {
        return activeQuests.values();
    }

    public void removeQuestById(String id) {
        activeQuests.remove(id);
    }

    public boolean containsQuest(String id) {
        return activeQuests.containsKey(id);
    }
}
```

QuestType.java

```
public enum QuestType {
    MURDER,
    DELIVERY,
    GATHERING,
    ESCORT,
    DEFENSE
}
```

ObjectiveType.java

```
public enum ObjectiveType {
    KILL,
    DELIVER,
    GATHER,
    ESCORT,
    DEFEND
}
```

ObjectiveModifier.java:

```
import java.util.List;
import java.util.Random;

public class ObjectiveModifier {
    private final ObjectiveType objectiveType;
    private String target;
    private String location;

    public ObjectiveModifier(ObjectiveType objectiveType, String target, String
location, int quantity) {
        this.objectiveType = objectiveType;
        this.target = null;
        this.location = null;
    }

    public void ensureFilled() {
        if (target == null || location == null) {
            DataQ data = ConfigQ.get(objectiveType.name());
            if (data != null) {
                if (target == null && data.targets != null &&
```

```
!data.targets.isEmpty()) {
                    List<Integer> weights =
ObjectiveGenerator.usageToWeights(data.targetsUsage);
                    int index = weightedRandomIndex(weights);
                    target = data.targets.get(index);
                    data.targetsUsage.set(index, data.targetsUsage.get(index) + 1);
                }

                if (location == null && data.locations != null &&
!data.locations.isEmpty()) {
                    List<Integer> weights =
ObjectiveGenerator.usageToWeights(data.locationsUsage);
                    int index = weightedRandomIndex(weights);
                    location = data.locations.get(index);
                    data.locationsUsage.set(index, data.locationsUsage.get(index) +
1);
                }

            }
        }
        if (objectiveType==ObjectiveType.DEFEND) {
            target = location;
        }
    }

    public ObjectiveType getObjectiveType() { return objectiveType; }
    public String getTarget() { return target; }
    public String getLocation() { return location; }

    public static int weightedRandomIndex(List<Integer> weights) {
        int total = weights.stream().mapToInt(Integer::intValue).sum();
        int rand = new Random().nextInt(total);
        int cumulative = 0;
        for (int i = 0; i < weights.size(); i++) {
            cumulative += weights.get(i);
            if (rand < cumulative) return i;
        }
        return weights.size() - 1;
    }


    @Override
    public String toString() {
        if (!target.equals(location)) {
            return "target = " + target + ", location = " + location;
        }
        return "location = " + location;
    }
}
```

ObjectiveGenerator.java:

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class ObjectiveGenerator {
    private static final Random random = new Random();

    public static ObjectiveModifier generateObjective(QuestType questType) {
        return switch (questType) {
            case MURDER -> new ObjectiveModifier(
```

```
                    ObjectiveType.KILL,
                    "",
                    "",
                    1
            );
            case DELIVERY -> new ObjectiveModifier(
                    ObjectiveType.DELIVER,
                    "",
                    "",
                    1
            );
            case GATHERING -> new ObjectiveModifier(
                    ObjectiveType.GATHER,
                    "",
                    "",
                    random.nextInt(3) + 2
            );
            case ESCORT ->
                    new ObjectiveModifier(
                    ObjectiveType.ESCORT,
                    "",
                    "",
                    1
            );
            case DEFENSE ->
                    new ObjectiveModifier(
                    ObjectiveType.DEFEND,
                    "",
                    "",
                    1
            );
        };
    }

    public static String generateQuestName(ObjectiveModifier objective) {
        ObjectiveType questType = objective.getObjectiveType();
        ConfigQ.loadDataFromJson("data/data.json");
        DataQ data = ConfigQ.get(questType.name());
        objective.ensureFilled();
        if (data != null && data.names != null && !data.names.isEmpty()) {
            List<Integer> weights =
ObjectiveGenerator.usageToWeights(data.namesUsage);
            int index = ObjectiveModifier.weightedRandomIndex(weights);
            String questName = data.names.get(index);
            data.namesUsage.set(index, data.namesUsage.get(index) + 1);
            questName = questName.replace("{target}", objective.getTarget())
                    .replace("{location}", objective.getLocation());
            return questName;
        }
        return "Unknown Quest Name";
    }

    public static String generateQuestDescription(ObjectiveModifier objective) {
        ObjectiveType questType = objective.getObjectiveType();
        ConfigQ.loadDataFromJson("data/data.json");
        DataQ data = ConfigQ.get(questType.name());
        objective.ensureFilled();
        if (data != null && data.descriptions != null &&
!data.descriptions.isEmpty()) {
            List<Integer> weights =
ObjectiveGenerator.usageToWeights(data.descriptionsUsage);
            int index = ObjectiveModifier.weightedRandomIndex(weights);
            String questDescription = data.descriptions.get(index);
```

```
            data.descriptionsUsage.set(index, data.descriptionsUsage.get(index) + 1);

            if (questDescription.contains("{target}")) {
                questDescription = questDescription.replace("{target}",
objective.getTarget());
            }
            if (questDescription.contains("{location}")) {
                questDescription = questDescription.replace("{location}",
objective.getLocation());
            }
            return questDescription;
        }
        return "Unknown Quest Description";
    }

    public static List<Integer> usageToWeights(List<Integer> usageCounters) {
        int max = usageCounters.stream().mapToInt(i -> i).max().orElse(0);
        List<Integer> weights = new ArrayList<>();
        for (int usage : usageCounters) {
            int weight = (max - usage) + 1;
            weights.add(weight);
        }
        return weights;
    }


    public static List<ObjectiveModifier> generateOV(QuestType questType) {
        List<ObjectiveModifier> vars = new ArrayList<>();

        ObjectiveModifier objective = switch (questType) {
            case MURDER -> new ObjectiveModifier(ObjectiveType.KILL, null, null, 1);
            case GATHERING -> new ObjectiveModifier(ObjectiveType.GATHER, null, null,
random.nextInt(3) + 2);
            case ESCORT -> new ObjectiveModifier(ObjectiveType.ESCORT, null, null,
1);
            case DEFENSE -> new ObjectiveModifier(ObjectiveType.DEFEND, null, null,
1);
            case DELIVERY -> new ObjectiveModifier(ObjectiveType.DELIVER, null, null,
1);
        };
        objective.ensureFilled();
        vars.add(objective);
        return vars;
    }
}
```

RewardType.java:

```
public enum RewardType  {
    GOLD,
    EXPERIENCE,
    ITEM,
    WEAPON,
    ARMOUR,
    MATERIAL,
    VALUABLE
}
```

RewardModifier.java:

```java
public class RewardModifier {
    private final RewardType rewardType;
    private final int amount;
    private final String itemName;

    public RewardModifier(RewardType rewardType, int amount, String itemName) {
        this.rewardType = rewardType;
        this.amount = amount;
        this.itemName = itemName;
    }

    public String toString() {
        return "[" +
                "rewardType = '" + rewardType + '\'' +
                (amount != 1 ? ", amount = " + amount : "") +
                (itemName != null ? ", itemName = '" + itemName + '\'' : "") +
                " ]";
    }
    public RewardType getRewardType() {
        return rewardType;
    }

    public int getAmount() {
        return amount;
    }

    public String getItemName() {
        return itemName;
    }
}
```

RewardGenerator.java:

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class RewardGenerator {
    private static final Random random = new Random();

    public static List<RewardModifier> generateRewards(double difficulty) {
        List<RewardModifier> rewards = new ArrayList<>();

        int minRewards;
        int maxRewards;
        int nXP = getBaseRewardValue(RewardType.EXPERIENCE);
        int nGold = getBaseRewardValue(RewardType.GOLD);
        int diff = (int) Math.round(difficulty);

        if (diff <= 10) {
            minRewards = 1;
            maxRewards = 3;
        }
        else if (diff <= 20) {
            minRewards = 2;
            maxRewards = 4;
        }
        else {
            minRewards = 3;
            maxRewards = 5;
        }
```

```
        int nRewards = minRewards + random.nextInt(maxRewards - minRewards);

        RewardType[] rewardTypes = RewardType.values();

        int qDiff = (int) Math.round(difficulty) / 5;

        for (int i = 0; i < nRewards; i++) {
            RewardType chosenType = rewardTypes[random.nextInt(rewardTypes.length)];
            switch (chosenType) {
                case RewardType.GOLD:
                    nGold += Math.round((random.nextInt(100) + 50) * qDiff);
                    break;
                case RewardType.EXPERIENCE:
                    nXP += Math.round((random.nextInt(200) + 100) * qDiff);
                    break;
                case RewardType.ITEM:
                    rewards.add(new RewardModifier(RewardType.ITEM, 1,
getRandomReward(RewardType.ITEM)));
                    break;
                case RewardType.ARMOUR:
                    rewards.add(new RewardModifier(RewardType.ARMOUR, 1,
getRandomReward(RewardType.ARMOUR)));
                    break;
                case RewardType.WEAPON:
                    rewards.add(new RewardModifier(RewardType.WEAPON, 1,
getRandomReward(RewardType.WEAPON)));
                    break;
                case RewardType.MATERIAL:
                    rewards.add(new RewardModifier(RewardType.MATERIAL, 1,
getRandomReward(RewardType.MATERIAL)));
                    break;
                case RewardType.VALUABLE:
                    rewards.add(new RewardModifier(RewardType.VALUABLE, 1,
getRandomReward(RewardType.VALUABLE)));
                    break;
            }
        }
        rewards.add(new RewardModifier(RewardType.EXPERIENCE, nXP, null));
        rewards.add(new RewardModifier(RewardType.GOLD, nGold, null));

        return mergeDuplicateRewards(rewards);
    }

    private static List<RewardModifier> mergeDuplicateRewards(List<RewardModifier>
rewards) {
        for (int i = 0; i < rewards.size(); i++) {
            RewardModifier base = rewards.get(i);
            for (int j = i + 1; j < rewards.size(); j++) {
                RewardModifier compare = rewards.get(j);

                boolean sameType =
base.getRewardType().equals(compare.getRewardType());
                boolean sameItem =
                        (base.getItemName() == null && compare.getItemName() == null)
||
                                (base.getItemName() != null &&
base.getItemName().equals(compare.getItemName()));

                if (sameType && sameItem) {
                    int combinedAmount = base.getAmount() + compare.getAmount();
                    rewards.set(i, new RewardModifier(base.getRewardType(),
combinedAmount, base.getItemName()));
                    rewards.remove(j);
```

```
                    j--;
                }
            }
        }
        return rewards;
    }


    private static String getRandomReward(RewardType rt) {
        ConfigI.loadDataFromJson("data/rewards.json");
        DataI data = ConfigI.get(rt.name());
        if (data != null && data.names != null && !data.names.isEmpty())
            return data.names.get(random.nextInt(data.names.size()));
        return "Unknown Reward";
    }
    private static int getBaseRewardValue(RewardType type) {
        ConfigI.loadDataFromJson("data/rewards.json");
        DataI data = ConfigI.get(type.name());
        return data.baseValue;
    }
}
```

DataI.java:

```
import java.util. List;

public class DataI {
    public int baseValue;
    List<String> names;
}
```

DataQ.java:

```
import java.util.ArrayList;
import java.util.List;

public class DataQ {
    List<String> targets;
    List<String> locations;
    List<String> names;
    List<String> descriptions;

    public List<Integer> targetsUsage = new ArrayList<>();
    public List<Integer> locationsUsage = new ArrayList<>();
    public List<Integer> namesUsage = new ArrayList<>();
    public List<Integer> descriptionsUsage = new ArrayList<>();

    public void initializeUsageCounters() {
        targetsUsage = initZeroList(targets);
        locationsUsage = initZeroList(locations);
        namesUsage = initZeroList(names);
        descriptionsUsage = initZeroList(descriptions);
    }


    private List<Integer> initZeroList(List<String> source) {
        List<Integer> result = new ArrayList<>();
        for (int i = 0; i < (source != null ? source.size() : 0); i++) {
            result.add(0);
        }
```

```
        return result;
    }
}
```

ConfigI.java:

```
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

public class ConfigI extends HashMap<String, DataI> {
    private static Map<String, DataI> configDataI = new HashMap<>();

    public static void loadDataFromJson(String filepath) {
        if (configDataI != null && !configDataI.isEmpty()) return;
        try {
            configDataI = ConfigILoader.loadConfig(filepath);
            if (configDataI.isEmpty()) {
                System.out.println("Config data at " + filepath + " is empty!");
            } else {
                System.out.println("Loaded config at " + filepath + "
successfully!");
            }
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    public static DataI get(String rewardsType) {
        return configDataI.get(rewardsType);
    }
}
```

ConfigILoader.java

```
import com.google.gson.Gson;
import com.google.gson.reflect.TypeToken;
import java.io.FileReader;
import java.io.IOException;
import java.util.Map;

public class ConfigILoader {

    public static ConfigI loadConfig(String filePath) throws IOException {
        Gson gson = new Gson();
        FileReader reader = new FileReader(filePath);
        Map<String, DataI> dataMapI = gson.fromJson(reader, new TypeToken<Map<String,
DataI>>(){}.getType());
        ConfigI configI = new ConfigI();
        configI.putAll(dataMapI);
        reader.close();
        return configI;
    }
}
```

ConfigQ:

```
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

public class ConfigQ extends HashMap<String, DataQ> {
    private static Map<String, DataQ> configData = new HashMap<>();

    public static void loadDataFromJson(String filepath) {
        if (configData != null && !configData.isEmpty()) return;
        try {
            configData = ConfigQLoader.loadConfig(filepath);
            if (configData.isEmpty()) {
                System.out.println("Config data at " + filepath + " is empty!");
            } else {
                System.out.println("Loaded config at " + filepath + "
successfully!");
            }
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    public static DataQ get(String questType) {
        return configData.get(questType);
    }
}
```

ConfigQLoader.java:

```
import com.google.gson.Gson;
import com.google.gson.reflect.TypeToken;
import java.io.FileReader;
import java.io.IOException;
import java.util.Map;

public class ConfigQLoader {

    public static ConfigQ loadConfig(String filePath) throws IOException {
        Gson gson = new Gson();
        FileReader reader = new FileReader(filePath);
        Map<String, DataQ> dataMap = gson.fromJson(reader, new TypeToken<Map<String,
DataQ>>(){}.getType());
        ConfigQ config = new ConfigQ();
        config.putAll(dataMap);
        reader.close();
        for (DataQ dq : config.values()) {
            dq.initializeUsageCounters();
        }
        return config;
    }
}
```

Додаток Б JSON-файли із даними для шаблонів

rewards.json:

```
{
  "GOLD": {
    "names": [],
    "baseValue": 50
  },
  "EXPERIENCE": {
    "names": [],
    "baseValue": 100
  },
  "ITEM": {
    "names": ["Mystic Cloak", "Ring of Speed", "Healing Potion", "Mana Potion",
"Magic Scroll", "Amulet"]
  },
  "WEAPON": {
    "names": ["Bastard Sword", "Falchion", "Bow of Despair", "Ice Daggers",
"Gauntlet", "Flint Pistol", "Longsword", "Crossbow"]
  },
  "ARMOUR": {
    "names": ["Light Helmet", "Sandals", "Robe", "Rough Chestplate", "Steel Horned
Helmet", "Rawhide Pants"]
  },
  "MATERIAL": {
    "names": ["Iron Ore", "Copper Ore", "Medical Herbs", "Rare Ores", "Spring Wood",
"Blessed Water"]
  },
  "VALUABLE": {
    "names": ["Poach of Gems", "Shiny Necklace", "Colourful Rock", "Boring Book"]
  }
}
```

data.json:

```
{
  "KILL": {
    "targets": ["Bandit Leader", "Wild Beast", "Corrupt Official", "Ferocious Demon",
"Enemy Scout"],
    "locations": ["Forest of Shadows", "Ruined Temple", "Mountain Pass", "Swamp of
Sorrows"],
    "names": [
      "Eliminate {target}",
      "Hunt down the {target}",
      "Neutralize {target}",
      "Put an end to {target}"
    ],
    "descriptions": [
      "You are to eliminate {target} near {location}.",
      "Find and kill {target} lurking around {location}.",
      "The {target} is causing trouble around {location}, stop them."
    ]
  },
  "DELIVER": {
    "targets": ["Ancient Relic", "Lost Scroll", "Secret Message", "Potion Crate"],
    "locations": ["Silver Town","Capital City", "Harbor", "Remote Village", "Merchant
Camp", "Riverside", "Lonely Watchtower"],
    "names": [
      "Deliver the {target}",
      "Bring {target} to its destination",
```

```
      "Transport {target}"
    ],
    "descriptions": [
      "Take the {target} safely to the {location}.",
      "Ensure the {target} arrives intact at the {location}.",
      "Bring {target} to {location} in one piece."
    ]
  },
  "GATHER": {
    "targets": ["Medical Herbs", "Rare Ores", "Sacred Water", "Monster Teeth"],
    "locations": ["Ancient Grove","Crystal Cavern", "Forest of Shadows", "Swamp of
Sorrows", "Highlands"],
    "names": [
      "Gather {target}",
      "Collect some {target}",
      "Get your hands on some {target}"
    ],
    "descriptions": [
      "Collect {target} at the {location}.",
      "Find and bring back some {target} from the {location}.",
      "Harvest some {target} at the {location}"
    ]
  },
  "ESCORT": {
    "targets": ["Merchant", "Scholar", "Messenger", "Noble", "Wounded Soldier"],
    "locations": ["Silver Town","Capital City", "Harbor", "Remote Village", "Merchant
Camp", "Riverside", "Lonely Watchtower"],
    "names": [
      "Escort the {target}",
      "Protect {target} on the road",
      "Accompany the {target} safely"
    ],
    "descriptions": [
      "Ensure {target} reaches {location} unharmed.",
      "Escort {target} to {location} avoiding all threats.",
      "Guide {target} on the journey to {location}.",
      "Safely escort {target} to {location}."
    ]
  },
  "DEFEND": {
    "targets": ["empty1", "empty2"],
    "locations": ["Silver Town","Capital City", "Harbor", "Remote Village", "Merchant
Camp", "Riverside", "Lonely Watchtower"],
    "names": [
      "Defend the {location}",
      "Hold the line at {location}",
      "Protect the {location} from incoming threat"
    ],
    "descriptions": [
      "Defend the area around {location} from enemy threats.",
      "Hold your ground at {location} until reinforcements arrive.",
      "Protect {location} from the incoming threat."
    ]
  }
}
```