Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій

Кафедра програмних систем і технологій

УДК 004.946 *На правах рукопису*

ВИПУСКНА КВАЛІФІКАЦІЙНА БАКАЛАВРСЬКА РОБОТА

Тема: “ Розробка гри в жанрі двовимірного rogue-like платформера ”

(*назва згідно з наказом ректора*)

Спеціальність – 121 “Інженерія програмного забезпечення”

ПОЯСНЮВАЛЬНА ЗАПИСКА

БР.ІПЗ - 14.00.00.000

(*позначення*)

Студент

ІПЗ-43___________/Андрій ПРУДЧЕНКО/

(шифр групи) (підпис) (дата)(розшифровка підпису)

Науковий керівник

д.т.н., с.н.с., доц________/Геннадій ПОРЄВ/

(посада) (підпис) (дата) (розшифровка підпису)

Консультант
з питань нормоконтролю
фахівець__________/Тамара ЧАПОВСЬКА/

(посада) (підпис) (дата) (розшифровка підпису)

Допускається до захисту

Завідувач кафедри

д.т.н., проф.___________/Олексій БИЧКОВ/

(посада) (підпис) (дата) (розшифровка підпису)

Київ – 2021

Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій

Кафедра програмних систем і технологій

Освітньо-кваліфікаційний рівень бакалавр

Спеціальність 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖЕНО

Зав.кафедри програмних систем і технологій

______________________(Олексій БИЧКОВ)

(підпис) (прізвище та ініціали)

ЗАВДАННЯ

НА ВИПУСКНУ КВАЛІФІКАЦІЙНУ БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТУ

Прудченку Андрію Олександровичу

1. Тема бакалаврської роботи “Розробка гри в жанрі двовимірного rogue-like платформера”, керівник роботи Порєв Геннадій Володимирович, д.т.н., с.н.с., доцент__________________________ затверджені наказом вищого навчального закладу від “11” листопада 2020 р. № 6.

2. Строк подання студентом роботи 18 лютого 2021 р.

3. Вихідні дані до роботи Базові концепції та розуміння проектування та розробки програмних технологій.

4. Зміст розрахунково - пояснювальної записки (перелік питань, які потрібно розробити)

1. Огляд жанру roguelike.
2. Аналіз існуючих методів реалізації ШІ.
3. Аналіз існуючих методів реалізації процедурної генерації рівнів.
4. Розробка прототипу roguelike гри на основі отриманих результатів.

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

Рис. 1.1 Типова сесія гри Rogue

Рис. 1.2 ADOM в плитковій графіці і ASCII

Рис. 2.1 Простий робочий процес відео гри

Рис. 3.1 Приклади процедурно згенерованих лабіринтів

Рис. 3.2 Приклад печероподібного рівня згенерованого клітинним автоматом

Рис. 3.3 Приклад рівня поділеного на листки

Рис. 3.4 Приклад листків із випадковими кімнатами всередині кожного

Рис. 3.5 Приклад листків із випадковими кімнатами всередині кожного без розділових ліній

Рис. 3.6 Приклад листків заповнених випадковими кімнатами і з’єднаними коридорами

Рис. 3.7 Приклад рівня згенерованого генетичними алгоритмами

Рис. 5.1 Робочий процес для розробленого прототипу гри

Рис. 5.2 Типова сесія гри, розробленої для дипломної роботи

Рис. 5.3 Приклад згенерованого рівня із виділеними кімнатою та найкоротшим шляхом

Рис. 5.4 Відношення часу генерації до розміру рівня для базового ітераційного підходу

Рис. 5.5 Відношення часу генерації до розміру рівня для техніки BSP-дерева

Рис. 5.6 Відношення часу генерації до розміру рівня для техніки клітинних автоматів

Рис. 5.7 Відношення часу обробки до загальної кількості персонажів для евклідової погоні

Рис. 5.8 Відношення часу обробки до загальної кількості персонажів для знаходження шляху за допомогою пошуку в ширину

6. Консультанти розділів роботи

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

Розробка прототипу гри жанру roguelike	Геннадій ПОРЄВ		

7. Дата видачі завдання 13 жовтня 2020 р.

Керівник ___________________________ (Геннадій ПОРЄВ)

Завдання прийняв до виконання _____________(Андрій ПРУДЧЕНКО)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломного проекту (роботи)	Строк виконання етапів проекту (роботи)	Примітка
1	Отримання завдання	13.10.2020	
2	Збір інформації	05.12.2020	
3	Вивчення варіантів реалізації та вибір варіанту для розробки	20.12.2020	
4	Проведення експериментів із різними техніками реалізації	25.12.2020	
5	Розробка прототипу	20.01.2021	
6	Оформлення дипломної роботи	12.05.2021	
7	Отримання допуску до захисту		

Студент – бакалавр ________________ Андрій ПРУДЧЕНКО
(підпис) (розшифровка підпису)

Керівник роботи _________________ Геннадій ПОРЄВ
(підпис) (розшифровка підпису)

АНОТАЦІЯ

Ігри в першу чергу є джерелом розваг, а також основою для розвитку, тестування та доведення теорій. Коли відеоігри почали популяризуватися, більш амбіційні проекти штовхали вперед розробку більш просунутих алгоритмічних методів для обробки графіки в режимі реального часу, динамічних великомасштабних віртуальних світів та інтелектуальних неігрових персонажів.

Мета цієї роботи — розробити гру жанру roguelike, а також проаналізувати та порівняти алгоритмічні прийоми, що використовуються для вирішення проблем пов’язаних з такою розробкою. Точніше, буде зосереджено на двох конструктивних особливостях, перша з яких — процедурна генерація рівнів, а друга — штучний інтелект для ворогів.

Деякі результати, які стосуються формування процедурних рівнів, показують, що методи простої ітерації та BSP-дерева можуть застосовуватися практично до будь-яких розмірів рівня, тоді як клітинні автомати та генерування лабіринту шляхом пошуку в глибину повинні бути оптимізовані або обмежені якимось чином для збільшення масштабованості. Крім того, що стосується штучного інтелекту ворогів, було досягнуто висновку, що і техніки з використанням станів, і без них можуть використовуватися практично для будь-якої кількості одночасно активних персонажів без помітної затримки. Однак було показано, що персонажі на основі алгоритмів із використанням станів можуть бути спроектовані для представлення більш складної, гнучкої та інтелектуальної поведінки. Стосовно алгоритмів пошуку шляхів було продемонстровано, що пошук в ширину є задовільним з точки зору ефективності, тоді як з точки зору продуктивності він працює погано, якщо не застосовуються алгоритми скорочення простору пошуку.

Загальний об’єм роботи : 84 сторінки, 18 рисунків, 3 таблиць, 10 джерел посилань.

Ключові слова: процедурна генерація рівнів, штучний інтелект, розробка ігор, Roguelike.

АННОТАЦИЯ

Игры в первую очередь являются источником развлечений, а также основой для развития, тестирования и доказания теорий. Когда видеоигры начали популяризироваться, более амбициозные проекты продвигали вперед разработку более продвинутых алгоритмических методов для обработки графики в режиме реального времени, динамических крупномасштабных виртуальных миров и интеллектуальных неигровых персонажей.

Цель этой работы — разработать игру жанра roguelike, а также проанализировать и сравнить алгоритмические приемы, используемые для решения проблем, связанных с такой разработкой. Точнее, будет сосредоточено на двух конструктивных особенностях, первая из которых — процедурная генерация уровней, а вторая — искусственный интеллект для врагов.

Некоторые результаты, касающиеся процедурной генерации уровней, показывают, что методы простой итерации и BSP-дерева могут применяться практически для любых размеров уровня, тогда как клеточные автоматы и генерирование лабиринта путем поиска в глубину должны быть оптимизированы или ограничены каким-то образом для увеличения масштабируемости. Кроме того, что касается искусственного интеллекта врагов, было достигнуто вывода, что и техники с использованием состояний, и без них могут использоваться практически для любого количества одновременно активных персонажей без заметной задержки. Однако было показано, что персонажи на основе алгоритмов с использованием состояний могут быть спроектированы для представления более сложного, гибкого и интеллектуального поведения. Относительно алгоритмов поиска путей было продемонстрировано, что поиск в ширину является удовлетворительным с точки зрения эффективности, тогда как с точки зрения производительности он

работает плохо, если не применяются алгоритмы сокращения пространства поиска.

Общий объём работы: 84 страницы, 18 рисунков, 3 таблиц, 10 источников литературы.

Ключевые слова: процедурная генерация уровней, штучный интеллект, разработка игор, Roguelike.

ANNOTATION

Games are mainly a source of entertainment, as well as a basis for developing, testing and proving theories. As video games began to popularize, more ambitious projects motivated the development of more advanced algorithmic methods for processing real-time graphics, dynamic large-scale worlds, and intelligent NPC.

The purpose of this work is to develop a game of the roguelike genre, as well as to analyze and compare algorithmic techniques used to solve problems associated with such development. More precisely, it will focus on two design features, the first of which is the procedural generation of levels, and the second is artificial intelligence for enemies.

Some results regarding the generation of procedural levels show that simple iteration and BSP tree techniques can be applied to virtually any level size, while cellular automata and in-depth maze generation must be optimized or limited in some way to increase scalability. In addition, regarding the artificial intelligence of enemies, it was concluded that techniques with and without states can be used for almost any number of simultaneously active characters without noticeable delay. However, it has been shown that characters based on state-machine can be designed to represent more complex, flexible, and intelligent behavior. Regarding path search algorithms, breadth-first search has been shown to be satisfactory in terms of effectiveness, while in terms of performance it works poorly if search space reduction algorithms are not used.

Total capacity: 84 pages, 18 images, 3 tables, 10 references.

Tags: procedural level generation, AI, game development, roguelike.

ЗМІСТ

Стр.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ 12
ВСТУП 13
РОЗДІЛ 1
ОГЛЯД ЖАНРУ ROGUELIKE
1.1 Визначення жанру roguelike 15
1.2 Історичний контекст 16
РОЗДІЛ 2
ОСОБЛИВОСТІ ДИЗАЙНУ
2.1 Загальні ознаки 20
2.2 Процедурна генерація рівнів 25
2.3 Штучний інтелект 27
РОЗДІЛ 3
ПРОЦЕДУРНА ГЕНЕРАЦІЯ РІВНІВ
3.1 Основна ітерація 30
3.2 Лабіринти 31
3.3 Клітинні автомати 34
3.4 Дерева бінарного розбиття простору 35
3.5 Генетичні алгоритми 38
3.6 Параметризація 40
3.7 Аналіз та порівняння 41
РОЗДІЛ 4
ШТУЧНИЙ ІНТЕЛЕКТ
4.1 Знаходження шляху 47
4.2 Персонажі без станів 50
4.3 Персонажі зі станами 51

4.4 Інтелект натовпу....................52
4.5 Генетичні алгоритми....................54
4.6 Персонажі, які базуються на емоціях....................56
4.7 Створення екземплярів ігрових сутностей....................57
4.8 Аналіз та порівняння....................58
РОЗДІЛ 5
РЕАЛІЗАЦІЯ
5.1 Середовище розробки....................66
5.2 Загальна реалізація....................66
5.3 Процедурна генерація рівня....................69
5.4 Штучний інтелект....................71
5.5 Тестування продуктивності....................73
ВИСНОВКИ....................80
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ....................83

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

RPG – (Role Playing Game) жанр комп'ютерних і настільних ігор.

NPC – (Non-Player Character) будь-який персонаж в грі, яким не може керувати гравець.

AI – (Artificial Intelligence) здатність інженерної системи здобувати, обробляти та застосовувати знання та вміння.

BSP – (Binary Space Partitioning) метод рекурсивного розбиття евклідового простору на опуклі множини за допомогою гіперплощин.

FSM – (Finite-State Machine) абстракції, що використовується для опису шляху зміни стану об'єкта в залежності від поточного стану та інформації отриманої ззовні.

PCG – (Procedural Content Generation) створення наповнення гри за допомогою рандомних параметрів.

RNG – (Random Number Generator) обчислювальний або фізичний пристрій, спроектований для генерації послідовності номерів чи символів, які не відповідають будь-якому шаблону, тобто є випадковими.

A* – евристичний алгоритм пошуку.

ВСТУП

Ігри використовувались протягом історії математики та інформатики для розробки, перевірки та доведення теорій. Багато з них є чудовими прикладами того, як математики і програмісти розробили алгоритми для перемоги над людьми в іграх, одним з яких є комп'ютерна програма Deep Blue, яка виграла шаховий матч проти гросмейстера Гаррі Каспарова. Коли відеоігри почали популяризуватися, більш амбіційні проекти штовхали вперед розробку більш просунутих алгоритмічних методів для обробки графіки в режимі реального часу, динамічних великомасштабних віртуальних світів та інтелектуальних неігрових персонажів (NPCs).

Дане дослідження обертатиметься навколо основи, яка буде описана в розділах 2 і 3, але перед тим, як вдаватися в деталі, важливо охарактеризувати жанр roguelike, а також дізнатися його історичну еволюцію, щоб мати чітке бачення того, де було поставлено цілі.

Мета заснована на тому, що було описано вище, пам'ятаючи також і про контекст області, якій ця робота була присвячена. Ціллю є розробити прототип гри жанру roguelike, а також проаналізувати та порівняти алгоритмічні прийоми, що використовуються для розв'язання супутніх проблем. Серед таких проблем буде зосереджено на двох: процедурній генерації рівнів і штучному інтелекті для ворогів. Більш конкретно буде:

- представлено основні завдання розробки гри жанру roguelike, як з обчислювальної так і дизайнерської точок зору;
- порівняно різні підходи вирішення для обох із наведених проблем;
- розроблено прототип roguelike гри як засіб тестування та підтвердження досліджуваних підходів;
- проаналізовано результати розробки, як із точки зору продуктивності так і грабельності;

- зроблено висновок про найкращу конфігурацію функцій та вказано напрямок можливої майбутньої роботи над грою.

РОЗДІЛ 1
ОГЛЯД ЖАНРУ ROGUELIKE

1.1 Визначення жанру roguelike

Просте визначення терміну Roguelike — сказати, що він характеризується іграми які були прямо чи опосередковано натхнені грою Rogue. Більш конкретно, це піджанр рольових ігор (РПГ), які поділяють певні особливості з архетипом жанру, наприклад, остаточна смерть персонажа гравця випадкові (або процедурно генеровані) підземелля та покрокові рухи [1].

Хоча це загальновизнане вільне визначення, були зроблені різні спроби визначити жанр більш жорстко. Один з найбільш відомих називається "Берлінська інтерпретація", яка була створена в Roguelike Development Conference 2008 [2]. Учасниками було обговорено кілька ігрових особливостей, і зважувались які фактори визначити як важливі для жанру. Серед факторів високої цінності зазначено:

- **Випадкове генерування середовища**. Світ формується процедурно, і, швидше за все, гравець ніколи не побачить однакового рівня підземелля двічі;
- **Перманентна смерть**. Коли персонаж гравця помирає, ним більше не можна грати;
- **Покроковість**. Гра не працює в режимі реального часу. Тобто вона змінюється лише тоді, коли користувач діє якимось чином;
- **Побудована на сітці**. Світ представлений однорідною сіткою плиток;
- **Нелінійність**. Гра є досить продуманою, тому існує кілька варіантів вирішення проблем. Це досягається шляхом декількох різних взаємодій між предметами та монстрами;

- **Управління ресурсами**. Гравець має обмежені ресурси, і він повинен придумати стратегії управління ними якомога кращим чином для просування в грі;
- **Hack'n'Slash**. Боротьба з великою кількістю монстрів є важливою частиною гри;
- **Дослідження**. Гравець повинен ретельно і сплановано досліджувати нові рівні для кожної нової гри.

Хоча можна інтуїтивно поглянути на гру і перевірити, відповідає вона лиш деяким чи більшій частині вищевказаних критеріїв, важко встановити конкретне визначення жанру.

Зрештою, замість того, щоб сказати, які ігри належать до цього жанру, ліпше визначати, наскільки гра є «Rogue-like». Це означає, що чим більше рис жанру вона має, тим більше вважається roguelike. Один із способів краще висвітлити такі особливості — показати, які ігри були історично пов’язані з жанром, і саме це буде продемонстровано нижче.

1.2 Історичний контекст

Перший крок до розуміння Roguelikes має розпочатися з розуміння рольових відеоігор в цілому. Комп’ютерні рольові відеоігри(CRPG) — це ігри, в яких гравець керує одним персонажем (або партією), зануреним у чітко визначений світ через комп’ютер. Вони, в свою чергу, були натхненні так званими настільними (або «pen and paper») рольовими іграми. Там гравці створювали персонажів і приймали їх ролі, керуючись набором правил та регулюючись спеціально призначеним гравцем, який називається ігровим майстром [3]. Вони грались офф-лайн, коли гравці знаходились за одним столом. По суті, вони надихнули перші CRPG, які були розроблені для мейнфреймів у середині 1970-х років. Dnd є одним із перших прикладів ігор натхненних настільними РПГ, таких як D&D (Підземелля та Дракони) Гері

Гігакса. Інші приклади ігор, які допомогли визначити жанр CRPG, включають Dungeon і pedit5, а пізніше серія Ultima [4].

З цієї основи вийшов канон для жанру, яким є Rogue. Натхненний вищезгаданими dungeoncrawlerами, він представляв особливий набір властивостей, що спонукав інших розробники створити для нього ряд варіацій, що зробило його архетипом для жанру roguelike. У кожній грі гравець починав «з нуля» в невідомому підземеллі, і мусив битися з навалами монстрів зростаючої складності. Для цього йому доводилося підбирати обладунки для його завдання, а також піднімати рівень свого персонажа, щоб стати могутнішим. Якщо його вбивали, єдине, що зберігалося, — це запис про найбільшу кількість набраних очок, що показувала його прогрес до цього моменту [5]. У 1980-х рр. в деяких університетах Rogue вважався дуже захоплюючою грою, особливо тому, що поєднання перманентної смерті та випадково згенерованих рівнів призвело до високого рівня реграбельності.

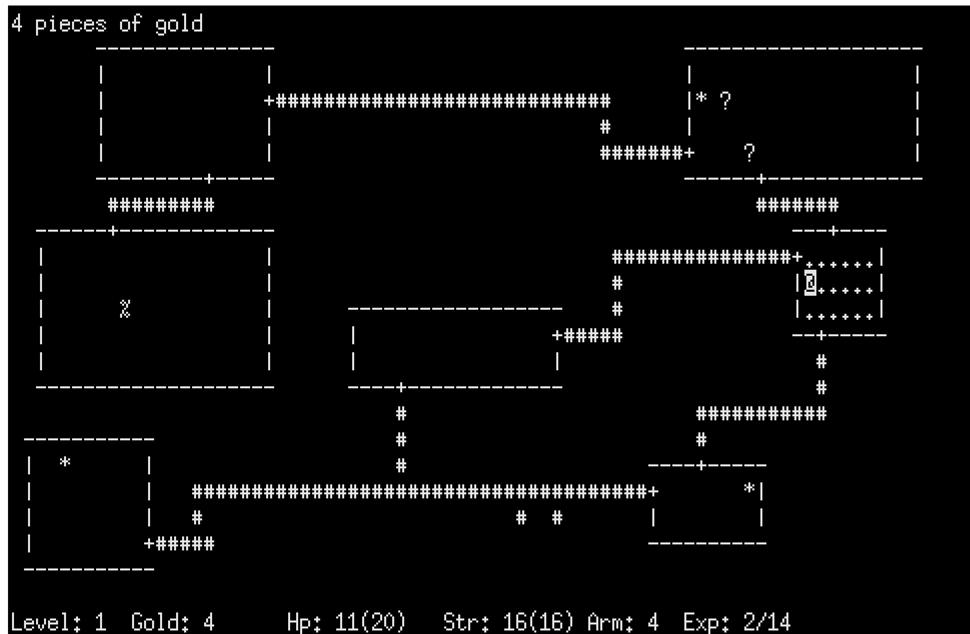

Рис. 1.1 Типова сесія гри Rogue

Двоє важливих нащадків Rogue — Hack і Moria — були спробою їхніх творців якимось чином поширити Rogue. Вони, в свою чергу, надихнули своїх, більш амбіційних, нащадків. Angband, найважливіший варіант Морії, мав 100 рівнів підземелля, деякі з них із унікальними особливостями, що називаються

сховищами, де гравець стикався з важчими ворогами, але нагороджувався скарбами. Angband також додав ряд унікальних предметів, які називаються артефактами, які були неоціненні для перемоги над великою кількістю унікальних монстрів, також доданих до гри.

NetHack, наступник Hack, об'єднав різні незалежні галузі розвитку свого попередника як засіб для поєднання внесених ним доповнень. Гра була перенесена на інші операційні системи і включала більше класів гравців, монстрів та рівнів підземель, а також більше унікальних предметів та монстрів.

Не так давно і NetHack, і Angband почали підтримувати використання графіки на основі плитки на додачу до традиційного символьного стилю ASCII.

У середині 90-х років багато Roguelikes були розроблені натхненними NetHack та Angband. Одна із них, Crawl, випущена в 1995 році Лінлі Хенцеллом. Вона слідувала принципу обмежених ресурсів із NetHack, що означало, що гравець не міг залишатися на тому самому рівні під час набуття досвіду. Його сюжет був також простим: спуститись у єдине підземелля, витягти артефакт внизу, а потім знову піднятися на поверхню.

Гра античні домени таємниць (ADOM), випущена в 1994 році, з іншого боку мала поєднання стійких та не стійких рівнів підземелля, що залишало деякий простір для так званого "гріндУ", тобто, залишаючись на тому самому місці, перемагати монстрів і отримувати досвід. Крім того, він містив набагато докладніший сюжет, ряд побічних квестів та потойбічний світ, щоб зв’язати їх. У 2012 році ADOM провів краудфандингову кампанію «ADOM. Воскресіння». В результаті, в грі з’явилося кілька доповнень, одне з яких — інтеграція з NotEye, що дозволило надати плиткову підтримку для гри [6]. На рисунку 1.2 показано знімок екрана ADOM як у ASCII (праворуч), так і в плитковій графіці.

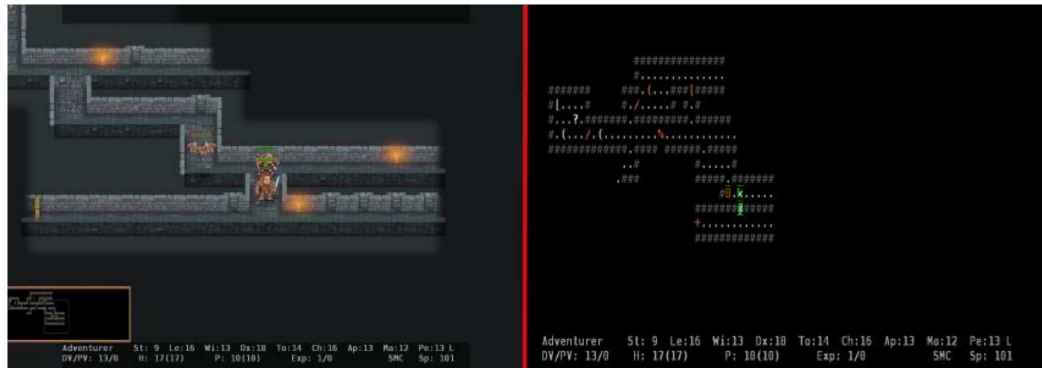

Рис. 1.2 ADOM в плитковій графіці і ASCII

Нарешті, варто згадати один із найвизначніших прикладів roguelike: Dwarf Fortress. У ньому можна грати як мандрівник і досліджувати величезний, випадково створений світ, або це може бути стимулятор побудови та управління цивілізації гномів. Підсумовуючи складність цього, у грі випадковим чином генеруються цілі світи, починаючи з цивілізацій, війон, міст, і закінчуючи простими NPC. Що стосується механіки бою, вона імітує окремі частини тіла (включаючи внутрішні органів) та декількох видів пошкоджень, від яких вони можуть постраждати, такі як поріз, опік, гниття та замороження і так далі [7].

З усього вищеописаного можна побачити, що розвиток жанру переплітається з розвитком сучасних комп'ютерів загалом, починаючи з мейнфреймів на яких запускались прості *dungeoncrawler*и, минаючи через ігри, все ще прості в графіці, але вже з великою глибиною вмісту, і доходячи до повноцінних комерційних ігор, з чудовою графікою та добре розробленою сюжетною лінією.

РОЗДІЛ 2
ОСОБЛИВОСТІ ДИЗАЙНУ

При розробці гри, в даному випадку roguelike, є кілька особливостей дизайну, на які слід звернути увагу. Для деяких з них існують прості (або складні, але стійкі до відмов) рішення, тоді як інші представляють труднощі як з точки зору складності обчислень, так і сприйнятливої грабельності. У наступному розділі буде перелічено деякі аспекти, пов'язані з розробкою прототипу гри. Після цього буде детальніше рознлянуто дві обрані конструктивні особливості для дипломної роботи.

2.1 Загальні ознаки

Нижче наведено деякі основні особливості, на які слід звертати увагу в будь-якому жанрі ігор, але з різним рівнем акценту. Наприклад, хоча CRPG значною мірою засновані на розвитку персонажів, пригодницькі платформери менше зосереджені на цьому, але значно зосереджені на користувацькому інтерфейсі. Таким чином, можна спробувати визначити особливості з точки зору обраного жанру.

2.1.1 Робочий процес гри

Одним із перших рішень, які дизайнер повинен прийняти стосовно гри, є робочий процес, тобто як будуватимуться різні ігрові події (або екрани) та як вони будуть співвідноситися між собою. Наприклад, гра зазвичай починається з головного меню, і звідти можна вибрати різні опції, наприклад, грати в нову гру, завантажити збережену гру, змінити параметри та вийти з гри. Зазвичай

робочий процес гри представлений кінцевим автоматом (FSM), де події є станами, а переходи відбуваються на основі вводу користувача. Можуть бути складені діаграми різних рівнів абстракції для представлення ланцюга подій, що відбуваються в грі.

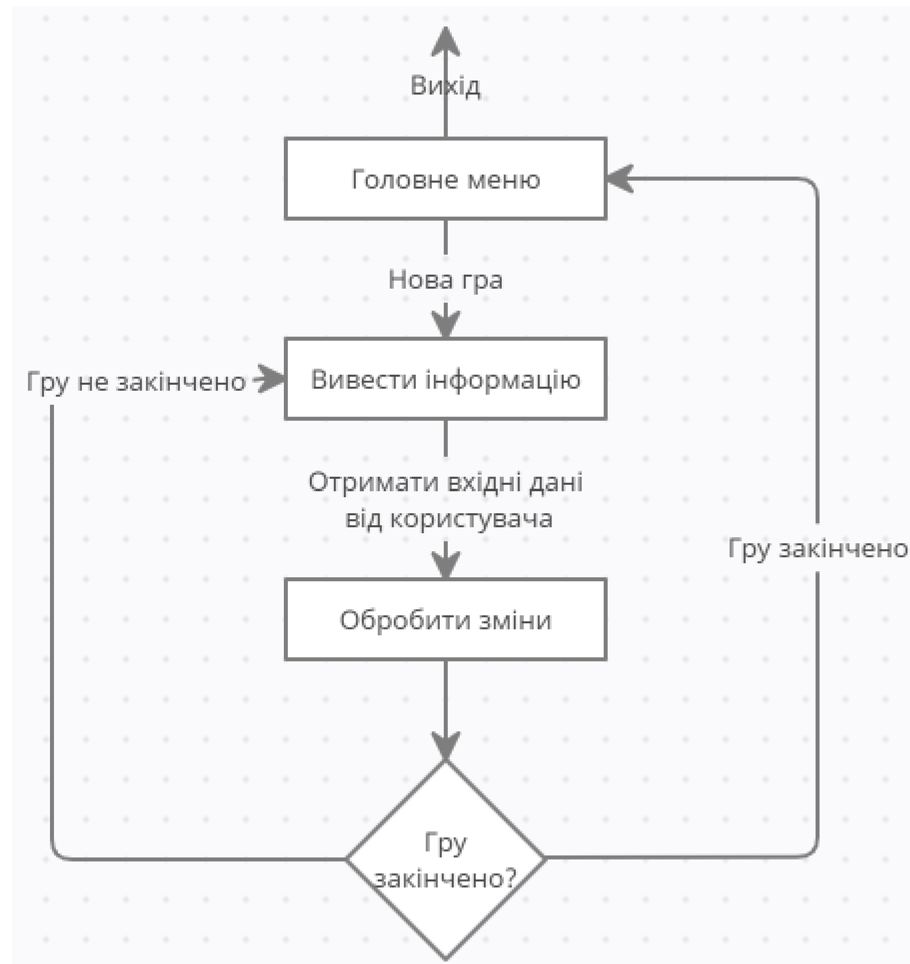

Рис. 2.1 Простий робочий процес відео гри

Одним із важливих аспектів, про який слід подумати, є темп гри, тобто те, як будуть відбуватися події в грі. Більшість ігор потрапляють або в покрокові категорії, або в режими реального часу, хоча деякі ігри можуть містити поєднання двох режимів. У покроковому темпі гра буде чекати ввід користувача і відповідно змінювати стан гри, тоді як у реальному часі гра буде прогресувати навіть тоді, коли гравець не виконує жодних дій. У випадку roguelike, більшість із них є покроковими, але деякі все ж використовують режим реального часу,

наприклад, варіація Angbanda Mangband, Diablo, який поділяє більшість рис roguelike, і тому часто зараховується в цей жанр, та Pyromancer.

Зі збільшенням продуманості гри ускладнюватиметься і робочий процес. Це означає, що перед початком розробки гри потрібно ретельно спланувати її особливості, режими та взаємодії між об’єктами. Гра, розроблена для цієї дипломної роботи, націлена на простий робочий процес, завдяки чому обрані функції можна буде протестувати без зайвих ускладнень. Цьому сприятиме і той факт, що обидві функції є модульними, тобто їхня логіка здебільшого не залежить від решти гри, тому теоретично вони можуть застосовуватися (з деякими можливими модифікаціями) до будь-якої гри, яка слідуватиме приблизно тим самим наборам правил.

2.1.2 Графіка

Що стосується відображення інформації про гру користувачеві, можуть використовуватися різні рівні графіки — від чистого тексту та ASCII графіки до повноцінних 3D-ігор. Хоча roguelike історично представлені і зазвичай асоціюються із ASCII сиволами. Причиною цього є те, що більшість розробників roguelike ігор вирішують зосередитись на ігровому контенті, а не на графіці, оскільки розробка гри зі складною графікою вимагає ресурсів. Крім того, важливо зазначити, що графіка сама по собі не робить гру приємною для користувача. Для цього при розробці гри також слід враховувати спосіб введення команд користувачем.

Гра, розроблена для цієї роботи, використовує плиткову графіку. В цілому, ASCII символи вважаються доволі архаїчним способом для представлення ігор, і на сьогоднішній день більшість roguelike ігор використовують саме плиткову графіку.

2.1.3 Інтерфейс користувача

Як було зазначено раніше, добре намальована графіка не гарантує позитивних емоцій гравця стосовно ігрового інтерфейсу. Дизайнер гри також повинен спланувати, як інформація відображатиметься гравцеві, а також як ігрові дії будуть контролюватися пристроями введеня. Наприклад, більшість roguelike ігор фокусують майже всі команди на введення з клавіатури, при цьому деякі незначні функції приписуються миші (наприклад, відображення інформації при наведенні). Оскільки roguelike схильні надавати пріоритет продуманості перед графічним дизайном, вивчення прив'язок клавіш (які клавіші що роблять) є важливою складовою для оволодіння певною грою. Такі ігри, як ToME, намагаються краще використовувати мишу, щоб забезпечити більш інтуїтивне введення команд користувачами.

В розробленій грі було дотримано традиційні прив’язки клавіш, які мають скоріше класичні RPG, аніж roguelike. Це означає, шо хоч переважна більшість команд буде прив’язана до клавіатури, миша також гратиме важливу роль в управлінні. Незважаючи на це, оскільки гра має відносно прості взаємодії, користувачу не складе труднощів запам’ятати їх.

2.1.4 Сюжетна лінія та квести

Хоча сюжетна лінія та квести зазвичай не є пріоритетними для жанру (особливо для старих ігор), зазвичай існує головна мета. У таких іграх, як Rogue, Moria і Nethack, та їх нащадків Angband, Crawl, ADOM та інших, мета полягає в тому, щоб спуститися на найглибший рівень підземелля, щоб або перемогти фінального боса, або отримати важливий артефакт. Окрім цього, більшість з них також мають побічні квести. Сюди входять перемога над проміжними босами, отримання артефактів, що полегшують або навіть є обов’язковими для просування в грі, а деякі навіть включають головоломки в гру (наприклад, головоломка Sokoban від NetHack та лабіринт ADOM).

Більшість із цих квестів та сюжетних ліній попередньо розробляються ігровим дизайнером, хоча є дослідження щодо їх процедурного генерування. Є ігри з процедурними квестами і сюжетними лініями, наприклад, генерація світу в Dwarf Fortress, якf створює цілий світ та імітує його історію, аж до біографій та стосунків окремих істот (людей та відомих монстрів).

В цій роботі було вирішено зробити ігрові цілі простими, щоб зосередитись на інших особливостях дизайну. Однак у розділі 6 буде вказано, як можна розширити гру в цьому напрямку.

2.1.5 Темп гри

Прогресування гри в часі стосується того, як гравець сприймає зміни в грі з плином часу, виходячи з того, як вона відображає інформацію на екрані та як обробляється введення користувачем. Ігри, як правило, або покрокові, або в режимі реального часу, як зазначено раніше. У покроковій грі інформація відображатиметься гравцеві, і гра не буде прогресувати, поки гравець не робить певні дії. З іншого боку, в реальному часі події будуть продовжувати відбуватися, навіть якщо гравець не робить жодних дій.

Це означає, з точки зору дизайну, що покрокова прогресія гри передбачатиме більш послідовний робочий процес, тоді як при прогресуванні в режимі реального часу відсутність дій з боку користувача не буде блокувати ігрові події, отже, дизайн гри доведеться адаптувати, інакше можуть виникнути проблеми з грабельністю. Наприклад, якщо немає ходів, чи будуть монстри продовжувати атакувати гравця навіть якщо він не активний? Або що станеться, якщо гравець і монстр спробують переміститися одночасно в одне місце?

Ці проблеми можна вирішити, додавши атрибут швидкості для кожного персонажа, і зробивши кожну дію вартою певній кількості учовних одиниць часу. Крім того, потрібно керувати порядком дій, виходячи з їх тривалості. Класичним рішенням для цього є динамічна черга. У розроблюваній грі

використовується прогресія в реальному часі, тому всі такі підводні камені потрібно було взяти до уваги.

2.2 Процедурна генерація рівнів

Процедурна (або випадкова) генерація рівнів (зазвичай підземель), перша із вибраних дизайнерських особливостей для фокусування, є частиною більшої концепції, яка називається Процедурне Створення Наповнення (Procedural Content Generation or PCG). PCG займається процедурною генерацією вмісту для ігор. Це включає рівні підземелля, квести та сюжети, монстрів та навіть музику та графіку, серед іншого. Вирішено сконцентруватися на цьому, оскільки:

- це впізнавана особливість roguelikes;
- з часом для цього було розроблено велику кількість технік;
- деякі з цих методів можуть дати хороші результати, хоча і погано масштабуються.

Тестування цих методів різного рівня складності може дати змогу зробити корисні висновки. Для початку, PCG можна класифікувати різними способами. Існує сім вільних типів процедурної генерації [8], а саме:

- **Генерація випадкових рівнів під час виконання**. Створення рівнів підземелля за допомогою рандомізованих алгоритмів під час гри, як правило, коли гравець змінює рівні (у випадку roguelikes, як правило, коли він спускається сходами).
- **Розробка вмісту рівня**. Використовується, коли вищезазначений метод не може стабільно давати задовільні результати. У цьому випадку карти генеруються випадковим чином в інструменті генерації рівнів, на відміну

від часу виконання, а потім перевіряються на коректність дизайнером рівнів.

- **Динамічне генерація світу**. Ця техніка використовує випадкове зерно для ітеративного збільшення ігрового поля шляхом перестановки за допомогою методів генерації псевдовипадкових чисел.
- **Створення екземплярів ігрових сутностей**. Полягає у рандомізації параметрів ігрових сутностей (наприклад, монстрів), щоб можна було створити великі сукупності унікальних сутностей з незначним шансом на повторення.
- **Опосередковане користувачем наповнення**. Цей метод використовує PCG, щоб створити цілий ряд можливостей, які за бажанням можуть бути вибрані та налаштовані користувачем.
- **Динамічні системи**. Відноситься до моделювання таких систем, як погода або поведінка натовпу, за допомогою методів PCG. За допомогою цього типу генерації створюються (статистично) неповторювані ситуації в грі, що збільшує реграбульність.
- **Процедурні головоломки та створення сюжету**. У цій категорії використовуються процедурні прийоми, щоб зробити історії та головоломки гри більш непередбачуваними. Наприклад, до графа залежностей квестів можна додати рандомізацію так, що перегляд матеріалів по проходженню буде менше ламати гру.

Серед цих типів буде використано генерацію випадкових рівнів під час виконання, яка займається створенням підземель під час гри. Крім того, будуть проведені експерименти із створенням екземплярів ігрових сутностей у Розділі 4. Цей тип PCG пов’язаний із рандомізацією параметрів для генерації наповнення, так щоб з’явилися унікальні.

2.2 Штучний інтелект

Штучний інтелект в іграх відноситься до роботи з персонажами, які не контролюються гравцем. Що стосується монстрів, це зазвичай означає планування способів атакувати (і перемагати) гравця. У більшості roguelike ігор класичний спосіб досягти цього — обчислити найкоротший шлях від монстра до персонажа гравця, рухатись у його напрямку та атакувати, якщо досить близько. Цей тип поведінки може бути змодельований персонажами без використання станів, які в будь-який момент часу перевірятимуть набір умов і діятимуть на їх основі, не враховуючи жодної внутрішньої інформації, яку вони могли отримати за ігровий сеанс. Проблема пошуку найкоротшого (або найцікавішого) шляху відома як path-finding.

Однак, якщо ігровий дизайнер має намір надати гравцю більш цікавий досвід, можуть бути використані різні прийоми для збагачення NPC:

- **Персонажі без станів.** Найпростіша форма персонажів, вони складаються з ряду if-умов (як правило, формують дерево умов), які перевіряються на кожному кроці. Залежно від результату перевірок, персонаж реагуватиме відповідно.
- **Персонажі із використанням станів**. Створивши внутрішні стани для акторів, можна моделювати більш складну поведінку, оскільки вони можуть стати менш «рефлексивними» і більш «когнітивними» сутностями.
- **Колективний інтелект**. Займається використанням децентралізованої колективної поведінки для додавання колективного інтелекту персонажам за рахунок відносно низьких обчислювальних витрат.
- **Рандомізація параметрів**. Раніше згадана як «створення екземплярів ігрових сутностей», вона може використовуватися в контексті штучного

інтелекту в поєднанні з іншими заснованими на параметрах методами для створення великої кількості унікальних персонажів.

- **Генетичні алгоритми**. Можуть використовуватися для еволюції процесу прийняття рішень певного виду монстрів шляхом випадкових мутацій.
- **Персонажі, які базуються на емоціях**. Складається з моделювання емоцій людини (або тварин) у набір рис та додавання їх персонажам, таким чином визначаючи їх «особистість», так щоб їхні характери визначали (або принаймні впливали) на їхні дії.

Вищезазначені категорії можна використовувати не тільки самі по собі, а й у поєднанні між собою для створення унікальної поведінки NPC для гри, а також для створення рівнів складності та реграбельності за бажання дизайнера. У розділі 5 буде представлено набір методів штучного інтелекту, реалізованих для розробленої гри, та їх практичні результати.

РОЗДІЛ 3

ПРОЦЕДУРНА ГЕНЕРАЦІЯ РІВНІВ

У цьому розділі буде сконцентровано на проблемі процедурного формування рівн. Як зазначалося в попередньому розділі, буде зосереджено на генерації підземелля під час виконання, тобто техніки будуть застосовуватися під час гри. Якщо для обчислення результатів потрібен помітний час, може знадобитися екран завантаження (який буде повідомляти користувача про необхідність обробки перед початком гри). В іншому випадку гра просто представить новий рівень гравцеві, як тільки він досягне перехідної точки рівня (наприклад, сходи або портали).

Для цього можна використати декілька прийомів, і всі вони якимось чином використовують випадковість. Для цього потрібно застосовувати техніку, яка називається генератором випадкових чисел (RNG). Хоча це, як правило, неявно, ігри насправді використовують методи генерування псевдовипадкових чисел (PRNG), які мають ряд переваг перед справжніми випадковими методами для подібних цілей. Два з них:

- **Генерація не обмежується**. Хоча справжні (або природні) джерела випадкових чисел мають обмеження пропускної здатності (через їх обмежену ентропію з часом), PRNG мають теоретично необмежені можливості (вони повинні бути побудовані таким чином, щоб забезпечити достатню кількість послідовностей без повторення для конкретного застосування).
- **Передбачуваність**. Для цілей розробки, тестування та виправлення багів факт що конкретне зерно (початкове значення, з якого витягуються випадкові числа) завжди дає однакову послідовність чисел є позитвним

оскільки справжні випадкові джерела будуть генерувати непередбачувані результати, що робить аналіз важче.

В наступних розділах буде представлено декілька методів рандомної генерації підземель. Вони різняться за складністю обчислень та якістю (хоча це і суб’єктивний параметр оцінки). Через це в розділі 3.7 вони будуть проаналізовані та порівняні у цьому відношенні.

3.1 Основна ітерація

Основна техніка, яка в тій чи іншій формі присутня в більшості класичних roguelike іграх, полягає у створенні ряду випадково розміщених прямокутників, які представлятимуть кімнати і з’єднанні їх коридорами. Щоб гравець міг дістатись до будь-якої кімнати з будь-якої стартової точки, потрібно перевірити з’єднанність кімнат. Простий спосіб гарантувати це: завжди з’єднувати новостворену кімнату з попередньою (після першої), таким чином переконуючись, що щоразу після додавання нової кімнати всі кімнати лишатимуться доступними між собою. Одна з характеристик цього підходу: з’єднання коридорів не враховуватиме створені раніше приміщення, тому деякі коридори можуть перетинати інші приміщення та коридори. Крім того, оскільки нові кімнати завжди підключені до останньої створеної, рівень може здаватися єдиним шляхом послідовних кімнат. І те, і інше може бути не бажаним ефектом для дизайнера.

Одним із способів уникнути цих проблем є моделювання рівня у вигляді графа, де вершини представляють кімнати, а ребра — коридори. Після цього можна зробити наступне:

- Почати з будь-якої кімнати, підключити її до іншої кімнати (на основі деяких критеріїв, наприклад, найближчої або навіть випадково обраної кімнати), а потім виконати послідовний пошук, використовуючи алгоритми, такі як пошук в глибину, пошук в ширину або Дейкстри, поки кожна кімната не буде підключена до інших.
- З'єднати кожну кімнату з іншими за допомогою коридорів, а потім знайти мінімальне дерево, що охоплює цей граф. Після цього просто обрізати непотрібні коридори. Одним із цікавих підходів до цієї проблеми є зробити центральну координату кімнат вершинами графіка, а потім застосувати до неї триангуляцію Делоне. Таким чином, отримані ребра представлятимуть непересічні коридори.

Після того, як можна буде гарантувати, що кімнати так чи інакше повністю пов'язані, можна додати додаткові коридори для більшої нелінійності. Наприклад, для збільшення кількості шляхів в рівні можна додати ряд коридорів виходячи із кількості кімнат. На додаток до цього можуть бути додані коридори, які нікуди не ведуть, що характеризує тупики.

3.2 Лабіринти

На відміну від класичного підходу "кімнати, з'єднані коридорами", лабіринти складаються здебільшого (якщо не тільки) з довгих звивистих коридорів, з яких гравець повинен вийти, щоб просунутися в грі. Деякі з підходів до створення лабіринтів включають:

- **Пошук в глибину**. Моделюючи простір пошуку як двовимірну сітку з квадратами в якості вершин та переходами між сусідами в якості ребер, а також рандомізуючи вибір відвідування сусідів, можна створити лабіринт. В залежності від бажаного результату, деяким варіантам відвідування

сусідів можна поставити більшу вагу. Так, наприклад, якщо горизонтальному відвідуванню віддано перевагу над вертикальним, алгоритм створить більше довгих горизонтальних коридорів.

- **Алгоритм Крускала**. Спочатку створюється список усіх стін і набір для кожного квадрата сітки, що при ініціалізації містить лише себе. Після цього, для кожної стіни (вибраної випадковим чином): якщо квадрати, розділені цією стіною, належать до різних наборів, тоді стіна знімається, а вихідні набори з'єднуються разом.
- **Алгоритм Пріма**. Будучи алгоритмом мінімального охоплюючого дерева, звичайний Прім дасть результати, подібні до тих, що були у Крускала. Однак замість того, щоб вести для нього список ребер, зберігається список сусідніх квадратів сітки. Роблячи це, і випадковим чином вибираючи сусідні квадратні сітки для відвідування квадратів з кількома сусідами, алгоритм матиме тенденцію до розгалуження більше в порівнянні зі звичайним підходом.
- **Клітинні автомати**. Їх також можна використовувати для генерації лабіринту. Зокрема, для цієї мети широко використовуються два набори правил для «Гри життя» Конвея, а саме Maze і Mazectric. Рядки правил для них, B3/S12345 та B3/S1234, означають, що мертва клітина оживе, якщо у неї є три живих сусіди, а жива клітина буде продовжувати жити, якщо має від 1 до 4 (або 5 , у випадку з Mazectric) живих сусіди. Інакше клітина помирає. Якщо ці правила будуть засновані на випадковій стартовій схемі (яку можна вважати зерном), призведуть до складних лабіринтних структур. Незважаючи на те, що створені шаблони є більш складними, ніж попередні підходи, він має деякі недоліки, найважливішим з яких є те, що зв'язок між двома точками не гарантований. Щоб це вирішити, потрібно використовувати якийсь обхідний шлях, наприклад, випадково розмістити сходи вгору та вниз, а

потім запустити алгоритм пошуку, щоб переконатися, що сходи з'єднані між собою. Інший варіант полягає в тому, щоб створити шлях, незалежний від лабіринту автомата, а потім переписати коридори, які до нього ведуть, таким чином гарантуючи принаймні один шлях між сходами. Іншою можливістю було б дозволити гравцеві копати стіни, таким чином, не турбуючись про взаємоз'єднання входів і виходів.

На рисунку 3.1 показані приклади лабіринтів, створених за допомогою вищезазначених методів. У верхньому лівому куті зображено лабіринт, породжений алгоритмом Крускала, у верхньому правому куті — модифікованим Прімом, у нижньому лівому — за правилом клітинного автомата "Maze" і внизу праворуч за правилом клітинного автомата "Mazectric".

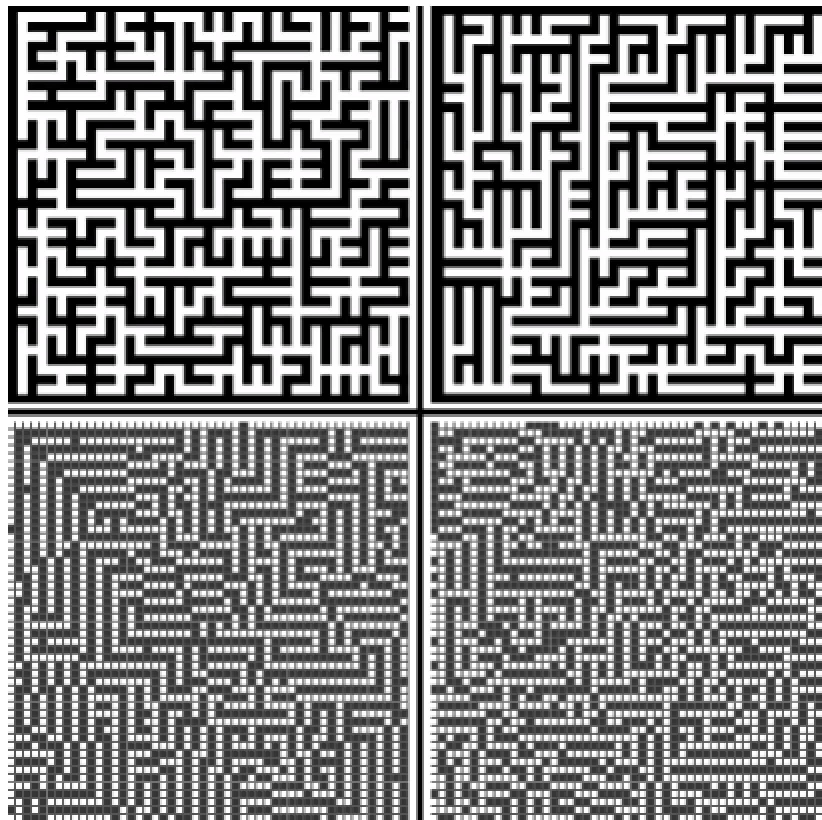

Рис. 3.1 Приклади процедурно згенерованих лабіринтів

3.3 Клітинні автомати

Окрім вищезазначеного застосування в лабіринтах (за правилами Maze та Mazectric), клітинні автомати можна використовувати для створення природних, схожих на печери рівнів. Для цього спочатку дизайнер повинен знайти відповідний набір правил, обираючи один із раніше перевірених, або створити власний. Загальний набір правил, що використовується для цього, відомий як правило «4-5», яке говорить, у випадку підземель, що клітина підлоги «народиться», якщо навколо неї більше п'яти таких клітин, і вона не «помре», якщо навколо неї чотири і більше клітини підлоги. Результатом цього є те, що клітини підлоги, як правило, залишаються в органічних структурах, а дрібнозернисті скупчення, як правило, зникають після певної кількості ітерацій моделювання.

Після вибору відповідного набору правил дизайнер повинен хаотично заповнити простір рівня клітинами. Потрібно знайти адекватне співвідношення стін/підлоги експериментально, але виявлено, що 40-50% підлог у просторі дає найкращі результати. Далі під час моделювання потрібно зробити певну кількість ітерацій, щоб хаос перетворився на бажану печерну структуру. Знову ж таки, потрібно буде експериментувати з кількістю кроків для досягнення найкращих результатів, але було виявлено, що через 3-5 кроків більшість «артефактів» зникають.

Нарешті, останнім кроком у цьому процесі є перевірка зв’язаності, оскільки в цьому процесі не рідко трапляються поодинокі ділянки. Одним із способів впоратися з цим є використання алгоритм «заливки потопом», який полягає в тому, щоб, починаючи з обраної точки (якими будуть початок рівня), рекурсивно “забарвлювати” 2D-квадратну сітку, поки вона не буде обмежена стінами. Зрештою, якщо всі місця в підземеллі, де може пройти персонаж

гравця, будуть кольоровими, це означає, що воно повністю зв’язане. Якщо будуть не з’єднані ділянки, потрібно згенерувати інший рівень, зв’язати його ділянки якимось чином, утриматися від додавання чого-небудь на недосяжні ділянки або дати гравцю можливість руйнувати стіни. На рисунку 3.2 показано приклад вищезазначеного процесу.

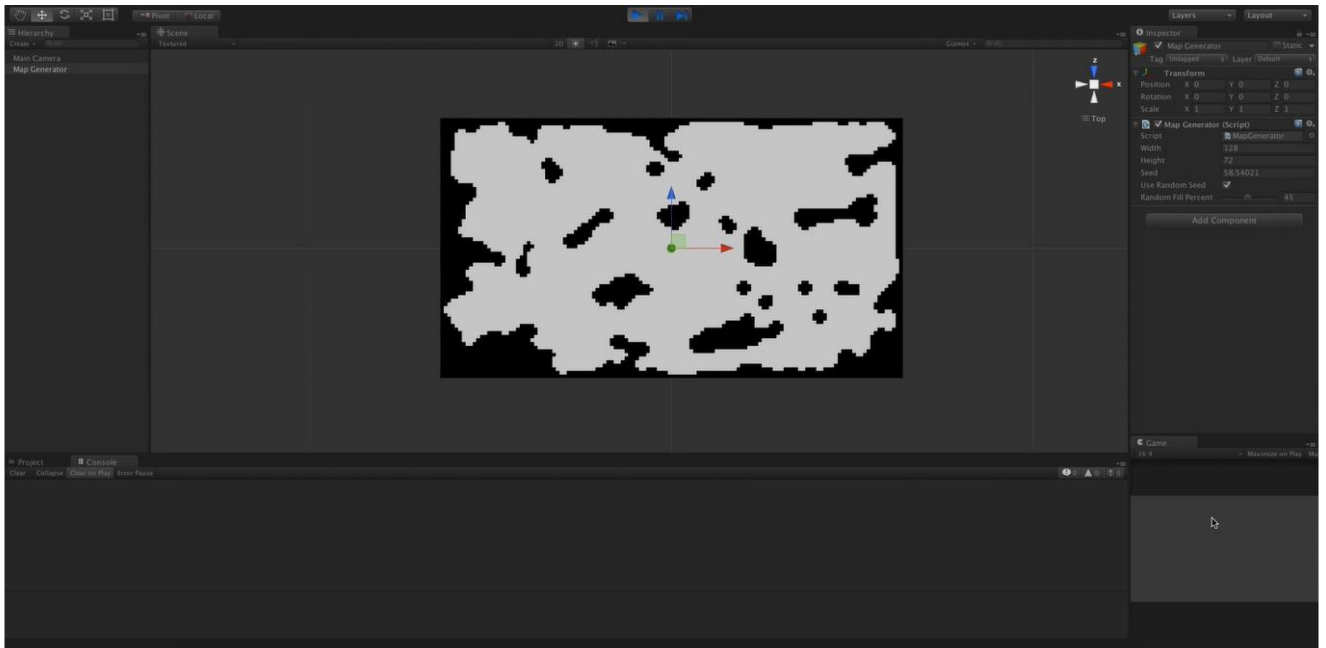

Рис. 3.2 Приклад печероподібного рівня згенерованого клітинним автоматом

3.4 Дерева бінарного розбиття простору

Метод бінарного розбиття простору (BSP) також може бути застосований до генерації рівнів. Він використовується для створення рекурсивних поділи заданого простору за допомогою гіперплощин. У випадку двовимірної гри на основі сітки, прямокутник з розмірами рівня рекурсивно ділиться, горизонтально або вертикально, на два менші прямокутники довільного розміру, в результаті чого утворюється структура, яка називається BSP-деревом. Рівень можна створити за допомогою цього методу, використовуючи таку процедуру:

1. Почати із прямокутного підземелля, заповненого стінами.
2. Випадково вибрати напрямок поділу, горизонтальний чи вертикальний, та відповідну координату поділу.
3. Розділити прямокутник рівня на два підрівня.
4. Повторити кроки 2 і 3 для кожного отриманого підрівня встановлену кількість разів, або допоки подальший поділ буде неможливим. Критерієм зупинки можна задати мінімальний розмір прямокутника.

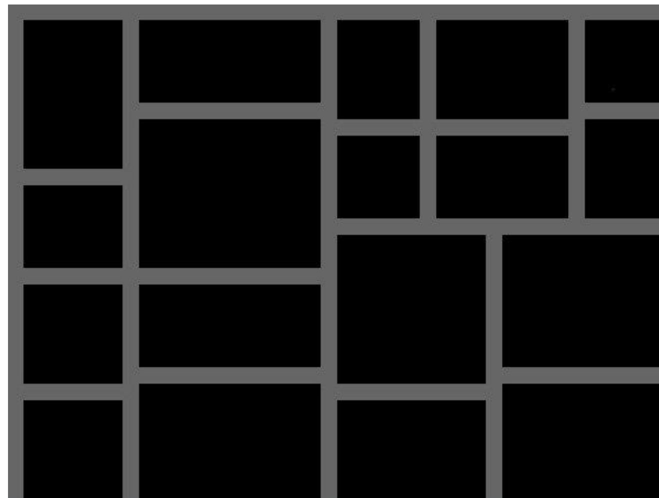

Рис. 3.3 Приклад рівня поділеного на листки

5. Для кожного листка створити у ньому кімнату, розмір якої варіюється від мінімального розміру кімнати до розміру цього підрівня.

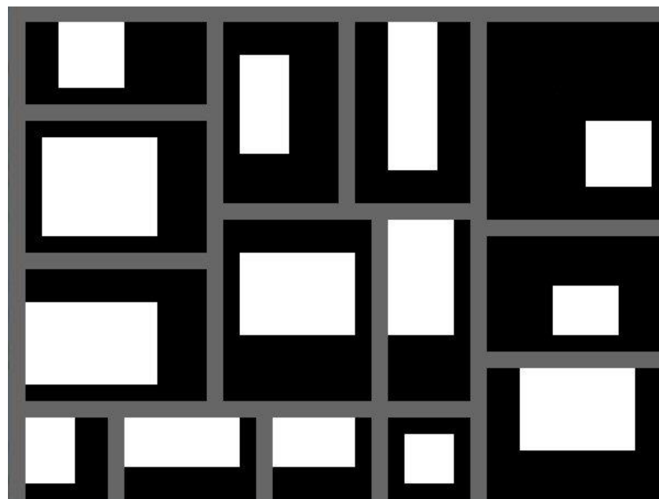

Рис. 3.4 Приклад листків із випадковими кімнатами всередині кожного

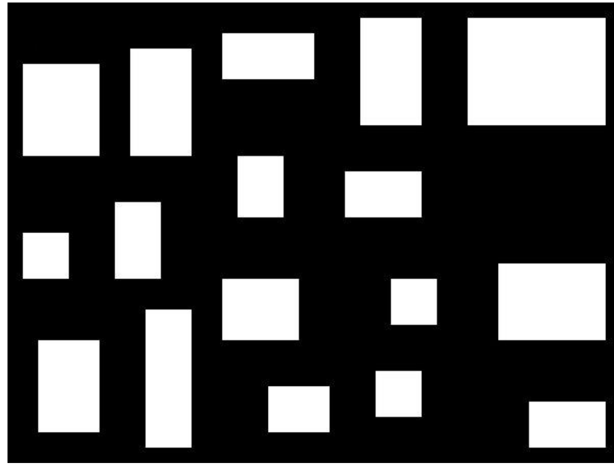

Рис. 3.5 Приклад листків із випадковими кімнатами всередині кожного без розділових ліній

6. Після створення всіх кімнат з'єднати всі листки дерева (підрівні після останнього поділу) з їхніми «сестрами».
7. Піднятися на один рівень у BSP-дерева і з'єднати всі субрегіони з їхніми сестрами, так само, як це було зроблено на кроці 6, таким чином під'єднавши кімнату в субрегіоні до кімнати його сестри або навіть до коридорів.
8. Повторювати крок 7, допоки кожен рівень не буде під'єднаним до своєї сестри.

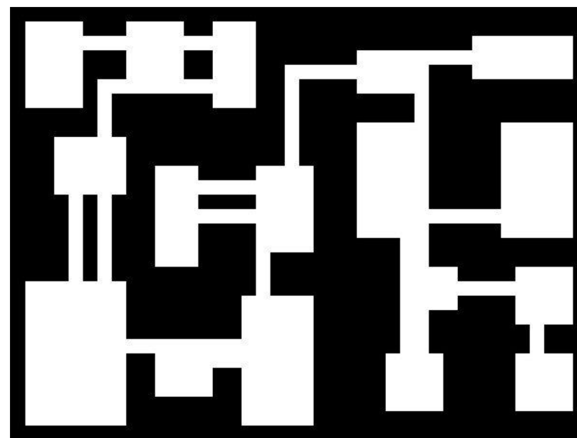

Рис. 3.6 Приклад листків заповнених випадковими кімнатами і з'єднаними коридорами

Однією з прямих переваг цього процесу є те, що завдяки властивостям BSP-дерева всі кімнати будуть доступні між собою.

3.5 Генетичні алгоритми

Генетичні алгоритми — це парадигма, натхненна біологічною еволюцією, при якій кандидати послідовно проходять мутацію та схрещування, а потім відбираються за допомогою функції фітнесу, так що, з часом, найкращі кандидати еволюціонують до найкращого можливого рішення. При застосуванні до генерації рівнів кандидатами є самі рівні, операції мутації та схрещування — це, відповідно, випадкові зміни рівня та поєднання двох різних кандидатів.

Першим кроком є пошук способу представити кандидатів, які зазвичай називаються хромосомами, у відповідному форматі. Проста (але непрактична) альтернатива — представити кожну плитку підземелля як біт, який буде увімкнено, якщо плитка є стіною, а в іншому випадку вимкнено. Інший спосіб — представити кімнати у вигляді дерева, вершини якого представляють кімнати, а ребра — коридори (або з'єднання) між кімнатами.

Потім повинні бути визначені методи мутації та схрещення. Прикладом оператора мутації для деревоподібної структури є випадковий вибір кількох вузлів із хоча б одними доступними дверима (які ведуть до коридору), а потім додавання до них іншої кількості дочірніх вузлів. Для схрещування можна проводити обмін випадковим піддеревом між двома кандидатами.

Нарешті, для відбору необхідно визначити функцію фітнесу. Це означає, що спосіб оцінки якості даного рівня підземелля повинен бути представлений функцією, щоб процес відбору поступово вдосконалював кандидатів на протязі

генерації. Фітнес-модель Вальчанова характеризується тим, що надає перевагу кластерам кімнат з невеликим простором між собою, з'єднаних коридорами. Вона також надає перевагу картам, що містять до трьох унікальних кімнат, розташованих неподалік від країв карти. Ці характеристики аналізуються таким чином, що спочатку деревоподібна структура перетворюється в фактичний рівень, а потім оцінюється.

Після визначення функції фітнесу, кандидати чергової генерації випадковим чином організовуються в групи, які називаються турнірами. Потім, кожен турнір сортує своїх кандидатів за фітнес-функцією, і нижня половина з них замінюється нащадками верхньої половини. Цей процес повторюється задану кількість генерацій або до досягнення певного порогу. На рисунку 3.7 наведено приклад карти, сформованої вищевказаним процесом. На кольорових ділянках вказані спеціальні кімнати [9].

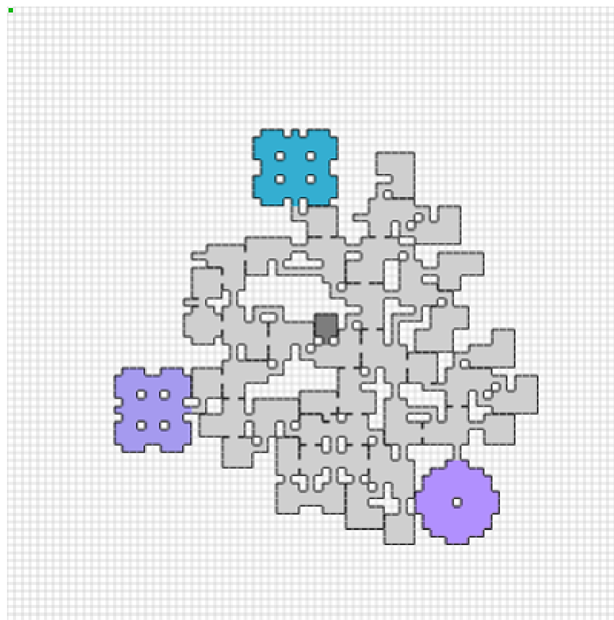

Рис. 3.7 Приклад рівня згенерованого генетичними алгоритмами

3.6 Параметризація

Параметризація — це концепція додавання параметрів, тобто правил за якими повинен працювати генератор, а потім варіювання їх значень, для збільшення різноманітності результатів. Вони також можуть бути встановлені гравцем при початку нової гри, як спосіб генерації однакових рівнів в масштабах всієї гри, а не лише одного рівня. Зазвичай параметрами виступають:

- **Зерно**. Константа (числове значення або рядок), яка використовується для побудови рівня (або підземелля) певним чином. Зазвичай його передають генератору псевдовипадкових чисел, який впливає на все підземелля (або світ). Кілька зерен також можуть бути використані для створення різних елементів гри.
- **Мотив**. Параметр, який представляє тип середовища, яке буде використовуватись при генерації. Наприклад, рівень можна створити в вулканічному, водному або печерному мотивах.
- **Тупики**. Коридори, які нікуди не ведуть, можуть бути створені з заданою частотою.
- **Унікальні особливості**. Цей параметр може означати частоту появи унікальних об’єктів, таких як сховища, магазини та храми у всьому підземеллі.
- **Розмір рівня**. Розмір підземелля можна встановити за допомогою параметра. Важливо зазначити, що якщо розмір підземелля більший за розмір ігрового вікна, необхідно прийняти рішення, який підхід буде використаний для регулювання цього. Наприклад, гра завжди може бути зосереджена на гравці, тому гра промальовує лише ті частини, які видно для поточного положення гравця. Інший спосіб — використовувати вікно-

слайдер, коли гра пересуває екран перегляду щоразу, коли гравець досягає його краю.

- **Тип коридорів**. Коридори можна встановити так, щоб утворювати прямі або більш звивисті візерунки.
- **Складність**. Встановлює складність, з якою гравець буде стикатися протягом гри. Це може впливати на гру по-різному, наприклад, частоту появи монстрів, їхню силу та інтелект, кількість пасток, тощо.

3.7 Аналіз та порівняння

У цій частині буде коротко проаналізовано всі описані вище методи і порівняно їх як за складністю, так і за якістю результатів, починаючи із стандартного алгоритму підземелля. При створенні підземелля з класичним плануванням «кімнати та коридори», техніка ітеративного створення кімнати та з’єднання її із останньою створеною кімнатою за допомогою коридору дасть результати в лінійний результат в плані часі. Визначимо метод setToWalkable для перетворення плитки в прохідну. Нехай n буде максимальною кількістю створених кімнат, і при кожному створенні кімнати доведеться перетворити на прохідні плитки прямокутник комірок з максимальною висотою h і шириною w. Тоді, в гіршому випадку, буде зроблено $n \times w \times h$ викликів методу setToWalkable. Однак, оскільки на практиці w і h обмежені невеликим максимальним значенням (наприклад, 10) через обмеження розмірів кімнат у roguelikes, можна сказати, що алгоритм буде запускати уявний метод makeRoom n разів, роблячи його лінійним за кількістю кімнат. Насправді експерименти показали, що після того як було створено певну кількість кімнат, більшість спроб створення чергової кімнати зазнають невдачі через брак місця, тому ця техніка буде працювати досить швидко навіть для великих рівнів.

З іншого боку лабіринтам буде потрібно трохи більше часу для генерації. Використання пошуку в глибину призведе до того, що всі можливі прохідні клітини будуть відвідані рівно один раз. Так, якщо кількість клітин (або вершин) $V = w \times h$, а $E = V \times 4$ (оскільки кожна клітинка матиме чотирьох сусідів у випадку недіагонального створення лабіринту якщо не враховувати краї карти), вони матимуть час виконання O(V). Важливо зазначити, що, оскільки ця техніка може включати глибоку рекурсію, особливо при великих рівнях, вона може спричинити проблеми із переповненням стеку, тому рекомендується реалізувати її за допомогою ітерацій, зберігаючи інформацію для відстеження в самому лабіринті.

Крускал і Прім, будучи алгоритмами мінімального охоплюючого дерева, будуть використовувати подальші обчислення, такі як сортування ваги ребер. Тому вони будуть працювати повільніше, ніж пошук в глибину. Показано, що Крускал відпрацьовує за $O(E \log V)$, тоді як Prim — за $O(|E| + |V| \log |V|)$. На практиці будь-яка з цих альтернатив буде працювати досить ефективно практично для будь-якого розміру рівня.

У випадку клітинного автомата найбільш інтуїтивний спосіб проведення моделювання для рівня фіксованого розміру — це при кожній генерації розрахувати кількість сусідів для кожної клітини та вирішити, чи стане вона живою, залишатиметься живою чи помре. Тоді нехай для g поколінь, розміру рівня $w \times h$ та максимальної кількості сусідів для клітини дев’ять (включаючи саму клітинку), алгоритм буде виконувати $9 \times w \times h \times g$ оновлень клітин. Однак кожне застосування клітинних автоматів, згадане в цій роботі, використовує фіксовану, невелику кількість генерацій, наприклад п'ять. Це означає, що в даному випадку найгіршим випадком буде $O(w \times h)$. На практиці, завдяки цій фіксованій кількості генерацій, можна припустити, що всі програми клітинних

автоматів у цій роботі будуть працювати досить ефективно практично для будь-яких розмірів карти.

Що стосується техніки розділення бінарного простору, описаної в цій дипломній роботі, кількість поділів, зроблених на карті, буде обмежена заданою кількістю ітерацій або до тих пір, поки вона не досягне мінімального розміру прямокутника. Для версії із заданою кількістю поділів k, алгоритму доведеться запустити функцію makeRoom 2^k разів. Таким чином, важливо зберегти k маленькою, інакше для великих карт кількість викликів функцій може перевищувати розмір стека. Втім, кількість ітерацій, як правило, невелика, наприклад, п’ять.

Згідно із Брауном [9], експерименти з генерацією підземелля за допомогою генетичних алгоритмів, при кількості генерацій до 500 і максимуму в 100 кімнат, рівень генерувався приблизно за 30 секунд, використовуючи одне ядро 2,4 ГГц процесора. Це означає, що якщо цю техніку використовувати при генерації підземелля під час виконання, слід застосувати якийсь прийом, щоб цей час обробки не впливав на ігровий процес. Можливі рішення включають додавання екрана завантаження після того, як користувач переходить на наступний рівень або створення наступного рівня, поки користувач проходить поточний.

Нарешті, оскільки параметризація — це просто спосіб збільшити різноманітність за допомогою конфігурації параметрів, а не самостійна техніка генерації рівнів, час виконання в цьому випадку залежить від методів, що використовуються такою конфігурацією.

Далі буде порівняно вищезазначені техніки на загальну якість. Хоча це доволі суб’єктивна міра, все ж можна виділити деякі характеристики, які допоможуть приблизно порівняти ці техніки:

- **Масштабованість**. Стосується того алгоритм працює для більших розмірів рівня. Ця характеристика тісно пов’язана зі складністю обчислень.
- **Самостійність (Connectivity)**. Визначає чи метод може використовуватися самостійно за замовчуванням, чи для гарантії його роботи потрібно використовувати інші методи.
- **Поєднуваність**. Простота поєднання з іншими техніками або унікальними конструктивними особливостями.
- **Надійність**. Визначає, наскільки часто техніка даватиме потрібний результати протягом визначеного часу.
- **Варіативність.** Наскільки відрізнятиметься кожна випадкова генерація від інших.
- **Унікальність**. Показує, чи дає метод результати, які не можуть дати інші методи.

Виходячи з вищезазначених характеристик та пам’ятаючи, що якість відносна, було вирішено візуально порівняти методи за допомогою таблиці. У таблиці (табл. 3.1) перераховано згадані методи із наведеними для них характеристиками. Коли порівняння не може бути застосоване, використовується «—».

Таблиця 3.1

Техніка	Масштабованість	Самостійність	Поєднуваність	Надійність	Варіативність	Унікальність
Основна ітерація	Масштабована	Самостійний	Так	Надійна	Середня	Не унікальна
Лабіринти	Масштабована	Самостійний	Ні	Надійна	Низька	Не унікальна
Клітинні автомати	Масштабована	Не самостійний	Ні	Не надійна	Висока	Унікальна
BSP-дерево	Не масштабована	Самостійний	Так	Надійна	Середня	Не унікальна
Генетичні алгоритми	Не масштабована	Самостійний	Так	Не надійна	Середня	Не унікальна
Параметризація	Масштабована	—	—	Надійна	Висока	Унікальна

РОЗДІЛ 4

ШТУЧНИЙ ІНТЕЛЕКТ

У цьому розділі буде описано, проаналізовано та порівняно ряд методів штучного інтелекту, які можна застосувати до персонажів в іграх. Більш конкретно, їх буде розглянуто з точки зору застосування в roguelike іграх.

Одна з відмінностей, яка виникає при визначенні штучного інтелекту для ігор є «псевдоінтелект». Це означає, що, незважаючи що персонаж працює як треба, тобто забезпечує виклик і розважає гравця, фактичне програмування іноді може вважатися позбавленим інтелекту. Наприклад, персонажі, що дотримуються набору правил if-then, не можуть вважатися раціональними, тоді як персонаж, який має складне тактичне планування, може навчатися та має емоції, що регулюють його взаємодію з навколишнім середовищем, можна справді назвати «штучним інтелектом». Незважаючи на це, у цій роботі хоч і зауважені ці відмінності, але врешті-решт вони будуть проігноровані. Причиною цього є те, що в досліджуваній області, а саме в іграх, насправді важливим є результат ігрового ефекту (а також продуктивність), а не філософські роздуми.

Наступні розділи будуть містити низку прийомів для штучного інтелекту персонажів, адаптованих до roguelike ігор. Після цього, у розділі 4.8, вони будуть проаналізовані з точки зору складності обчислень, переваг та недоліків, так щоб ми можна було зрозуміти, які методи застосовувати в практичній реалізації самі по собі або в поєднанні.

4.1 Знаходження шляху

Для початку варто рознлянути техніку, яка, незважаючи на те, що фактично є проблемою пошуку найкоротшого шляху графа, є наріжним каменем для більшості ігор, в яких беруть участь персонажі, що пересуваються по місцевості. Вона полягає у пошуку найкращого шляху, який персонаж має пройти, щоб досягти певної мети. Однак визначення «найкращого» шляху залежить від програми, оскільки дизайнер гри повинен встановити параметри для оцінки якості шляху. Наприклад, найпростішим (і найпоширенішим) параметром є відстань, що у випадку roguelike ігор означає кількість квадратів у сітці, яку персонаж повинен пройти, щоб досягти своєї мети. Інші параметри включають відкритість шляху для ворогів, плавність шляху та уникнення небажаних ділянок.

Загалом є кілька технік знаходження шляху, які використовуються в roguelike іграх:

- **Евклідова відстань**. У цій техніці персонаж обчислює пряму лінію між ним та ціллю і рухається в цьому напрямку. Евклідову відстань можна розрахувати таким чином:
  1. Обчислити вертикальну та горизонтальну відстані, *dx* і *dy*, між персонажем та його ціллю;
  2. Обчислити евклідову відстань між ними, використовуючи формулу (4.1.1).

$$\mathrm{D} = \sqrt{(\mathrm{dx})^2 + (\mathrm{dy})^2} \tag{4.1.1}$$

  3. Нормалізувати цю довжину та розділити на дві вісі використовуючи формули (4.1.2) і (4.1.3).

$$dx = \text{round}(\frac{dx}{D}) \quad (4.1.2)$$

$$dy = \text{round}(\frac{dy}{D}) \quad (4.1.3)$$

4. Перемістити персонажа, змістивши його поточне положення на *dx* і *dy*.

Незважаючи на те, що ця техніка проста у впровадженні і потребує незначних обчислювальних витрат, персонаж може рухатися неприродно і застрявати в кутах, тому дизайнеру доведеться знайти способи вирішення цих проблем.

- **Пошук в ширину**. У цій техніці персонаж буде шукати місцезнаходження своєї цілі шляхом ітеративного збільшення діапазону пошуку. Почне з перегляду всіх сусідніх комірок (тобто відстані 1). Якщо ціль знайдена це означатиме, що вона знаходиться поруч з персонажем, і він може діяти. Якщо ні, це збільшить діапазон пощуку на 1, таким чином охоплюючи всі квадрати відстані 1 і 2. Важливо зазначити, що процес не починається спочатку, коли діапазон збільшується, а натомість просто продовжується з раніше відвіданих вузлів. Коли ціль знайдено, алгоритм відстежує квадрати, що привели до цілі, і будує шлях. Цей процес гарантовано закінчується та знаходить ціль, якщо такий шлях існує.
- **Алгоритм Дейкстри**. Це техніка пошуку найкоротшого шляху між двома вузлами в графі. Алгоритм Дейкстри може бути застосований до roguelike ігор, моделюючи карту підземелля як граф, де квадрати в сітці є вершинами, а ребра представляють сусідів квадрата до яких можна переміститись, знаходячись на цьому квадраті. Після цього алгоритм можна застосовувати безпосередньо за призначенням, відвідуючи всі інші квадрати, починаючи від квадрата персонажа, і послідовно обчислюючи

найкоротші відстані до них, поки не буде відвіданий цільовий квадрат, що в подальшому буде означати найкоротшу відстань до нього.

- **Алгоритм A***. Хоча алгоритм Дейкстри гарантовано знайде найкоротший шлях (якщо такий є), в залежності від розміру підземелля та кількості одночасно задіяних персонажів, вартість найгіршого сценарію — це відвідування кожної вершини графа. Алгоритм A* — це узагальнення алгоритму Дейкстри, обчислювальна вартість якої це не реальна вартість відстані а її наближення за допомогою евристики. Завдяки цьому, хоч і ціною втрати гарантії завжди знаходити найкращий можливий шлях, кількість пошукових вузлів може бути значно зменшена, якщо при виборі вдалої евристики.

Нарешті, важливо згадати, що для знаходження шляху можна зробити кілька оптимізацій. Це може бути як оптимізація продуктивності так і естетичності. У випадку A*, як приклад естетичної оптимізації, можна використати процедуру випрямлення шляхів. У ньому, маючи на увазі, що до цілі може бути декілька найкоротших шляхів, ідея полягає в тому, щоб трохи зменшити евристичне значення непрямих шляхів, наприклад, на -0,0000001, так що в кінцевому підсумку найбільш прямий шлях вважатиметься трішки коротшим ніж альтернативи, і таким чином буде обраний. Однак у цьому випадку дизайнер гри повинен пам’ятати про деяку втрату продуктивності, оскільки ця процедура змусить пошуковий алгоритм розглянути набагато більше перестановок, щоб знайти найбільш прямий шлях.

Один із способів вирішення цієї проблеми, який також є прикладом оптимізації продуктивності, полягає у використанні ієрархії шляхів. Ідея полягає в розбитті місцевості на зони (у випадку roguelike ігор, кімната була б ідеальною характеристикою зони), так що персонаж, який хоче досягти цілі в іншій зоні, обчислить лише шлях до наступної сусідньої зони на шляху до цілі

замість обчислення цілого шляху. В результаті, перестановки шляхів суттєво зменшаться, і додаткові кроки для випрямлення шляхів будуть мати незначну обчислювальну вартість.

4.2 Персонажі без станів

Персонажі без станів найпростіша форма штучного інтелекту для ворогів. Вони складаються з набору причинно-наслідкових правил, які персонаж перевірятиме під час кожного ігрового циклу та діятиме на їх основі. Це означає, що вони взагалі не мають внутрішньої пам’яті. Наприклад, простий персонаж міг би виконати таку процедуру:

1. Якщо я можу дістати гравця, то я атакую його;
2. В іншому випадку, якщо я не можу дістати гравця, але бачу його, то рухаюся до нього;
3. Інакше, стою на місці.

Повторюючи цю процедуру кожного циклу, вороги сліпо рухатимуться до гравця і атакуватимуть його. Можна додати подальші перевірки та реакції, щоб симулювати більш складну поведінку. Наприклад, персонаж може перевірити, чи його здоров’я занадто низьке, і втекти від гравця, або застосовувати дальню зброю, коли знаходиться на відстані від персонажа гравця, і застосовувати атаки ближнього бою, коли перебуває біля персонажа.

4.3 Персонажі зі станами

Персонажі, які використовують стани, мають внутрішню пам’ять, тобто вже закладену або отриману під час ігрової сесії інформацію, що допоможе їм діяти більш гнучко. Що стосується закладеної інформації, це може бути знання, про внутрішні параметри персонажа, наприклад, кількість поточних та загальних очків здоров’я, які він має. З іншого боку, інформація, отримана через досвід, отримується завдяки спостереженню за подіями, які відбуваються протягом періоду існування персонажа. Наприклад, ворог може оцінити силу персонажа гравця, спостерігаючи, як він б’ється з іншими ворогами, і втекти, якщо персонаж занадто сильний, або негайно напасти, якщо занадто слабкий. Іншим прикладом може бути ворог, що злиться і переходить із стану обережності на гнів, якщо він спостерігає, як персонаж гравця вбиває іншого ворога того ж виду.

Тип використання станів, який було продемонстровано «тактичним станом». Це форма внутрішньої пам’яті, яка вказує, в якій тактичній ситуації знаходиться персонаж в даний момент. Наприклад, він може ігнорувати гравця, а на основі якоїсь події в грі, наприклад, нападу гравця або будучи загнаним в кут, він може стати ворожим.

Таблиця (табл. 4.1) показує приклад реалізації персонажа зі станами і переходами між ними. У ньому стовпець «Стан» позначає назву стану, «Спостереження» означає рівень уважності персонажа, де «гучний шум» — це звуки, достатньо голосні, щоб пробудити персонажа, «звичайне» — монстр спостерігає за тим, що випадково потрапляє в поле зору, і «розширене» означає уважне спостереження за усім. Стовпці «Гучний шум», «Зниклий скарб», «Голодний» та «Немає» — це правила переходу.

Так, наприклад, якщо актор, який перебуває в стані СОН, чує гучні звуки, він переходить у стан АКТИВНИЙ. На додаток до цього, для кожного стану існує відповідний шаблон поведінки.

Таблиця 4.1

Стан	Спостереження	Гучний шум	Зниклий скарб	Голодний	Немає
СОН	Гучний шум	АКТИВНИЙ		ГОЛОДНИЙ	
АКТИВНИЙ	Звичайне		РОЗЛЮЧЕНИЙ	ГОЛОДНИЙ	ЗАЦІКАВЛЕНИЙ
РОЗЛЮЧЕНИЙ	Звичайне				АКТИВНИЙ
ГОЛОДНИЙ	Звичайне		РОЗЛЮЧЕНИЙ		
ЗАЦІКАВЛЕНИЙ	Розширене		РОЗЛЮЧЕНИЙ	ГОЛОДНИЙ	

4.4 Інтелект натовпу

Інтелект натовпу (або ройовий інтелект) — це техніка інтелекту без використання станів, при якій низка персонажів координує свої рухи так, що ірраціональна поведінка кожного учасника створює колективний інтелект. Одним із прикладів техніки інтелекту натовпу був натхненний способом за яким перелітні птахи формують ключі. Є чотири правила, які називаються кермова поведінка, які можна використовувати для управління групами автономних персонажів, щоб вони представляли реалістичні моделі поведінки:

- **Відокремлення**. Персонаж повинен уникати зіткнень із союзниками.

- **Вирівнювання**. Персонаж повинен рухатись в напрямі середнього напрямку натовпу.
- **Згуртованість**. Персонаж повинен рукатись до геометричного центру натовпу.
- **Уникнення**. Персонаж повинен уникати зіткнень із місцевими перешкодами і ворогами.

Ідея в тому, щоб обчислити вектор швидкості для кожного персонажа в натовпі в кожен інтервал часу, керуючись наведеними вище правилами. З точки зору roguelike гри це означає, що потрібно розрахувати напрямок, в якому рухатиметься кожен персонаж, щоб ці правила виконувались якнайкраще. Крім того, потрібно прийняти рішення як діяти, коли ці правила суперечать одне одному. Простий спосіб зробити це — встановити пріоритети в правилах, наприклад, віддавати пріоритет відокремленню, щоб персонаж намагався з усіх сил не зіткнутись із союзниками.

Дизайнер також може визначити ряд обмежень, які визначатимуть, як персонажі можуть рухатися та реагувати. Це змінить спосіб загальної поведінки натовпу. Одним із найважливіших параметрів можна назвати діапазон сприйняття кожного агента, тобто те, як далеко він може помічати союзників, перешкоди та ворогів. Ще одне обмеження стосується швидкості, яка, у випадку roguelike, — це час, необхідний для переміщення або виконання дії.

Хоча ці чотири правила були визначені в контексті інтелекту натовпу, загальна ідея може бути застосована і до інших моделей. Наприклад, правила можна було б визначити таким чином, щоб учасники натовпу завжди намагалися наблизитися до однієї цілі одночасно, що характеризувало б засідки замість формування натовпу. Існує також кілька інших прийомів для інтелекту натовпу, таких як алгоритм бджіл та колонії мурах, але оскільки їхнє

застосування в roguelike іграх доволі сумнівне, їх не буде досліджено в цій роботі.

4.5 Генетичні алгоритми

Генетичні алгоритми також можуть використані для штучного інтелекту персонажів. Тоді хромосоми представлятимуть набір вбудованих знань та поведінки, які будуть поступово модифікуватися генетичними операторами за допомогою фітнес-функцій. У цьому випадку фітнес-функція вибиратиме найкращі риси для персонажа, наприклад зважене прийняття рішень та природна поведінка. Як це зазвичай буває з генетичними алгоритмами, дизайнер повинен змоделювати та налаштувати алгоритм спеціально для його програми. У випадку з roguelike іграми є кілька параметрів, які потрібно налаштувати[10]:

- **Чисельність популяції**. Загальна кількість учасників у моделюванні. Велика кількість персонажів призведе до того, що процес триватиме занадто довго для співставлення, тоді як мала кількість може призвести до швидкого співставлення, але невражаючих результатів.
- **Швидкість мутації**. Це відображає як частоту, з якою мутації траплятимуться у персонажей, так і кількість змін, до яких призведе кожна окрема мутація. Цей параметр варто розглядати у поєднанні з різними типами операторів мутації та фітнес-функціями.
- **Карта генів**. Карта генів, яку також називають хромосомою, як правило є вектором або послідовністю значень, які будуть представляти характеристики кандидат. Одна з корисних властивостей хромосом це локальність. Це означає, що характеристики, які впливають одна на одну,

повинні бути розташовані близько одне до одного на векторі. Таким чином, якщо для методу комбінації встановлено звичайний оператор схрещення, який полягає у передачі суміжних наборів характеристик обох батьків новому кандидату, то тісно пов'язані характеристики будуть передані разом. Зручний спосіб визначити карту генів для персонажів без станів, які по суті є вкладеними if-умовами, є представити їхню деревоподібну структуру умов у вигляді вектора, при цьому кожне значення представлятиме if-умову. Таким чином, проміжні вузли в дереві представлятимуть if-умови, а листя будуть діями. Це буде чудово працювати із оператором комбінацій як описано нижче.

- **Метод комбінацій**. Завдяки деревоподібній структурі можна отримати ізоморфні відношення між різними кандидатами. Таким чином той самий ген завжди буде означати одне і те ж для будь-якого кандидата. Крім того, оператор комбінації можна визначити як заміну піддерева. Це означає, що можна видалити гілку одного кандидата та замінити її іншою гілкою з іншого кандидата, зберегти складні ієрархічні відносини. Тому бажані комбінації генів будуть вибрані разом під час моделювання, і таким чином співставлення до найкращого можливого набору генів буде швидшим.
- **Фітнес-функція**. Зазвичай найважливіший і найважчий для визначення параметр. І roguelike ігри не є винятком. Фітнес-функції особливо важко визначити для рольових ігор, оскільки більшість ворогів мають дуже коротке життя (час, необхідний гравцеві, щоб побачити і дістатись до них). Одна з можливостей — визначити критерій успішності як середній час життя персонажа, але це означатиме, що вороги почнуть максимально уникати гравця, що навряд буде бажаною поведінкою для більшості видів ворогів. Іншим варіантом є величина пошкоджень, завданих персонажу гравця. Але багато монстрів у roguelike взагалі не атакують гравця, а найскладніші завдають гравцю труднощів за допомогою більш творчих

способів. Врешті решт, непоганою технікою для вимірювання успішності персонажа буде оцінити наскільки дорого він коштуватиме гравцеві в сенсі пошкоджень, витрачених заклинань, пошкоджених обладунків, тощо.

- **Елітарність**. Показує, наскільки більш успішному кандидату буде віддана перевага для розповсюдження генів перед іншими менш успішним. Важливо знайти золоту середину між високою елітарністю, яка потенційно може відкинути хороший генетичний матеріал, і повністю випадковим вибором, який не надасть жодної переваги більш успішним кандидатам. Найліпшою практикою вважається відбір трьох випадкових кандидатів, упорядкуванням їх за успішністю, і призначенні шансів на подальше розмноження і бути заміненими залежно від їхнього рангу в цій вибірці.

4.6 Персонажі, які базуються на емоціях

Використання психологічних моделей емоцій для персонажів, може змусити їх поводитись більш природно і несподівано. Наприклад, монстр буде більш схильний до втечі при зустрічі, якщо його параметр мужності буде нижчим за середній. Подібні моделі емоцій взяті з досліджень у таких сферах, як психологія та когнітивна наука.

Однією з широко використовуваних моделей емоцій є модель OCC [11]. У ній емоції розбиваються на три категорії з урахуванням їх часових рамок:

- **Емоції**. Ця категорія представлена реакціями на події, предмети та персонажей, що виражаються в невеликому часовому масштабі, наприклад хвилинах і навіть секундах. Прикладом для roguelike може

бути монстр, який спостерігає, як гравець атакує іншого монстра тієї ж категорії і злиться через це.

- **Настрій**. Емоційний стан який триває від кількох днів до кількох місяців. Простий спосіб представити це: емоційний стан персонажа змінюється в діапазоні від -1 до 1. Таким чином, негативні значення означатимуть «поганий настрій», що впливатиме на те, як актор сприймає події.
- **Особистість**. Особистість - це сукупність рис, які керують діями персонажа протягом усього його існування, при цьому рідко змінюючись. Модель OCEAN відокремлює такі риси, як відкритість, доброчесність, погоджуваність, екстравертність та невротизм. Кожна з цих ознак представлена числом із плаваючою комою в діапазоні від 0 до 1. Таким чином, наприклад, актор з невротизмом, встановленим на 1, був би дуже схильний атакувати можливого ворога з першого погляду, тоді як якщо встановити значення 0, то завжди чекав би провокації перед нападом.

4.7 Створення екземплярів ігрових сутностей

Створення екземплярів ігрових сутностей — це техніка процедурної генерації контенту (PCG), яка може бути застосована до штучного інтелекту ворогів. Використовуючи її, параметри при створенні персонажа рандомізуються. В результаті виходить набір унікальних персонажів, із статистично незначним шансом повторення. Наприклад, якщо параметри персонажів, що базуються на емоціях, будуть рандомізовані, то з’являтьсь персонажі з різними «особистостями». Приклади параметрів, які можна рандомізувати під час створення персонажа включають:

- **Атрибути**. Силу, спритність, кмітливість та інші можна змінювати, створюючи унікальних ворогів. Хоча вони самі по собі не вважаються методами штучного інтелекту, монстра можна запрограмувати діяти відповідно до своїх сильних і слабких сторін. Так, наприклад, слабкий монстр міг би віддавати перевагу бою на відстані.
- **Особистість**. Як вже зазначалося, методи, засновані на емоціях, можна рандомізувати, щоб популяція монстрів мала більшу різноманітність. У цьому випадку більш статичні або навіть фіксовані аспекти емоцій повинні бути рандомізовані. Такі риси особистості, як екстравертність, невротизм та погоджуваність, можуть бути рандомізовані так, щоб персонажі могли постати перед гравцями з непередбачуваною емоційною поведінкою.

Важливо зазначити, що якщо сліпо рандомізувати параметри персонажів, можуть з’явитися непідходящі результати. Наприклад, ворог, який був рандомізований, щоб мати надзвичайно низьку кількість здоров’я, високі показники дальнього бою, але мав би емоційну схильність до сліпого нападу на гравця з першого погляду, був би непрактичним. Таким чином, важливо, щоб дизайнер керував процесом якимось чином, наприклад, додаючи обмеження та перехресні виключення, щоб не можна було створювати небажані комбінації

4.8 Аналіз та порівняння

У цьому розділі буде коротко проаналізовано та порівняно описані вище методи штучного інтелекту, застосовувані до roguelike. Для початку буде розглянуто проблему знаходження шляху, яка в тій чи іншій мірі присутня для всіх персонажів CRPG.

Знаходження шляху — це проблема пошуку найкоротшого шляху до певної цілі. Воно може заключатися в сліпому руху до цілі по прямій лінії, як у алгоритмі евклідової відстані, або може включати проходження кожної плитки підземелля хоча б один раз. У випадку евклідової відстані процедура є незмінною за часом. У випадку алгоритму найкоротшого шляху Дейкстри при використанні куп Фібоначчі, складність у найгіршому випадку становить $O(|E| + |V| \log |V|)$, де V - кількість плиток на рівні, а E - кількість суми можливостей пересування для кожної плитки (тобто кількість її країв).

Для алгоритму A* складність залежить від використовуваної евристики. Якщо є лише одна ціль, простір пошуку нескінченний і похибка евристичної функції буде зростати не швидше, ніж логарифм гіпотетично досконалої евристики, тоді її складність буде многочленом коефіцієнта розгалудження. У випадку класичних roguelike із квадратною сіткою, кофіцієнт розгалудження ніколи не перевищує восьми, оскільки відвідати можна не більше 8 сусідніх квадратів. Крім того, у roguelike іграх простір пошуку обмежений (обмежений розміром рівня), і для пошуку присутня одночасно лиш одна ціль — персонаж гравця або цільовий квадрат. На додаток, простір пошуку можна значно зменшити, якщо використовувати ієрархію шляхів пошуку, так як із графа всіх плиток він зменшиться до значно меншого графа взаємоз’єднаних кімнат.

Далі буде розглянуто складність персонажів без станів. Враховуючи їх простий рефлексивний характер, їх обчислювальна вартість є незначною. Вони складаються з циклу перевірок, де кожна перевірка — це if-умова із однаковим часом обробки. Таким чином, ця техніка має сталу обчислювальну вартість. Однак якщо якась із перевірок включає операцію, обробка якої займає більше часу, то вся процедура матиме обчислювальну складність цієї операції. Наприклад, якщо персонаж має можливість напасти на всіх інших персонажів, включаючи союзників (він може бути в розлюченому стані), і він повинен

оцінити найкращу ціль для нападу, тоді йому доведеться перевірити список усіх видимих персонажей для такої оцінки.

Таким чином, у найгіршому випадку, коли персонаж може бачити всіх інших персонажів на рівні (наприклад, у великій повністю освітленій кімнаті), обчислювальна складність буде лінійно залежати від кількості монстрів на рівні. Це призводить до висновку, що обчислювальна складність для персонажів без станів точно може бути визначена лише після розробки конкретної процедури. На практиці, як уже зазначалося раніше, їхня обчислювальна складність є незначною.

Подібно до акторів без станів, цикл перевірок для персонажів із станами доволі незначний. Персонаж перевіряє свій поточний стан, вхідні дані та внутрішню інформацію, і використає відповідну поведінку штучного інтелекту [11]. Крім того, якщо вхідні дані спровокують зміну стану, це просто змінить змінну, яка відповідає за поточний стан, що практично не потребує обчислювальних витрат. Таким чином, часова складність для цього типу персонажів залежить від видів методів штучного інтелекту, що застосовуються для кожного з його тактичних станів.

Інтелект натовпу — це форма групової поведінки, яка передбачає персонажів без інтелекту, які діють скоординовано, а це означає, що це також форма інтелекту без станів. Обчислювальна вартість для цього виду техніки залежить від конкретної поведінки, яка буде імітуватися. Наприклад, у випадку зграй, кожен персонаж перевіряє положення всіх видимих союзників і обчислює вектор руху, який би максимально підходив для нього. У найгіршому випадку персонаж не матиме обмежень щодо видимості, і в кожен момент часу він буде бачити всіх союзників одночасно. Це означає, що часова складність буде лінійно залежати від розміру зграї. На практиці більшість методів, які

використовують зграї додають обмеження видимості персонажам, тим самим зменшуючи кількість союзників, яких треба прийняти до уваги.

На відміну від попередніх методів, генетичні алгоритми, застосовані до roguelike ігор, займають занадто багато часу для обчислювання під час виконання програми. Тобто вони вважаються технікою попередньої обробки, за якої закладені знання і попередній досвід використовуються для визначення фітнес-функції, яка буде використана в процесі відбору. Всі параметри процесу повинні бути точно відрегульовані, щоб збіжність до бажаних кандидатів відбулася за необхідну кількість часу. Наприклад, чисельність популяції не повинна бути занадто великою, інакше кожен крок моделювання займатиме неприпустимо великий час. Крім того, оператори повинні бути протестовані так, щоб знайти хороші комбінації, інакше процес може застрягти в локальних оптимумах або навіть взагалі не закінчитися. Таким чином, ігровий дизайнер повинен вирішити, коли варто використовувати цю техніку.

Для представлення фреймворку для персонажів заснованих на емоціях, його ліпше розбити на компоненти, щоб проаналізувати їх окремо:

- **Дерево поведінки**. Дерево поведінки виступає спеціалістом по знанням у дошковій системі. Воно складається з організації виконання планів у дереві завдань. Це означає, що час обробки залежить від складності самих завдань. Наприклад, задача пострілу стрілою у вибрану ціль складається з серії підзавдань, а саме: перевірити, чи є стріли, дістати стрілу, прицілитися в ціль, вистрілити. Всі ці завдання є незмінні в часі, отже, весь процес також є незмінним. Однак бувають випадки, коли потрібні завдання із непостійним часом обробки, такі як вирішити, куди бігти, при втечі від сильного ворога, що потребує використання алгоритмів знаходження шляху.

- **Оцінка**. Виступає в якості основного вхідного процесу в системі. Вона відображає вхідні дані у зміни на дошці. Вона може додавати або видаляти цілі, а також впливати на емоційні значення персонажів, що зберігаються на дошці. Прикладом завдання для модуля оцінки є повернення відсортованого списку сутностей відповідно до їх емоційних значень, використовуючи запит від дерева поведінки. Такий процес лінійно залежить від кількості об'єктів, що перебувають у зоні видимості персонажа. Іншим завданням є просто змінити емоційне значення певного персонажа на дошці, що завжди відбуватиметься за однакову кількість часу.
- **Дошка**. Дошкова система, будучи динамічним сховищем знань, цілей та часткових рішень цілей, не має трудомістких процесів обробки, оскільки це просто місце, де спеціалісти (оцінка, пересування, дерево поведінки, тощо) шукають інформацію та діють відповідно до неї.

Тому, можна сказати, що складність цього фреймворку залежить від особливостей застосування і складності його спеціалізованих модулів. У випадку коли фреймворк емоційних персонажів застосовується до roguelike ігор, він, швидше за все, буде добре масштабуватись практично для будь-яких розмірів рівня та кількості персонажів, оскільки змінні, що беруть участь у вхідних даних, переміщенні та навіть дереві поведінки, як правило, прості.

Обчислювальну вартість методу створення екземплярів ігрових сутностей можна вважати незначною оскільки він просто рандомізує параметри персонажів, які в будь-якому випадку будуть згенеровані.

Далі буде порівняно методи з точки зору їх загальної якості, щоб можна було краще зрозуміти їх сильні та слабкі сторони. Таким чином буде легше вибрати, які з них використовувати самостійно або в поєднанні під час розробки гри. Деякі бажані характеристики цих методів:

- **Масштабованість**. Чи буде техніка працювати достатньо швидко для великих кількостей персонажів.
- **Поєднуваність**. Показує те, наскільки ефективно техніку можна поєднувати з іншими.
- **Надійність**. В рамках штучного інтелекту для персонажів це показує як часто дана техніка дає позитивні результати. При цьому негативним результатом, наприклад, може бути застрягання персонажа чи вчинення помітно безглуздих дій.
- **Унікальність**. Показує, чи буде техніка генерувати поведінку, яку не можуть відтворити інші.

Далі буде порівняно техніки відповідно до цих характеристик. Це може допомогти зрозуміти, які методи найкраще підходять для тих чи інших типів програм. На відміну від розділу 3, порівняння технік буде в текстовій формі, оскільки деякі з них залежать від кількох модулів, які використовуються для їх проектування. Тому таблиця була б недостатньо інформативною і мала б кілька винятків.

Стосовно **масштабованості**, яка тісно пов’язана зі обчислювальною складністю, методам без станів невластиві складні обчислення, і вони масштабуються практично для будь-якої кількості одночасно діючих персонажів для Roguelike ігор, за умови, що їх складові модулі також мають незначні обчислювальні складності. Це також актуально для методів із станами, оскільки додавання станів не спричиняє значних додаткових обчислень. Щодо персонажів, заснованих на емоціях, якщо завдання їхнього дерева поведінки не надто складні і їх модуль оцінки не повинен обробляти занадто багато складних взаємодій з навколишнім середовищем, що, як правило, характерно для Roguelike ігор, вони теж масштабуються. Генетичні алгоритми, навпаки, не повинні мати дуже великий розмір популяції, і їх оператори повинні бути точно

налаштовані, щоб швидко досягти збіжності, інакше використання цієї техніки буде недоцільним. Що стосується знаходження шляху, якщо використовується евклідова відстань або A* з ієрархією шляхів, ця техніка може бути використана для великих рівнів з великою кількістю одночасно діючих персонажів без помітних затримок. Нарешті, створення екземплярів ігрових сутностей не вимагає додаткової обробки, отже, масштабується для будь-якої кількості персонажів.

Що стосується **сумісності**, знаходження шляхів використовується будь-якою іншою технікою штучного інтелекту персонажів, тому він повністю поєднується. Інтелект натовпу, навпаки, не поєднується з іншими технiками, оскільки має свої правила поведінки. Емоційний фреймворк є формою інтелекту зі станами, тому він не сумісний з персонажами без станів. Процедура генетичного алгоритму вибирає найкращого кандидата, і його характеристики визначатимуться його генами, тому ця техніка не поєднується з іншими методами. Нарешті, створення екземплярів ігрових сутностей буде працювати для кожної техніки, що має параметри, які можна рандомізувати.

Надійність методів знаходження шляху для персонажів із та без використання станів завжди працюватимуть як очікувалося. Однак інші техніки можуть дати несподівані результати. У випадку з персонажами, заснованими на емоціях, деякі комбінації рис особистості можуть призвести до поведінки, що шкодить персонажам, наприклад, до суїцидальної або надмірно нерішучої поведінки. Що стосується генетичних алгоритмів, то, оскільки потрібно зробити багато налаштувань, процес часто призводить до помилок, і в результаті може відбутися зближення до «дефективних» персонажів. Те саме стосується, хоч і в меншій мірі, методів інтелекту натовпу, оскільки поведінка та обмеження управління персонажів також повинні бути відрегульовані, тому без ретельного тестування може відбуватися нестабільна поведінка. Нарешті, рандомізація

параметрів без якихось обмежень може призвести до непрактичних комбінацій параметрів, тому дизайнеру потрібно продумати обмеження для цієї техніки.

Нарешті, коли мова заходить про **унікальність**, варто зазначити, що більш унікальна, цікава поведінка трапляється, коли використовуються більш досконалі методи. Так, наприклад, хоча персонажі без станів мають просту природу, додаючи до них велику кількість умов, дизайнер може очікувати передбачуваних, але більш цікавих результатів. Однак унікальну поведінку легше відтворити використовуючи техніки з різними пропозиціями, таких як інтелект натовпу та додавання емоцій до персонажів. Також варто згадати, що, генетичні алгоритми можуть давати цікаві результати, і більше того можуть бути запрограмовані як для персонажів із станами, так і без них. Зрештою, найкращою способом додати унікальності до поведінки персонажів є використання рандомізації параметрів під час створення екземплярів об’єктів.

РОЗДІЛ 5

РЕАЛІЗАЦІЯ

У цьому розділі буде докладно розглянуто процес розробки roguelike гри для цієї дипломної роботи. У наступному розділі буде описано та проаналізовано загальну реалізацію гри з практичної точки зору. Потім, у наступних розділах, буде обговорено подальший розвиток деяких методів, розглянутих у розділах 4 та 5. Більш конкретно, буде обґрунтовано вибір деталей реалізації, зроблених для цих функцій після їх теоретичного аналізу. Нарешті, в останньому розділі ми представимо результати процесу порівняльного аналізу, зробленого для цієї роботи, який включає впроваджені методики для створення підземелля та штучного інтелекту. Крім того, на основі цих результатів ми порівняємо ці методи щодо продуктивності.

5.1 Середовище розробки

Мовою програмування, яка використовувалася для розробки прототипу, був C#. Розробка проводилася на ПК з використанням процесора Intel® Core™ i7-7500U на 2,7 ГГц, з 8 ГБ оперативної пам'яті. Операційною системою була Microsoft Windows 10 Home. Для розробки використовувався двигун Unity версії 2019.4.15f1.

5.2 Загальна реалізація

У цьому розділі буде докладно розказано про загальну реалізацію прототипу. Як згадувалося в розділі 2, ігровий дизайнер повинен вирішити, як

будувати ігрові процеси та переходи між ними, спричинені взаємодією гравця з грою. Такий процес називається ігровим робочим процесом. Рисунок 5.1 показує високорівневу абстракцію робочого процесу для гри, розробленої для цієї дипломної роботи.

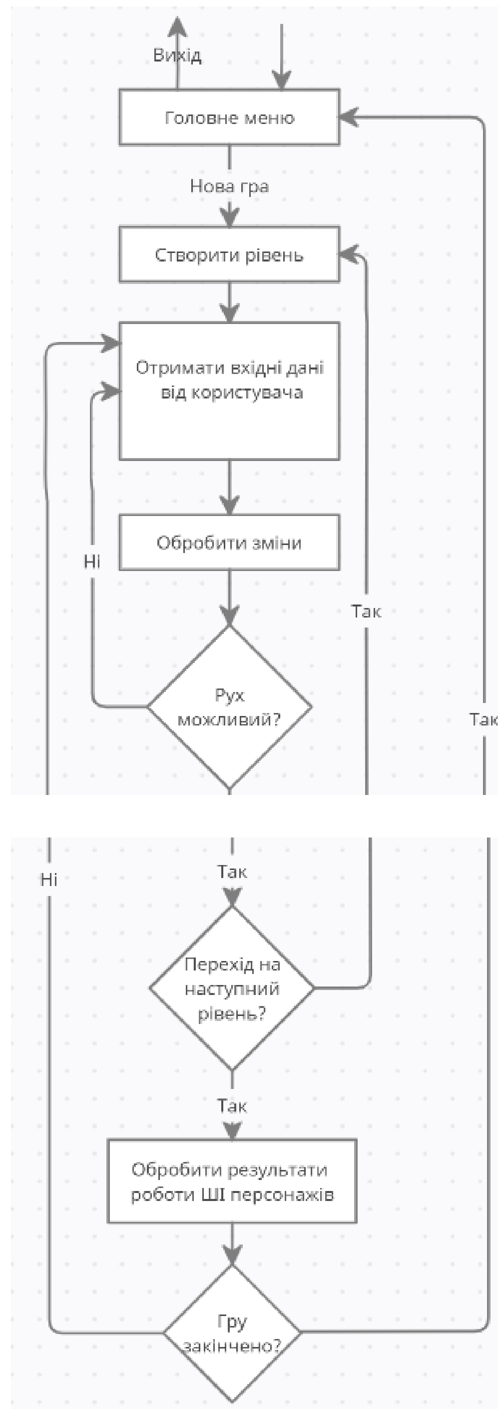

Рис. 5.1 Робочий процес для розробленого прототипу гри

У грі, використовується режим реального часу: персонажі діють одночасно і незалежно одне від одного.

Для графічного представлення гри частково використовувалися спрайти, завантажені із Unity Store (для персонажів) і частково — власне створені (для рівнів). Для фону головного меню і рівня також використовувалися зображення із Unity Store. На малюнку 5.2 показано типовий сеанс гри. У ньому гравець вже дослідив певну частину рівня і вирішує, куди піти далі.

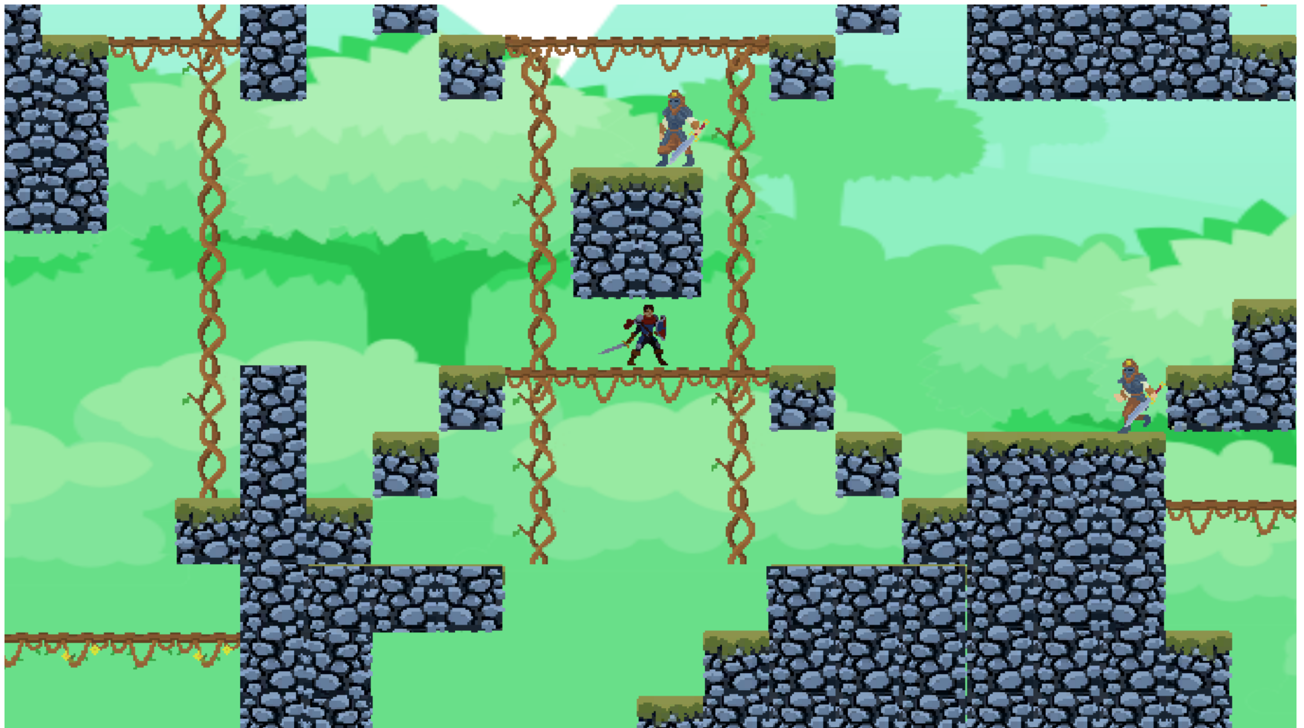

Рис. 5.2 Типова сесія гри, розробленої для дипломної роботи

В інтерфейсі користувача, було дотримано традиційної для РПГ-ігор системи управління, яка покладається на взаємодію як із клавіатурою, так і мишею. У ньому набір клавіш зіставляється з відповідними ефектами в грі, наприклад: A, S, D використовуються для переміщення персонажа, “Space” для стрибку, “E” для взаємодії з об’єктами, ЛКМ для ближньої атаки, ПКМ для дальньої атаки.

5.3 Процедурна генерація рівня

Першим методом генерації підземелля, з яким проводилися експерименти, був базовий ітераційний підхід, який полягав у створенні кімнат різних розмірів у випадкових положеннях на рівні, а потім їх з’єднанні. Його названо ітераційним підходом через спосіб з’єднання: для кожної нової кімнати центр з’єднується із центром раніше створеної кімнати. Таким чином, зв’язок між кожною кімнатою є гарантованим. Параметри для цього процесу — це максимальна кількість кімнат (яка може не бути досягнута залежно від розміру підземелля) та мінімальний та максимальний розміри кімнат.

Наступним методом, який було реалізовано, були клітинні автомати, в яких життя «організмів» моделюється за допомогою двовимірного прямокутника на основі сітки використовуючи набір правил розмноження. Спочатку весь рівень складався тільки із платформ, і визначав "організми" як пустий простір крізь який персонаж гравця зможе рухатись. Потім було додано випадково розміщені плитки простору по всьому рівню у співвідношенні 50% від кількості платформ. Після цього було проведено симуляцію, дотримуючись правила 4-5, яке говорить, що плитки з менш ніж 4 живими сусідами помирають, від 4 до 5 залишаються незмінними, а з більш ніж 5 стають живими (що в даному випадку означає стають простором). Параметри цього процесу — кількість поколінь та стартове співвідношення стін / підлога. На рисунку 5.5 показано підземелля, створене за допомогою техніки клітинних автоматів за правилом 4-5.

Далі буде розглянуто останню техніку, яка і була реалізована в кінцевому прототипі.

Рівень складається із 24 кімнат. Кожна кімната має габарити 10*10 плиток. Кімнати поділяються на 4 типи в залежності від того, з якого боку до

неї розташовані входи: Bottom-Top, Bottom-Top-Left, Bottom-Top-Right і Bottom-Top-Right-Left. Кожну із цих кімнат було створено вручну. При генерації рівня використовується такий алгоритм:

1. Згенерувати стартову кімнату в лівій нижній частині карти.
2. Згенерувати кімнату зліва, справа або знизу від попередньо згенерованої.
3. Створити екземпляр випадкової кімнати на згенерованій на кроці 2. кімнати так, щоб її вхід співпадав із виходом попередньої.
4. Повторювати кроки 2 і 3 допоки не буде досягнуто кінця рівня.
5. Заповнити решту рівня кімнатами так, щоб входи і виходи співпадали з уже створеними кімнатами.

Такий алгоритм дозволяє отримати процедурно згенерований рівень, використовуючи попередньо розроблені кімнати. При цьому гравець завжди матиме змогу пройти від початку рівня і до кінця. На рисунку 5.3 зображено приклад всього рівня. Зеленим кольором виділено одну із 24-х кімнат, жовтими стрілками показано найкоротший шлях від початку до кінця рівня.

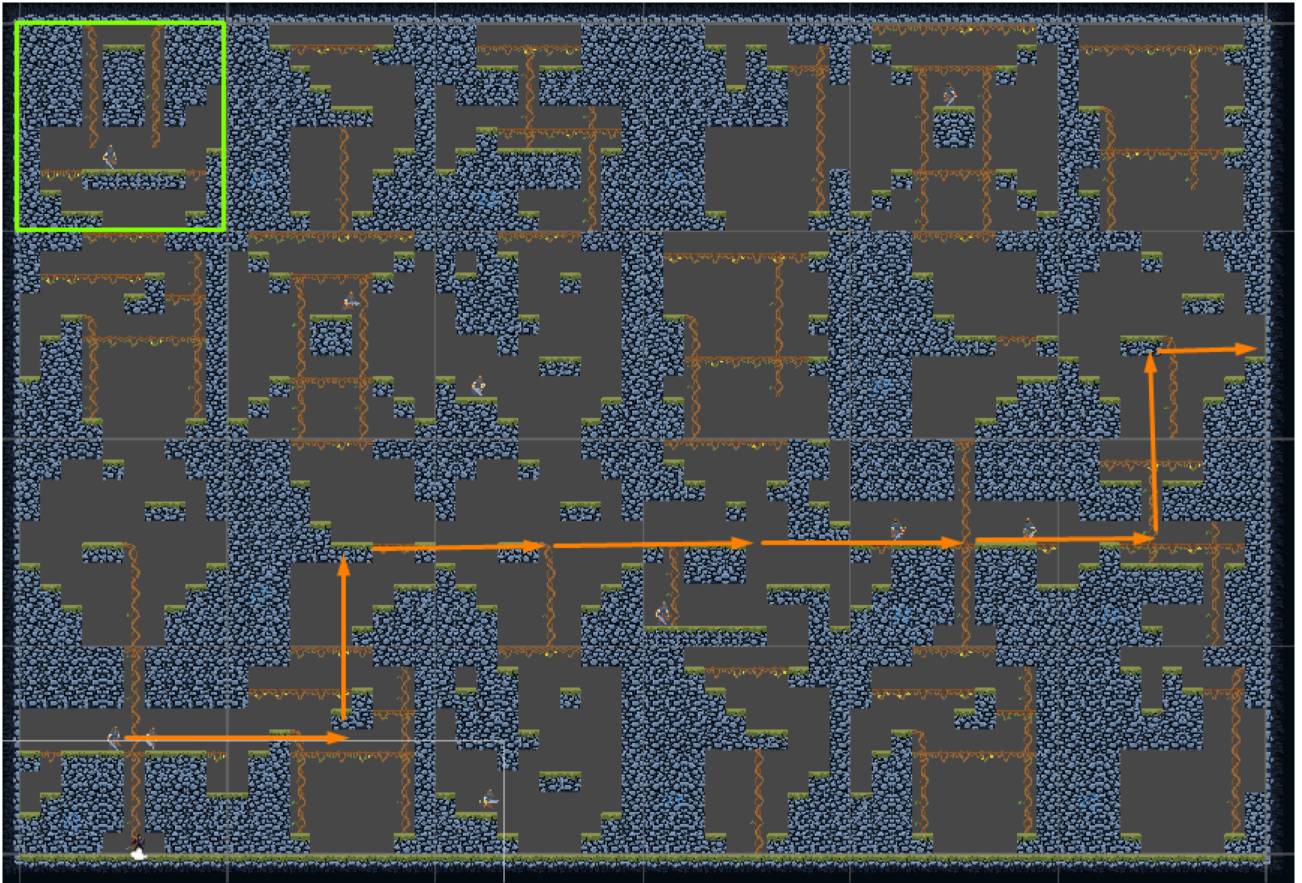

Рис. 5.3 Приклад згенерованого рівня із виділеними кімнатою та найкоротшим шляхом

5.4 Штучний інтелект

Першим кроком при впровадженні штучного інтелекту в roguelike грі є вирішення питання про те, як персонажі рухатимуться до своїх цілей (як правило, до гравця). Це називається проблемою знаходження шляху. У розробленому прототипі проводились експерименти з двома різними підходами:

- **Евклідове переслідування**. В цій техніці персонаж, який помічає персонажа гравця, розраховує найкоротший вектор відстані до нього, використовуючи метод евклідової відстані. Після цього він рухається до наступного квадрата у напрямку до гравця. Хоча цей процес незначний із

точки зору обчислювальних витрат, він має недолік: якщо прохід вже зайнятий (стіною , наприклад), персонаж буде стояти на місці. Таким чином, потрібно було перевірити, чи зайнятий прохід, і якщо так, рухатися до сусідньої плитки в цьому напрямі;

- **Пошук в ширину**. Процес пошуку цілі шляхом пошуку в ширину є безпомилковим, хоча і дорогим. Пошук рекурсивно розглядає усі квадрати в заданому радіусі перед тим, як збільшити значення радіуса 1. Через це навіть невелика кількість одночасно діючих персонажів може призвести до помітної затримки, якщо не буде запроваджено якихось обмежень.

Що стосується фактичної поведінки ворогів, спочатку персонажа було реалізовано без використання станів. Це означає, що персонаж не мав ні внутрішньої інформації, ні станів, на яких він може базувати свою поведінку. З одного боку, це означає, що персонажі такого типу будуть менш гнучкими і важче адаптуватися до різних непередбачених ситуацій під час гри. З іншого боку, їх легко реалізувати і, як правило, для них потрібні незначні обчислювальні витрати. Розроблена версія складалася з простого персонажа, який просто сліпо переслідував гравця, щойно бачив його.

Актори із використанням станів, навпаки, мають внутрішню інформацію, яка може бути вбудованою набутою під час ігрової сесії. У даній реалізації інформація відображається у формі поведінкових станів, які змінюються в залежності від отриманої інформації. У таблиці (табл. 5.1) показано логіку переходу між станами такого персонажа, реалізованого для остаточної версії прототипу. «Бездіяльність» означає, що персонаж стоїть на місці, «Патрулювання» — переміщається в межах заданої зони, «Атака в ближньому бою» — атакує персонажа гравця використовуючи атаку ближнього бою,

«Атака в дальньому бою» — атакує персонажа гравця використовуючи атаку дальнього бою.

Таблиця 5. 1

Стан	Є ворог в полі зору	Немає ворога в полі зору	Ворог достатньо близько	Ворог не достатньо близько
Бездіяльність	Патрулювання	Патрулювання		
Патрулювання	Атака в дальньому бою	Бездіяльність		
Атака в ближньому бою		Бездіяльність		Атака в дальньому бою
Атака в дальньому бою		Бездіяльність	Атака в ближньому бою	

Також була використана техніка створення екземплярів ігрових сутностей: у згенерованих ворогів було рандомізовано певні атрибути (такі як швидкість і кількість здоров’я).

5.5 Тестування продуктивності

При побудові бенчмаркових структур для реалізованих технік потрібно було визначити метрики та параметри:

- Для **генерації рівнів** — інтервал розмірів рівня, який буде перевірятися, а також приріст розміру між кожною ітерацією. Потрібно було знайти найкращу можливу конфігурацію, щоб зафіксувати зростання часу обробки в залежності від обраної техніки, і в той же час уникати розмірів, занадто великих для практичних цілей. Через було визначено розміри рівня в **діапазоні від 100*100 до 1000*1000, з кроком 50** для обох вимірів. Також довелося експериментувати з кількістю циклів в кожній ітерації, щоб не перевантажувати операційну систему. Було встановлено, що **20 циклів** дають задовільні результати, не витрачаючи при цьому непрактично велику кількість часу.
- Для методів штучного інтелекту, інтервал кількості одночасно діючих персонажів був встановлений **10-200, з кроком 10** персонажів для кожної ітерації. Так само як і для генерації рівнів, було проведено **20 циклів** для кожної кількості персонажів.
- Нарешті, для кожної техніки також було визначено їх конкретні параметри, такі як мінімальний та максимальний розмір кімнати. Їх буде описано при аналізі відповідної техніки.

Перша протестована техніка була базова ітераційна генерація рівнів. Максимальна кількість кімнат була встановлена до 1000. Причиною є зберегти фіксоване значення для цієї величини у всіх бенчмарках, а також те, щоб кількість кімнат не була меншою за розмір рівня, оскільки така техніка створить додаткові кімнати тільки якщо для них залишається місце. Діапазон розмірів кімнати було встановлено 9-10 для обох вимірів. На рисунку 5.4 наведено графік аналізу ефективності для базового ітераційного підходу. З нього видно, що інтервал від 100 до 250 можна вважати постійним у часі (незначна тенденція до зниження зумовлена коливаннями), що можна пояснити тим, що для цього діапазону розмірів рівня алгоритм виконується швидше, ніж верхня межа

точності функції вимірювання часу. Після цього спостерігається лінійне зростання функції, що підтверджує очікування щодо лінійності цього процесу від кількості кімнат доданих до рівня. Таким чином, можна зробити висновок, що цей прийом буде працювати і для набагато більших розмірів рівня.

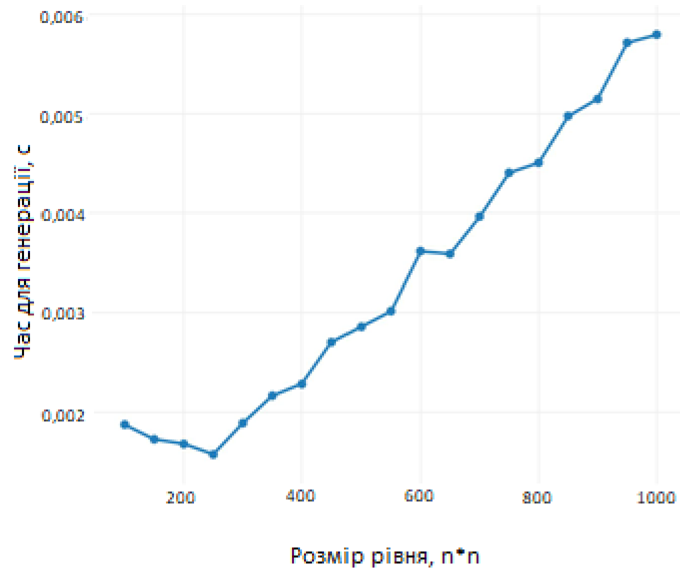

Рис. 5.4 Відношення часу генерації до розміру рівня для базового ітераційного підходу

Для тестування техніки бінарного розділення простору, було встановлено кількість ітерацій до 4, які генерували ідеальну кількість та розмір кімнат. Розмір кімнати коливався в межах від 9 до 10, як і в базовому ітераційному підході. Нарешті, горизонтальне і вертикальне співвідношення було встановлено до 1.0, що, здавалося, давало задовільні результати після певного тестування. На рисунку 5.5 показано аналіз ефективності BSP-дерева. Порівняно з базовим ітераційним підходом, можна бачити, що BSP метод є на порядок більш трудомісткою. Це пов’язано з експоненціальним характером техніки, який обмежений через малу кількість ітерацій, яку було встановлено.

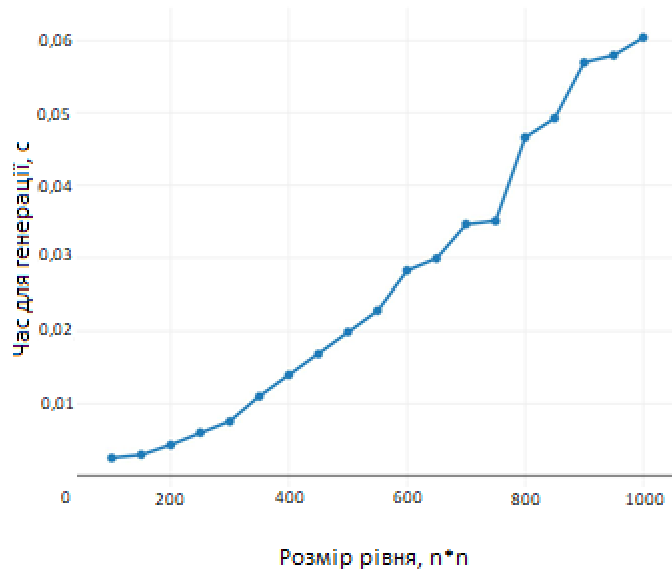

Рис. 5.5 Відношення часу генерації до розміру рівня для техніки BSP-дерева

Для клітинних автоматів було визначено однакові шанси що плитка буде згенерована як платформа або як простір. Тобто в середньому, половина рівня буде «живою» при створенні. Крім того, кількість циклів було встановлено до 20, що може здатися надмірним, але після експериментів було виявлено, що додаткові цикли роблять отриманий рівень більш органічним. На рисунку 5.6 наведено графік аналізу продуктивності техніки клітинних автоматів. Видно, що результуюча функція росту прагне до параболи (що було б ще більш наочним, якби було протестовано на більших розмірах рівня, але було вирішено протестувати всі методи з однаковим діапазоном розмірів). Також варто зауважити, що алгоритм на кілька порядків повільніший за попередні методи. Ці факти свідчать, що техніка стає лінійною від загального розміру рівня, або квадратичною від n. Більш конкретно, вона перевіряє сусідів для кожної плитки (включаючи саму себе) на кожній ітерації, що в цілому становить $9n^2$ перевірок для кожного циклу.

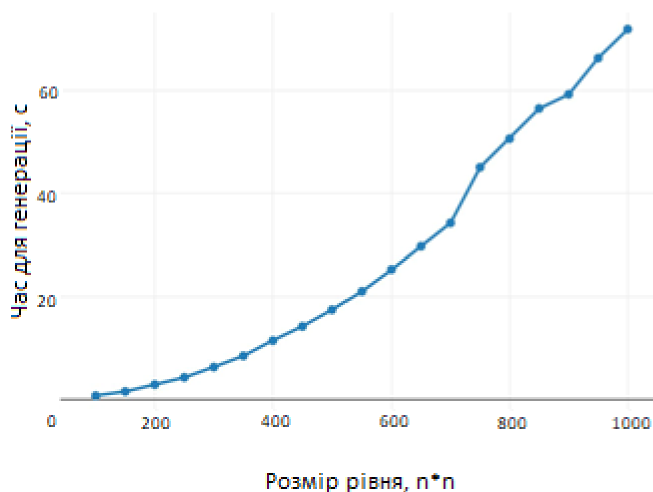

Рис. 5.6 Відношення часу генерації до розміру рівня для техніки клітинних автоматів

Остання техніка, яка була протестована — це генерація рівнів на основі спроектованих кімнат. Ця техніка показала майже миттєві результати, - 0,001 секунда для рівня 1000*1000 і була вибрана для подальшого проектування.

Після цього було порівняно три техніки штучного інтелекту для знаходження шляху: евклідове переслідування, пошук у ширину, та пошук у ширину з оптимізацією. Важливим параметром, який слід визначити, є розмір рівня, оскільки він є визначальним фактором для пошукових алгоритмів знаходження шляху. Було встановлено розмір рівня як 200*200 для всіх випробувань, оскільки таким чином буде достатньо місця для всієї кількості одночасно діючих персонажів. Причина, через яку було вирішено протестувати штучний інтелект з точки зору кількості одночасно діючих персонажів, а не за іншими видами метрик є те, що такий підхід допоможе зрозуміти практичні обмеження наведених технік з точки зору одночасності, що і було метою.

Для евклідової погоні для ворогів не було встановлено обмежень поля зору, що означало, що всі вороги бачитимуть персонажа в будь-який час. Крім цього, ніяких інших параметрів встановлювати не потрібно. На рисунку 5.7 показано аналіз ефективності техніки евклідової погоні. З графіка видно, що загальна вартість обчислень усіх одночасно діючих персонажів відповідає

лінійній функції зростання. Це відповідає очікуванням, оскільки для кожного персонажа витрачається незмінна кількість часу, тому обчислювальна складність лінійно залежить від кількості персонажів.

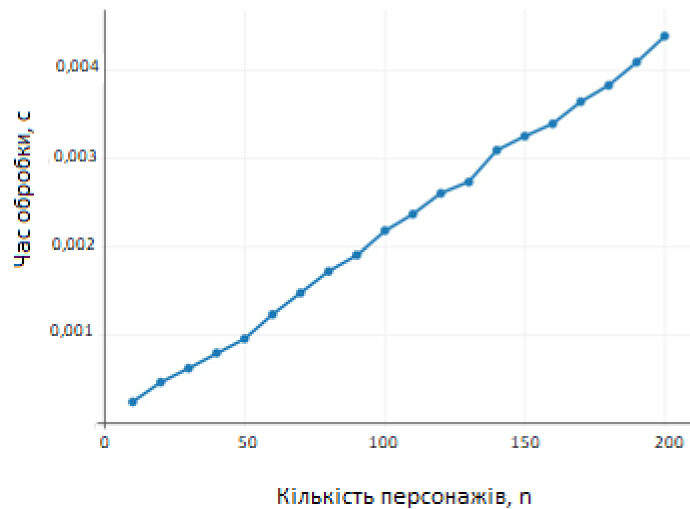

Рис. 5.7 Відношення часу обробки до загальної кількості персонажів для евклідової погоні

Нарешті, буде розглянуто знаходження шляху за допомогою пошуку в ширину. Проаналізовано дві версії цієї техніки:

- Найгірший сценарій, коли всі вороги можуть бачити гравця (необмежене поле зору). Крім того, для цього тесту не було додано жодних заходів оптимізації, щоб можна було порівняти ці результати з оптимізованою технікою.
- Оптимізована шляхом обчислення шляхів лише для ворогів, які бачать персонажа гравця. Для цього експерименту діапазон поля зору було встановлено до 9.

На рисунку 5.8 показано аналіз ефективності для обох сценаріїв пошуку в ширину. Видно, що просто обмежуючи знаходження шляху для ворогів, які бачать гравця можна різко зменшити необхідні обчислення. Важливо зазначити, що, оптимізована версія виглядає константою через невелике поле зору, що означає, що лиш кілька ворогів виконуватимуть пошук одночасно. Насправді,

найгірший сценарій — це, та сама техніка за винятком того, що з необмеженим діапазоном поля зору всі вороги будуть виконувати пошук.

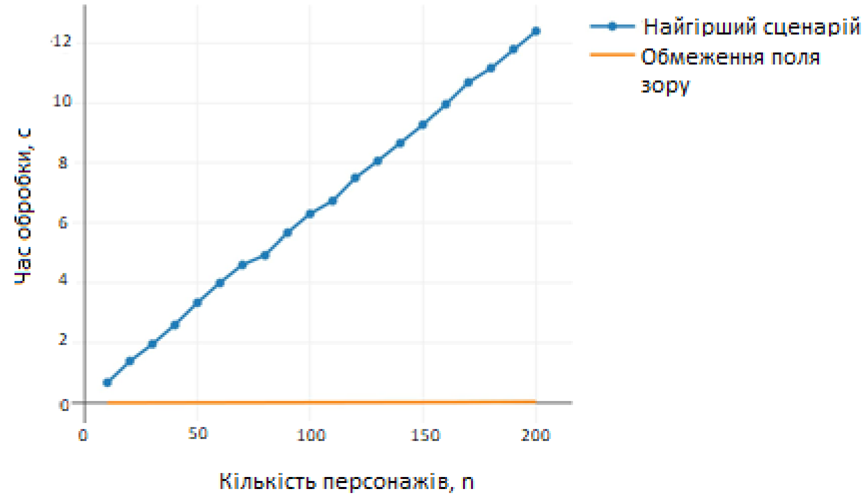

Рис. 5.8 Відношення часу обробки до загальної кількості персонажів для знаходження шляху за допомогою пошуку в ширину

ВИСНОВКИ

У цій роботі було створено дизайн і розроблено гру жанру roguelike. Було зосереджено на двох основних конструктивних особливостях, а саме на процедурному генеруванні рівнів та штучному інтелекті для ворогів. По-перше, було описано декілька технік для реалізації обох особливостей для вибраного жанру. Потім було реалізовано деякі з цих методів та досліджено на продуктивність і загальну якість, щоб можна було зробити практичні висновки щодо їх життєздатності для ігрового прототипу, розробленого для цієї дипломної роботи.

Стосовно процедурної генерації рівня було досягнуто висновку, що методи базової ітерації та BSP- дерева можуть бути використані для створення рівнів практично будь-якого розміру через їх низькі обчислювальні витрати. Для клітинних автоматів було виявлення, що для того, щоб він працював на більших розмірах рівнів на практиці, потрібно або зменшити кількість ітерацій до невеликої кількості (нижче 5), або впровадити деякі із запропонованих методів оптимізації для процесу моделювання. Генерація рівнів шляхом використання вже розроблених кімнат показала найліпші результати при низьких обчислювальних витратах.

Що стосується штучного інтелекту для ворогів, було виявлено, що техніка знаходження шляхів за допомогою евклідової відстані може бути використана для будь-якої кількості одночасно діючих персонажів. Однак через ігнорування перешкод необхідно додавати обмеження і додаткові умови, щоб персонажі не застрягали в кутах чи позаду інших персонажів. Також для знаходження шляхів було показано, що використовувати пошук і ширину без суворого обмеження простору пошуку недоцільно. Однак після простого обмеження кількості персонажів, що обчислюють шлях, процес стає незначним за вартістю

обчислення. Що стосується фактичної поведінки персонажів, було показано що як персонажі із використанням станів, так і без них можуть використовуватися без значних обчислювальних витрат. Однак було доведено що використання персонажів зі станами забезпечує більш яскравий ігровий процес та непередбачувану поведінку, в порівнянні із персонажами без станів.

Щодо практичних знань, які було отримано в процесі розробки можна навести:

- Створення гри вимагає великої віддачі та ретельно продуманого проектування, щоб з часом можна було легко вносити необхідні модифікації;
- Розробники roguelike ігор історично завжди віддавали перевагу глибині контенту над естетичною привабливістю, оскільки для впровадження навіть простих графічних удосконалень потрібен значний час;
- Вивчення нової теми за допомогою незнайомого ігрового рушія має переваги, такі як отримання практичного досвіду. Однак воно також має і недоліки. Головним чином: повільніший процес розробки, оскільки також доводиться вивчати особливості рушія і різні нюанси.

Крім того, хочеться вказати на деякі напрямки, в яких ця робота, а також гра, можуть бути досліджені далі:

- Впровадження більшої кількості унікальних ворогів із своєю логікою;
- Розробка більшої кількості контенту: квестів, артефактів, обладунків, дружніх NPC, тощо;
- Розширення можливостей персонажа гравця: додати можливість руйнувати платформи, робити ривок.

Можна сказати, що було досягнуто кінцевої мети роботи, яка полягала в тому, щоб впорядкувати загалом неформальний, нестандартизований процес

розробки гри жанру roguelike, за допомогою алгоритмічних прийомів, які були широко вивчені та формалізовані в науковому середовищі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Класична Roguelike (https://blog.roguetemple.com/roguelike-definition/)

2. «Берлінська інтерпретація» (http://www.roguebasin.com/index.php?title=Berlin_Interpretation)

3. Дж.Г.Ковер: “Створення сюжету в настільних рольових іграх”, Джефферсон, Північна Кароліна: МакФарланд і компанія, 2010.

4. М.Бартон: «Створення комп’ютерних рольових ігор. Частина 1: ранні роки» (https://www.gamasutra.com/view/feature/3623/the_history_of_computer_.php)

5. Г.Р.Віхман: «Коротка історія гри Rogue» (http://www.wichman.org/roguehistory.html)

6. «Воскреси розробку ADOM» (https://www.indiegogo.com/projects/resurrect-adom-development#/)

7. Т.Адамс: «Dwarf Fortress» (http://www.bay12games.com/dwarves/features.html)

8. А.Доул: «Погибель дизайнера рівнів: генерація процедурного наповнення в іграх» (http://roguelikedeveloper.blogspot.com/2008/01/death-of-level-designer-procedural.html)

9. В.А.Браун, «Еволюція рівнів *dungeoncrawler*ів за допомогою відносного розташування», із матеріалів п’ятої Міжнародної конференції Комп’ютерних Наук і Програмної Інженерії, Нью-Йорк, США, с. 27-35, 2012

10. Р.Диллінжер «Інтелект Roguelike — Генетичні Алгоритми і Еволюціонуюча машина станів для ШІ» (http://www.roguebasin.com/index.php?title=Roguelike_Intelligence_-_Genetic_Algorithms_and_Evolving_State_Machine_AIs)

11. IEEE International Conference on Advanced Trends in Information Theory (ATIT), А.О. Прудченко, К.В.Герасименко – «Проблема вибору СУБД в сучасних інформаційних системах» (https://ieeexplore.ieee.org/document/9030517)